EMCE - Institute of Electrodynamics, Microwave and Circuit Engineering
Vienna University of Technology

Gusshausstrasse 25
1040 Vienna
www.emce.tuwien.ac.at

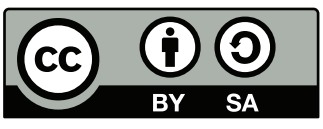
# Diploma Thesis

## FPGA-based Load-Pull Measurement System

Gernot Vormayr
0425210

September 28, 2015

Supervisors
Ass.Prof. Dipl.-Ing. Dr.techn. Holger Arthaber
Univ.Ass. Dipl.-Ing. Dr.techn. Thomas Faseth



## Abstract

Load-pull is a non-linear measurement setup which operates by presenting a specific impedance to a device under test. One type of such setups is active load-pull which retransmits a modified received wave back to the device under test. Generally, a load-pull system is intended to characterize non-linear devices like transistors and can also be used to test devices under different operating conditions. Traditional load-pull measurement systems are narrowband in nature.

During the course of this thesis an active FPGA based load-pull measurement system capable of synthesizing reflections with a constant reflection coefficient over a bandwidth of 20 MHz was developed and verified in measurements. The goals were achieved by synthesizing and filtering the reflection waveform digitally in baseband. This filtering allowed to apply phase correction to the synthesized waveforms for a cyclic signal which was used to achieve a zero phase delay between the reflected wave and a later cycle of the received wave at the load reference plane. Furthermore, an iterative target algorithm based on measurements using an also realized 1-port vector network analyzer was developed to achieve accurate reflection coefficients. This algorithm needs no calibration and allows the verification of highly non-linear devices under test which can exhibit load dependent behaviour.

The digital reflection generation was implemented in hardware on an FPGA. In combination with an efficient pipelined design this allows for fast update rates. Additionally, software running on a processor contained within the FPGA was developed to control the digital hardware via Ethernet which is needed for automated test bench setups.

Finally, the implementation was verified by comparing the realized reflection coefficients to measurements carried out using a commercially available vector network analyzer connected to the load-pull setup as the device under test.

## Kurzfassung

Load-Pull ist eine nicht lineare Messtechnik bei der einem Prüfobjekt während einer Messung verschiedene Terminierungen angeboten werden. Aktive Load-Pull Systeme lösen diese Aufgabe mit zurück transmittieren einer modifizierten Version des Ausgangssignals an das Prüfobjekt. Diese Art Messung wird benötigt um nicht lineare Bauteile, wie z.B. Transistoren, zu charakterisieren und kann auch verwendet werden um Bauteile unter realen Bedingungen zu testen. Traditionelle Load-Pull Systeme sind von ihrer Natur aus eher Schmalbandsysteme.

Im Zuge dieser Diplomarbeit wurde ein aktives Load-Pull Messsystem aufgebaut und charakterisiert, das Reflexionen über eine Bandbreite von 20 MHz mit konstanten Reflexionskoeffizienten erzeugen kann. Dies wurde mittels digitaler Reflexionssynthese und Filterung im Basisband erzielt. Die Filterung ermöglicht das Ausgleichen der Gruppenlaufzeit verursacht durch Kabel im Messaufbau unter Nutzung eines zyklisch periodischen Eingangssignals durch angleichen der Phase des synthetisierten reflektierten Signals an zukünftige Signalperioden. Weiters wurde ein iterativer Algorithmus entworfen, der, basierend auf Messungen eines zusätzlich implementierten Netzwerkanalysators, den gewünschten Reflexionskoeffizienten erreichen kann. Dieser Algorithmus setzt keine Kalibration voraus und wird benötigt um Prüfobjekte mit lastabhängigem Verhalten testen zu können.

Die digitale Reflexionsynthese ist in einem FPGA implementiert. Diese effiziente, pipeline-basierende Implementierung erlaubt hohe Aktualisierungsraten. Weiters wurde eine Kommunikationssoftware entwickelt die auf dem in dem FPGA integrierten Prozessor läuft. Damit wird die Fernsteuerung der digitalen Hardware mittels Ethernet ermöglicht was in automatisierten Prüfaufbauten benötigt wird.

Abschließend wurde das Messsystem verifiziert indem die erzeugten Reflexionskoeffizienten mit Messungen eines kommerziell erhältlichen Netzwerkanalysators verglichen wurden.

## Acknowledgements

I would like to thank my parents for their support during my long journey through university. Especially I would like to thank Katharina for proofreading this thesis and for giving me the strength to withstand the sprint during the end of my study.

Additional thanks go to my supervisors who always responded quickly to my questions, provided me with lots of suggestions, and helped me understanding RF.

I also wish to thank the members of the microwave laboratory for their continuing help, support, and food supply despite their lack of use of the microwave oven for cooking.

# Contents

# List of Figures

# Acronyms

**ADC** analog-to-digital converter. 9, 15–19, 22, 23, 25–28, 31, 36, 54, 55, 64, 69, 71

**API** application programming interface. 36, 44

**CF** compact flash. 35

**CPU** central processing unit. 22, 23, 33, 34, 44

**DAC** digital-to-analog converter. 9, 20

**DC** direct current. 8, 14–16, 24, 33, 68

**DDR** double data rate. 33

**DFT** discrete Fourier transform. 10, 11

**DRAM** dynamic random access memory. 34

**DUT** device under test. 2–5, 7, 8, 11–13, 19, 20, 37, 47, 49, 50, 52, 53, 55, 56, 60, 63, 71

**ELP** envelope load-pull. 7, 8, 14, 16, 52

**EMT** electromechanical tuner. 4

**FFT** fast Fourier transform. 10, 19, 20, 25, 28–31, 47, 48, 55, 57, 67

**FIR** finite impulse response. 7, 19, 25, 55, 57

**FPGA** field programmable gate array. 9, 18, 21–24, 26–28, 32–37, 45, 46, 50, 52, 55, 60, 63, 64, 70, 71

**GPIO** general-purpose input/output. 34, 35

**HTML** HyperText Markup Language. 45

**IF** intermediate frequency. 7, 8, 14–17, 19, 52, 53

**IP** internet protocol. 45

**IQ** in-phase/quadrature-phase. 7–9, 16, 19, 21, 22, 24, 25, 28, 30–33, 63, 67

**JSON** JavaScript Object Notation. 36, 45, 46

**LED** light emitting diode. 35

**LO** local oscillator. 19, 23, 71

**LP** load-pull. 2–7, 19

**LUT** lookup table. 29

**LVDS** low voltage differential signalling. 16, 17, 21, 33, 66

**MW** microwave. 1, 2

**OS** operating system. 23, 35, 36, 46, 65

**OSERDES** output serializer/deserializer. 33

**PA** power amplifier. 2, 3, 63

**PC** personal computer. 8–10, 17, 22–24, 36, 52, 61

**PLB** processor local bus. 23, 26, 27, 31, 35

**RAM** random access memory. 22–25, 27–29, 32, 34

**RF** radio frequency. 1, 2, 14–16, 52, 57, 64

**ROM** read only memory. 34

**S-parameter** scattering parameter. 1, 2, 48, 53, 63

**SATA** serial AT attachment. 17, 18, 22, 25, 26

**SMA** sub-miniature version A. 17, 18

**SNR** signal-to-noise ratio. 19, 55

**TCP** transport control protocol. 46, 63

**UI** user interface. 36

**VHDL** very high speed integrated circuit hardware description language. 22, 32, 65

**VNA** vector network analyzer. 2, 8–10, 12, 15, 17, 37, 47, 52–56, 58, 60–63, 71

**XPS** Xilinx Platform Studio. 34, 35, 43

# List of Symbols

$a$ incident wave. 1–12, 14, 15, 17, 21, 48–50, 52

$b$ reflected wave. 1–12, 14, 15, 17, 48–50, 52

$\mathcal{DFT}$ discrete Fourier transform. 11, 19

$f$ frequency. 10, 11, 13, 53, 54

$f_0$ fundamental frequency, center frequency. 7, 8, 15, 16, 52

$f_s$ sample frequency. 10, 16

$\Gamma$ complex reflection coefficient. 2, 13, 14

$\Gamma_L$ complex load reflection coefficient. viii, ix, 3–15, 17, 21, 22, 24, 32, 36, 37, 47–53, 55–60, 62–64, 68

$j$ imaginary unit. 7–9, 13, 17, 22, 24

$L$ overlap add block size. 19, 20, 25, 30, 31, 45, 46, 55, 60, 65, 67

$N$ number of samples, periodicity of signal. 10, 11, 20, 28, 31–33, 47, 55, 60

$n_{\text{fft}}$ fast Fourier transform size. 19, 20, 25, 29, 31, 47, 55, 60, 67

$S$ scattering matrix. 1, 2, 11–13, 47, 49, 56

$x[n]$ sampled signal $x$ at sample position $n$. 11, 19, 20, 23

$Z$ complex impedance. 2, 13

$Z_0$ complex characteristic impedance of system. 1–3, 13

$Z_L$ complex load impedance. 3, 5, 7, 8, 10, 13, 15, 17, 52

# 1 Introduction

At radio frequency (RF) and microwave (MW) frequencies, the circuit theory with lumped elements, where voltage and current do not vary over the physical dimension of the elements, is of limited value. The wavelength at these frequencies is of the order of the circuit element dimensions. This means that transmission line theory has to be used instead [1]. This theory applies circuit theory to infinitesimally small pieces of the lumped elements and introduces the concept of forward and backward traveling power waves.

For this reason, instead of impedance- and admittance matrices, scattering parameters (S-parameters) are commonly used in RF and MW circuit engineering to describe N-ports (see figure 1.1 and equation (1.1) for a 2-port). $a_k$ denotes the incident and $b_k$ the reflected power wave. The complex valued $S_{kk}$ represents the part of $a_k$ that gets reflected at port $k$, where as $S_{kl}$ is the part of $a_l$ that is transmitted to $b_k$. A set of S-parameters is only valid for a specific frequency, the characteristic impedance $Z_0$ of the system, and a well defined reference plane (port 1 and port 2 in figure 1.1).

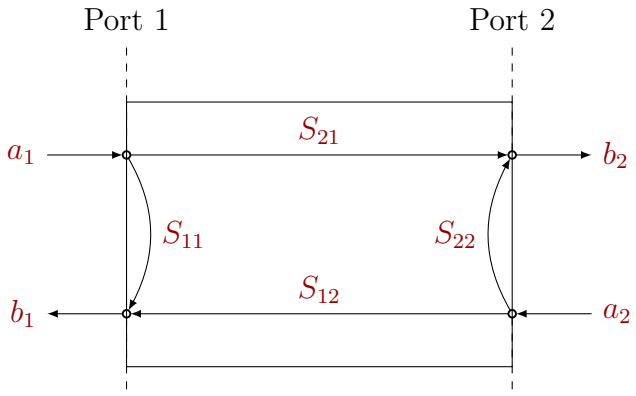



Figure 1.1: S-parameters of a 2-port

$$\begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \tag{1.1}$$

For a 1-port there is only the parameter $S_{11}$, which is equivalent to the reflection co-

efficient $\Gamma$ and can be also expressed by the input impedance $Z_{in}$ (see equation (1.2) as shown in [1]). This does not necessarily hold for 2-ports, since for a generic multi-port configuration, also reflections from the devices connected at the other ports can be seen.

$$\Gamma = S_{11} = \frac{Z_{in} - Z_0}{Z_{in} + Z_0} \tag{1.2}$$

One way to evaluate the S-parameters at a specific frequency would be connecting the reference impedance $Z_0$ to the ports which are not under test. For example, to measure $S_{11}$ of a 2-port, a matching impedance equal to the ports reference impedance has to be connected to port 2. According to equation (1.2) the reflected wave $a_2$ at port 2 is zero in this case. This means that measurements at port 1 can only see the reflections caused by $S_{11}$. Consequently $S_{11}$ can now be calculated by measuring the incident and reflected wave at port 1 (see equation (1.3)) [2].

$$S_{11} = \left.\frac{b_1}{a_1}\right|_{a_2=0} \tag{1.3}$$

$$S_{21} = \left.\frac{b_2}{a_1}\right|_{a_2=0} \tag{1.4}$$

Because of connectors and cabling it is often impossible to connect exact matches. Therefore, S-parameters are measured by sending a power wave into port 1 of the device under test (DUT), measuring $a_k$ and $b_k$. After that the measurement is repeated with port 2. With measurements from both ports and calibration measurements, which are necessary to account for systematic errors in the measurement setup, it is possible to determine every S-parameter. Vector network analyzers (VNAs) use this method and can make automated measurements at various frequencies.

As long as a network behaves linearly, what is typical for small incident signals, the S-parameters fully describe this network at a specific frequency. Thus S-parameters can also be used to model transistors, as long as those exhibit controlled and linear behaviour. For example, transistors used in class-A power amplifiers (PAs) behave nearly linear and S-parameters can therefore be used to describe the small signal behaviour. But those PAs exhibit a very low efficiency of around 50 %. In modern RF and MW applications the efficiency is increased with specially designed input and output matching networks, that improve the performance. This comes at the cost of non-linear behaviour of the transistor in use [3].

Because of these non-linearities more complex models and verification systems are needed. One way to measure the characteristics of a non-linear device is the so called load-pull (LP) technique (see figure 1.2). This measurement system presents a controllable load impedance (output tuner) to the DUT. The fixed input tuner is needed to match the input of the DUT to the source. LP can be used for obtaining

the DUT characteristics and/or for verifying an implementation. LP also allows to test the DUT under realistic operational conditions.

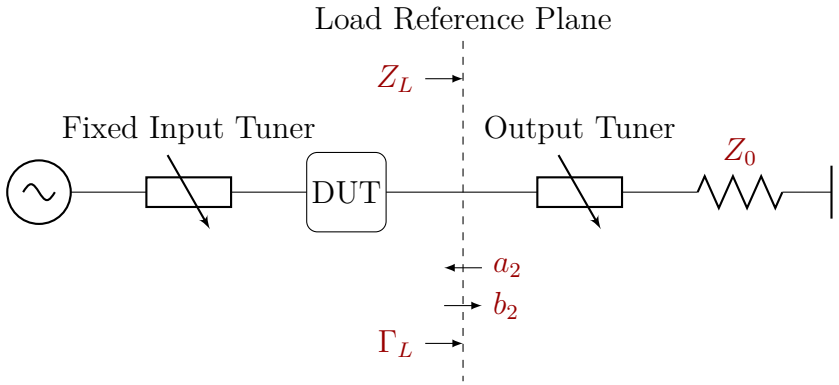



Figure 1.2: Load reflection coefficient

The equations (1.5) and (1.6) show the relations between the load reflection coefficient $\Gamma_L$, the incident wave $a_2$, the reflected wave $b_2$, and the load impedance $Z_L$ at port 2 of figures 1.1 and 1.2. $Z_0$ is the reference impedance of the system where the DUT is going to be used [4].

$$\Gamma_L = \frac{a_2}{b_2} \tag{1.5}$$

$$\Gamma_L = \frac{Z_L - Z_0}{Z_L + Z_0} \tag{1.6}$$

The output tuner in figure 1.2 synthesizes a desired $\Gamma_L$ either by varying the phase and magnitude of the reflected wave $a_2$ or by varying the load impedance $Z_L$. This means, that it is possible to build a LP setup by either using a passive tuner, or feeding a modified wave back to the DUT.

There are various types of LP measurement setups, which have different characteristics. One important feature of a LP setup is the repeatability of reflection coefficients. The repeatability is needed to ensure accurate application specific device models. Another important factor is the tuning range, which depicts the maximal achievable range of the reflection coefficient $|\Gamma_L|$ (e.g. figure 1.3). Usually passive LP systems have a more limited tuning range than active ones, but provide a better repeatability [3]. Tuner resolution is an additional performance characteristic of LP measurement systems. High resolution is needed since PAs are often highly sensitive to impedance variations. However a high resolution incurs a high number of measurement points which increases the overall measurement time. Power handling capability is another extremely important factor. The LP setup has to be capable to sustain the power presented to the tuner without damage. Which LP setup to choose for a specific DUT depends on these requirements.

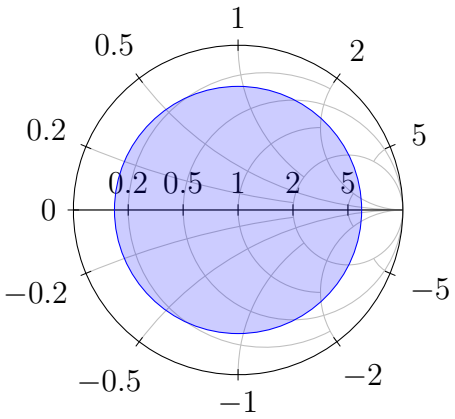



Figure 1.3: Achievable tuning range passive LP system

Passive LP systems are based on the block diagram in figure 1.2. Depending on the desired measurements, additional circuit elements like directional couplers, power meters, and oscilloscopes are needed. For higher power measurements additional amplifiers might also be needed. These additional components, the cabling, and connectors add additional loss on the reflection path, leading to achievable reflection levels $|\Gamma_L| < 1$ with a maximum usually around 0.75 at the load reference plane [5] (see figure 1.3).

Typically used tuners consist of a transmission line and a probe that introduces a mismatch by adding a parallel susceptance. Varying the position of the probe along the transmission line changes the phase of the impedance mismatch and the distance the magnitude [4]. Automated positioning can be achieved by adding motors. These tuners are called electromechanical tuners (EMTs). Those have to be calibrated before use and typically synthesize around 10,000 points [3].

If higher $|\Gamma_L|$-levels are needed (even levels >1), active LP setups have to be used. There are two categories: Closed- and open-loop.

Active closed loop LP setups synthesize a modified reflected wave $a_2$ by modifying the phase and magnitude of $b_2$ and feeding it back to the DUT. The functional block diagram, which can be seen in Figure 1.4, is an example of such an active closed loop LP setup. With the circulator the power wave $b_2$ is fed to a variable attenuator, a phase shifter and an amplifier. These elements modify the phase and the magnitude of $b_2$, which is again fed back to the DUT with the circulator. Because of limited isolation provided by real world circulators the loop will oscillate if the loop gain exceeds the isolation [3]. The isolation can be improved with an additional isolator after the amplifier. Another way to improve the stability of the loop is to introduce a bandpass filter into the loop which prevents oscillations at frequencies outside the band of interest.

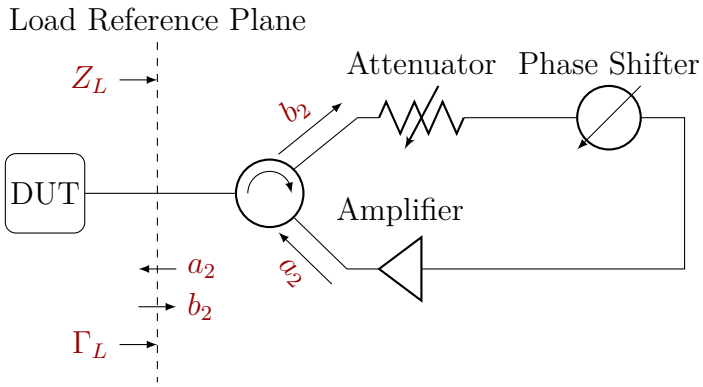



Figure 1.4: Active closed loop LP block diagram

Active open loop LP setups work by synthesizing a phase coherent wave with an external signal generator (see figure 1.5). The open loop approach has the advantage, that no loop oscillations are possible, since there is no closed loop. It works by synthesizing a signal with a source that is locked to the generator driving the source port of the DUT. Phase and magnitude can be adjusted with the attenuator and phase shifter. Additionally the amplifier has to be protected from possibly damaging input signals with an isolator. A disadvantage of the open loop system is that the synthesized waves $a_2$ are not derived from $b_2$. This implies that the reflection coefficient $\Gamma_L$ not only depends on the phase shifter and attenuator settings, but also on the DUT itself. Therefore, an iterative approach is needed to achieve desired $\Gamma_L$ [6].

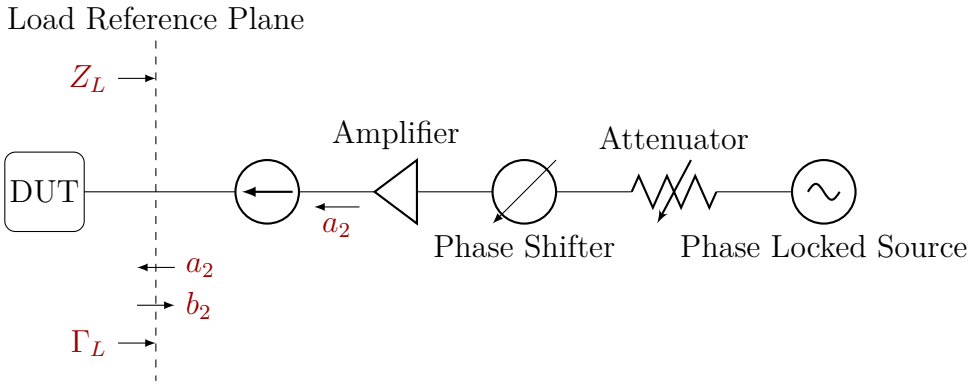



Figure 1.5: Active open loop LP block diagram

In order to provide a realistic termination with an LP setup the variation of the reflection coefficient $\Gamma_L$ should be minimized at the load reference plane. Because of phase variations caused by cable connections needed to reach the DUT, all of the above mentioned setups are only capable of synthesizing narrowband reflections. Therefore, a phase correction is needed for wideband measurements which are needed for modulated signals used in modern communication systems.

During the course of this thesis an active closed loop LP system that uses a digital filter for phase correction is presented. With this filter reflections with minimal $\Gamma_L$ variation at the load reference plane over a bandwidth of 20 MHz can be synthesized. This wideband system works by modifying the incoming wave $b_2$ according to the desired $\Gamma_L$, applying an overall negative phase correction at the load reference plane by filtering the signal, and sending the modified wave back to the device. The negative phase correction is only possible for a later repetition of the signal. Therefore, cyclic signals have to be used which allow completely compensating the group delay by matching the phase of the reflected wave $a_2$ to the phase of the incident wave $b_2$ at a later signal cycle at the load reference plane.

# 2 FPGA-based Load-Pull Measurement System

The aim of this thesis was to setup a wideband active LP measurement system, capable of providing a bandwidth of 20 MHz. As mentioned in chapter 1 traditional classic LP systems can only synthesize narrowband reflections. To overcome these limitations an envelope load-pull (ELP) measurement system, which synthesizes the reflections in the digital domain, was chosen as the base for this work. This approach solves part of the stability problem and the problem of the intermediate frequency (IF) calibration with a fully configurable digital finite impulse response (FIR) filter. Furthermore, this filter is capable of compensating the group delay caused by the measurement setup and the cabling needed to reach the DUT. This allows synthesizing constant reflection coefficients over a wide bandwidth, which in turn allows the use of modulated signals used in modern applications during measurements.

