Peter Uran

Design of an FPGA-based Data Acquisition System for a Shore-based Maritime Radar Network

Master's thesis in Electronic Systems Design and Innovation
Supervisor: Egil Eide
Co-supervisor: Zolve AS
July 2021

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Peter Uran

Design of an FPGA-based Data Acquisition System for a Shore-based Maritime Radar Network

Master's thesis in Electronic Systems Design and Innovation
Supervisor: Egil Eide
Co-supervisor: Zolve AS
July 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Abstract

A shore-based network of maritime radars is to be developed by NTNU and Sintef as part of the Ocean Lab Node 2 infrastructure. The purpose of the network is to track both autonomous and traditional vessels in the Trondheim fjord and contribute to a shared situational awareness. This report presents a system architecture for the remote radar sites, based on sampling the radar data on-site and transmitting the data via the emerging 5G network in Trondheim to minimize the cost of infrastructure. A data acquisition system is developed based on the Analog Devices AD-FMCDAQ2 ADC and an iWave Intel Arria10 SoC/FPGA FMC+ Development Kit. A Simrad Halo 20+ pulse compression radar is currently considered for the system. The radar waveforms are measured and revealed to transmit both pulses and chirps with varying center frequencies and bandwidths. The sweep bandwidth is measured to a maximum of 30 MHz, which is expected to result in 1.0 Gbit/s of uplink data. The data acquisition system is shown to successfully perform IQ demodulation, clock domain crossing and pulse compression in simulation. Pulse compression is achieved by dynamically creating a matched filter from the transmitted chirp to account for the alternating center frequency and bandwidth of the chirp. The required filter length was not achieved due to over-utilization of the FPGA multipliers, to which multiple solutions are discussed. A driver issue prevented the testing of the dechirping in hardware. The project also makes successful use of a continuous integration workflow for simulating the FPGA design with the VUnit test framework and the open source GHDL simulator. It is also demonstrated how VUnit can be used to verify the signal processing chain by generating and validating simulation data in Python, rather than pure VHDL.

Sammendrag

Et landbasert nettverk av maritime radarer skal utvikles av NTNU og Sintef som en del av Ocean Lab Node 2-infrastrukturen. Hensikten med nettverket er å spore både autonome og tradisjonelle fartøy i Trondheimsfjorden for å bidra til en delt situasjonsbevissthet. Denne rapporten presenterer en systemarkitektur for eksterne radarsteder, basert på sampling av radardata på stedet og opplasting via det nye 5G-nettverket i Trondheim for å minimere infrastrukturkostnadene. Et datainnsamlingssystem er utviklet basert på Analog Devices AD-FMCDAQ2 ADC og et iWave Intel Arria10 SoC / FPGA FMC+ utviklingskort. En Simrad Halo 20+ pulskompresjonsradar er tiltenkt for systemet som del av prosjektet. Radarbølgeformene måles og viser seg å overføre både pulser og *chirps* med varierende senterfrekvenser og båndbredde. Båndbredden for en *chirp* måles til maksimalt 30 MHz, som forventes å resultere i 1.0 Gbit/s opplinkingsdata. Datainnsamlingssystemet er vist å utføre IQ-demodulering, klokkedomenekryssing og pulskompresjon i simulering. Pulskompresjon oppnås ved dynamisk å lage et *matchet* filter fra den overførte *chirp*-en for å ta hensyn til den alternerende senterfrekvensen og båndbredden til *chirpen*. Den nødvendige filterlengden ble ikke oppnådd på grunn av overutnyttelse av FPGA-multiplikatorene, som flere løsninger blir diskutert til. Et driverproblem forhindret testing av *dechirping* i maskinvare. Prosjektet bruker også *continuous integration* for å simulere FPGA-design med VUnit-testrammeverket og GHDL-simulatoren med åpen kildekode. Det er også demonstrert hvordan VUnit kan brukes til å verifisere signalbehandlingskjeden ved å generere og validere simuleringsdata i Python, fremfor bruk av ren VHDL.