Instead of traditional active closed loop LP, this system synthesizes the load coefficient at baseband or IF. The basic principle of this ELP system can be seen in figure 2.1. A circulator is used to split the incident and reflected wave. The mixer *mix1* is used for shifting the spectrum to the baseband and for in-phase/quadrature-phase (IQ) demodulation. This IQ signal is then multiplied by the complex valued

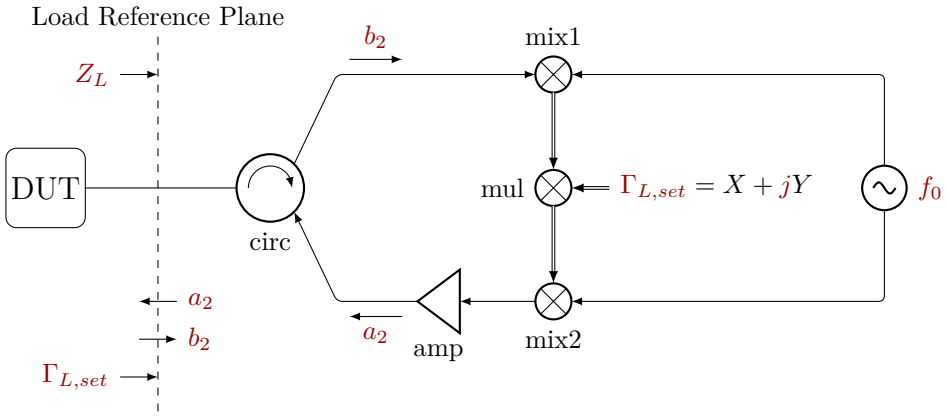



Figure 2.1: Generic block diagram ELP

$\Gamma_{L,set}$ (*mul1*), thus creating a specific reflection coefficient $\Gamma_L$. Mixer *mix2* is used for modulating the IQ signal and upconverting the signal back to the desired frequency [7].

The multiplication can be done in the analog domain, as has been described by [7]. But this has the disadvantage that an additional image will be generated by the multiplicator *mul1* (figure 2.1), if there are amplitude imbalances in the output of the demodulator *mix1*. Since this image is very close to the carrier signal, it can't be filtered out [8].

Phase variation with the frequency at the reference plane caused by the cabling needed to reach the DUT prevents the ELP measurement system from synthesizing realistic terminations. Additionally, the frequency responses of the components used to build the measurement setup cause variations in the magnitude and phase of $\Gamma_L$ with the frequency. Therefore, a negative phase correction and a magnitude correction is needed. The design in [9] solves this problem with a configurable digital delay line which is able to compensate the phase for cyclic signals. This works by matching the phase of the reflected wave $a_2$ to the phase of the incoming wave $b_2$ at a later signal cycle at the load reference plane. However, the design in [9] uses direct conversion, which creates additional direct current (DC) components in the IQ signals. Since these have to be removed by filters, the band around 0 Hz in the baseband is not usable for reflection synthesis. Furthermore, the delay line approach can only compensate phase differences and has a limited resolution.

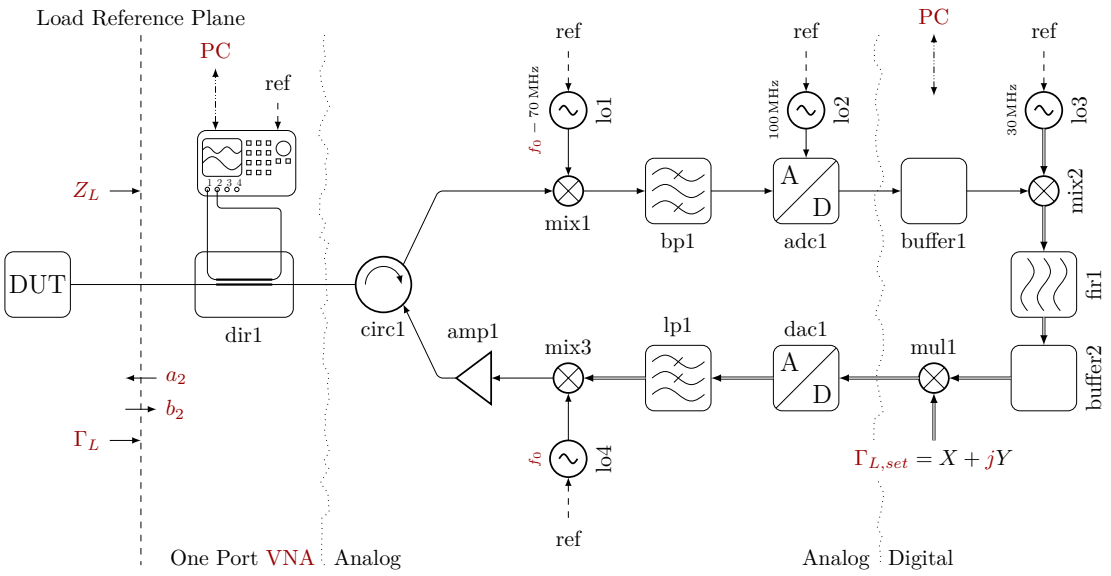



Figure 2.2: System overview

As can be seen in figure 2.2, instead of direct conversion a superheterodyne design with digital IF processing was used. With this design the band around 0 Hz is

usable and the measurement system can't exhibit IQ imbalances. Furthermore, the delay line was replaced by a fully configurable filter *fir1*. This filter is able to provide a negative phase correction for a later repetition of the measured signal $b_2$. Therefore, this filter allows synthesizing reflections with a nearly frequency independent $\Gamma_L$ over the required bandwidth of 20 MHz for cyclic signals. The setup consists of three distinct parts:

1. A one port VNA, to measure the current $\Gamma_L$. This part is needed, to reach a specific $\Gamma_{L,target}$ iteratively and is described in detail in section 2.1. The iterative algorithm can be found in section 4.3.

2. The analog part, which contains the mixers for the frequency shifting operation, necessary filters, a digital-to-analog converter (DAC), an analog-to-digital converter (ADC), and an amplifier. A detailed description can be found in section 2.2.

3. A digital processing chain is implemented in an field programmable gate array (FPGA), which is controlled with a personal computer (PC) running Matlab. The FPGA implementation is described in chapter 3. The software running on the processor contained in the FPGA, as well as the Matlab code, in chapter 4.

## 2.1 One Port VNA

The one port VNA in figure 2.3 consists of a directional coupler *dir1* and a sampling oscilloscope. The directional coupler is needed to split up the signal into the incident power wave $b_2'$ and the reflected power wave $a_2'$. With the help of the oscilloscope both signals can be measured in the time domain. Additionally a PC connected to the oscilloscope is needed for extracting the wave parameters from the measured samples and the necessary error correction calculations, which are explained in the rest of this section.

According to

$$\begin{aligned} \Gamma_L' &= \frac{a_2'}{b_2'} \\ &= \frac{|a_2'|}{|b_2'|} e^{j(\arg a_2' - \arg b_2')} \end{aligned} \tag{2.1}$$

only $\frac{a_2'}{b_2'}$ is of interest for calculating $\Gamma_L'$. Since only the ratio of the magnitudes and the phase difference is needed, the exact point in time, when the signals are taken does not matter, as long as both are measured at the same time which is guaranteed

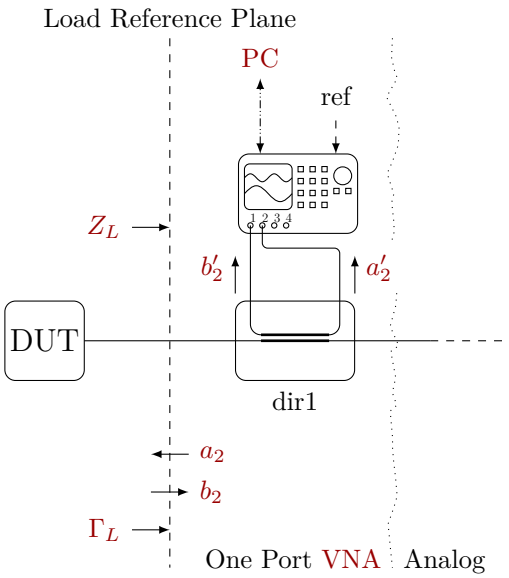



Figure 2.3: One port VNA part

by the use of the oscilloscope. Therefore, no special triggering or synchronization is necessary.

Magnitude and phase of the measured waves can be calculated by using the discrete Fourier transform (DFT). If multiple frequency components are of interest, the computational power needed can be lowered by using the fast Fourier transform (FFT). With both methods spectral leakage will occur, which is caused by the windowing since only a limited number of samples is used [10]. Another problem is that the frequency resolution $\Delta f$ is limited by the sampling frequency and the number of recorded samples:

$$\Delta f = \frac{f_s}{N} \tag{2.2}$$

Higher resolutions can be achieved with a higher sample rate $f_s$ and/or more samples $N$. Both are oscilloscope and setup dependent parameters and potentially increase the time needed for transferring the data to the PC. A preferably way to improve the results is applying a window function to the measured samples.

The window function used in this work (see section 4.3) is the flat top window. This window function has a very high amplitude accuracy [11]. Since it's frequency response is very flat in a small frequency range around the selected $\Delta f$-bin, frequency components leak into the surrounding bins with the same amplitude as in the original bin. This enables acquiring frequency components, that are not $\Delta f$-aligned [11]. In this work, the periodic version of the built-in Matlab flat top window `flattopwin` was used (see section 4.3 and [12]).

$$k = \left\lfloor \frac{f}{\Delta f} \right\rfloor \tag{2.3}$$

$$w[n] = \texttt{flattopwin}(N, \texttt{'periodic'}) \tag{2.4}$$

$$A'[n] = \mathcal{DFT}(a'[n]\, w[n]) \tag{2.5}$$

$$B'[n] = \mathcal{DFT}(b'[n]\, w[n]) \tag{2.6}$$

$$\Gamma'_{L,k} = \frac{A'[k]}{B'[k]} \tag{2.7}$$

Using equations (2.2) to (2.7) the reflection coefficient $\Gamma_L$ at frequency $f$ (frequency bin $k$), with the measured $N$ samples $a'[n]$ of wave $a'_2$ and $b'[n]$ of wave $b'_2$, can be calculated. First, the frequency resolution $\Delta f$ is needed, to calculate the index $k$ of the frequency bin, which contains the frequency $f$. Next, the window function $w$ with length $N$ is obtained with the Matlab function `flattopwin` as mentioned earlier. This window function is element-wise multiplied with the input signal in equations (2.5) and (2.6) before applying the DFT. After the transformation $A'[n]$ and $B'[n]$ contain the signals in the frequency domain. Because of the linearity of the DFT [13], the value at index $k$ can be directly used in equation (2.1) leading to equation (2.7).

The measurements acquired using this type of setup contain systematic errors. These are caused by mismatches and imperfections in the equipment, which are the limited directivity of the coupler and imperfect connectors and distort the measurements. Furthermore, the needed cabling causes the reference plane to be shifted to another place, than depicted in figure 2.3. Those errors can be corrected using vector error correction. By measuring the systematic errors with known calibration standards it is possible to calculate the error model and use this model to remove the systematic errors from the subsequent measurements [14].

Figure 2.4 depicts the error model for the measurement system in figure 2.3. It consists of the DUT and an error box, containing all the systematic errors of the measurement system. This error model can cancel out three different errors:

1. Source match $S_{22}$, the mismatch between the measurement system and the DUT.

2. Directivity $S_{11}$, which characterizes signal leakage and imperfections in the coupler.

3. Reflection tracking $S_{12}$, which characterizes the difference in the frequency response between the two oscilloscope ports, including loss in the couplers, transmission lines, and other components.

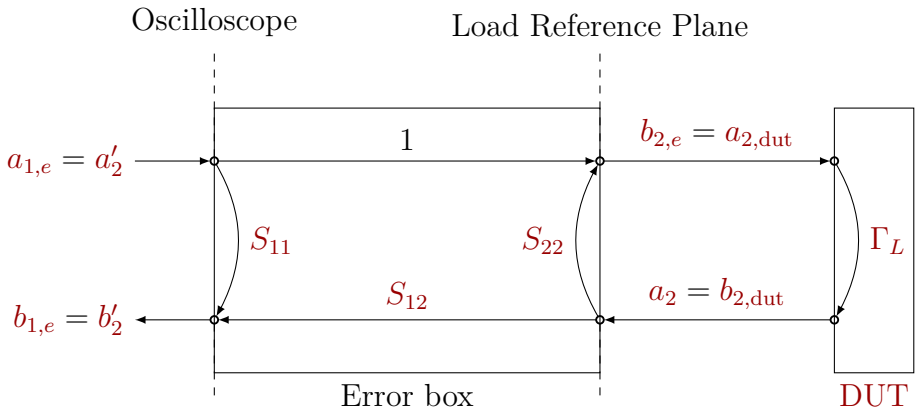



Figure 2.4: Error box for the one port VNA

From the error model in figure 2.4 the following equations can be derived:

$$b_{2,e} = a_{1,e} + S_{22}a_{2,e} \tag{2.8}$$
$$b_{1,e} = S_{11}a_{1,e} + S_{12}a_{2,e} \tag{2.9}$$
$$a_{2,e} = \Gamma_L b_{2,e} \tag{2.10}$$

Equations (2.7) to (2.10) can be combined into

$$\begin{aligned}\Gamma_{L,\text{corr}} &= \frac{S_{22} - \frac{b_{1,e}}{a_{1,e}}}{S_{11}S_{22} - S_{11}\frac{b_{1,e}}{a_{1,e}} - S_{12}} \\ &= \frac{S_{22} - \Gamma_{L,k}^{-1}}{S_{11}S_{22} - \Gamma_{L,k}^{-1}S_{11} - S_{12}}\end{aligned} \tag{2.11}$$

With the help of this equation the corrected reflection coefficient $\Gamma_L$ can be calculated. $\Gamma_{L,k}$ is the measured value from the oscilloscope (see equation (2.7)) and $S_{22}$, $S_{11}$, and $S_{12}$ denote the different error terms mentioned above, that have to be measured and calculated, which is discussed in the following paragraphs.

Since there are three error terms, at least three different measurements have to be taken. These measurements are usually conducted using short, open, and load reflection standards from commercially available calibration kits, where load is normally the characteristic impedance of the system [14]. Other impedances could also be used, which is done for example at higher frequencies with different shorts, because it is more difficult to characterize open and loads at these frequencies [15]. Another way commonly used in commercial VNA is the electronic calibration, where different impedances are available via a semiconductor switch through the same connector. With this technique it isn't necessary to change the connections between the different targets, therefore minimizing calibration time and risk for an operator error. Nevertheless, the calibration used in this work was carried out with short, open, and load (see Section 5.1).

Open targets are usually specified by a frequency-dependent capacitance (see equation (2.12)) and a transmission line length. Short targets use the same specifications, but instead of a capacitance a frequency-dependent inductance is needed (see equation (2.13)). For loads, a shunt capacitance, a series inductance, a resistance, and the transmission line length are needed.

$$C = C_0 + C_1 f + C_2 f^2 + C_3 f^3 \tag{2.12}$$

$$L = L_0 + L_1 f + L_2 f^2 + L_3 f^3 \tag{2.13}$$

Because of the inherent transmission line, the impedance on the input of this line has to be calculated. As shown in [1] this can be achieved using equations (2.14) and (2.15). In these equations the impedances calculated from the capacitance/inductance provided by equations (2.12) and (2.13), the material dependent propagation velocity $c$, the frequency $f$ and the transmission line length $l$ are needed. Therefore the actual frequency dependent impedances can be calculated using the values from the data sheet of the used calibration kit.

$$\beta = \frac{2\pi}{\lambda} = \frac{2\pi f}{c} \tag{2.14}$$

$$Z_{in} = Z_0 \frac{Z_L + jZ_0 \tan(\beta l)}{Z_0 + jZ_L \tan(\beta l)} \tag{2.15}$$

These values don't incorporate eventual adapters needed for the measurements. Hence it is important to use the calibration kit with the correct (the same as the DUT) gendered connectors, or the connectors have to be additionally accounted for in the error corrections.

By using three different $\Gamma_{L,k}$ measurements, acquired with each of the three different calibration standards in place of the DUT in figure 2.3, a system of three equations (2.16) to (2.18) can be set up from equation (2.11). The three different $\Gamma_L$ and $\Gamma_{L,k}$ represent the short ($\Gamma_S$), the open ($\Gamma_O$) and the match ($\Gamma_M$) calibration standard.

$$\Gamma_{L_S} = \frac{S_{22} - \Gamma_{L,k_S}^{-1}}{S_{11}S_{22} - \Gamma_{L,k_S}^{-1} S_{11} - S_{12}} \tag{2.16}$$

$$\Gamma_{L_O} = \frac{S_{22} - \Gamma_{L,k_O}^{-1}}{S_{11}S_{22} - \Gamma_{L,k_O}^{-1} S_{11} - S_{12}} \tag{2.17}$$

$$\Gamma_{L_M} = \frac{S_{22} - \Gamma_{L,k_M}^{-1}}{S_{11}S_{22} - \Gamma_{L,k_M}^{-1} S_{11} - S_{12}} \tag{2.18}$$

Solving these equations for the error terms $S_{11}$, $S_{12}$ and $S_{22}$ leads to equations (2.21) to (2.23) with the common term $g$ from equation (2.20) and $\Gamma_1$ from equation (2.19).

Those terms were used to enable a more compact representation. These equations were derived using Wolfram Mathematica and are implemented in a Wolfram Mathematica generated Mathworks Matlab module (see section 4.3).

$$\Gamma_1 = \frac{a_{1,e}}{b_{1,e}} = \Gamma_{L,k} \tag{2.19}$$

$$g = (\Gamma_{1_O} - \Gamma_{1_S})\Gamma_{1_M}\Gamma_{L_O}\Gamma_{L_S} + \Gamma_{L_M}((\Gamma_{1_M} - \Gamma_{1_O})\Gamma_{1_S}\Gamma_{L_O} + (\Gamma_{1_S} - \Gamma_{1_M})\Gamma_{1_O}\Gamma_{L_S}) \tag{2.20}$$

$$S_{11} = \frac{(\Gamma_{1_O} - \Gamma_{1_S})\Gamma_{L_O}\Gamma_{L_S} + \Gamma_{L_M}((\Gamma_{1_M} - \Gamma_{1_O})\Gamma_{L_O} + (\Gamma_{1_S} - \Gamma_{1_M})\Gamma_{L_S})}{g} \tag{2.21}$$

$$S_{12} = \frac{(\Gamma_{1_O} - \Gamma_{1_M})(\Gamma_{1_M} - \Gamma_{1_S})(\Gamma_{1_O} - \Gamma_{1_S})(\Gamma_{L_M} - \Gamma_{L_O})(\Gamma_{L_M} - \Gamma_{L_S})(\Gamma_{L_O} - \Gamma_{L_S})}{g^2} \tag{2.22}$$

$$S_{22} = \frac{(\Gamma_{1_S} - \Gamma_{1_O})\Gamma_{1_M}\Gamma_{L_M} + (\Gamma_{1_M} - \Gamma_{1_S})\Gamma_{1_O}\Gamma_{L_O} + (\Gamma_{1_O} - \Gamma_{1_M})\Gamma_{1_S}\Gamma_{L_S}}{g} \tag{2.23}$$

## 2.2 Analog Part

The analog part, which was designed for the ELP system can be seen in figure 2.5. It consists of everything needed to prepare the signal for analog-to-digital conversion and back. Furthermore it contains a circulator (*circ1*), which is needed for splitting the incident power wave $b_2$ and the reflected power wave $a_2$. Instead of a circulator a directional coupler could be used (see chapter 5 for an example) [3].

The upper analog processing chain in figure 2.5 handles the incident power wave $b_2$. It is responsible for shifting the frequency spectrum of the power wave from RF to an IF of 70 MHz. This superheterodyne design, with baseband mixing implemented in the digital part (see section 2.3), was chosen because it exhibits no amplitude imbalances between I and Q. This setup has the additional feature, that the DC-component caused by the analog mixer *mix1* is outside the band of interest at IF. Therefore, there is no gap around 0 Hz in baseband and the whole bandwidth is usable. 70 MHz was chosen as IF, because it is a widely used IF in radar and microwave applications [16–19] leading to many available components for this frequency band and compatible microwave laboratory equipment [20].

After shifting the band of interest to the IF, the signal is converted from analog to digital (*bp1* and *adc1*). For the sampling rate 100 MS/s was chosen, since it provides enough headroom to support the earlier mentioned bandwidth. Furthermore the chosen vector signal generator supports this sample rate at the digital input port.

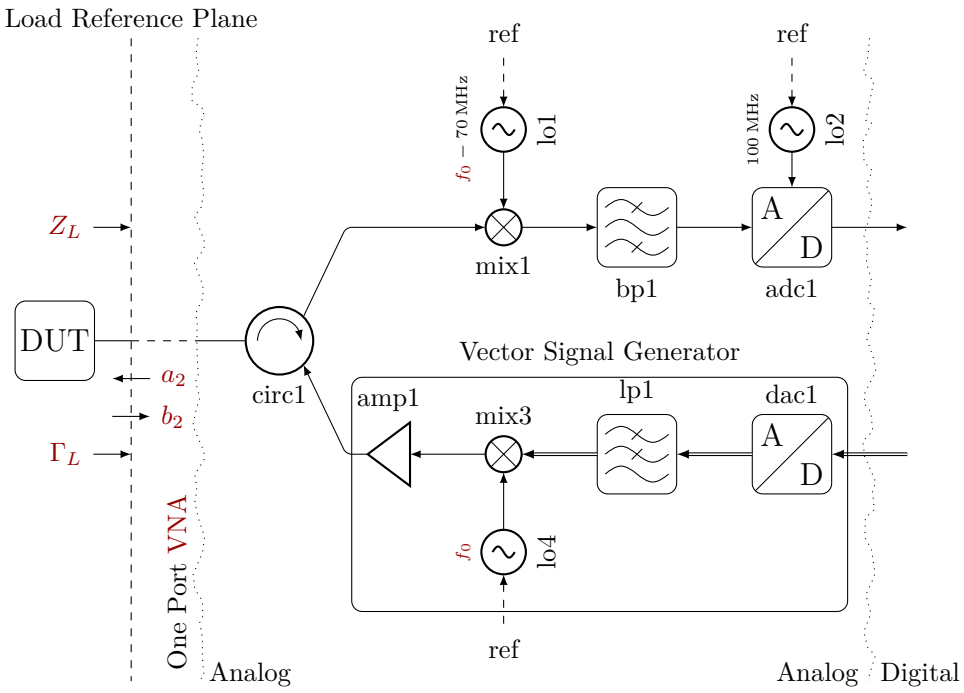



Figure 2.5: Analog part

To fulfill the sampling theorem, this sampling rate is too low for the chosen IF, therefore bandpass sampling was used. This necessitates that a bandpass filter is used as alias filter *bp1*.

Figure 2.6 explains the frequency spectra for the complete down conversion chain (including the digital mixer). The first plot sketches the spectrum of interest in RF around the frequency $f_0$. After the mixer *mix1*, the spectrum of interest lies at 70 MHz and $2f_0 + 70\,\mathrm{MHz}$, which is caused by the analog mixer. Because of this, even if only single tones within the determined bandwith around $f_0$ are generated, at least a low pass has to be used as filter *bp1*. As mentioned earlier, the mixer also generates a DC component, which is depicted as vertical dashed arrow. During sampling spectral aliases occur. These are located at $(k100\,\mathrm{MHz} + 70\,\mathrm{MHz})$ and in mirrored form at $(k100\,\mathrm{MHz} - 70\,\mathrm{MHz})$. This mirroring would cause mixing of frequency components around $(f_0 - 40\,\mathrm{MHz})$ into the band of interest, if no bandpass filter is used. The last step is the digital mixer, which shifts the spectrum by $-70\,\mathrm{MHz}$ which is equivalent to $+30\,\mathrm{MHz}$ caused by the 100 MHz sampling. After that the band of interest lies around 0 Hz (see section 2.3).