Acknowledgements

First and foremost I want to thank my colleagues in Zolve for their support. While the FPGA-world is fascinating, it is certainly not traveled alone. I appreciate all the experience and advice you have been sharing with me the last year. One should also not underestimate the value of engineers discussing problems by the coffe machine. There are few other companies with such a joyous and nurturing culture as Zolve, which is deeply appreciated. I very much look forward to start working full time in Zolve this autumn.

I also want to thank my supervisor Egil Eide for sharing his expertise and giving valuable feedback throughout the project. As both an outstanding source of radar as well as humor, our meetings have been something to look forward to. While your are an entrepreneur with many ambitious projects, I hope you continue to spend time on educating the next generation of engineers. Your lectures were much appreciated.

Last but not least, a special thanks goes to Lars Lundheim for taking it upon himself to transform the way we teach electronics to university students. You have been a good source of motivation and knowledge even years after I had you as a lecturer. Elsys would never have existed without you, and I hope you continue to teach for years to come.

Abbreviations

Abbreviation	**Definition**
BFM	Bus Functional Module. Software model applying stimuli to a bus under simulation.
COTS	Commercially available off-the-shelf
DSP slice	Special logic block providing arithmetic for large numbers
DUT	Device Under Test
Entity	VHDL language construct defining a module, its ports and content
Fabric	Reconfigurable logic part of the FPGA silicon
FIFO	First In, First-Out. Simple memory structure
FMC HPC	FPGA Mezzanine Card High Pin Count. Also called FMC+
FMC LPC	FPGA Mezzanine Card Low Pin Count. Often denoted as only FMC
FPGA	Field-programmable gate array
GNSS	Global Navigation Satellite System
GNSSDO	GNSS Disciplined Oscillator
HPS	Hard Processor System
IP Core	Intellectual Property core. Ready-to-use entity designed by vendor or third party
Logic block	Basic FPGA building block providing reconfigurable logic
LUT	Lookup table
Process	VHDL language construct providing sequential execution of code
PPS / 1PPS	(1) Pulse Per Second
Quartus	Intel/Altera Quartus Prime FPGA design tool
R/W	Read/Write
SoC	System on Chip. Term describing a FPGA with a HPS
UDP	User Datagram Protocol
UVVM	Universal VHDL Verification Methodology
VHDL	VHSIC-HDL, Very High Speed Integrated Circuit Hardware Description Language

Contents

Chapter 1

Introduction

In 2016, The Norwegian Coastal Administration authorized the Trondheim Fjord as the first testbed for autonomous vessels in the world [2]. These vessels are also known as Maritime Autonomous Surface Ships (MASS). Since then, the fjord has been a prime target for innovation in the transport, fishing and aquaculture industries through the emergence of autonomous vessels [31]. This includes the NTNU research vessel MilliAmpere [34] and its recent commercialization through ZeaBuz [3]. The new autonomous vessels need to co-exist with and safely navigate traditional crafts and recreative vessels, as well as sporting crafts such as sailboats and kayaks. This is imperative to sustain a safe environment, establish trust and preserve the various and diverse interests of all the different groups using the fjord.

As part of the effort to sustain safe transport on the fjord, NTNU and SINTEF wish to develop a radar network to track both autonomous and non-autonomous vessels. The radar network is to be part of the Ocean Lab Node 2 infrastructure [35]. The objective of this infrastructure is to help detect small craft and objects, assist navigation in confined waters and automatic operation in ports, as well as contribute to a shared situational awareness for the vessels. Furthermore, the information can be used in control rooms to monitor the traffic and quickly react to emerging situations. The radar network will also be useful to gather more information for further research on autonomous vessels.

The radar network will consist of several shore based radars stationed at remote sites around the fjord, as shown in Figure 1.1. A data acquisition system on site shall be used to sample and process the radar data. Each site will transmit the raw radar data to a central processing hub using the emerging 5G mobile broadband network in Trondheim [5]. Observing the objects from multiple angles allows for better coverage and high reliability of detection, as well as reduction of sea clutter [19].

Existing solutions, such as the Vessel Traffic System (VTS), are mostly based on professional personnel supervising and hailing vessels to regulate traffic and ensure safe passage. Autonomous vessels pose a challenge as they cannot be contacted as conventional piloted vessels. At the same time, they provide both the potential to resolve situations themselves as well as do it faster and more accurately than human personnel could achieve. This requires an accurate shared awareness between the autonomous vessels. Other research projects are also addressing this problem, albeit with slightly different solutions. The University of Florence develop a similar system, albeit based on AIS and shared data from ship-borne radars [37]. Kongsberg Maritime AS also develop several systems for situational awareness based on radar, cameras and AIS [28].

A secondary objective of the project is to offer users of recreative vessels with a real-time map of the various vessels in the fjord. This service is dubbed "Augmented AIS". This service is largely targeted at small recreational vessels with onboard radar. These vessels are susceptive to tilting during wind and high waves, potentially blinding the radar. The Augmented AIS, therefore, has a comparative advantage to the on-board radar, in addition to avoiding the investment of a radar. There should be less than a one-second delay from the radar detecting a vessel to the end-user being updated.

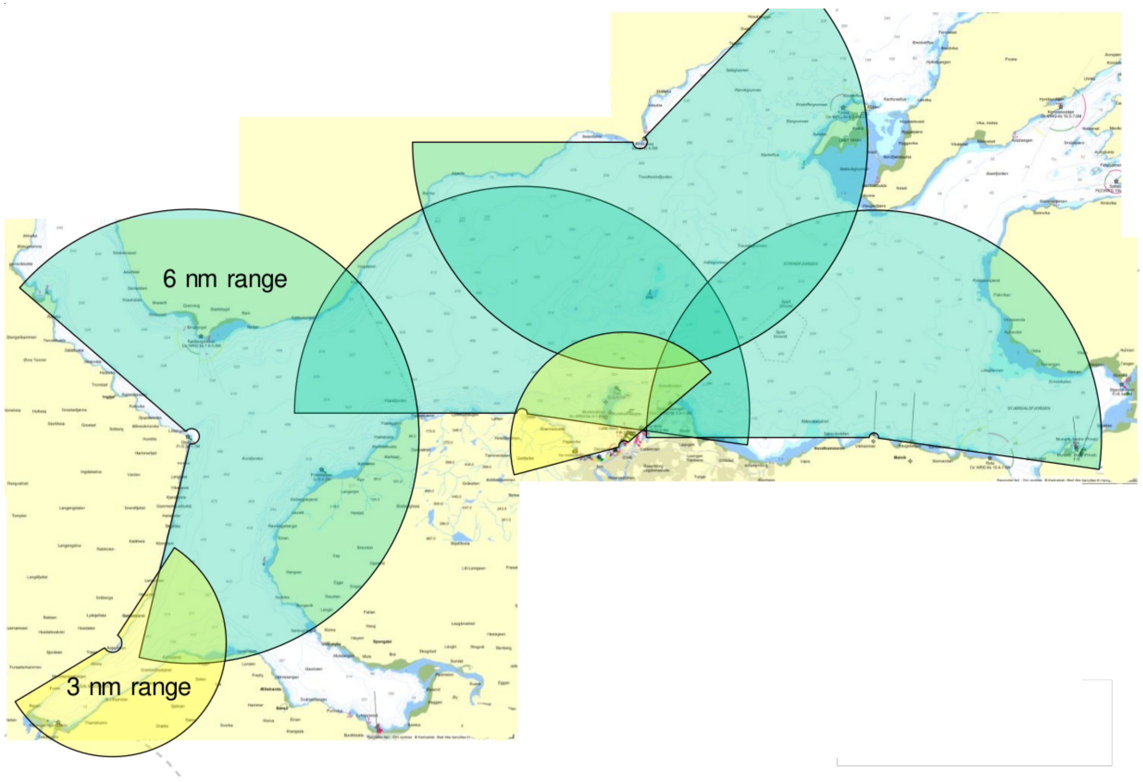


Figure 1.1: Overview image of the fjord showing potential radar sites and coverage [19].