The ADC chosen for this work was an LTC2274. This 16 bit ADC has an input bandwidth of 700 MHz and is capable of 105 MS/s [21]. With these specifications and above discussed filtering techniques for limitation of the signal bandwidth it is suited for sampling the IF of 70 MHz with the sample rate of 100 MS/s. This

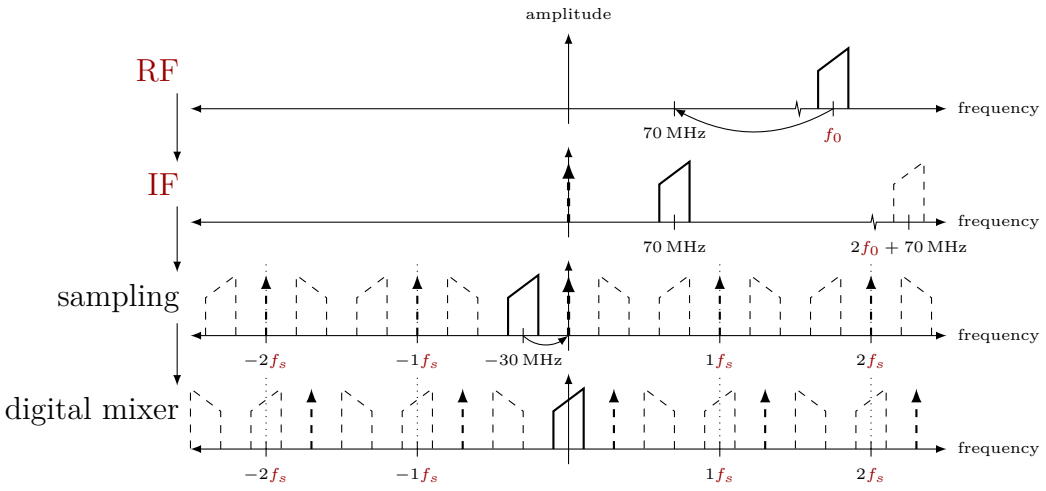



Figure 2.6: Down conversion frequency spectra ($f_s = 100\,\text{MHz}$)

ADC has a JESD204 compliant high speed serial interface [21]. This interface uses 8b/10b line coding over a low voltage differential signalling (LVDS) connection [22]. The line coding is responsible for keeping a balanced number of ones and zeros on the line, therefore keeping a long term DC balance on the line. This allows transmitting the data stream through a high pass channel. Clock recovery is also possible, since this protocol ensures frequent transitions in the bit stream. Any other ADC capable of handling these requirements and a similar digital interface can be used as a drop in replacement for this ELP system.

Digital to analog conversion and up conversion is handled by the lower analog processing chain in figure 2.5. In this work the signal vector generator SMBV100A from Rohde & Schwarz with digital IQ input support (R&S SMBV-K18 [23]) was used for these tasks. Like for the ADC a different signal vector generator can be used as a drop in replacement, if it uses the same digital interface (see section 2.3).

## 2.3 Digital Part

The digital part in figure 2.7 consists of a single 16 bit digital signal processing chain operating at 100 MS/s. It is responsible for the reflection synthesis. This reflection synthesis is achieved with the filter *fir1* and the multiplicator *mul1*. Furthermore it contains the mixer *mix2*. This mixer is needed for IQ-demodulation and down conversion from IF to baseband. The filter *fir1* needs a static digital representation of the samples for the filter calculations. Therefore, the additional sample buffers *buffer1* and *buffer2* are included into the signal processing chain.

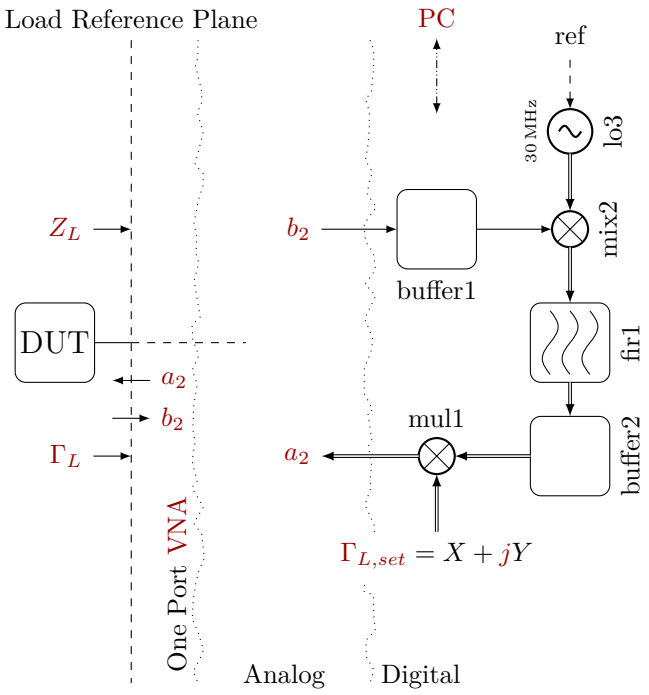



Figure 2.7: Digital part

As can be seen in figure 2.7, a digital representation of the IF signal $b_2$ is provided by the analog part. As mentioned in section 2.2 the LTC2274 with a JESD204 compatible high speed serial interface was chosen as ADC. Instead of special sequences marking word and byte boundaries during communication the receiver interface has to be synchronized before data can be received. During this synchronization mode a JESD204 compatible serial interface sends a specific code sequence instead of data. This code sequence can be used to detect word boundaries and byte ordering. This synchronization mode has to be requested explicitly via dedicated pins of the ADC. In order to minimize needed cabling a communication scheme was developed that initiates synchronization mode by briefly stopping the clock signal.

To minimize the hardware development time, a Linear Technology DC1151A-D evaluation board including the ADC was used in this work. This board consists of a sub-miniature version A (SMA) connector for the analog input, two SMA connectors for data+ and data– of the LVDS connection, an SMA connector for clock input and two pins to enable the synchronization mode. For easier wiring an adapter circuit board was designed to use a serial AT attachment (SATA) cable for clock and data. Furthermore, a synchronization circuit which implements the above mentioned synchronization method was developed.

The synchronization circuit in figure 2.8 consists of three parts. The first part is the power detector *IC1* which is connected via $C_5$ to the TX+ pin of the SATA

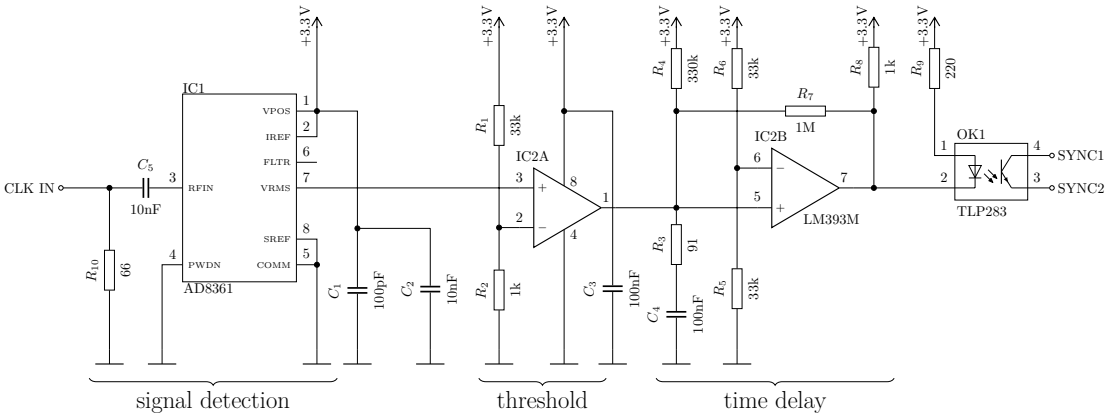



Figure 2.8: Synchronization signal generator for ADC

connector. The TX– pin of the SATA connector is directly connected to an SMA connector. This enables clocking the ADC board using a clock signal delivered by the FPGA. The second part is the comparator *IC2A*. It is used to define a minimum threshold of about 15 $\mathrm{mV_{RMS}}$ for valid clock signals which is determined by $R1$, $R2$, and the power detector *IC1*. The last part is a time delay element. This time delay starts after a valid clock signal is detected and lasts for about 30 ms. During this time the pins *SYNC1* and *SYNC2* are connected via the optocoupler *OK1*, which enables the synchronization mode of the ADC. The implemented adapter circuit board atop the ADC evaluation board can be seen in figure 2.9.

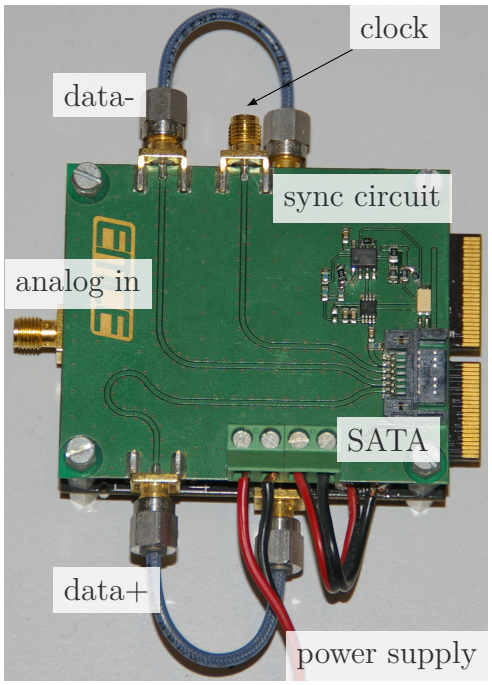



Figure 2.9: ADC adapter board with disconnected clock atop LTC2274 demo board

Samples provided via this connection by the ADC are stored in the sample buffer *buffer1*. As mentioned earlier this LP system needs a cyclic signal for reflection synthesis. Therefore, the buffer needs to store one or multiple periods of the signal. The signal periodicity additionally allows averaging the signal which helps improving the signal-to-noise ratio (SNR) of the ADC [24]. This averaging was implemented by increasing the sample size of this buffer to 19 bit. Furthermore, samples can be accumulated from up to eight consecutive signal cycles. Averaged samples can then be read by dividing the accumulated samples by the number of cycles.

After this buffer, the IF sampled data is IQ-demodulated into baseband with mixer *mix2*. The local oscillator (LO) *lo3* synthesizes the needed $-70\,\mathrm{MHz}$ ($\equiv +30\,\mathrm{MHz}$ with 100 MHz sampling) sine and cosine waveforms using a fixed lookup table. This approach has the advantage of a highly accurate IQ-demodulation without any amplitude or phase imbalances. After baseband conversion, a low pass filter would be needed to suppress the aliases generated from the sampling process, as can be seen in figure 2.6. This task can also be achieved with the configurable filter *fir1* in the next processing step. Therefore, the aliasing filter was left out of the design to keep resource usage at a minimum.

Next in the signal processing chain is the FIR filter *fir1*. As mentioned earlier this filter is needed to compensate the group delay and frequency responses caused by the measurement setup. In combination with a cyclic signal this filter is capable of synthesizing negative phase shifts which is needed to compensate the group delay introduced by cabling needed between the DUT and the measurement setup. Implementing FIR filters consisting of a high number of filter coefficients uses significantly more resources with a tapped delay line than with the FFT. Therefore, the filter *fir1* was implemented with linear convolution:

$$\begin{aligned} y[n] = h[n] * x[n] &= \sum_{m=0}^{n_{\mathrm{fft}}-1} h[n]x[n-m] \\ &= \mathcal{DFT}^{-1}\left\{\mathcal{DFT}(x[n])\, H[n]\right\} \end{aligned} \tag{2.24}$$

with

$$H[n] = \mathcal{DFT}(h[n]) \tag{2.25}$$

This filter is a causal FIR filter with the input $x[n]$, the length $n_{\mathrm{fft}}$, the impulse response $h[n]$, and the output $y[n]$. The input samples $x[n]$ represent the samples in buffer *buffer1* and the output samples $y[n]$ the samples in buffer *buffer2*. To further reduce the needed hardware resources the implemented filter *fir1* uses the transfer function $H[n]$ instead of the impulse response $h[n]$ (see equation (2.25)).

Since implementing the FFT in hardware uses a lot of resources, overlap add was used to allow for larger signal periods than $n_{\mathrm{fft}}$. As shown by [25] this algorithm splits the signal $x[n]$ into non-overlapping subsequences of length $L$ (see figure 2.10).

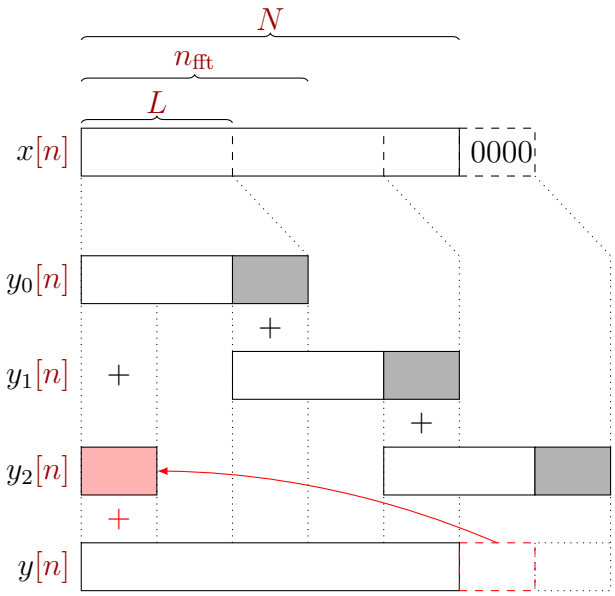



Figure 2.10: Overlap add algorithm (linear convolution, with circular in red)

Therefore, $x[n]$ can be expressed by a sum of shifted finite-length sequences:

$$x[n] = \sum_{i=0}^{\infty} x_i[n - iL] \qquad x_i[n] = \begin{cases} x[k + iL] & k = 0, 1, \ldots, L - 1 \\ 0 & \text{else} \end{cases} \tag{2.26}$$

Applying the linear convolution to this sum leads to

$$y[n] = h[n] * x[n] = \sum_{i=0}^{\infty} x_i[n - iL] * h[n] = \sum_{i=0}^{\infty} y_i[n - iL] \tag{2.27}$$

where $y_i[n]$ is the linear convolution of $x_i[n]$ with $h[n]$. Because every $y[n]$ is of length $N = L + n_{\text{fft}} - 1$, equation (2.24) utilizing n-point FFTs can be used to calculate the results. Every convoluted subsequence $y_i[n]$ overlaps by $n_{\text{fft}} - L$. Since this overlapping points are summed up by equation (2.27), this method is called overlap add. A visualisation can be seen in figure 2.10. The red part in the figure is the circular extension of the linear convolution, which is done by adding the $n_{\text{fft}} - L$ points after the sequence $y[n]$ to the beginning of $y[n]$.

The filter *fir1* needs exclusive access to the output samples during computation. Another requirement is, that samples are continuously output to the DAC. This is needed to ensure that the DUT does not leave eventual operating points during measurements. Therefore, the buffer *buffer2* after the filter was implemented with double buffering. This means that this sample buffer consists of an active and an inactive buffer. The active buffer is used to continuously play back the output signal, while the inactive buffer can be used during the filter calculations. After a new

signal period has been computed the buffers can be swapped without interrupting the output signal.

The last element in the processing chain is the complex multiplier *mul1*. This multiplier is responsible for the actual reflection generation. It allows adjusting the phase and amplitude of the reflection coefficient $\Gamma_{L,set}$ by varying $X$ and $Y$. Samples from *buffer2* are continuously multiplied with this reflection coefficient and output to the analog part as signal $a_2$.

The digital interface needed to forward the processed samples to the analog part was implemented according to the digital IQ interface supported by the used SMBV100A signal generator from Rohde & Schwarz. According to [26], this interface is implemented according to the channel link serializer described in [27]. This serializer uses eight LVDS data lines and one LVDS clock line. Each of the lines is clocked at 700 MHz for a sample rate of 100 MHz. An adapter board was developed for this work that allows connecting pin headers of an FPGA board with the needed mini D ribbon connector [26]. The connector and the wiring schema for the adapter were laid out according to the data sheet in [26]. One missing detail from the documentation, which was found out empirically during this work, is that the S_CLK pin of the connector has to be connected to ground. This signal is marked for future use in the data sheet [26]. Without this ground connection the SMBV100A signal generator does not recognize a connected peripheral at the digital IQ port. A photo of the implemented pin header to digital IQ adapter including the fix can be seen in figure 2.11.

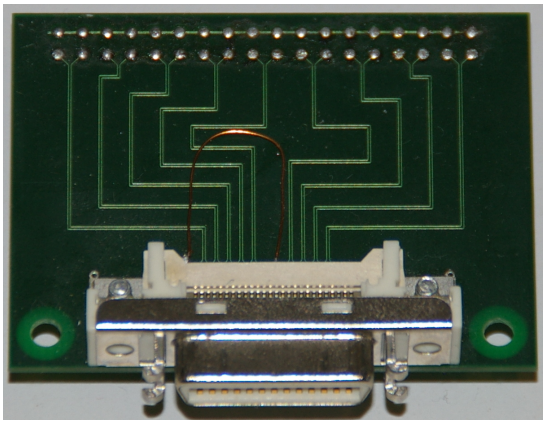
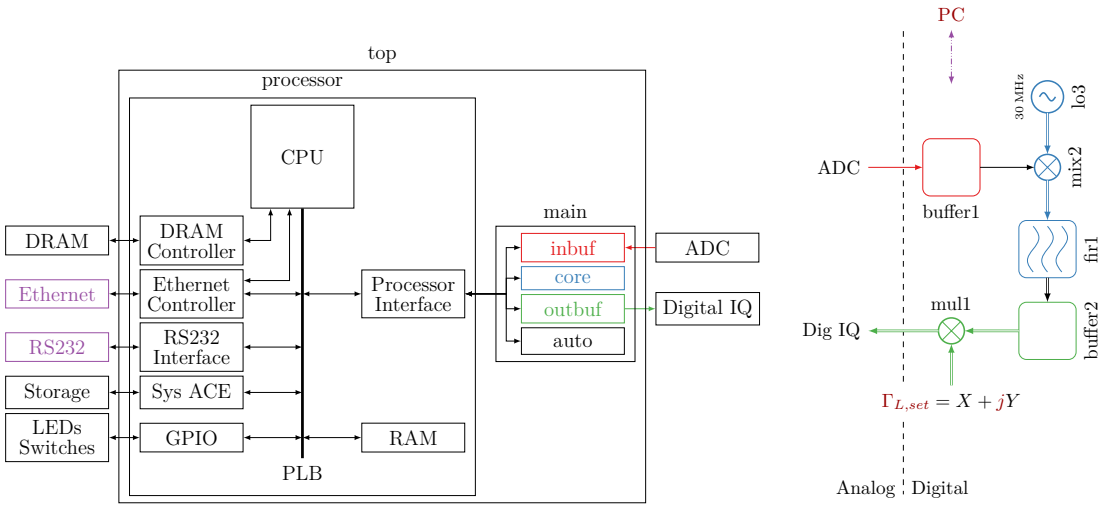


Figure 2.11: Digital IQ adapter circuit board with S_CLK fix

The complete digital signal processing chain described in this section was implemented in an FPGA. A detailed description of the digital hardware implementation can be seen in chapter 3.

# 3 FPGA Implementation

The digital part, as described in section 2.3, was realized in very high speed integrated circuit hardware description language (VHDL). It was specifically tailored to a Xilinx Virtex-5 FXT on an ML507 evaluation board. The FPGA model XC5VFX70T on this evaluation board provides the necessary high speed transceivers, sufficient block random access memories (RAMs), and dedicated digital signal processing hardware [28]. Furthermore the ML507 board contains an SATA connector needed for the ADC and pin headers with high speed differential signal routing to the FPGA which can be used for the digital IQ interface. Additionally, the board contains an Ethernet port enabling high speed data exchange with a PC. Although, according to [29], the SATA headers are only rated up to 1.5 Gbit/s, it was confirmed during tests that the necessary 2 Gbit/s, as required by the ADC [21], are also technically feasible. This FPGA also contains a hardwired PowerPC central processing unit (CPU), which was used for controlling the digital components and as a communication bridge to the PC.




(a) FPGA modules and interconnections

(b) Digital signal processing chain

Figure 3.1: FPGA design overview and digital signal processing chain with color coded modules

The design overview, as can be seen in figure 3.1a, describes the high level modules

of the design and connections between them. Names used in this design overview and the following overviews in this chapter are the actual module names used in the source code. The overall design was split into two parts. The first part is the digital signal processing chain in figure 3.1b (see section 2.3) implemented in the *main* module. The *main* module is described in section 3.1. The second part is the *processor* module which contains a complete embedded processor system. This module was used to implement the necessary protocols for communicating with a PC in software. It is described in section 3.2.

This design uses the unrelated clocks from the processor local bus (PLB) and the ADC. Therefore clock synchronization was needed to minimize the chance of metastable processes. This was achieved using a two-stage synchronizer as the base synchronization circuit. Signals with short pulses were synchronized with an additional pulse shaping to prevent lost pulses. Signal buses were synchronized with an open loop approach. This approach leaves out the acknowledgement of the synchronization which is sufficient for this design because the *processor* module is not fast enough to change the bus value within the synchronization period. Block RAMs in the used FPGA are true dual port memories. Therefore no synchronization is necessary for read access. If one port is used for writes to the memory the other port must not be used to access the same location at the same time [30]. Care was taken to avoid this situation by disallowing memory access from the other port during writes.

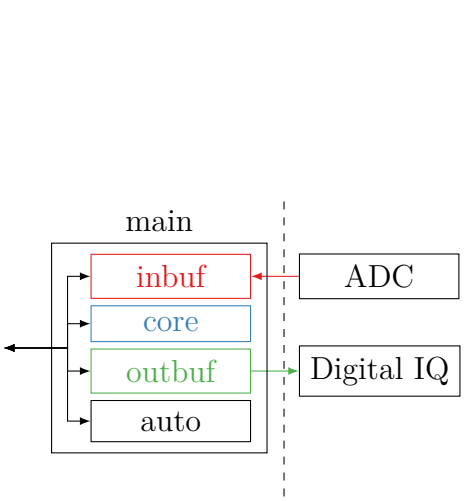
To ease software development a full operating system (OS) was used as software for the CPU. Linux was chosen as the OS for the *processor* module, since it is freely available and can be configured specifically for this target. As will be discussed in section 3.2 the *processor* design includes the modules needed to use Linux. An overview of the complete Linux implementation can be seen in chapter 4.

A reference to all hardware source codes and project files, which are needed to generate the hardware, can be found in appendix A.1.

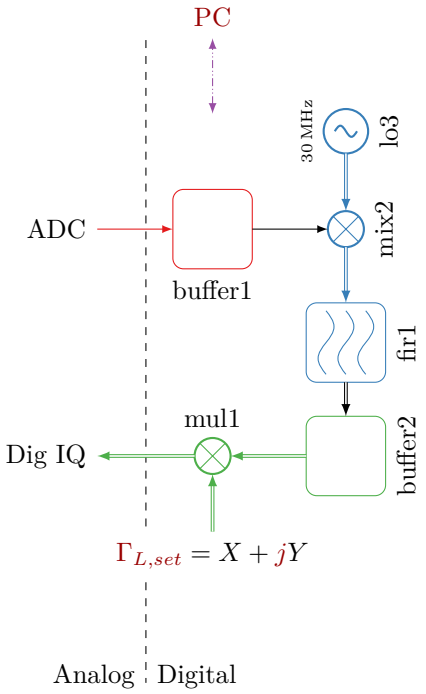
## 3.1 Digital Signal Processing Chain — main

The *main* module contains the digital signal processing chain. As can be seen in figure 3.2 the different parts of the processing chain (figure 3.2b) are mapped to three modules (figure 3.2a). The module *inbuf* contains the digital receive interface for the ADC and the sample buffer *buffer1*. This module is described in section 3.1.1. The *core* module contains the frequency mixer *mix2*, the LO *lo3*, and the digital filter *fir1* with the accompanying buffer $H$ containing the transfer function $H[n]$. A detailed description can be seen in section 3.1.2. The third part is

the *outbuf* module. This module contains the sample buffer *buffer2*, the multiplier *mul1*, and the digital IQ interface and is described in section 3.1.3. The filter *fir1* is not capable of handling a continuous data stream (see section 3.1.2). Therefore the *auto* module was implemented to emulate continuous behaviour by sequentially activating the modules *inbuf*, *core*, and *outbuf*. A more detailed description of the *auto* module can be seen in section 3.1.4.