The scope of this project is to explore a possible system architecture and its validity for the projects remote site systems. A proof of concept for the on-site data acquisition system is to be developed, in addition to the hardware selection for this system.

Chapter 2

Radar characteristics

This chapter presents the Simrad Halo 20+ pulse compression radar currently considered for the project. Measurements to characterize the radar waveforms are also presented, as well as some background material for radar systems.

2.1 Radar background

The following section aims to provide a short introduction to radar systems and important terms therein, such as IQ samples, complex signals and pulse compression.

Radar system introduction

A typical radar transmitter and receiver is shown in Figure 2.1. An oscillator generates a radio frequency (RF) signal which is transmitted by an antenna. The resulting radio wave will propagate until it hits a target. This will cause a fraction of the energy to be transmitted back to the radar. The received signal will be very weak and must first be amplified before it is mixed down to an intermediate frequency (IF). It this stage, a detector can be used to determine whether a target is present or not. Alternatively, the radar signal can be sampled by an analog to digital converter (ADC) as Figure 2.1 suggest. This allows more complicated signal processing to be performed digitally using a signal processor.

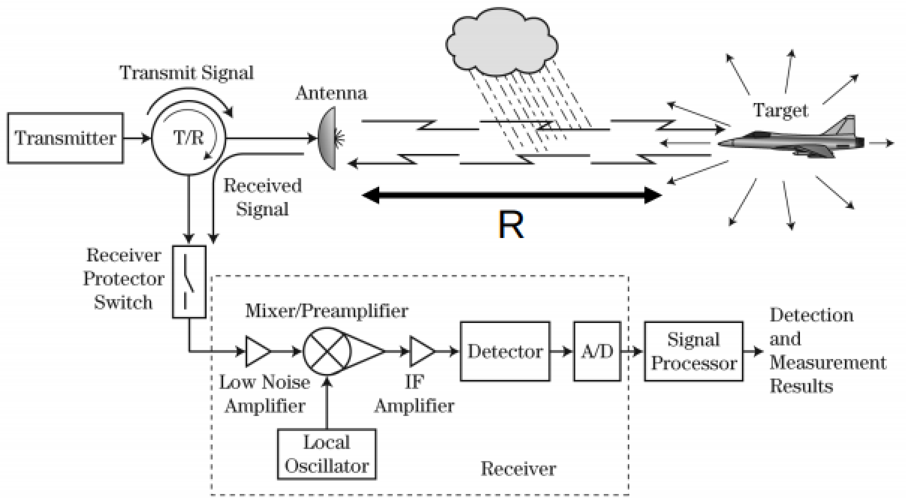


Figure 2.1: Typical radar system showing the major elements of the transmit (TX) and receive (RX) processes [33].

IQ samples and complex signals

When sampling radar signals it is often desired to measure phase changes between transmit and receive. This is referred to as a coherent receiver and is useful for measuring Doppler shifts [33]. Coherent sampling is achieved by sampling the signal twice with a 90° phase difference in between. These are called the in-phase (I) and in quadrature (Q) samples, respectively, which together make up an IQ-sample. The signal can then be represented as a complex-valued signal, where the I-channel is real and the Q-channel is imaginary. For instance, $x_{IQ}(t)$ is a complex signal represented by its I- and Q-components,

$$x_{IQ}(t) = x_I(t) + j \cdot x_Q(t), \tag{2.1}$$

where j is the imaginary unit.

Figure 2.2 illustrates this concept in practice, where a real-valued signal $x[n]$ is split into an I- and Q-channel after being sampled by an ADC. This is done by multiplying the sampled signal with a sine and cosine function, which are separated by a 90° phase offset. Note that the IF signal is band-limited to avoid aliasing.

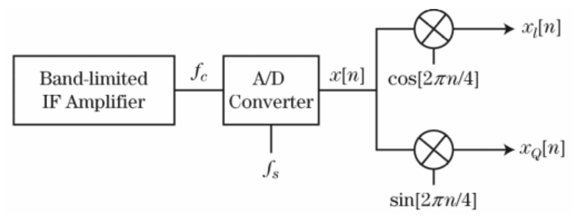


Figure 2.2: Figure illustrating how an IQ-signal is constructed after sampling an IF signal [33].

Pulse compression

A basic pulse radar is based on transmitting short duration pulses to detect targets. The radar can transmit more energy by transmitting longer pulses, which increases the signal strength and improves the signal-to-noise ratio (SNR). Longer pulses will however also decrease the radar's ability to distinguish between targets that are very close. This is called the radars range resolution and is given by

$$\Delta R = c_0 \frac{\tau}{2}, \tag{2.2}$$

where c_0 is the speed of light and τ is the pulse length in seconds. Because the of the inverse relationship between resolution and pulse length, a pulse radar must trade between SNR and range resolution.

This tradeoff can be overcome by using a chirp radar to decouple the pulse length and range resolution. A chirp radar transmits *chirps*, which are frequency modulated signals that increase or decrease in frequency. One such chirp is shown in Figure 2.3, which clearly shows the increasing frequency from 0 Hz to 6 Hz. A radar can create a chirp by modulating the frequency of a sine wave. By using knowledge of the modulation and frequency change, the radar can demodulate the reflected chirp back to a simple pulse. This technique is called pulse compression, which yields a narrow bandwidth pulse to resolve close targets without decreasing the radars SNR.

2.2 Simrad Halo 20+ Radar

A Simrad Halo 20+ Pulse Compression radar is currently considered for the project. While the exact radar to be used in the system is yet to be determined, the Halo 20+ serves as a reference radar for initial development. It can therefore be used to extract design specifications and constraints. Since this is a commercial product, its implementation details and internal workings are largely unknown. Some key specifications are however available from the manual [4], given in Table 2.1.

Frequency and bandwidth specifications are of special interest for sampling the radar signal. Table 2.1 states that the radar operates in the X-band with a center frequency between 9.4 GHz

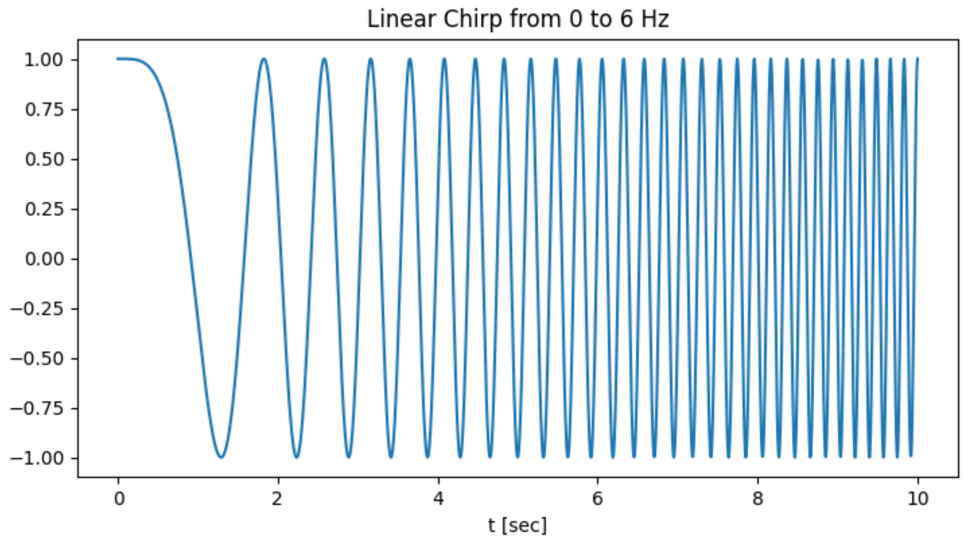


Figure 2.3: Example of a chirp signal increasing in frequency.