(a) Overview of the *main* module




(b) Digital signal processing chain

Figure 3.2: *main* module overview and digital signal processing chain with color coded modules

Necessary bit width truncations throughout the digital signal processing chain use convergent rounding. This rounding mode was used to prevent DC offsets. It rounds to even integers in case of a tie. For example rounding 4.5 with this method results in 4. The same would be true for 3.5. Thus rounding does not introduce an offset towards infinity or zero, as would be the case for round half up. Additionally, overflows are signalled and saturation is used where applicable.

The sample buffer size was chosen according to the available block RAM in the FPGA. A maximum number of 148 block RAMs capable of storing up to 36 kbit is available in the Virtex-5 on the ML507 board [28]. Those block RAMs can also be used as two independent 18 kbit block RAMs. As mentioned in section 2.3, the digital signal processing chain needs to support a sample width of 16 bit. Input averaging was chosen to support a maximum of 8 averages, resulting in an overall needed bit with of 19 bit for the input buffer. Since the output buffer uses double

buffering and needs to store IQ signals, two buffers with a bit width of 32 bit are needed. In order to achieve a high memory utilization, constrained by the possible block RAM bit widths, a single data bit of a buffer was realised with a 36 kbit and an 18 kbit block RAM. This leads to a total number of 124.5 block RAMs where 0.5 denotes an 18 kbit block RAM. This leaves enough for the embedded processor and the FFT implementation. Using this memory layout only a 2-to-1 multiplexer for the output is needed. Since the highest address line can be used as selection between both block RAMs, no address decoder is needed. This ensures fast operation of the overall memory. In one bit mode only 32 kbit (16 kbit) are available. This leads to a theoretical buffer depth of 49,152 samples. However as will be explained in section 3.1.2 (see also section 2.3), if circular convolution with a block size of $L$ and an FFT size of $n_{\text{fft}}$ is used for the FIR filter, the full buffer depth is not usable. This is caused by the implementation of the overlap add algorithm which needs additional buffer space of $L - n_{\text{fft}}$ samples after the signal.

The memory buses of the *inbuf* module and the *outbuf* module are connected to the *core* module. The same bus connections are also exposed via a memory interface of the *core* module. To prevent access violations the exposed memory interface can't access the internal modules while one of them is active. Also further module activations are not possible during this period. With these restrictions in place no memory corruptions can occur. The *core* module also exposes every configuration and status signal of the internal modules. These signals are used by the processor interface described in section 3.2.1 for controlling the whole signal processing chain.

### 3.1.1 Data Acquisition — inbuf

The *inbuf* module is responsible for serial to parallel conversion of the ADC data, descrambling, triggering, averaging, and storing the acquired samples. As can be seen in figure 3.3, this module consists of the main modules *receiver*, which handles receiving the data stream from ADC, and *average_mem*, which handles storing the samples in a buffer and averaging. The additional modules *trigger*, *prepare*, and *wallclk* are responsible for controlling the data acquisition and keeping the time, which is needed for the IQ demodulation that will be explained in section 3.1.2.

The serial to parallel conversion of the received data was implemented using the built in GTX transceiver. Since the ML507 board features two SATA ports, two identical receivers were implemented. Since the GTX blocks need a lot of configuration settings, they were instantiated using the recommended method, which is the transceiver wizard documented in [31]. The settings used in this work were chosen according to the data sheet of the ADC [21]. The receiver was configured to handle the specifications as described in the section 2.3. These are 8b/10b line

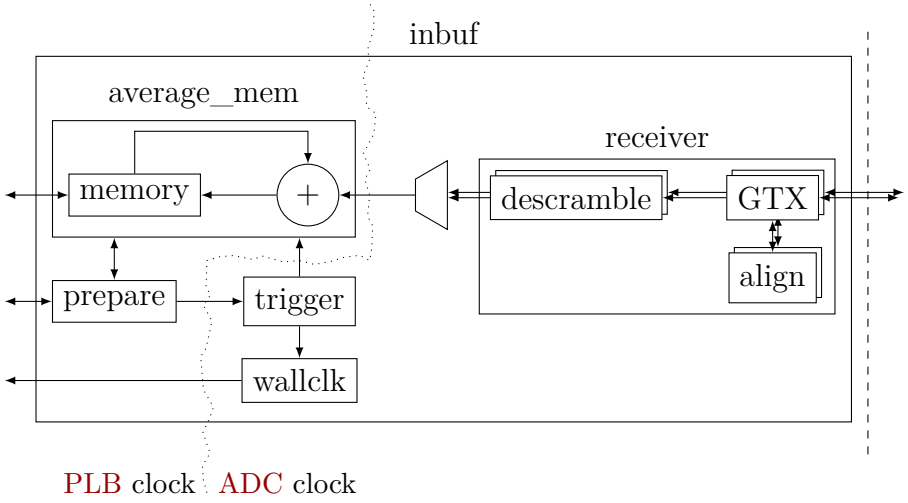



Figure 3.3: Block diagram of the *inbuf* module

coding, a data width of 16 bit and a target line rate of 2 Gbit/s. Synchronization was configured for an enabled idle synchronization mode (ISMODE) of the ADC. In this mode the transmitter of the ADC sends idle ordered sets instead of commas during synchronization [21]. An idle ordered set is a special sequence used for synchronization consisting of a K28.5 comma followed by either D5.6 or D16.2 code word. A comma is a special code sequence of the 8b/10b line coding, which does not represent valid data [32]. To synchronize the GTX receiver according to these specifications, the comma alignment was set to align to even byte boundaries and to detect the K28.5 comma. After serial to parallel conversion, the endianness is converted to the internal representation. The recovered clock from the received data stream is used as the internal sampling clock. This setup allows the use of an externally provided sampling clock, by connecting it to the ADC instead of the clock provided by the ADC adapter circuit board mentioned in section 2.3. The receiver and the transmitter share the internal clock generation of the GTX. Therefore the transmitter had to be configured for the same line rate. To transmit a clock signal with the transmitter, the 8b/10b line coding was disabled for the transmitter and a fixed clock pattern was applied to the parallel input.

It is not possible to provide an externally generated clock directly to the SATA connected GTX transceivers because of design limitations of the ML507 evaluation board. Since using a clock, which is routed through the global clock network of the FPGA introduces jitter [32], an externally provided sampling clock connected to the ADC is the preferred mode of usage.

The module *align* controls the clock signal, which is transmitted using the GTX. This module contains a state machine that blanks the clock signal for 41 ms after a loss of sync of the receiver. In combination with the synchronization circuit

described in section 2.3 this clock blanking enables synchronization mode for the ADC. A loss of sync is detected if either a comma value or a value that is not part of the 8b/10b line coding is detected. After the blanking period the clock signal is reactivated and the GTX is put into alignment mode, which searches for the above mentioned comma K28.5. If no comma is detected during a further 41 ms period the synchronization re-starts from the beginning.

Succeeding the GTX transceiver, the *descrambler* module descrambles the data according to the data sheet of the ADC [21]. Data scrambling is used to lessen the noise caused by the digital transmission in the analog part. The scrambler implemented in the ADC is based on the generator polynomial given in equation (3.1) [22]. This scrambler can be bypassed if a different ADC is used.

$$g(x) = 1 + x^{14} + x^{15} \tag{3.1}$$

Averaging the data is handled by the *average_mem* module. This module consists of the above mentioned (19 × 49,152) bit memory. Averaging is achieved by reading the appropriate sample from the buffer and adding it to the current value. For the first run the read sample is replaced with the value zero. Using this method the sampled values are accumulated over a configurable number of zero, two, four, or eight rounds. Averaging is achieved during memory access from outside the module by shifting the accessed values by zero, one, two, or three bit. This is equal to dividing the samples by the number of rounds used for averaging. This method needs to read the sample from the last round from the buffer while writing the current sample to the buffer. Therefore, both ports of the memory are needed by the implementation which implies that the synchronization technique using different ports for the different clock domains as described above is not feasible. Hence, the memory interface allowing access to the sample buffer from outside the *inbuf* module needs to share the ports with the averaging mechanism. Since those two parts don't share the same clock, clock multiplexing with dedicated FPGA hardware was implemented.

If the data acquisition is not in use, the *average_mem* is powered by the PLB clock. Before data acquisition, the *prepare* module transitions the clock signal to the ADC clock. Undefined behaviour can occur if the timing conditions of the address signals of the block RAM are violated while the enable signal is high. This can lead to memory corruption even if write enable is not asserted [30]. Therefore, the enable signal of the block RAM is switched to low by the *prepare* module before the clock transition. Since this is not possible for an unstable ADC clock, the validity of the sample buffers is not guaranteed after connecting or disconnecting an ADC. After the data acquisition finishes, the *prepare* module transitions the clock signal back to the PLB clock. Write operations to the memory are ignored during data acquisition. Read operations during data acquisition don't interfere with the process but the read out data is invalid.

The *trigger* module generates the start signal for the *average_mem* and the *wallclk* module. During the first run and after a reset, the trigger can either be triggered externally or internally. The external trigger is sampled from the pin AN33 of the FPGA, which is connected to HDR1_64 on the evaluation board. After the first successful trigger, consecutive triggers are only generated at multiples of the configured signal period $N$.

Subsequent to a trigger event, the *wallclk* module takes a snapshot of the jiffy counter which marks the time the first sample of the current acquisition was acquired. A jiffy is 10 ns in this FPGA design which is the time needed for one sample for a sample period of 100 MHz. The 30 MHz signal for the IQ demodulation is not generated in real time. Therefore the snapshot of the jiffy counter is necessary for the *core* module to generate the 30 MHz signal with the correct phase (see sections 2.3 and 3.1.2).

The *inbuf* module is fully runtime configurable. Changing the active receiver or connecting an ADC resets the whole module automatically. This prevents undefined behaviour which could result from the clock change. Since the trigger source is only decisive for the first trigger event changing the source is only possible if the *trigger* module has not fired yet. Therefore, the trigger module should be reset after changing the source. Trigger events, averaging finished, and ADC connection status are reported with separate signals. The memory interface of the *inbuf* module has a 16 bit wide data bus, a 16 bit wide address bus, and a read access latency of 2 cycles. Accessing samples at addresses $\geq$49,152 result in undefined behaviour.

### 3.1.2 Overlap Add — core

An overview of the *core* module can be seen in figure 3.4. It consists of the buffer $H$, which is needed to store the transfer function, and the module *overlap_add*, which contains the signal processing. The *overlap_add* module itself contains the module *wave*, which is responsible for IQ demodulation, the *fft* module, which can calculate the FFT and inverse-FFT, a complex adder, a complex multiplier, and additional block RAM, needed as temporary buffer space (*scratch*).

The IQ demodulation module *wave* consists of a hard coded look up table, that generates a 30 MHz sine and cosine waveform for a sampling rate of 100 MHz. Furthermore it contains multipliers, which multiply the incoming samples with the sine and cosine waveforms, for converting the signal into I and Q samples. Instead of performing this demodulation in real time, it is performed during filter processing. To ensure the correct phase of the 30 MHz waveform it is evaluated at the point in time calculated from the jiffy counter and the position of the currently processed sample. The jiffy counter is contained in the *wallclk* module as explained

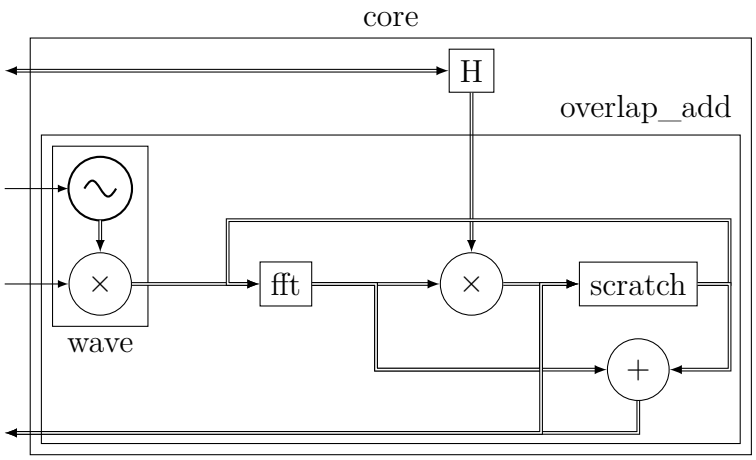



Figure 3.4: Block diagram of the *core* module

in section 3.1.1. This counter marks the time the first sample of the current signal was taken.

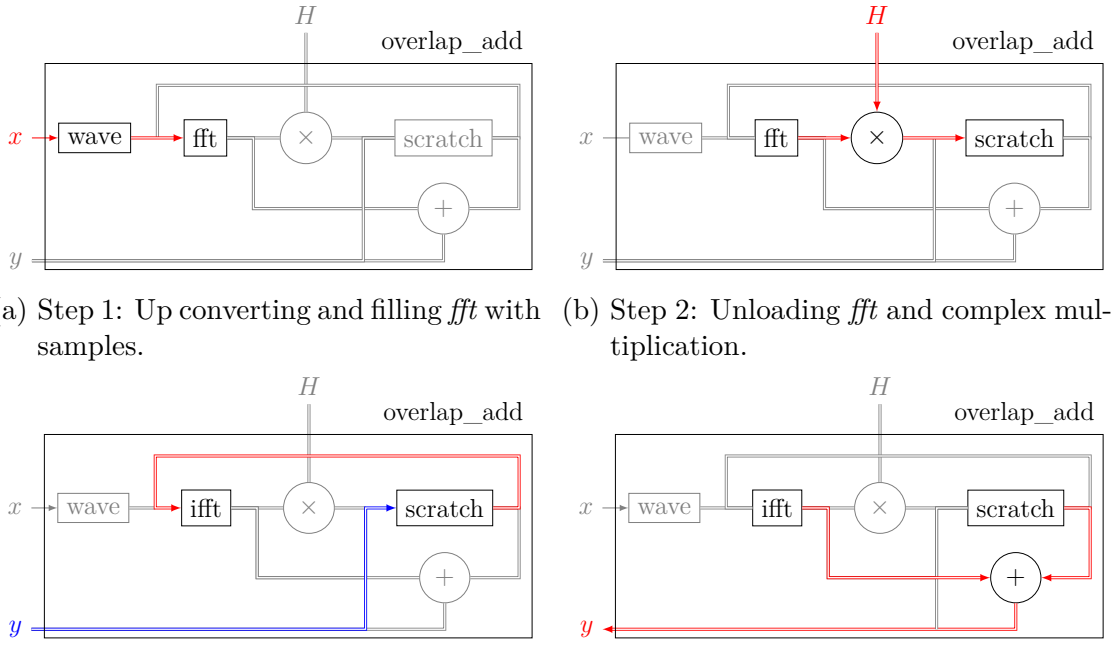
A Xilinx LogiCore IP was used as the *fft* module [33]. In order to minimize block RAM usage and increase the performance the pipelined version was used. A further reduction in block RAM usage was achieved by limiting the IP core to the minimum setting of three stages in block RAM. The remaining stages needed for this configuration are implemented in distributed RAM which results in a higher lookup table (LUT) usage of the *fft* [30, 33]. Distributed RAM was preferred over block RAM because LUT usage was not a limiting factor in this design. The *fft* implementation was set to allow a modification of the FFT length at runtime, with a maximum length of 4,096. A further runtime configuration setting is, that this core can be switched between FFT and inverse FFT. This allowed using only one *fft* module. Therefore reducing the resource usage further. Every computational part of the FFT, called a butterfly, consists of an addition and a multiplication that preserves the magnitude of the complex valued input of the butterfly. The addition increases the needed bit width after every butterfly by one bit to represent every possible value. The multiplication can result in an overall bit width growth of one bit for the whole FFT if the magnitude represented by the complex valued input is greater than one. Combining both factors leads to equation (3.2) given in [33].

$$bits_{out} = bits_{in} + \underbrace{\log_2(n_{\text{fft}})}_{\text{addition per butterfly}} + \underbrace{1}_{\text{complex rotation}} \tag{3.2}$$

According to equation (3.2), keeping every bit for an FFT length $n_{\text{fft}}$ of 4,096 and input width $bits_{in}$ of 16 bit would result in a maximum output width $bits_{out}$ of 29 bit. Using such high bit widths is not possible because of resource constraints. Therefore, scaling was used and is implemented with a divider after every group, where one group consists of two butterflies. Every divider can divide the samples by one, two, four, or eight. Since the possibility of overflows depends on the input

data the right scaling schedule can't be determined beforehand. According to [33] the best scaling schedule for the FFT and inverse FFT is found by starting with a divider schedule of one and incrementing the divider schedule until the computation stops overflowing. Input data for the FFT core needs to be in natural order and output data is in bit reversed order [33].

The complex multiplier was implemented with a runtime changeable scaling schedule. This schedule can be set to shift the result by 14 to 17 bits. Therefore, the transfer function should always be scaled to the maximum possible values. The best overall scaling schedule can be found by first configuring the scaling schedule of the FFT, then the complex multiplier and after that the inverse FFT.




(a) Step 1: Up converting and filling *fft* with samples.

(b) Step 2: Unloading *fft* and complex multiplication.

(c) Step 3: Loading the *fft* with the computed samples and loading scratch with the previous block.

(d) Step 4: Unloading *fft* and adding the overlapping part from *scratch*.

Figure 3.5: Overlap add algorithm hardware implementation

The overlap add algorithm, as explained in section 2.3, was implemented with two separate state machines. This allows speeding up the process since some parts can be calculated in parallel. The first state machine controls the FFT and the complex multiplication part of the algorithm. It is called *fftncmul*. Inverse FFT and complex addition is handled by *ifftnadd*. The overall process is described in figure 3.5. Every block of size $L$ needs to be processed within four steps. In the first step (figure 3.5a), the *fft* module is loaded with $L$ IQ demodulated samples from the source memory $x$. According to the overlap add algorithm, after $L$ samples the data input of the *fft* is switched to zero. After the *fft* module has finished

processing, the second step (figure 3.5b) starts. In this step the data is multiplied with $H$ and stored in the scratch buffer. During the next step (figure 3.5c), the *fft* module is switched to inverse operation and the samples from the *scratch* buffer are loaded into the *fft* module. At the same time the *scratch* buffer is filled with the previous computed block from the target memory $y$. After the *fft* module has finished the computation, the last step (figure 3.5d) is executed. During this step the overlapping part from the previous block is added and the samples are unloaded into the target memory. Since the FFT module is pipelined, step one of the next block is started after step three of the current block has finished. This means, that the FFT module computes the samples of two different blocks at the same time. With this technique, the duration of the overlap add algorithm is reduced by one stage per $L$ sized block. If the circular mode is enabled, then the block after the signal is added to the beginning (see section 2.3). Since this process needs $L - n_{\text{fft}}$ samples after the end of the signal, the sample buffer can't be used to the full extent.

The overall module is freely configurable and accepts FFT sizes 8, 16, 32, 64, 128, 256, 512, 1,024, 2,048, and 4,096. The signal size $N$ has to be between 8 and 65,535 and the block size $L$ between 1 and 4,096. There are no plausibility checks in the hardware, which means that wrong settings will lead to undefined behaviour. The *core* module can be stopped by issuing a reset signal. Numerical overflows are separately reported for FFT, inverse FFT, and complex multiplication.

### 3.1.3 Vector Signal Generator Interface — outbuf

The *outbuf* module consists of two independent sample buffers *mem_0* and *mem_1*, a complex multiplier and the digital IQ interface *transmitter*. The two sample buffers are both 32 bit wide, to store the 16 bit wide I and Q signals. Both can store up to 49,152 samples. A general overview can be seen in figure 3.6.

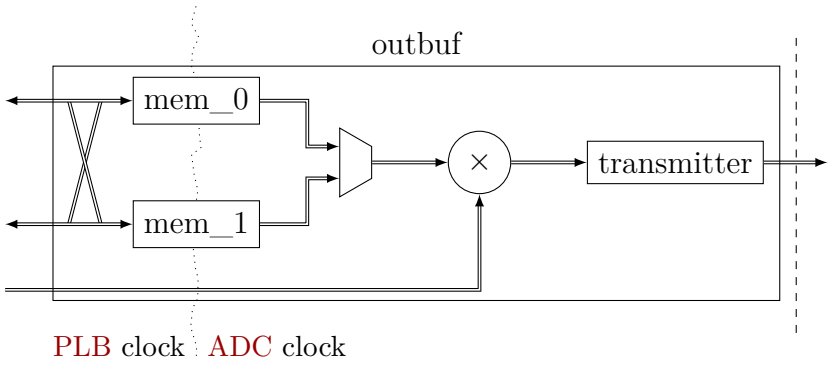



Figure 3.6: Block diagram of the *outbuf* module

Sample buffers *mem_0* and *mem_1* can only be accessed by the logical names *active* and *inactive* from outside the *outbuf* module. The *active* sample buffer contains the sample data, which is streamed via the *transmitter* module to the digital IQ interface. This memory can only be used in read mode via the internal interface. The *inactive* sample buffer can be read and written to. Maintaining this logical to physical assignment is handled by the network, indicated to the left of the sample buffers, and the multiplexer in figure 3.6. Both sample buffers consist of true dual port block RAMs. In combination with the unsupported writes to the *active* sample buffer, this ensures that no timing conditions are violated, due to the different clocks.

The *outbuf* module uses a configurable signal period $N$. This period can only be changed by resetting the module or by asserting the *resync* signal. Sample data of length $N$ is continuously read from the *active* buffer. Swapping the *active* and the *inactive* buffer can be initiated by asserting the *toggle_buf* signal for one cycle. To prevent signal distortion, the actual swapping occurs only before reading sample zero from the buffer. Accesses to the sample buffers during this operation result in undefined behaviour and can corrupt the buffer content. This is prevented by the access protection mechanism implemented in the *core* module.

The complex multiplier, between the multiplexer and the *transmitter* module in figure 3.6, is responsible for the reflection generation. It multiplies the signal output to the transmitter with $\Gamma_{L,set}$ (see section 2.3). As mentioned earlier, this multiplier features a bit shifter at the output. This bit shifter shifts the output by a configurable range of 2 bit–17 bit. With that operation it is possible to reach full scale output also for small input signals. This is necessary if, for example, a high scaling factor is needed for the *core* module (see section 3.1.2). After this scaling and convergent rounding, the multiplier is also capable of saturating the output in case of overflows. This would prevent a wrap around and therefore limit signal distortion.

According to [26] the Rohde & Schwarz digital IQ interface uses the encoding scheme and digital interface of a DS90CR485 channel link serializer. This serializer encodes data from a 24 bit input on both clock edges resulting in a total of 48 bit per clock cycle [27]. As specified by [26] 20 bit I data, 20 bit Q data, enable, valid, marker bits, and trigger bits need to be encoded in these 48 bit. Enable and valid are connected to high in this implementation. The marker and trigger bits are not supported by the Rohde & Schwarz SMBV100A vector signal generator in digital IQ input mode. Therefore, these were permanently disabled in this work. The FPGA internal 16 bit IQ implementation was mapped onto the 20 bit link by left shifting the data by 4 bit.

The *transmitter* module is an implementation of the DS90CR485 channel link serializer specification in VHDL. Since [26] does not mention which operating mode

of the channel link serializer is needed, the full DS90CR485 specification according to [27] was implemented during this work. According to this specification, the 48 bit, as mentioned in the last paragraph, need to be partitioned onto eight LVDS links clocked at 700 MHz. It is not possible to achieve such high frequencies in the FPGA fabric. Therefore the serialization was implemented utilizing output serializer/deserializers (OSERDES) in 4 bit double data rate (DDR) mode. The DDR mode reduces the necessary clock rate to 350 MHz for the OSERDES. Furthermore, the 4 bit mode decreases the needed clock for the parallel data to 175 MHz. By connecting a test implementation of the transmitter it was verified that the Rohde & Schwarz digital IQ interface uses the DS90CR485 channel link serializer with switched off DC Balance mode.

The *outbuf* module is runtime configurable. After changing the signal length $N$ the *outbuf* module has to be reset or a *resync* operation needs to be initiated. Overflows from the multiplicator are reported with a status signal which has to be manually reset. This prevents interrupt storms if the signal is used as an interrupt. The memory interface of the *outbuf* module has a 16 bit wide data bus, a 16 bit wide address bus, and a read access latency of 2 cycles. Accessing samples at addresses $\geq$49,152 results in undefined behaviour.

### 3.1.4 Auto Module

For the complete signal computation, the previously described blocks *inbuf* (see section 3.1.1), *core* (see section 3.1.2) and *outbuf* (see section 3.1.3) need to be triggered in this order. To minimize the delay and CPU usage, the *auto* module contains a state machine that performs this task. It supports single shot and continuous operation.

During a single run, the *auto* module first triggers the *inbuf* module. After data acquisition is finished, the filter operation is carried out by starting the *core* module. If an overflow occurs during *core* module operation, the automatic mode is stopped. After a successful filter operation the *outbuf* buffer is toggled. If the *auto* module is switched to continuous mode, then the whole process repeats.

Because of the way the *auto* module was implemented for this thesis, it is necessary to clear eventual overflows of the *core* module before enabling the automatic mode.

## 3.2 Processor

The embedded processor design was realized using the Xilinx Platform Studio (XPS). The design consists of every additional module needed to run Linux and communicate via Ethernet. As can be seen in figure 3.7, this includes a dynamic random access memory (DRAM) controller, an Ethernet controller, a RS232 interface, a System ACE controller, a general-purpose input/output (GPIO) module, and an additional memory controller with a connected block RAM. Additionally, an interrupt controller is needed which is not included in figure 3.7.

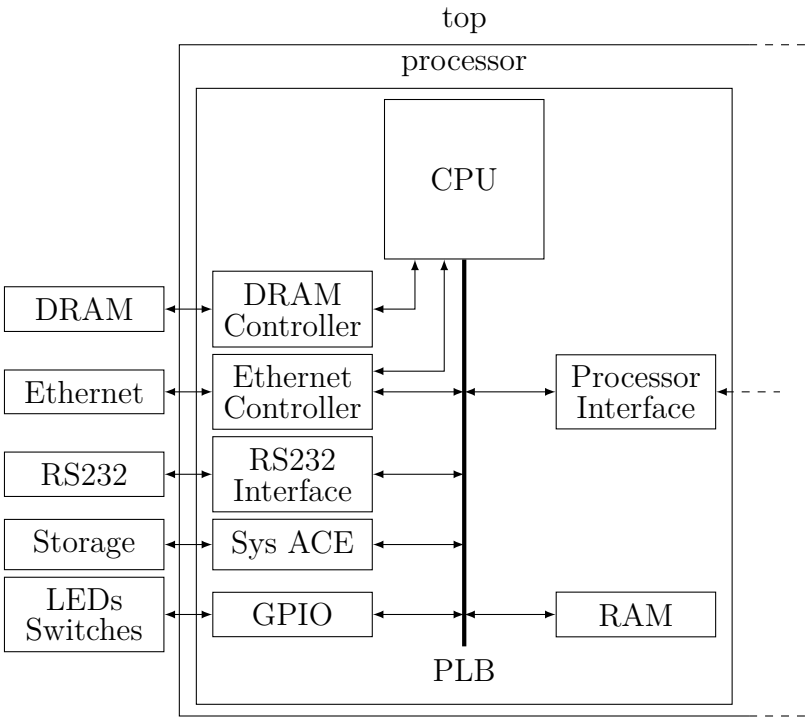



Figure 3.7: Block diagram of the *processor* module

The internal block RAM is needed because the PowerPC 440 embedded in the used FPGA has a reset address which resides in the highest addressable page [34]. Since the DRAM is mapped at the lowest address and is only 256 MiB large [29], the processor would try to access a non existing address after a reset. The CPU starts executing code before the complete setup is finished initializing. To prevent such undefined behaviour after programming the FPGA, a memory has to be placed at this address. This memory needs to be filled with a boot loop which is a simple program consisting of an endless loop. Since Linux also uses this address range, this small memory can't be implemented as a read only memory (ROM).

To enable communication with a computer the RS232 module and the Ethernet module are needed. The RS232 module is not necessary for production use, but

needed for debugging purposes. The System ACE module is required to provide access to the compact flash (CF) card on the ML507 board. With this module, a network boot setup is unnecessary, therefore enabling easier deployment of the whole system. The GPIO module was included into this setup, to provide the OS with access to the light emitting diodes (LEDs) and switches for indicating purposes.

### 3.2.1 Processor Interface

The processor interface, allowing the processor to control the digital signal processing chain, consists of three modules. An XPS module which interfaces with the PLB, called *proc2fpga*, a module implementing the registers, called *proc_register*, and a module implementing a small memory controller, called *proc_memory*. Separating the PLB interface into an XPS module allowed keeping the whole *processor* module fixed during development. See section 4.1 for a description of the software needed to control the processor interface.

The PLB interface was implemented with the peripheral wizard in XPS. It uses the burst variant of the PLB slave [35]. This module exposes four user memories with 16 bit addressing and 32 bit data bus, six 32 bit registers, and 16 interrupt lines to the FPGA fabric. The interrupt lines are triggered on the positive edge. In order to enable wide memory access the burst variant is needed to use memory mapping in Linux. In this mode, Linux handles the size of the memory access, which results in reading or writing in whole cache lines. The PLB is 128 bit wide, therefore, the non burst variant can't handle these operations, since the module interface is only 32 bit wide. The burst variant handles this situation by queuing four consecutive 32 bit operations. Without this, a bus error would be generated.

An overview of the register contents can be seen in appendix A.2. These registers and the interrupt generation are implemented in the *proc_register* module. Every register has an access time of a single cycle.

The memory controller implemented in the *proc_memory* module generates the necessary acknowledge signals for the processor. Furthermore it generates the write and enable signals from the chip select and write request signals of the processor interface. Special care was taken to allow multi cycle requests as described above.

A description of a Linux kernel module implementation using this interface and the needed infrastructure can be seen in chapter 4.

# 4 Software Implementation

As described in chapter 3, the FPGA contains a general purpose processor. To enable controlling the digital signal processing chain described in sections 2.3 and 3.1 with a PC, software was developed for this processor. This software is needed, for instance, to control the filter transfer function which is needed for correcting the frequency response (see section 3.1.2) or setting the targeted hardware reflection coefficient $\Gamma_{L,set}$ (see section 3.1.3). To allow high speed transfers of the filter transfer function or sample data, Ethernet was chosen as the communication interface. Furthermore this communication interface is cheap, widely supported by PC-hardware, and can be used by PC-software such as Matlab with the Instrument Control Toolbox.

To simplify the needed development process an OS was used as the base which provides the necessary drivers supporting the needed peripherals and network communication. Linux was chosen since it can be targeted specifically for this platform due to its open nature. The cross compilation tool chain needed for compiling the software components and the base system were built using buildroot [36]. A reference to the configuration files needed for configuring buildroot can be found in appendix A.1 and a guide for building the base software in appendix B.2.

Controlling the hardware from within the Linux operating system can be achieved with either memory mapping */dev/mem* or by writing a kernel module [37]. Memory mapping has the disadvantage that interrupts cannot be used and concurrent access leads to undefined behaviour. Another disadvantage is that directly using */dev/mem* is only possible by the privileged user [37]. In opposition to that a kernel module can provide an easy application programming interface (API) that can be used by any programming language, shell script, or even shell commands like `echo`. Because of these advantages a kernel module was implemented. A description of this kernel module can be found in section 4.1.

A background process, called a daemon, providing access via network to the API exposed by the kernel module was implemented. Additionally a web-based user interface (UI) was developed to allow manual control and provide a visual feedback of events like ADC connected (see section 3.1.1) or output multiplier overflow (see section 3.1.3). This web-based UI communicates with the daemon with JavaScript Object Notation (JSON) messages via the websocket protocol. In addition to the

websocket based protocol a text based protocol enabling simple communication with software like Matlab was implemented. An overview of the complete software architecture including the communication paths can be seen in figure 4.1. A detailed description of the implemented daemon can be found in section 4.2.

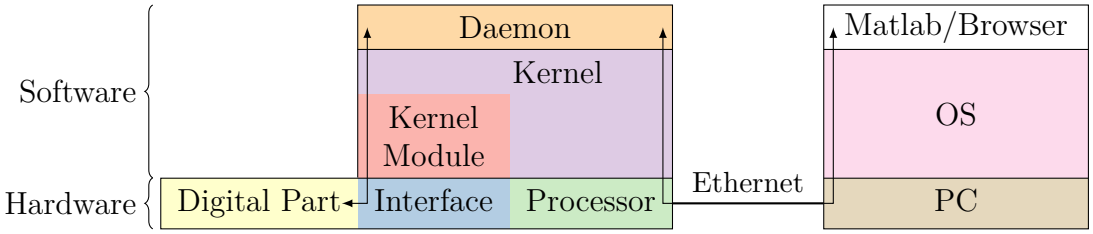



Figure 4.1: Soft- and hardware architecture including communication paths. FPGA on the left and PC on the right.

The calibration and error correction routines needed for the one port VNA functionality were implemented as Matlab scripts. These scripts automate the measurements and are able to perform the calculations needed for correcting the systematic errors (see section 2.1). Furthermore, a Matlab function for performing an iterative target algorithm in order to achieve a specific $\Gamma_{L,target}$ was developed. This algorithm is necessary, because reflections caused by the measurement setup and non-linear DUTs influence the resulting effective $\Gamma_L$ which is controlled by the $\Gamma_{L,set}$ setting provided by the digital signal processing chain. Using this algorithm the calibration routine for the digital filter was implemented. This filter enables the frequency response compensated broadband reflection synthesis (see section 2.3). A description of the implemented Matlab scripts can be found in section 4.3.

## 4.1 Kernel Module

The kernel module implemented specifically for the processor interface in the FPGA takes care of initializing the hardware, provides direct access to the sample buffers, provides access to hardware knobs via registers, and enables utilizing the interrupts (see section 3.2.1 and appendix A.2). Since these tasks are handled independently, the following description is split into the parts memory access (section 4.1.1), register access (section 4.1.2), and interrupts (section 4.1.3).

To keep the driver as generic as possible a platform device driver [38] was developed. Platform device drivers are matched against a hardware description called a device tree [38]. The kernel takes care of invoking the `probe` and `remove` functions upon device tree initialization. Besides the common needed module configuration macros in listing 4.1 line 28 only a **`struct`** describing the driver (line 16), a list of device names (line 9), and the accompanying platform driver macros in lines 14

```
static int emce_of_probe(struct platform_device *ofdev) {
        ...
}

static int emce_of_remove(struct platform_device *of_dev) {
        ...
}

static const struct of_device_id emce_of_match[] = {
        { .compatible = "xlnx,proc2fpga-3.00.b", },
        { /* end of list */ },
};

MODULE_DEVICE_TABLE(of, emce_of_match);

static struct platform_driver emce_of_driver = {
        .driver = {
                .name = DRIVER_NAME,
                .owner = THIS_MODULE,
                .of_match_table = emce_of_match,
        },
        .probe = emce_of_probe,
        .remove = emce_of_remove,
};

module_platform_driver(emce_of_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Gernot␣Vormayr␣<notti@fet.at>");
MODULE_DESCRIPTION("driver␣for␣custom␣fpga␣interface");
```

Listing 4.1: Basic elements of a platform driver module

and 26 are needed. These macros create the module instantiation code. The only code that needs to be supplied are the bodies of the `probe` and `remove` functions (lines 1 and 5). The `probe` function needs to perform device initialization, has to allocate memory for the device data, reserve ownership of device associated memories, allocate devices and files, setup the interrupt function, and initialize the interrupt registers. For freeing the resources, everything has to be released in the `remove` function. Omitting this prevents unloading the driver cleanly which causes locked resources that can only be used again after a reboot. All the code examples in this chapter are taken from the developed kernel module source. A reference to the complete source code can be found in appendix A.1.

### 4.1.1 Memory Access

The sample buffers *inbuf* and *outbuf* as well as the transfer function $H$ of the filter (see section 3.1) are exposed to user space via character devices [37]. A character device is a special file that can be opened, closed, read, and written to. Instead of operating on a real file in the file system, these operations are handled by the attached kernel module. Read and write operations are possible in byte sized

```
#define USER_MEM 4

struct user_mem
{
        unsigned long start;
        unsigned long size;
        void __iomem *base_address;
};

struct emce_device {
        struct cdev cdev;
        dev_t dev;
        ...
        struct user_mem mem[USER_MEM];
        ...
};

...

static const struct file_operations emce_fops = {
        .owner = THIS_MODULE,
        .read = mem_read,
        .write = mem_write,
        .open = mem_open,
        .mmap = mem_mmap,
        .llseek = mem_lseek,
};

static struct class *emce_class;

static int emce_of_probe(struct platform_device *ofdev)
{
        struct emce_device *edev = NULL;
        ...

        cdev_init(&edev->cdev, &emce_fops);
        kobject_set_name(&edev->cdev.kobj, "mem");
        if(cdev_add(&edev->cdev, edev->dev, USER_MEM)) {
                ...
        }

        emce_class = class_create(THIS_MODULE, DRIVER_NAME);
        if(IS_ERR(emce_class))
                ...

        for(minor = 0; minor < USER_MEM; minor++)
                device_create(emce_class, dev, MKDEV(MAJOR(edev->dev), minor),
                              NULL, "emce%d", minor);
        ...
}
```

Listing 4.2: Character device initialization

chunks, hence the name character device.

Character device initialization was implemented in the `probe` function mentioned above. To set up a character device a `struct file_operations` [37] is needed. This `struct` has an entry for every possible file operation which needs to point to the implementation of this operation. Not implemented functions need to be null pointers which is taken care of by leaving out the fields in the GNU style initialization seen in listing 4.2 line 20. In the example in listing 4.2 the `close` function is left out, since it is not needed. This function would normally be used for cleaning up allocations done in the `open` function. However, in this kernel module only physical memory, which is owned by this module, is accessed via the character devices. Therefore, no allocations are necessary.

Allocating the necessary character devices and initialization is taken care of by the functions in lines 36 to 48 in listing 4.2. The most important functions are `cdev_init` which allocates the driver structure and initializes the operations, and `device_create` which allocates and creates the actual devices [37]. Every device needs a major and a minor number which are used to identify which device a file belongs to. In this example the major number is automatically allocated and assigned and the minor number corresponds to the internal memory number. The rest of the code causes the kernel to create the device nodes */dev/emce0* to */dev/emce3*. Without these routines these nodes would need to be created manually using `mknod`.

```
ssize_t mem_read (struct file *file, char __user *buf,
                  size_t count, loff_t *ppos) {
        struct user_mem *mem = file->private_data;

        if(*ppos >= mem->size)
                return 0; //EOF

        if(*ppos + count >= mem->size)
                count = mem->size - *ppos;

        if(copy_to_user(buf, mem->base_address+*ppos, count))
                return -EFAULT;

        *ppos+=count;
        return count;
}

static int mem_open(struct inode *inode, struct file *file) {
        struct emce_device *edev;

        if(MINOR(inode->i_rdev)>=USER_MEM)
                return -ENODEV;

        edev = container_of(inode->i_cdev, struct emce_device, cdev);
        file->private_data = &edev->mem[MINOR(inode->i_rdev)];
        return 0;
}
```

Listing 4.3: Character device access functions

If user space calls the function `open` on one of those device nodes, the kernel deduces from the major number that this driver is the owner and calls `mem_open` (see line 18 in listing 4.3). As mentioned earlier, almost no setup needs to be done in this function. This function only checks if the minor number is out of range. The minor number is used as an index into the four memories *inbuf*, *H*, inactive *outbuf*, and active *outbuf*. After the range check the memory area pointer is assigned to the private data of the *inode* [37]. Storing this information within the *inode* provides the other access functions (e.g. `read`, `write`) directly with information about the memory operated on.

Since the `read` and `write` operations are very similar, only `mem_read` is shown in listing 4.3 line 2. Upon invoking the `read` function user space provides a buffer to write to (`buf`) and the number of bytes to read (`count`). The current file position is provided with the variable `ppos`. Since the requested information is already in memory, the only things that need to be done are:

1. Boundary check (lines 5 to 9).

2. Copy the data to user space (line 11). This must be done with the kernel provided `copy_*_user` macros. Kernel memory and user memory must never be mixed, since this can cause security problems [37].

3. Advance the file position (line 14).

4. Return the number of bytes actually read.

In addition to the traditional access methods the function `mmap` was implemented. This function allows user space to directly map the memory represented by the character device. With this technique a user space program can access the memory directly via a pointer, which avoids the copy operation mentioned above. In this case memory protection is handled by the memory management unit instead of the above mentioned `copy_*_user` macros. This method is different to mapping */dev/mem* directly by providing the correct offset and bounds of the access buffer. Therefore, this method doesn't bear the risks of system crashed by writing to the wrong address.

### 4.1.2 Register Access

Access to the registers is provided via files in *sysfs* [39]. To ease user space handling and following the guidelines of *sysfs* with one setting per file, every single flag from the registers is implemented as a file in *sysfs*. A *sysfs* file can be allocated with a `device_attribute` structure, which needs a function pointer to a `show` and a

`store` function, access rights, and a file name. Setting one of the function pointers to `NULL` disallows the respective access. `show` is called if user space reads from a file and `store` if user space writes to a file. Since these files are supposed to represent a single value there are no `open` and `close` functions.

If user space writes a value to one of these files, then the `store` function is called. Additionally, a pointer to a single page containing the written contents and the number of written bytes is passed to the function. The `show` function is called if user space reads from a file in *sysfs*. A pointer to one pre-allocated page is provided to the function. The function has to return the number of bytes written.

The *sysfs* files have to be assigned to attribute groups which in turn represent subdirectories below the driver directory [39]. For easier handling generalized `show` and `store` functions were developed. The generalized `store` function named `fpga_flag_store` converts the textual representation of the passed number to an integer and performs out of bound checking. This number is then shifted to the right position in the register and the memory mapped register updated with a read modify write cycle. This read modify write cycle is protected by a spin lock to ensure that possibly concurrent write operations to the registers don't interfere. This spin lock isn't actually needed by a uniprocessor system and gets optimized away during the compilation. Since this code could also be used on a different processor it is best practice to keep the spin lock. The generalized `show` function `fpga_flag_show` reads the desired value from the register, shifts it to the correct position, and truncates the value to the correct length. This value is then converted to a textual representation and written to the buffer.

Since the digital signal processing chain features a lot of configuration options, macros that allow instantiating a single configuration option with one line of code were developed. A single configuration option is called flag and it possesses a register address, a bit position within the register, a bit width, and possibly a maximum value. The following macros allow instantiating the above mentioned *sysfs* functions:

`FPGA_FLAG(base, name, access, register, bit, length)`
: This macro creates a flag which represents a part of a register within the group `base` with the name `name` and access rights `access`. The physical base address of the register has to be provided with `register`, the bit number within the register with `bit`, and the length of the flag with `length`.

`FPGA_FLAGM(..., max)`
: This macro has the same arguments as `FPGA_FLAG` with an additional `max`, which is the maximum value + 1, that is allowed.

`FPGA_FLAGC(..., show, store)`

Same as `FPGA_FLAGM` but with the two additional arguments `show` and `store` which allow the use of custom functions.

A reference to the source code implementing and using this macros can be found in appendix A.1 and an overview of the registers including the flags can be seen in appendix A.2.

### 4.1.3 Interrupts

With the function `request_irq` a function can be registered that is supposed to handle the interrupt with a specific interrupt number [37]. The interrupt number is set by the hardware implementation in XPS. An example usage of the register function can be seen in listing 4.4 line 24. In this example the function `edev_isr` in line 1 is registered as the interrupt handler.

```
static irqreturn_t edev_isr(int irq, void *dev_id)
{
        struct device *dev=(struct device*)dev_id;
        struct emce_device *edev = dev_get_drvdata(dev_id);

        u32 status;
        int i;

        status = in_be32(edev->base_address + EMCE_INTR_IPISR_OFFSET);
        out_be32(edev->base_address + EMCE_INTR_IPISR_OFFSET, status);

        for(i=0; edev->int_nodes[i]; i++)
                if((status >> (15 - i)) & 1)
                        sysfs_notify_dirent(edev->int_nodes[i]);

        return IRQ_HANDLED;
}

static int emce_of_probe(struct platform_device *ofdev)
{
        struct emce_device *edev = NULL;
        ...

        if(request_irq(edev->irq, edev_isr, IRQF_SHARED, DRIVER_NAME, dev))
        {
            ...
        }
        ...
    }
```

Listing 4.4: Driver interrupt routine

Upon invocation the interrupt handler reads the interrupt register (listing 4.4 line 9). This register contains the different interrupt sources of the processor interface (see appendix A.2). The hardware does not clear the interrupts automatically after this read operation. Therefore, the interrupts are cleared by writing the

read value back to the interrupt register (line 10). This call resets only the interrupts registered in `status`. Next the different interrupt sources are checked and the `sysfs_notify_dirent` function called with a *sysfs* directory entry accordingly (lines 12 to 14). Those *sysfs* entries are created in the `emce_of_probe` function. After successfully handling the interrupt the handler has to return `IRQ_HANDLED` to notify the kernel. Without this the kernel searches for other interrupt handlers for this interrupt number.

The `sysfs_notify_dirent` function wakes up user space processes that wait with the `poll` system call for `POLLPRI` and `POLLERR` events [40]. An example program using this interface can be seen in listing 4.5. In order to receive the notifications, the file passed as the first argument to the script is opened for reading in line 5. In lines 6 to 7 the `poll` system call is configured with the above mentioned event codes on the opened file. In order for the notification to work, the file has to be read completely which is dictated by the kernel API. This is done by resetting the file position to 0 (line 9) followed by reading the complete contents (line 10). Resetting the file position ensures that further invocations after the first event also work. Next the `poll` system call is invoked which blocks execution of the program until the interrupt occurs (line 11). After an interrupt occurrence a line is printed and the waiting procedure is repeated.

```
#!/usr/bin/python
import select
import sys

with open(sys.argv[1], "r") as f:
    p = select.poll()
    p.register(f, select.POLLPRI | select.POLLERR)
    while 1:
        f.seek(0)
        f.read()
        p.poll()
        print sys.argv[1], "fired!"
```

Listing 4.5: Wait for interrupt example in Python

During the blocked wait the program is not running and, therefore, not consuming CPU cycles. Multiple programs and threads can wait for events on the same file utilizing this mechanism.

## 4.2 Network Daemon and Web-Interface

A network daemon with an accompanying web interface was developed to allow controlling the digital signal processing chain via Ethernet. This daemon is written in Python using the event-driven network library Twisted [41]. This network

library encapsulates the different network layers and protocols into Python classes simplifying network development.

Protocol classes in Twisted provide a `Received` method. This method is called with the received data as argument after a complete line for a text based protocol, a complete packet for a packet based protocol, or binary chunk for a binary protocol was received. By sub-classing base protocols two communication methods were implemented. A websocket based protocol for a web interface and a text based protocol to enable communication with Matlab.