Table 2.1: Key radar specifications taken from the Simrad Halo 20+ Radar manual [4].

Range	36	NM	
Rotation speed	20-60	RPM	Dependent on mode and display (MFD)
Frequency	9.4-9.5	GHz	X-band
Transmitter peak power	25	W	
Polarization	Horizontal		
Minimum range	6	m	
Sweep repetition frequency	700-2400	Hz	Mode dependent
Pulse length	0.04-64	us	± 10%
Sweep bandwidth	48	MHz	Max
Horizontal beamwidth	4.9	degrees	TX and RX
Noise figure	5	dB	max

and 9.5 GHz. The radar sweeps over a maximum of 48 MHz, which dictates the sampling frequency. Another Simrad Halo model additionally lists its intermediate frequency (IF) stage as $f_{IF} = 70\,\mathrm{MHz}$ [4]. It can be assumed that this is equal to, or at least close to, the IF frequency of the Halo 20+.

Time of flight

The radar's two way time of flight (TOF) is also of interest to determine how long the delay is between a transmitted and received signal. Figure 1.1 suggests that a radars in this aplication should be operated with a range of 6 nautical miles, or 11 km. The time of flight t is then given by

$$t = \frac{2 \cdot R}{c_0} = \frac{2 \cdot 11.1\,\mathrm{km}}{3 \cdot 10^8\,\mathrm{m/s}} = 74.1\,\mathrm{\mu s}, \tag{2.3}$$

Antenna position

It is necessary to know the antenna position to discern where the radar targets are. While the time of flight determines the distance to the target, the position of the antenna determines the direction of the target. The radar does not explicitly state which communication protocol is used to communicate with the antenna rotator. It is however likely that it is based on NMEA 2000, as the radar already uses this to communicate with external systems [4]. Other common antenna rotator interfaces include RS232 or CAN-bus, or simply as a pulsed signal.

2.3 Waveform characteristics

This section presents the waveform characteristics of the Simrad Halo 20+ radar acquired by measuring the radar signal. The measurements are important to gain further insight into the functionality of the radar, as well as narrowing down constraints regarding bandwidth and pulse lengths. The measurement setup is also presented, along with a brief discussion of the measurement results.

2.3.1 Measurement setup

Both time domain and frequency domain representations of the radar waveforms are of interest for further examination. Table 2.1 lists the radar frequency between 9.4 GHz and 9.5 GHz, with a max sweeps bandwidth of 48 MHz. In order to measure the waveform with good resolution, it should be mixed down to the order of a few 100 MHz using a mixer and a local oscillator (LO) before being measured with an oscilloscope. The proposed measurement setup is shown in Figure 2.4.

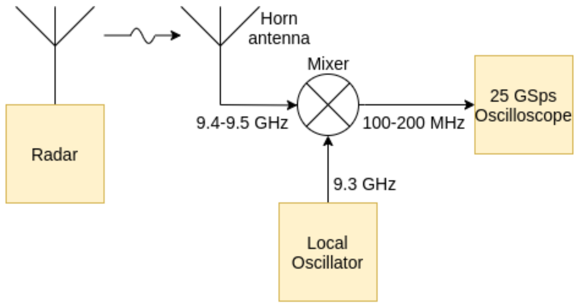


Figure 2.4: Block diagram of the measurement setup for determining the radar waveforms.

The measurement setup is shown in Figure 2.5. An AnaPico APSIN20G (100 kHz-20 GHz) was used as a local oscillator and set to 9.3 GHz. Considering the frequency and bandwidth of the signal, a signal between 100 and 200 MHz was left after mixing with the local oscillator output. The signal was sampled using an Agilent MSO9254A (2.5 GHz, 25 GSaps) set to 4 GSps. This yields an oversampling rate of at least 20, giving good resolution. The radar was set to different range settings and the data was exported from the oscilloscope to create a spectrogram for each setting. To avoid fading due to the rotating antenna, the radar was put on its side as shown in Figure 2.6. The results are shown in table Table 2.2.