The web interface uses JSON encoded packets for communication. The JSON packets contain an object with at least a `"cmd"` parameter. This parameter provides the allowed commands `"set"`, `"do"`, and `"get"`. With the additional parameters `"target"` and `"value"` the signal processing setting specified by target can be set to a specific value or can be queried. An example can be seen in listing 4.6. The daemon replies with `"update"` set as `"cmd"`, the specific target name and the new value. Additionally interrupts are reported with `"update"` set as the `"cmd"` value and the interrupt target name. The target names directly correspond to the *sysfs* files provided by the kernel module described in section 4.1. Therefore, the daemon just opens the file given by target and performs the required read or write operation. The `"do"` command is the same as `"set"` except that a value of 1 is assumed.

```
{
    "cmd": "set",
    "target": "core/L",
    "value": "2000"
}
```

Listing 4.6: Example JSON formated packet to set $L = 2000$

The web interface utilizing this protocol was developed in HyperText Markup Language (HTML) version 5 and JavaScript and exposes every possible setting of the kernel module. This web interface can be reached by opening the internet protocol (IP) address of the FPGA module in the web browser. The web interface is served by a small web server built into the daemon. The IP address is hard coded in the software image and defaults to `192.168.2.2`. A screen shot of the web interface can be seen in figure 4.2.

The Matlab communication protocol consists of two different protocols. It is not possible to use asynchronous and synchronous network communication over a single connection within Matlab. Therefore, a synchronous line based protocol was implemented. One command can be transmitted per line and every line starts with a command, followed by a space separated list of arguments. The commands are the same as in the JSON based protocol (`get`, `set`, `do`). Since this protocol is synchronous, the `do` command waits for the appropriate interrupt before replying. The

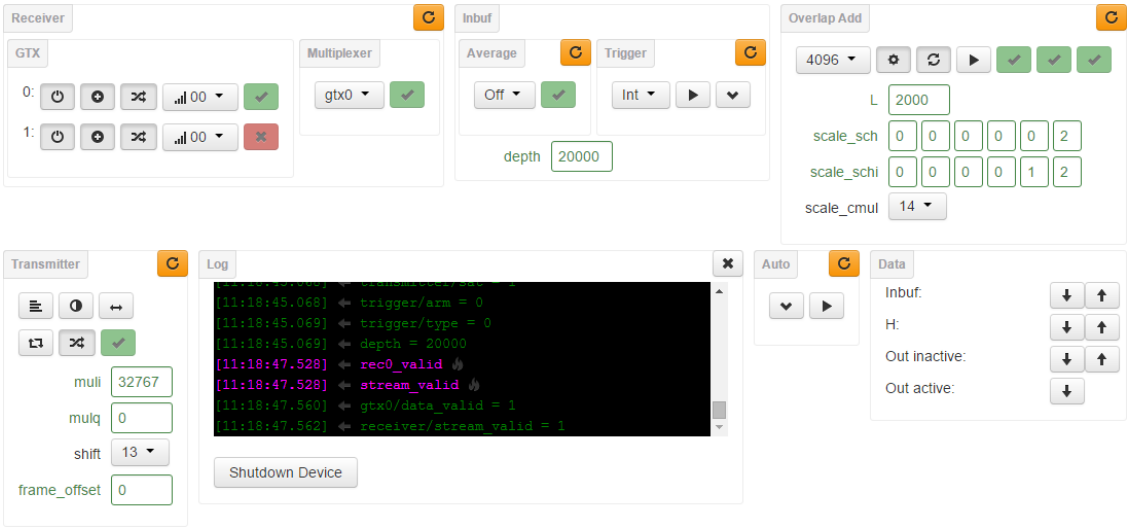



Figure 4.2: Screen shot of the web interface

daemon replies `OK` in the case of success or `ERROR` in case of an error (e.g. timeout). An example command can be seen in listing 4.7. The additional commands `read` and `write` implement interacting with the sample buffers. Both commands need the name of the target memory as the first argument (`emce0` to `emce3`) and the number of bytes as the second argument. The `write` command has to be followed by the specified number of bytes of raw data after the newline. The `read` command provides the specified number of raw bytes instead of the `OK` reply. The raw data is composed of 16 bit signed integers with big endian byte ordering. This protocol implementation can be reached via the transport control protocol (TCP) port `8000`.

```
set core/L 2000
```

Listing 4.7: Example line to set $L = 2000$ with the text based protocol

Additionally to this protocol a second asynchronous protocol was implemented that allows to be notified about interrupts. Any input to this protocol is ignored. Interrupts are reported via this protocol with the name of the interrupt followed by a newline. The asynchronous protocol listens on TCP port `8001`.

This daemon is started automatically during the OS boot sequence. The JSON- and text based protocol both provide an additional shutdown command that allows shutting down the OS. The OS should always be shut down before switching off the FPGA to prevent file system corruption. A reference to the source code for the daemon can be found in appendix A.1.

## 4.3 Matlab Algorithms

The text based protocol described in section 4.2 was implemented in a set of Matlab classes to ease controlling the hardware. These Matlab classes depict the same naming scheme as the *sysfs* interface with a dot instead of a slash as the path separator. The sample buffers can be accessed via the properties `inbuf` for *inbuf*, `H` for $H$, `out_inactive` for inactive *outbuf*, and `out_active` for active *outbuf* (see section 3.1). Those dependent properties automatically convert the raw data to complex vectors and vice versa. The interrupts are exposed via Matlab events. All the classes are fully documented with Matlab comments and a reference to the source codes can be found in appendix A.1. A description of all the properties can be found in appendix A.2. An example usage can be seen in listing 4.8.

```
ml507 = ML507.ML507(); %instantiate the class

ml507.depth = 20000; %set signal period N = 20,000
ml507.transmitter.mul = 32767; %set Γ_{L,set} = 32,767
ml507.transmitter.shift = 5; %set output multiplier shift to 5
ml507.core.n = 4096; %set n_fft = 4,096
ml507.core.L = 2000; %set L = 2,000
ml507.core.scale_sch(1:6) = [2 0 0 0 0 0]; %schaling schedule for FFT
ml507.core.scale_schi(1:6) = [2 1 0 0 0 0]; %schaling schedule for inverse FFT
ml507.core.scale_cmul = 1; %complex multiplier scaling
ml507.core.iq = 1; %enable iq decode
ml507.core.circular = 1; %enable circular convolution
ml507.transmitter.resync(); %resync transmitter to set signal period
ml507.trigger.arm(); %arm trigger
ml507.trigger.fire(); %fire trigger
```

Listing 4.8: Example usage of the Matlab driver for the hardware

Utilizing these classes the one port VNA as described in section 2.1 was implemented in a Matlab script. A description of this script can be found in section 4.3.1. Additional reflections caused by the measurement setup and the DUT cause the setup to exhibit a different $\Gamma_L$ than was set by $\Gamma_{L,set}$. These imperfections can be compensated with an error box as has been done for the one port VNA (see section 2.1). Since highly non-linear DUT can experience load-dependent $S_{22}$, this approach could prove futile. Therefore, an iterative target algorithm was developed to find the correct $\Gamma_{L,set}$ needed to achieve a specific $\Gamma_{L,target}$. This algorithm is described in section 4.3.2. Using this target algorithm the filter calibration script, described in section 4.3.3, was developed.

### 4.3.1 One Port VNA

The one port VNA as described in section 2.1 determines the reflection coefficient $\Gamma_L$ by measuring the magnitude and the phase of the incident and reflected power

waves. This measurement has to be corrected for systematic errors with the error box described in section 2.1. To use this error box additional calibration measurements have to be carried out beforehand.

The sampled data and the timebase from the oscilloscope were acquired using the Instrument Control Toolbox in Matlab. To extract the magnitude and the phase with high accuracy for specific frequencies the samples were multiplied with a flat top window (see section 2.1). Next, this modified data was transformed with the FFT into the frequency domain. After that the reflection coefficient was calculated by dividing the transformed data from the incident and the reflected power wave at the indices corresponding to the desired frequencies. The Matlab function implementing this functionality can be seen in listing 4.9.

```
function [rho, A, B] = manyRho(xincrement, a, b, freqs)
    A = fft(a.*flattopwin(length(a), 'periodic'));
    B = fft(b.*flattopwin(length(b), 'periodic'));

    df = 1/xincrement/length(A);
    freqs = floor(freqs./df)+1;

    A = A(freqs);
    B = B(freqs);

    rho = A./B;
end
```

Listing 4.9: Function for calculating $\frac{a}{b}$ from sampled data at multiple frequencies

The calibration of the error box was realized by measuring three different targets at multiple frequencies. In order to enable measurements with the possible frequency resolution of the oscilloscope, the calibration measurements were linearly interpolated. A Rohde & Schwarz Z132 (female) calibration kit was used for this calibration. To calculate the impedance values of the calibration kit the Matlab function `calcZ132` was developed. This function calculates the short, match, and open impedance for a given frequency according to the data sheet [42]. The Matlab function called `calcErrorBoxM` was generated with a Mathematica notebook which was used to solve the equations derived from the error box in section 2.1. With the impedance values from the calibration kit and the calibration measurement values this function calculates the S-parameters of the error box.

The developed function `calcGl` is able to calculate a corrected $\Gamma_L$ with these S-parameters and a reflection coefficient calculated with `manyRho` from acquired samples. A reference to the described functions and the example script `measureVNA` can be found in appendix A.1. This example script is split into four parts. The first part initializes the used instruments, the second part carries out the calibration measurements, and the third part implements the interpolation and error box calculations. The fourth part contains the measurement used to generate the verification data for section 5.1.

### 4.3.2 Target Algorithm

As described above, the resulting $\Gamma_L$ of the complete setup is influenced by additional reflections resulting from imperfections in the measurement setup. Furthermore, highly non-linear DUT can experience load-dependent $S_{22}$ which influence the resulting $\Gamma_L$ additionally. Therefore, the following iterative target algorithm was developed.

The iterative target algorithm used in this work makes the following steps:

1. Start at arbitrary start point by setting $\Gamma_{L,set}$.
2. Measure the resulting $\Gamma_L$.
3. If $|\Gamma_L - \Gamma_{L,target}| < accuracy$ then the algorithm is done. The accuracy used in this work was a magnitude difference $<10^{-3}$ and a phase difference $<0.25°$.
4. Scale $\Gamma_{L,set}$ according to $\frac{\Gamma_{L,target}}{\Gamma_L}$.
5. Pause to compensate for lag time introduced by the digital signal processing chain.
6. Repeat from step 2.

The pause is needed, because the digital signal processing chain experiences lag times up to 1 ms. Since the target algorithm and the digital signal processing chain run completely independent, the algorithm can experience stability issues if the pause of the algorithm is too short. A visualization of the different phases of the digital signal processing chain and the algorithm can be seen in figure 4.3. In this visualisation it can be seen that the algorithm starts by measuring $\Gamma_L$. After the new $\Gamma_{L,set}$ is set, the wave $a_2$ is modified immediately, which in turn results in a new $b_2$. After the next acquire, filter, and switch cycle of the digital signal processing chain, a new wave $a_2$ is generated, which results again in a new $b_2$. If the pause of iterative algorithm is too short, then the next step will be based on a wrong $\Gamma_L$ measurement, which can lead to a high number of algorithm steps.

Preliminary tests showed that this iterative target algorithm is capable of reaching multiple specific $\Gamma_{L,target}$. Therefore, the algorithm is capable of compensating the reflections introduced by the measurement setup. These are caused by connectors and mismatches between the components. The iterative target algorithm also worked after additional reflections have been introduced intentionally in the measurement setup. A more detailed verification of this algorithm can be found in section 5.2.1.

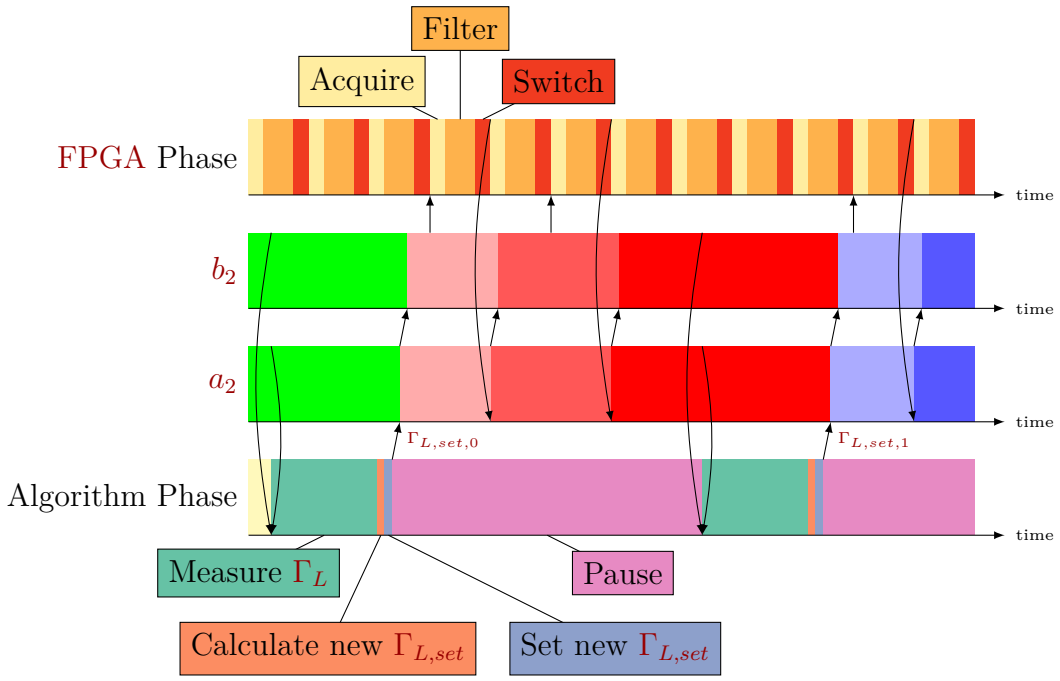



Figure 4.3: Visualization of the different phases of the FPGA implementation and target algorithm over time (not to scale)

The iterative target algorithm was implement in the Matlab function `findTarget`. The needed arguments are a `ML507` object for controlling the signal processing chain, the target, a function handle to a function implementing the $\Gamma_L$-measurement, and the pause time. Additionally, a start point can be specified. The `findTarget` function returns upon algorithm termination the actual $\Gamma_L$ and the needed $\Gamma_{L,set}$. Additionally, the complete trajectory consisting of every reached $\Gamma_L$, including start and stop point, is returned.

### 4.3.3 Filter Calibration

The digital filter in the signal processing chain is needed to synthesize a constant $\Gamma_L$ over a wide frequency range. This filter is required to compensate the group delay caused by the measurement setup and cabling needed for reaching the DUT. This is achieved with a negative phase correction for cyclic signals which works by modifying the phase of the signal to achieve a zero phase delay between the reflected wave and the incident wave at a later signal cycle. Furthermore, it is able to compensate the frequency responses of the components used in the measurement setup.

Calibrating this filter was carried out by measuring the needed $\Gamma_{L,set}$ for a specific

calibration point over the required frequency range. This calibration data was linearly interpolated to cover every frequency point of the digital filter. Since the measured values experience very high phase differences, the data was split into magnitude and unwrapped phase for the interpolation. After interpolation, the $\Gamma_{L,set}$ values were added to the filter by multiplying them with the transfer function. An example of the resulting transfer function can be seen in section 5.2.2.

A reference to an example script named `filterCalibration` is provided in appendix A.1 which acquires a filter calibration for the points 1, $-1$, $0.5\angle 135°$ and $0.5\angle -45°$. Additionally it calculates filters for the mean of the measurements 1 and $-1$, and $0.5\angle 135°$ and $0.5\angle -45°$. After these calculations the script performs verification measurements for the different filter calibrations. The results of these measurements can be found in section 5.2.2.

# 5 Verification of the Measurement System

The implemented ELP design was tested with the setup presented in figure 5.1 at a center frequency $f_0$ of 900 MHz. This test setup consists of the three parts which have been described in chapter 2. As mentioned in section 2.2, instead of a circulator, the directional coupler *dir2* in combination with attenuator *att1* was used. Furthermore, since during verification only waves synthesized inside the specified bandwidth of 20 MHz were used, the low-pass filter *lp2* was used instead of a bandpass filter. A detailed list of used equipment can be seen in appendix C.

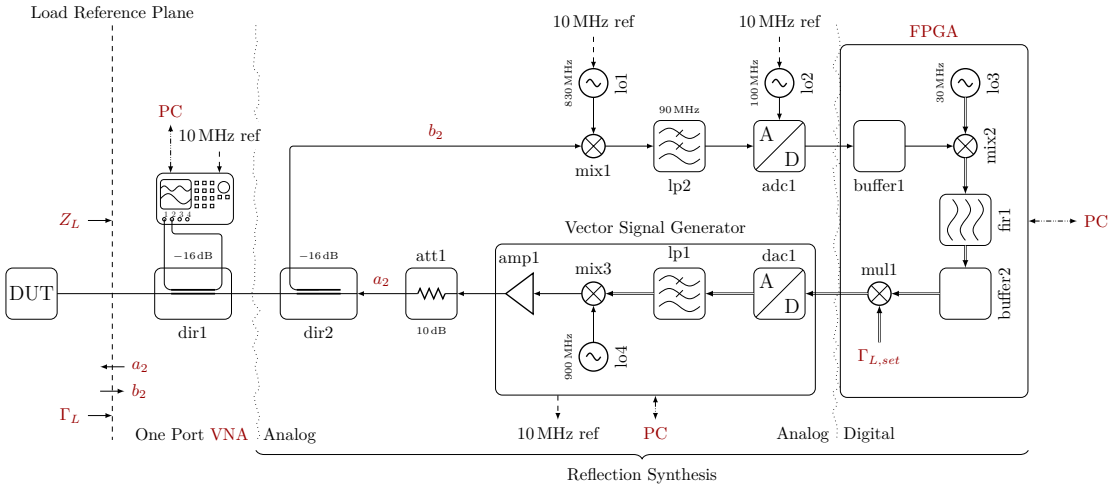



Figure 5.1: Measurement system verification test setup

The overall setup consists of two independent systems. The one port VNA and the reflection generation consisting of the analog and the digital part. The one port VNA is needed to measure the current $\Gamma_L$ (see section 2.1). Verification of this part can be found in section 5.1. As described in section 2.2, the analog part is responsible for separating the incident wave $b_2$, converting it to IF, and digitally sampling the signal. Furthermore it handles digital-to-analog conversion, up-conversion to RF and feeding back the reflected wave $a_2$ to the DUT. The digital part as described in section 2.3 handles filtering the signal to enable broadband reflection synthesis and applying the reflection coefficient $\Gamma_{L,set}$. Since analog and

digital part can't be verified separately, verification of the combined reflection synthesis part can be found in section 5.2. During these measurements, a noticeable phase drift was observed, and, therefore, an additional measurement was acquired, as seen in section 5.3, to verify the origin of this phase drift.

## 5.1 One Port VNA Verification

For the calibration and verification of the one port VNA, the modulator of the vector signal generator was turned off. Stepping through the frequency range was achieved by stepping the local oscillator *lo4* (see figure 5.1). This allowed for easier test automation since only the vector signal generator had to be controlled. The vector signal generator was set to an output power of 0 dBm which results in about −10 dBm at the load reference plane (see figure 5.1). A ZV-Z132 (female) calibration kit from Rohde & Schwarz was used for the calibration setup in place of the DUT. Since the calibration is only valid for a single frequency, the full calibration measurements were acquired for 21 $\Delta f$-aligned points within a 25 MHz range at a center frequency of 900 MHz. This was achieved by connecting the appropriate port of the calibration kit and acquiring $\Gamma_{L,f}$ values for every frequency point. After the measurement, the different $\Gamma_{L,f}$ were linearly interpolated at every $\Delta f$ bin. Finally, the S-parameters of the error box (see section 2.1) were pre-calculated for every bin utilizing the values from the data sheet of the calibration kit [42].

For the verification of the calibration, a stub tuner from Maury Microwave was tuned to a specific position and then measured utilizing the one port VNA in this work. Measurements were acquired over the whole 25 MHz bandwidth at the calibrated points. Additional measurements were acquired at an offset of $\frac{1}{2}\Delta f$, to test the performance of the chosen window function. The performance of the interpolation was further tested by measurements between the calibrated points. To verify the $\Gamma_L$, which was measured using the above described setup, those measurements were compared to a measurement acquired using a commercially available VNA. The VNA used for verification was an Agilent Technologies E8364A, calibrated with an Agilent Technologies N4433A calibration kit (see appendix C). VNA settings used were −10 dBm test port power, 35 kHz IF bandwidth, and 1,601 measurement points.

Every point was measured 100 times to quantify the non systematic errors. With those $N$ observations of every point, the measured samples $x$, and the mean of the measured samples $\overline{x}$, the corrected sample standard deviation $s$ was calculated according to equation (5.1). With the sample standard deviation $s$ the standard error $SE$ was calculated according to equation (5.2).

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2} \tag{5.1}$$

$$SE = \frac{s}{\sqrt{N}} \tag{5.2}$$

The standard error and the difference to the results from the Agilent VNA roughly stay the same over the complete 25 MHz range. There is also no noticeable difference between the three different types of points. Thus, the linear interpolation and the chosen window function seem to be appropriate for this type of measurement. Since there are only passive parts involved in the one port VNA, it should also be possible to use less calibration points to achieve a faster calibration. The measurement result of three exemplary points can be seen in table 5.1.

| **frequency** | | **mean** | **standard error** | **ΔVNA** |
|---|---|---|---|---|
| MHz | | 1 | 1 | 1 |
| 887.5 | (cal) | $581.0232 \times 10^{-3}$ | $37.6 \times 10^{-6}$ | $2.1 \times 10^{-3}$ |
| 887.505 | ($\frac{1}{2}\Delta f$) | $581.0589 \times 10^{-3}$ | $38.2 \times 10^{-6}$ | $2.2 \times 10^{-3}$ |
| 888.095 | (between cal) | $582.3070 \times 10^{-3}$ | $36.6 \times 10^{-6}$ | $2.1 \times 10^{-3}$ |

(a) Magnitude

| **frequency** | | **mean** | **standard error** | **ΔVNA** |
|---|---|---|---|---|
| MHz | | ° | ° | ° |
| 887.5 | (cal) | −157.5029 | $8.7 \times 10^{-3}$ | 2.2 |
| 887.505 | ($\frac{1}{2}\Delta f$) | −157.4787 | $8.1 \times 10^{-3}$ | 2.3 |
| 888.095 | (between cal) | −157.7972 | $9.7 \times 10^{-3}$ | 2.3 |

(b) Angle

Table 5.1: Measured reflection coefficient with standard error and difference to Agilent VNA

The setup of the one port VNA consists of an oscilloscope and only passive components. Therefore, the standard error notably depends on the performance of the oscilloscope. It is mainly caused by the ADCs contained in the oscilloscope and imperfections in the triggering and sampling. This also means that these results are only valid for the used oscilloscope in this work.

The difference to the results from the Agilent VNA are mainly caused by the used calibration kit. This is caused by the performance of the error box as described in section 2.1 which was used to correct the systematic errors. The calibration of the

error box depends only on measurements of the calibration standard. Therefore, the performance can be increased with a better calibration kit or by additional calibration measurements.

Further care should be taken to always ensure a strong enough signal power at the oscilloscope ADCs. According to [43] the SNR caused by quantization noise of ADCs depends on the bit width and the amplitude. Therefore, small signals cause a low SNR. Additionally, a small $\Gamma_L$ value implies a small reflected wave and therefore a lower SNR. Therefore, the resolution decreases for small $|\Gamma_L|$. This can be compensated with averaging which would increase the effective bit width at the cost of longer measurement times [24].