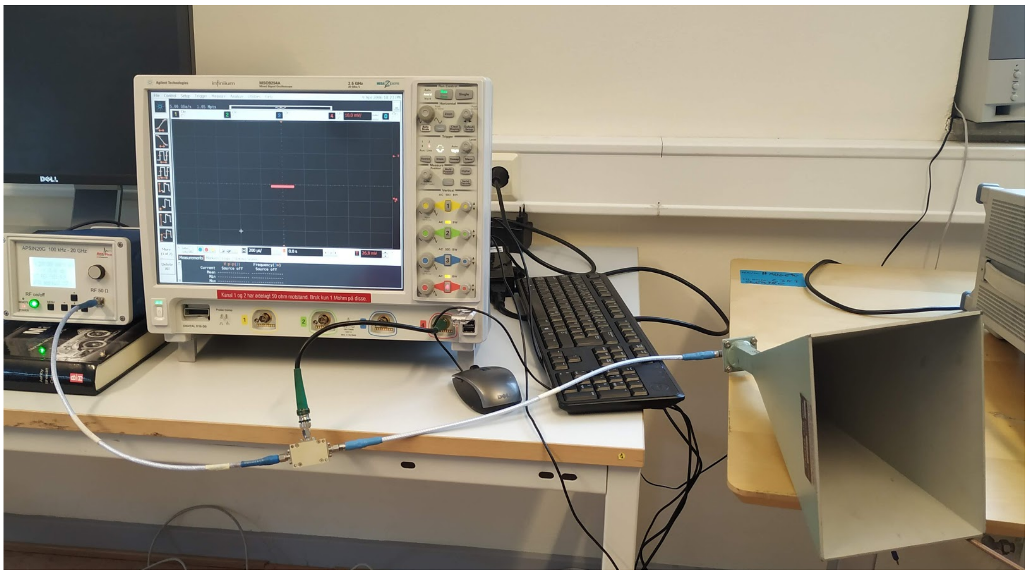


Figure 2.5: Image showing the radar measurement setup with a local oscillator, mixer signal analyzer and a horn antenna.

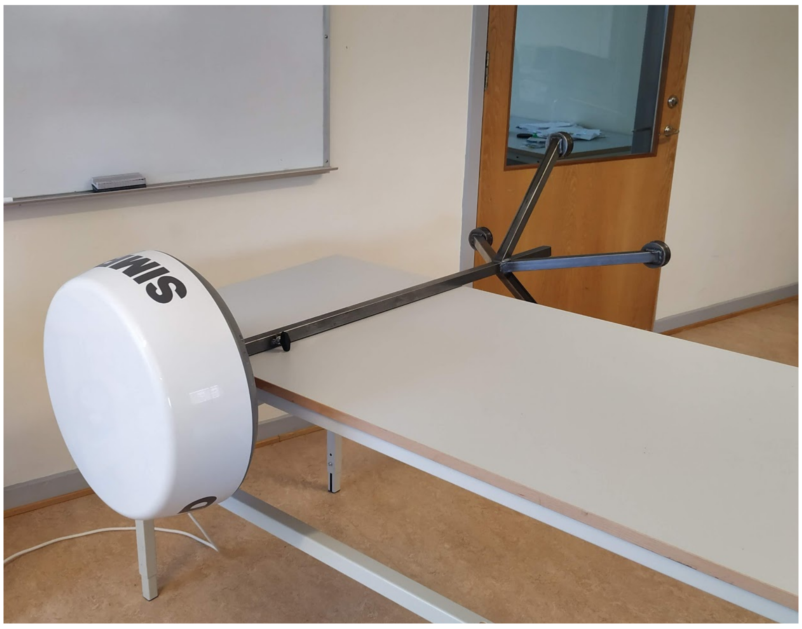


Figure 2.6: Image showing the radar being put on its side to avoid fading due to the rotating antenna.