## 5.2 Reflection Generation Verification

The reflection measurements were carried out using a test setup as displayed in figure 5.1. Here an Agilent VNA was connected instead of the DUT to verify the synthesized reflection coefficients. Furthermore, the Agilent VNA was synchronized to the setup using the same 10 MHz reference in order to synthesize the needed phase coherent signals for the test setup. This allowed verifying the synthesized $\Gamma_L$, and calibrating the FIR filter *fir1* more accurately, than using the built in one port VNA.

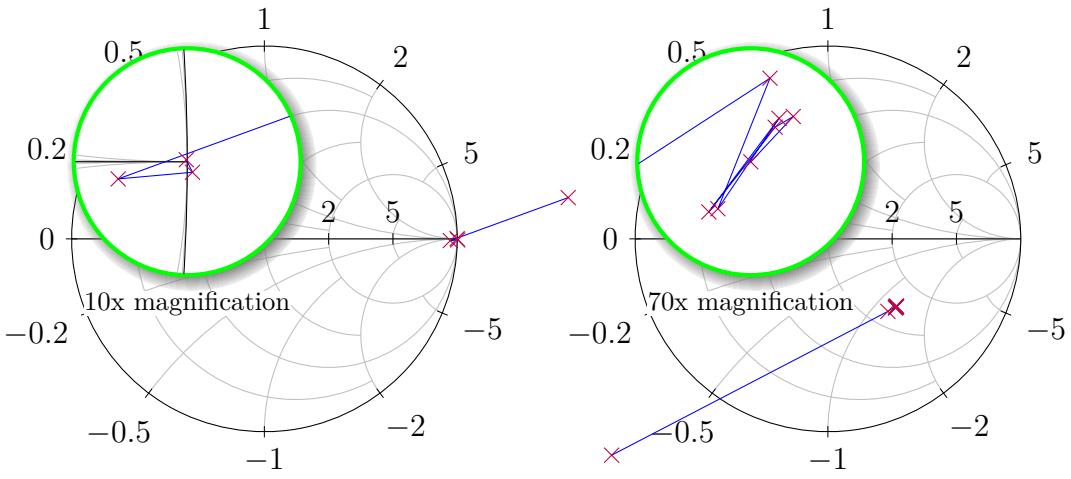
All the reflection generation verification measurements used 20,000 samples as signal period $N$, an overlap add block size $L$ of 2,000, and an FFT width $n_{\text{fft}}$ of 4,096 (see figure 2.10). The filter *fir1* was preloaded with the impulse response of a low pass filter with a cut off frequency of 25 MHz to filter out the aliases (see section 2.3). For the duration of these measurements, the FPGA implementation was switched to automatic mode which emulates continuous signal processing (see section 3.1.4). Changes at the input of the digital part propagate with a lag time of about 1 ms to the output using automatic mode and the above mentioned signal processing configuration.

The reflection generation consists of two tasks. First, the iterative target algorithm described in section 4.3 is needed, to synthesize specific $\Gamma_L$. The performance of this algorithm can be seen in section 5.2.1. Second, this algorithm was then used for calibrating and verifying the filter *fir1*. Measurement results for the verification measurements utilizing the previously calibrated filter can be seen in section 5.2.2.

### 5.2.1 Iterative Target Algorithm

The target algorithm is able to find the correct $\Gamma_{L,set}$ with a given $\Gamma_{L,start}$ in order to synthesize a given $\Gamma_{L,target}$. It works by setting the current $\Gamma_L$ into relation with the given $\Gamma_{L,target}$ and scaling $\Gamma_{L,set}$ accordingly (see section 4.3). This iterative target algorithm is needed to reach a specific $\Gamma_L$ even with highly non-linear DUT. These DUT can exhibit load dependent $S_{22}$ which renders error box based approaches unsuitable.

As mentioned above, the digital reflection generation introduces a large latency on the order of about 1 ms. Therefore, a delay time of 100 ms was used for the iterative algorithm (see section 4.3). This allows the system to settle on a $\Gamma_L$ before continuing with the algorithm. If the wait time is too short, the iterative target algorithm can start to cause stability issues and the completion time of the algorithm actually increases while a long wait time increases the completion time. The high delay time of 100 ms was chosen since the Matlab function **`pause`** is not capable of short delays and exhibits very high inaccuracy [44].




(a) $\Gamma_{L,target} = 1$ (b) $\Gamma_{L,target} = 0.5\angle{-45°}$

Figure 5.2: Exemplary trajectories of the target algorithm

Two descriptive example trajectories can be seen in figure 5.2. In figure 5.2a the algorithm terminated just after three steps, whereas in figure 5.2b the worst observed case can be seen. The trajectory in figure 5.2b oscillates around the target before reaching it. The reason can either be found in a too stringent termination condition or in a bad resolution of the VNA used for measuring $\Gamma_L$ or the digital part of the reflection synthesis in the target range.

Depending on the starting point $\Gamma_{L,start}$ and the target $\Gamma_{L,target}$ the performance of the algorithm varied. A histogram showing 108 different observations for different

$\Gamma_{L,start}$ different $\Gamma_{L,target}$ and different frequencies can be seen in figure 5.3. The trajectory in figure 5.2b is the only one where 8 iterations have been observed.

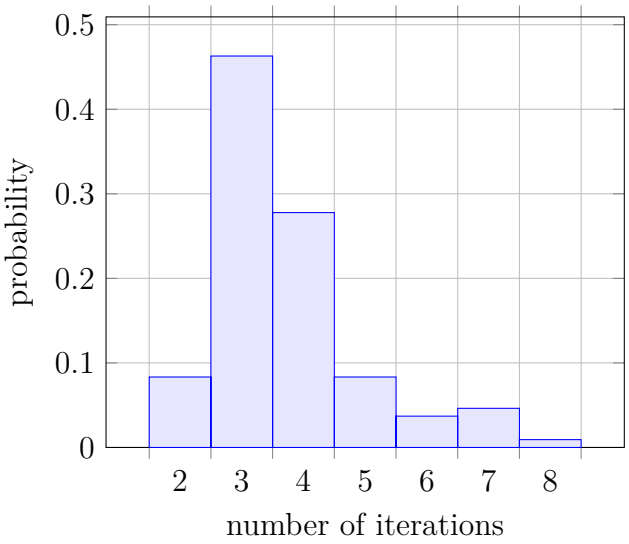



Figure 5.3: Histogram of algorithm iterations needed to find a specific $\Gamma_L$

As can be seen in figures 5.2 and 5.3 the algorithm is capable of reaching a specified reflection coefficient $\Gamma_{L,target}$ within a reasonable amount of time. Keeping the limitations above in mind, the algorithm can further be accelerated by adjusting the termination condition and the settling time.

### 5.2.2 Filter Performance

For verification, the filter *fir1* in figure 5.1 was calibrated by measuring the needed $\Gamma_{L,set}$ to achieve a specific $\Gamma_L$ over a baseband range from −13 MHz to 13 MHz at a RF of 900 MHz. The filter calibration is necessary to correct the frequency response of the reflection generation setup (see chapter 2). The needed $\Gamma_{L,set}$ were acquired in 1 MHz steps. Those values were normalized to represent the needed multiplier in order to achieve a constant $\Gamma_L$ over the whole bandwidth. Afterwards the multipliers were interpolated at the frequency points of the FFT of the FIR filter (see section 4.3).

Filter calibration was carried out at the following $\Gamma_L$ values: 1, −1, $0.5\angle 135°$ and $0.5\angle -45°$. Additional verification measurements were carried out with the mean of the acquired measurements at 1 and −1 as well as with the mean of the measurements at $0.5\angle 135°$ and $0.5\angle -45°$.

For the following verification measurements, the respective calibration measurement was combined with a 12.5 MHz low-pass filter by applying the acquired $\Gamma_{L,set}$ to

the transfer function of the filter. This calibrated filter transfer function was then loaded in the filter implementation *fir1*. An example of the uncalibrated low pass transfer function and two different calibrated filter transfer functions can be seen in figure 5.4.

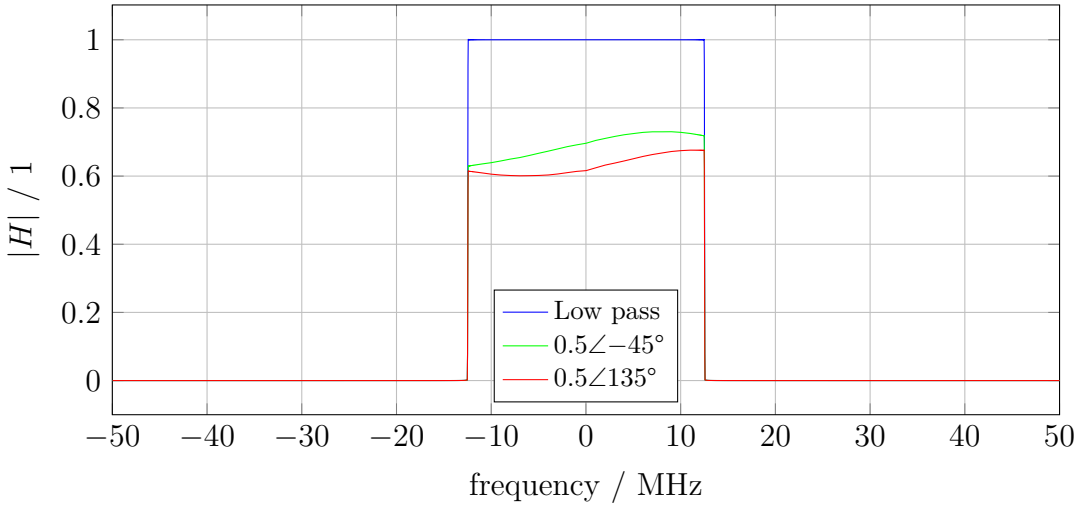
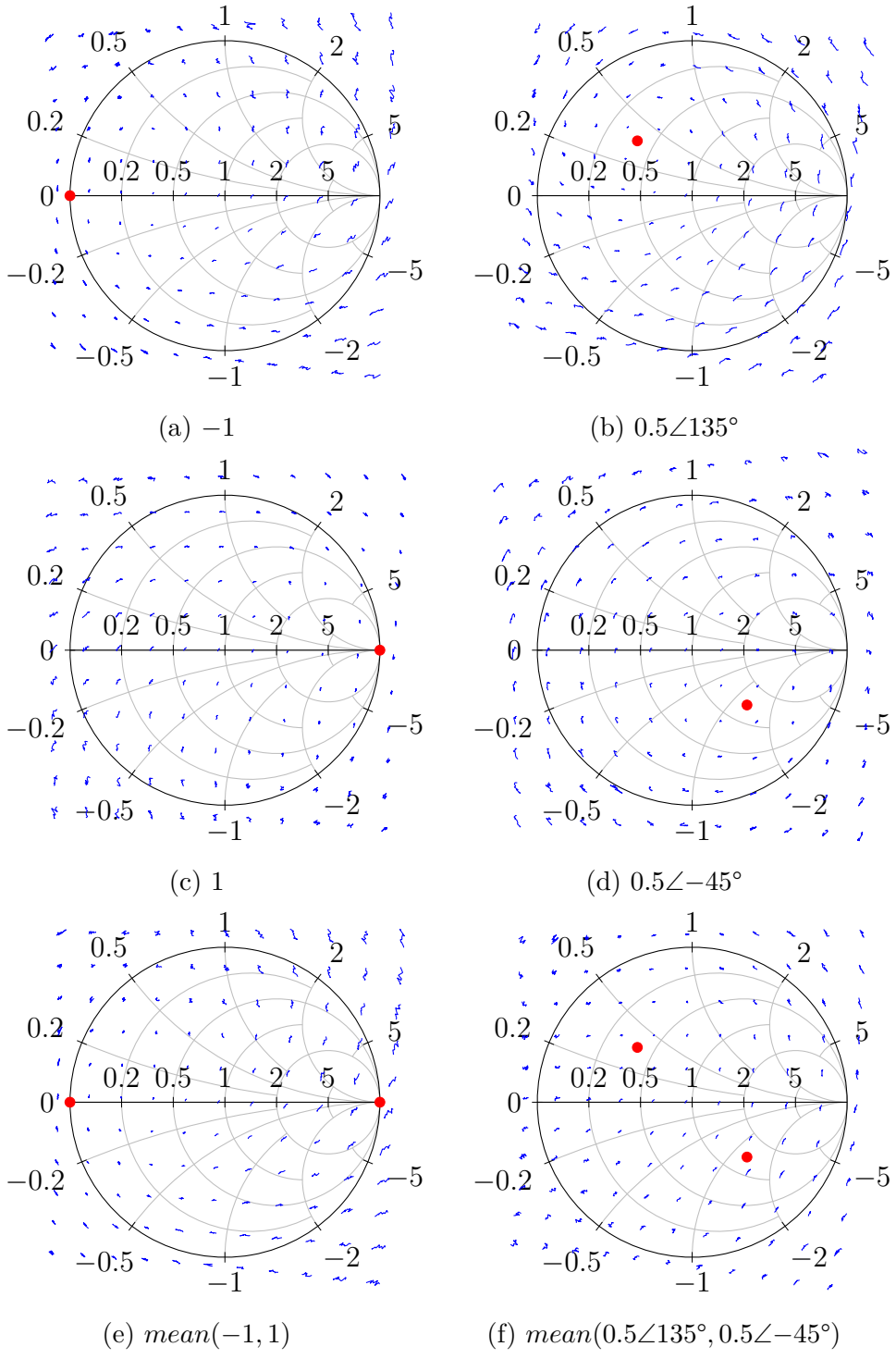



Figure 5.4: Uncalibrated and calibrated filter transfer functions

The verification measurements were acquired with the Agilent VNA at frequencies ranging from 895 MHz to 905 MHz in 1 MHz steps with a settling time of 0.5 s for every frequency point. This settling time was needed to ensure the digital signal processing chain has settled on the changed input. To verify the performance across the whole range, measurements were acquired over the possible $\Gamma_{L,set}$ range by stepping through an 11 by 11 grid. These measurements were acquired with the different filter calibrations mentioned above to compensate for the frequency response of the reflection generation setup.

The results, which can be seen in figure 5.5, contain every acquired trajectory in blue and the calibration point in red. Since only the filter *fir1* was calibrated, but neither the target algorithm nor an error model was used for the $\Gamma_{L,set}$ values, the overall grid is slightly distorted. As can be seen in these smith charts, the trajectories spread out at points far away from the calibration points. As expected, the filter calibrations utilizing the mean values achieve better trajectories between the calibration points. Since the trajectories around the calibration points show the best performance, it is advisable to use calibration points near the expected target $\Gamma_L$ and avoid calibration points at $|\Gamma_L| = 1$. Noticeable trajectory spreading near the calibration points is caused by the limited numerical precision of the filter implementation of *fir1* (see section 3.1.2).



(a) $-1$ (b) $0.5\angle 135°$

(c) $1$ (d) $0.5\angle -45°$

(e) $mean(-1, 1)$ (f) $mean(0.5\angle 135°, 0.5\angle -45°)$

Figure 5.5: 11 by 11 grid of reflection measurements over maximum range of $\Gamma_{L,set}$ values with calibrated filter at distinct calibration points

## 5.3 Phase Drift

During the measurements a phase drift could be noticed. In order to examine the cause of these phase drifts, a test measurement was acquired to observe the extent of the phase drift. For this measurement the test setup displayed in figure 5.1 and, again, the Agilent VNA (see appendix C) was used as DUT. The VNA was also connected to the 10 MHz reference, in order to synthesize phase coherent signals. The same settings for the FPGA implementation have been used ($N$ = 20,000, $L$ = 2,000, $n_{\text{fft}}$ = 4,096). According to section 2.3 *fir1* needs to be a low pass filter to filter out the aliases and was preloaded with the impulse response of a 25 MHz low pass filter. For the duration of the measurements, the FPGA implementation was switched to automatic mode to achieve continuous signal processing (see chapter 3). After those preparations the system was set up to realize a reflection coefficient $\Gamma_L = 1$ with the target algorithm. Measurements were acquired every 10 s for a duration of 1,000 min. The results of those phase drift measurements can be seen in figure 5.6, which shows the angle of $\Gamma_L$ over time. Since the magnitude did not change noticeably during those measurements it was left out of the plot.

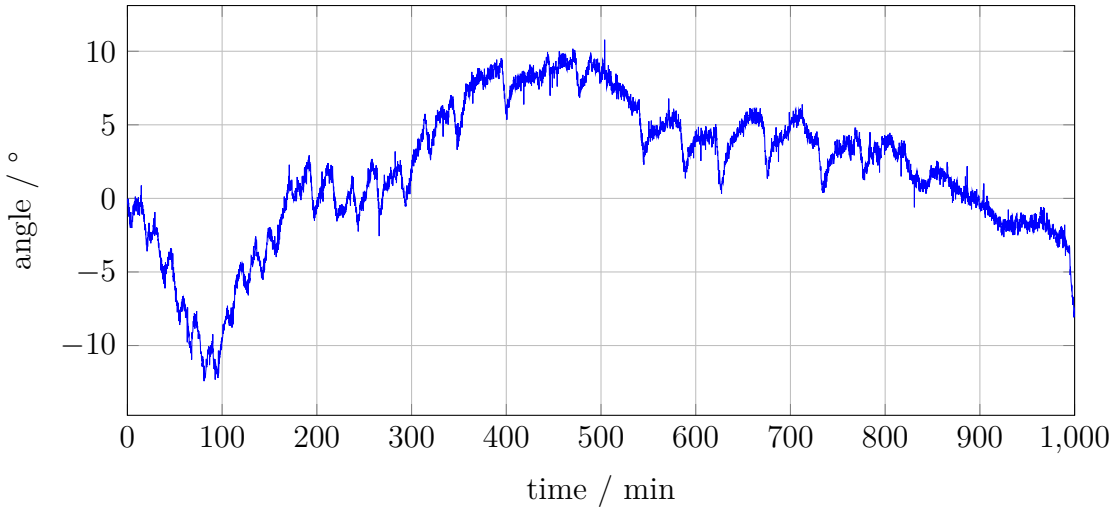



Figure 5.6: Phase drift of $\Gamma_L$ over time ($\Gamma_{L,start} = 1$)

These results indicate that the phase drift is not caused by the filter implementation of *fir1*, which could only cause constant offsets or linear drifts. Because of that, the test setup in figure 5.7 was extended to additionally measure the phase drift of all the used local oscillators in the setup. This setup consists of oscilloscopes to measure the different signals synthesized by the signal generators. To keep the setup as close to the previous measurements as possible, the vector signal generator was chosen as the reference source. Since the frequencies of the generators were 830 MHz, 100 MHz and 900 MHz, the additional function generator *trigger* (see figure 5.7) was needed to synthesize the greatest common divisor of those frequencies,

which is 10 MHz. Unfortunately, it was not possible to synthesize a stable and spectrally clear trigger signal using the function generator *trigger* that allowed reliably triggering for measuring the phase of the different signals. Therefore, the signal of the function generator *trigger* was also recorded. To measure all the signals, a second oscilloscope had to be used. Since triggering the second oscilloscope with the first one resulted in a worse performance than feeding both oscilloscopes with the same trigger, a chained setup was not used. Finally, after the measurement had taken place the different signals were corrected by the phase offset caused by the trigger signal, to achieve a phase offset of the trigger signal equal to zero. Consequently, the results of both oscilloscopes would be the same if the oscilloscopes would have been triggered perfectly. As before, a phase offset was acquired every 10 s for 1,000 min.

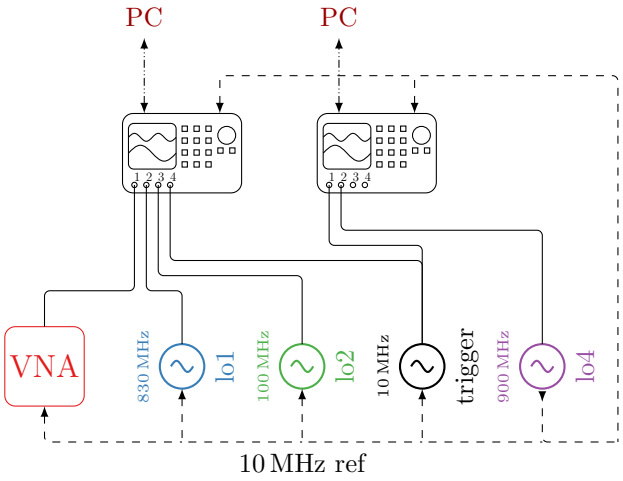



Figure 5.7: Phase drift measurement setup

The measurement results, which can be seen in figure 5.8, show a trend of the phase drift of every instrument. The phase drift is dominantly caused by the low reference frequency of 10 MHz. During a single period of the reference signal, a lot of periods need to be synthesized by the instruments, allowing for few synchronisation points. This means that a better result can be achieved by using a faster reference clock. This is also reflected in figure 5.8, which shows that the generators with the higher frequencies (*lo1*, *lo4*, VNA) experience more phase drift than the slower ones (*lo2*). This is caused by the fact that higher frequencies have shorter signal periods and, therefore, small time intervals cover bigger parts of a signal period as would be the case with low frequencies.

A further component is temperature drift, since the room temperature of the laboratory was not held constant. For these reasons the measurement results in figure 5.8 are only exemplary and are not accurate, since the trigger source *trigger* and the two oscilloscopes also experience phase drift. It is impossible to remove those measurement errors from the results.

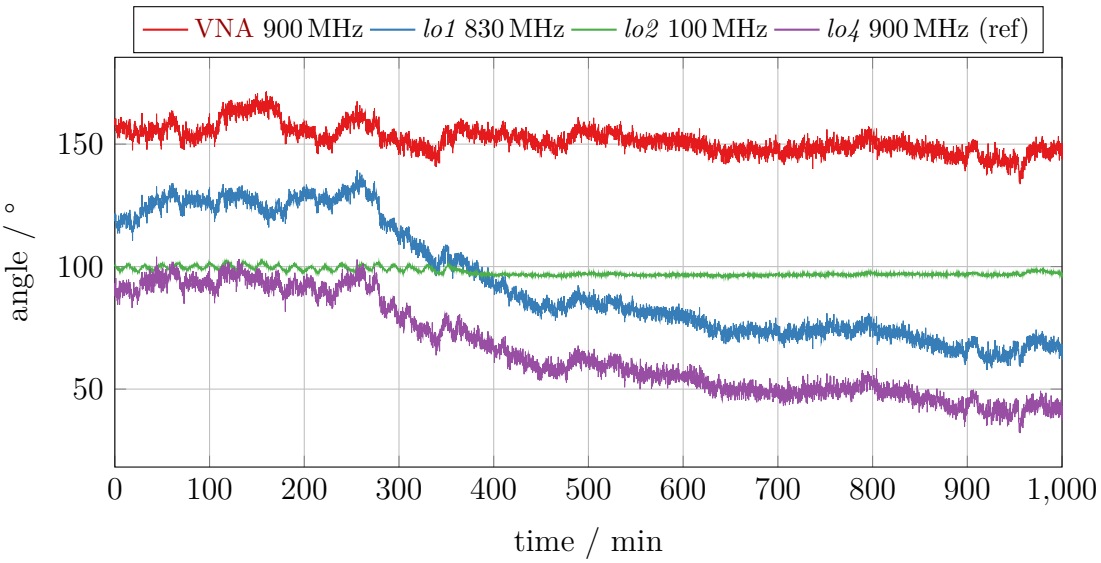



Figure 5.8: Phase drift of signal generators over time

The above mentioned phase drift needs to be kept in mind during measurements. This drift causes additional phase errors in the resulting $\Gamma_L$ and, depending on the accuracy needed of the measurements, limits the valid time of the calibration of the different parts of the measurement setup.

# 6 Conclusions and Outlook

Within this thesis an active FPGA based load-pull measurement setup capable of synthesizing an almost constant reflection coefficient over a wide bandwidth was realized. This is required, for instance, to characterize PAs used in modern communication systems. Furthermore, the realized setup allows using modulated signals which enables testing PAs with modulated signals as they are found in high speed communication applications [3]. Additionally, the presented active load-pull measurement system is capable of providing highly reflective environments with reflection coefficients $|\Gamma_L| = 1$ which is needed for e.g. class F PAs [3].

These objectives were achieved by implementing the reflection generation digitally in baseband. Additionally, the system uses digital IQ demodulation instead of a direct conversion approach which also allows using the frequency band around 0 Hz and minimizes IQ imbalance. Performing the reflection synthesis digitally allowed the use of a digital filter with fully configurable frequency response which would have been impossible in the analog domain. This filter allows, for instance, compensating the frequency response or the group delay introduced by the cabling needed in the setup. As has been presented in section 5.2.2, this filter allows synthesizing almost constant $\Gamma_L$ values over approximately a bandwidth of 20 MHz. Additionally, this filter helps to mitigate oscillations at frequencies outside the band of interest.

Since the components of the measurement setup can cause additional reflections and highly non-linear DUT can experience load-dependent S-parameters, an iterative target algorithm was developed. Therefore, an additional one port VNA was integrated into the setup in order to allow the target algorithm to determine the current $\Gamma_L$. As has been verified in section 5.2.1, this algorithm is capable of reaching specific $\Gamma_{L,target}$ values within the accuracy of this measurement system. Furthermore, given a calibrated one port VNA, this target algorithm needs no further calibration allowing for faster setup times.

The implementation of the digital hardware used within this thesis provides easy to use communication interfaces. For automated test bench setups a text based interface operating via TCP was developed. Accompanying Matlab classes allow controlling the complete digital part from within Matlab. Additionally, a web based interface using a websocket based protocol was designed for manual control and

status feedback.

This work can be used as a base to build a harmonic load-pull setup capable of synthesizing an almost constant reflection coefficient over a wide bandwidth. This could be achieved, for instance, using triplexers for splitting and combining the signal into the frequency components and three load-pull modules [45]. Each consisting of down-mixing, an ADC, an FPGA, and a vector signal generator. Many of the components of the digital signal processing chain could be shared between those modules. Nevertheless, extending the implementation to such a setup will need a larger FPGA containing more memory.

For future work on the realized load pull setup several improvements or extensions can be thought of. For instance, the duration which was needed to reach a specified $\Gamma_L$ by the target algorithm can be accelerated. Furthermore, using for instance RF signal sources that are capable of using reference signals with a higher frequency or direct RF synchronization seems promising. This would, for example, minimize the phase drift which has been observed during the verification measurements (see section 5.3), minimize noise, or allow for more easily achieving a synchronous active broadband load-pull measurement system capable of directly handling modern high bandwidth communication standards.

# A Sources and Documentation

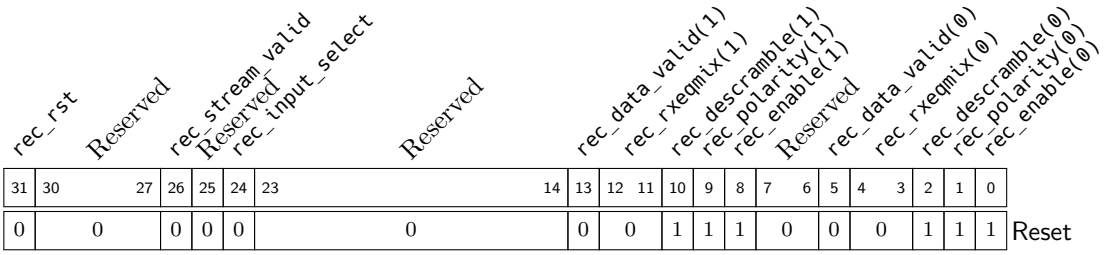
## A.1 Source Codes

The VHDL sources and project files needed to generate the hardware can be found at https://github.com/notti/load_pull-hw. Buildroot configuration files for building the OS necessary for the integrated processor design can be found at https://github.com/notti/load_pull-buildroot. The kernel module necessary for controlling the developed hardware, web interface and protocol server, Matlab source codes, and raw verification results can be found at https://github.com/notti/load_pull-sw. The complete LuaLaTeX source code and processed verification results necessary to build this document can be found at https://github.com/notti/load_pull.

## A.2 Register Assignment and Protocol Reference

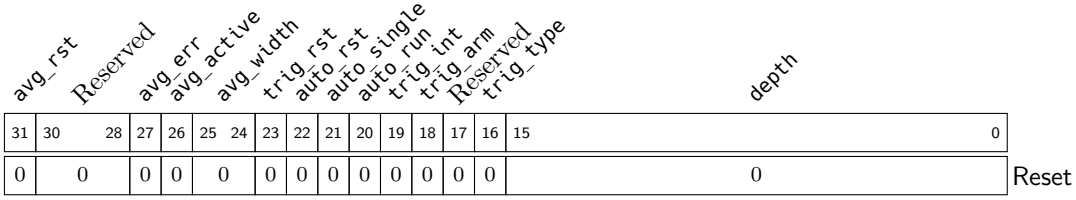
The following list of registers includes every signal exposed by the *main* module which can be used to control the complete digital signal processing chain. Additionally, the path names of the *sysfs* interface implemented by the kernel module are listed (see section 4.1).

The Matlab source code repository (see appendix A.1) contains a package named `ML507` which in turn contains the class `ML507.ML507`. This class can be used to control the hardware implementation. If not otherwise noted, the Matlab property is the same as the path names listed in the following tables with a dot instead of a slash. For instance $L$ can be set to 2,000 with the Matlab command `instance.core.L = 2000;` where instance is an instance of `ML507.ML507`.




| Signal name | Path name | r/w | Description |
|---|---|---|---|
| rec_rst | receiver/rst | w | Reset receiver. |
| rec_stream_valid | receiver/stream_valid | r | Indicates a valid data stream. |
| rec_input_select | receiver/input_select | rw | Select active receiver. |
| rec_data_valid(n) | gtx(n)/rec_data_valid | r | Indicates a valid data stream. |
| rec_rxeqmix(n) | gtx(n)/rxeqmix | rw | Controls equalizer of receiver [32]. |
| rec_descramble(n) | gtx(n)/descramble | rw | Enables descrambler (see section 3.1.1). |
| rec_polarity(n) | gtx(n)/polarity | rw | Controls polarity of LVDS pair [32]. |
| rec_enable(n) | gtx(n)/enable | rw | Enables transceiver. |

Figure A.1: Register 0 (0x00)

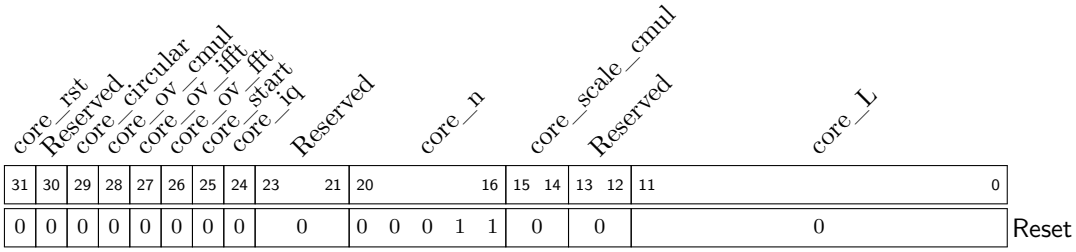



| Signal name | Path name | r/w | Description |
|---|---|---|---|
| avg_rst | average/rst | w | Reset *average_mem*. |
| avg_err | average/err | r | Indicates error during last acquisition. |
| avg_active | average/active | r | Indicates active data acquisition. |
| avg_width | average/width | rw | Number of samples for averaging. |
| auto_rst | auto/rst | w | Reset *auto*. |
| auto_single | auto/single | w | Execute single *auto* cycle (see section 3.1.4). |
| auto_run | auto/run | rw | Enable automatic mode (see section 3.1.4). |
| trig_rst | trigger/rst | w | Reset *trigger*. |
| trig_int | trigger/int | w | Manually trigger. |
| trig_arm | trigger/arm | rw | Arm trigger. |
| trig_type | trigger/type | rw | 0: Internal trigger. 1: External trigger. |
| depth | depth | rw | Number of samples to acquire (1–49,152). |

Figure A.2: Register 1 (0x04)

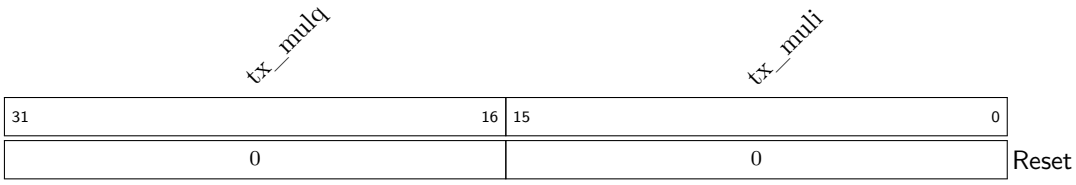

| Signal name | Path name | r/w | Description |
|---|---|---|---|
| core_scale_sch | core/scale_sch(n) | rw | Scaling Schedule for FFT (see |
| core_scale_schi | core/scale_schi(n) | rw | section 3.1.2) |

Figure A.3: Register 2 (0x08)



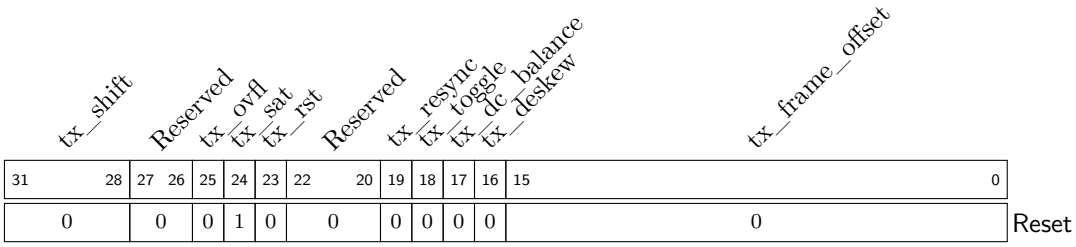

| Signal name | Path name | r/w | Description |
|---|---|---|---|
| core_rst | core/rst | w | Reset *core*. |
| core_circular | core/circular | rw | Enable circular convolution. |
| core_ov_cmul | core/ov_cmul | r | Indicates complex multiplication overflow. |
| core_ov_ifft | core/ov_ifft | r | Indicates inverse FFT overflow. |
| core_ov_fft | core/ov_fft | r | Indicates FFT overflow. |
| core_start | core/start | rw | Start filter execution. |
| core_iq | core/iq | rw | Enable IQ demodulation. |
| core_n | core/n | rw | Transform size in $\log_2(n_{\text{fft}})$ (3–12). |
| core_scale_cmul | core/scale_cmul | rw | Scaling schedule for complex multiplication. |
| core_L | core/L | rw | $L$. See section 3.1.2. |

Figure A.4: Register 3 (0x0C)



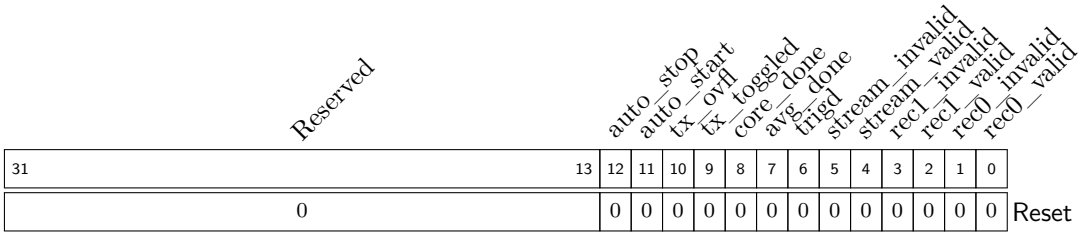

| Signal name | Path name | | r/w | Description |
|---|---|---|---|---|
| tx_muli | transmitter/muli | transmitter.mul | rw | $\Gamma_{L,set}$ |
| tx_mulq | transmitter/mulq | | rw | |

Figure A.5: Register 4 (0x10)




| Signal name | Path name | r/w | Description |
|---|---|---|---|
| tx_shift | transmitter/shift | rw | Scaling schedule output multiplier (0–15). |
| tx_ovfl | transmitter/ovfl | r | Indicates output multiplier overflow. Write a 0 to reset overflow. |
| tx_sat | transmitter/sat | rw | Enable saturation for complex multiplier. |
| tx_rst | transmitter/rst | w | Reset *outbuf*. |
| tx_resync | transmitter/resync | w | Resynchronize sample buffer to input buffer. |
| tx_toggle | transmitter/toggle | w | Toggle output buffers. |
| tx_dc_balance | transmitter/dc_balance | rw | Enable DC balance for output [27]. |
| tx_deskew | transmitter/deskew | w | Start deskew cycle [27]. |
| tx_frame_offset | transmitter/frame_offset | rw | Frame offset. |

Figure A.6: Register 5 (0x14)




| Signal name | Path name | Description |
|---|---|---|
| auto_stop | intr/auto_stop | Automatic mode stopped. |
| auto_start | intr/auto_start | Automatic mode started. |
| tx_ovfl | intr/tx_ovfl | Transmitter overflow. |
| tx_toggled | intr/tx_toggled | Output buffer toggled. |
| core_done | intr/core_done | Filtering done. |
| avg_done | intr/avg_done | Averaging done. |
| trigd | intr/trigd | Triggered. |
| stream_invalid | intr/stream_invalid | Stream data invalid (ADC disconnected). |
| stream_valid | intr/stream_valid | Stream data valid (ADC connected). |
| rec(n)_invalid | intr/rec(n)_invalid | Receiver n data invalid (ADC disconnected). |
| rec(n)_valid | intr/rec(n)_valid | Receiver n data valid (ADC connected). |

Figure A.7: Interrupt Register (0x220)

# B Build Instructions

## B.1 Hardware

A reference to the project files needed to build the netlist and FPGA configuration files can be found in appendix A.1. For compiling the project files, the Xilinx ISE Design Suite version 14.7 including all updates is needed. Since this design has very strict timing conditions, the design needs to be built using the SmartXplorer iterating different cost tables.

## B.2 Software

To build the complete software, the buildroot-add repository as specified in appendix A.1 and buildroot version 2015.02 [36] need to be downloaded. Furthermore, all the prerequisites listed at [36] need to be installed. After downloading buildroot-add and buildroot, both need to be extracted into a temporary directory. Next the following commands need to be executed from within the buildroot directory:

```
make BR2_EXTERNAL=/path/to/buildroot-add ml507_defconfig
make
```

These commands build the complete image including all the necessary software. The target image can be found in the directory `output/images/`. The kernel image called `simpleImage.virtex440-final.elf` needed for booting the processor is in the folder `output/build/linux-3.18.6/arch/powerpc/boot/`.

# C Used Equipment

| | |
|---|---|
| Directional coupler *dir1* | Krytar Model 1850 |
| Directional coupler *dir2* | Krytar Model 1850 |
| Attenuator *att1* | 10 dB Mini-Circuits VAT-10W2+ |
| Frequency mixer *mix1* | Hittite HMC208MS8E |
| Low pass filter *lp2* | Mini-Circuits SLP-90+ |
| ADC *adc1* | Linear Technology 1151A-D Eval Board (LTC2274) |
| FPGA | Xilinx ML507 Eval Board (XC5VFX70T) |
| Vector signal generator | Rohde & Schwarz SMBV100A |
| LO *lo1* | Rohde & Schwarz SMIQ 06B |
| LO *lo2* | Rohde & Schwarz SMGU |
| Oscilloscope | Agilent Technologies MSO7104A |
| 2. Oscilloscope | Agilent Technologies MSO7104A |
| Trigger | Hewlett Packard 33120A |
| Stub tuner | Maury Microwave 1819B |
| Calibration kit | Rohde & Schwarz ZV-Z132 (female) |
| VNA (DUT) | Agilent Technologies E8364A |
| calibrated with | Agilent Technologies N4433A |

# Bibliography


[1] D. M. Pozar, *Microwave Engineering*. Wiley, 2011, vol. 4.

[2] *S-Parameter Design*, Agilent, 2006, AN 154. [Online]. Available: http://cp.literature.agilent.com/litweb/pdf/5952-1087.pdf

[3] F. M. Ghannouchi and M. S. Hashmi, *Load-Pull Techniques with Applications to Power Amplifier Design*, ser. Springer Series in Advanced Microelectronics. Dordrecht: Springer Netherlands, 2013, vol. 32.

[4] M. Hashmi, F. Ghannouchi, P. Tasker, and K. Rawat, "Highly Reflective Load-Pull," *IEEE Microwave Magazine*, vol. 12, no. 4, pp. 96–107, Jun. 2011.

[5] F. De Groote, J.-P. Teyssier, T. Gasseling, O. Jardel, and J. Verspecht, "Introduction to measurements for power transistor characterization," *IEEE Microwave Magazine*, vol. 9, no. 3, pp. 70–85, Jun. 2008.

[6] J.-E. Muller and B. Gyselinckx, "Comparison of active versus passive on-wafer load-pull characterisation of microwave and mm-wave power devices," in *Microwave Symposium Digest, 1994., IEEE MTT-S International*, vol. 2, May 1994, pp. 1077–1080.

[7] T. Williams, J. Benedikt, and P. Tasker, "Experimental evaluation of an active envelope load pull architecture for high speed device characterization," in *Microwave Symposium Digest, 2005 IEEE MTT-S International*, Jun. 2005, pp. 1509–1512.

[8] M. S. Hashmi, A. L. Clarke, J. Lees, M. Helaoui, P. J. Tasker, and F. M. Ghannouchi, "Agile harmonic envelope load-pull system enabling reliable and rapid device characterization," *Meas. Sci. Technol.*, vol. 21, no. 5, May 2010. [Online]. Available: http://iopscience.iop.org/0957-0233/21/5/055109

[9] S. Hashim, M. Hashmi, T. Williams, S. Woodington, J. Benedikt, and P. Tasker, "Active Envelope Load-Pull for Wideband Multi-tone Stimulus Incorporating Delay Compensation," in *Microwave Conference EuMC*, Oct. 2008, pp. 317–320.

[10] F. Harris, "On the use of windows for harmonic analysis with the discrete Fourier transform," *Proceedings of the IEEE*, vol. 66, no. 1, pp. 51–83, Jan. 1978.

[11] G. Heinzel, A. Rüdiger, and R. Schilling, "Spectrum and spectral density estimation by the Discrete Fourier transform (DFT), including a comprehensive list of window functions and some new at-top windows," Max-Planck-Institut für Gravitationsphysik, Hannover, 2002, internal report.

[12] *Matlab documentation - flattopwin*, Mathworks. [Online]. Available: http://de.mathworks.com/help/signal/ref/flattopwin.html

[13] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*. Prentice Hall, 2009, vol. 3.

[14] *Applying Error Correction to Network Analyzer Measurements*, Agilent, 2002, AN 1287-3. [Online]. Available: http://cp.literature.agilent.com/litweb/pdf/5965-7709E.pdf

[15] *Understanding VNA Calibration*, Anritsu, 2012. [Online]. Available: http://anlage.umd.edu/Anritsu_understanding-vna-calibration.pdf

[16] E. P. J. Tozer, *Broadcast Engineer's Reference Book*. Focal Pr, 2004.

[17] S. V. Ahamed and V. B. Lawrence, *Design and Engineering of Intelligent Communication Systems*, 1997th ed. Springer, Feb. 1997.

[18] J. C. Whitaker, Ed., *The RF Transmission Systems Handbook*, 1st ed. CRC Press, May 2002.

[19] J. T. J. Penttinen, *The Telecommunications Handbook: Engineering Guidelines for Fixed, Mobile and Satellite Systems*, 1st ed. Wiley, Mar. 2015.

[20] *PSA Series Spectrum Analyzers Option H70, 70 MHz IF Output*, Agilent, 2003, Technical Overview.

[21] "LTC2274 16-Bit, 105Msps Serial Output ADC," Linear Technology, data sheet. [Online]. Available: http://cds.linear.com/docs/en/datasheet/2274fb.pdf

[22] *Serial Interface for Data Converters*, JEDEC Std. JESD204B.01, 2011.

[23] *R&S® SMBV100A Vector Signal Generator*, Rhode & Schwarz, Operating Manual.

[24] W. Kester, *ADC Input Noise: The Good, The Bad, and The Ugly. Is No Noise Good Noise?*, Analog Devices, 2006.

[25] M. Hayes, *Schaums Outline of Digital Signal Processing*, 2nd ed. McGraw-Hill Education, Sep. 2011.

[26] *Option: Digital Baseband Interface, R&S® FSQ-B17*, Rhode & Schwarz, Operating Manual.

[27] "DS90CR485 133MHz 48-bit Channel Link Serializer (6.384 Gbps)," Texas Instruments, data sheet. [Online]. Available: http://www.ti.com/lit/ds/symlink/ds90cr485.pdf

[28] *Virtex-5 Family Overview*, Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf

[29] *ML505/ML506/ML507 Evaluation Platform User Guide*, Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf

[30] *Virtex-5 FPGA User Guide*, Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug190.pdf

[31] *LogiCORE IP Virtex-5 FPGA RocketIO GTX Transceiver Wizard v1.7*, Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/v5_gtxwizard_ds601.pdf

[32] *Virtex-5 FPGA RocketIO GTX Transceiver User Guide*, Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug198.pdf

[33] *LogiCORE IP Fast Fourier Transform v7.1*, Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/xfft_ds260.pdf

[34] *Embedded Processor Block in Virtex-5 FPGAs Reference Guide*, Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug200.pdf

[35] *PLBv46 Slave Burst (v1.01a)*, Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/plbv46_slave_burst.pdf

[36] Buildroot - Making Embedded Linux Easy (version 2015.02). [Online]. Available: http://buildroot.org/

[37] J. Corbet, A. Rubini, and G. K. Hartman, *Linux Device Drivers*, 3rd ed. O'Reilly, 2005.

[38] J. Corbet, "Platform devices and device trees," *LWN*, June 2011. [Online]. Available: https://lwn.net/Articles/448502/

[39] P. Mochel, "The sysfs Filesystem," in *Proceedings of the Linux Symposium*, ser. Linux Symposium 2005, vol. 1. Linux symposium, 2005, pp. 313–326.

[40] *Portable Operating System Interface (POSIX®) Base Specifications*, IEEE Std. 1003.1, Rev. 7, 2013.

[41] Twisted (Version 14.0.2). [Online]. Available: https://twistedmatrix.com/trac/

[42] "R&S®ZV-Z121/ R&S®ZV-Z132 Calibration Kits," Rhode & Schwarz, data sheet.

[43] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-time Signal Processing*. Prentice-Hall, 1999, vol. 2.

[44] *Matlab documentation - pause*, Mathworks. [Online]. Available: http://de.mathworks.com/help/matlab/ref/pause.html

[45] M. Hashmi, A. Clarke, S. Woodington, J. Lees, J. Benedikt, and P. Tasker, "An Accurate Calibrate-Able Multiharmonic Active Load-Pull System Based on the Envelope Load-Pull Concept," *IEEE Transactions on Microwave Theory and Techniques*, vol. 58, no. 3, pp. 656–664, Mar. 2010.

# Code of Conduct

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

| | |
|---|---|
| Unterschrift | Datum |