2.3.2 Measurement results

Table 2.2 shows the measurement results for various range settings of the radar. The radar emits bursts consisting of up two pulses followed by up to three chirps, as shown in Figure 2.7. The exact number of pulses and chirps depends on the radar setting. The time between two bursts is denoted the Burst Repetition Interval, while the offset denotes the time since the first pulse in a burst. Two such bursts are shown in Figure 2.8 for the 0.75 NM setting, consisting of two pulses and a single chirp. Figure 2.9 shows the time domain and spectrogram plots of a single chirp, showing how the chirp increases in frequency.

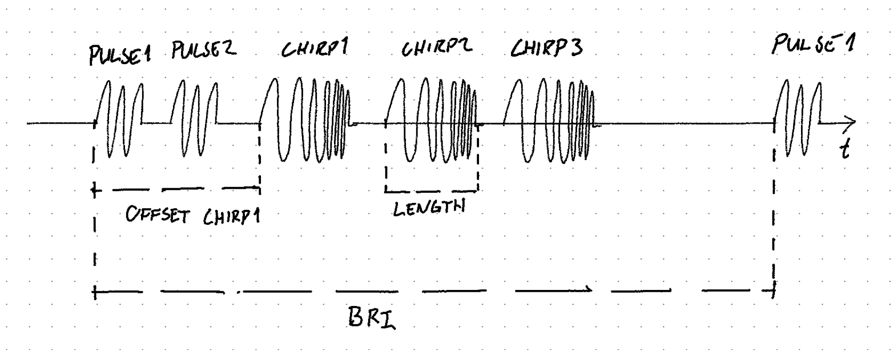


Figure 2.7: Illustration of a radar burst containing two pulses and three chirps. Examples of offset and length, along with burst repetition interval (BRI), are annotated.

The table indicates that the radar emits a single pulse for range settings below 1 NM and two for ranges above that. Likewise, the radar will emit up to three pulses for the longest range. Pulse and chirp center frequencies are listed after downmixing for readability. The true frequencies lie 9.3 GHz above the listed frequencies.

It can be noted that the pulses are of constant frequency and the length independent of the range. The varying frequency of the first pulse is likely attributed to measurement error.

Table 2.2: Radar waveforms for the Simrad Halo 20+ radar. Not all burst repetition intervals (BRI) could be calculated using the measurements.

Radar Range (NM)	**Burst description**									
	Pulse #1			**Pulse #2**			**Chirp #1**			
	f1 (MHz)	**Length(us)**	**Offset (us)**	**f2 (MHz)**	**Length(us)**	**Offset (us)**	**fc (MHz)**	**BW (MHz)**	**Length(us)**	**Offset (us)**
0.125	100	0.1	0							
0.25	90	0.1	0				105	30	1.75	38
0.5	90	0.1	0				105	30	1.76	38
0.75	100	0.1	0				110	30	1.5	38
3.00	90	0.1		110	2		150	20	10	
6.00	80	0.1	0	110	2	40	150	12	8	102
12.00	90	0.1	0	110	2	38	150	15	16	100

	Chirp #2				**Chirp #3**				**BRI (us)**	**Remarks**
	fc (MHz)	**BW (MHz)**	**Length(us)**	**Offset (us)**	**fc (MHz)**	**BW (MHz)**	**Length(us)**	**Offset (us)**		
0.125										Single pulse only
0.25										Single pulse and chirp
0.5										Single pulse and chirp
0.75	150	20	8	39					415	Single pulse and two chirps
3.00	130	10	16							Two pulses and two chirps
6.00	133	8	15	182	18	4	30	300	965	Two pulses and three chirps
12.00	135	20	15	185	19	4	60	300	1020	Two pulses and three chirps

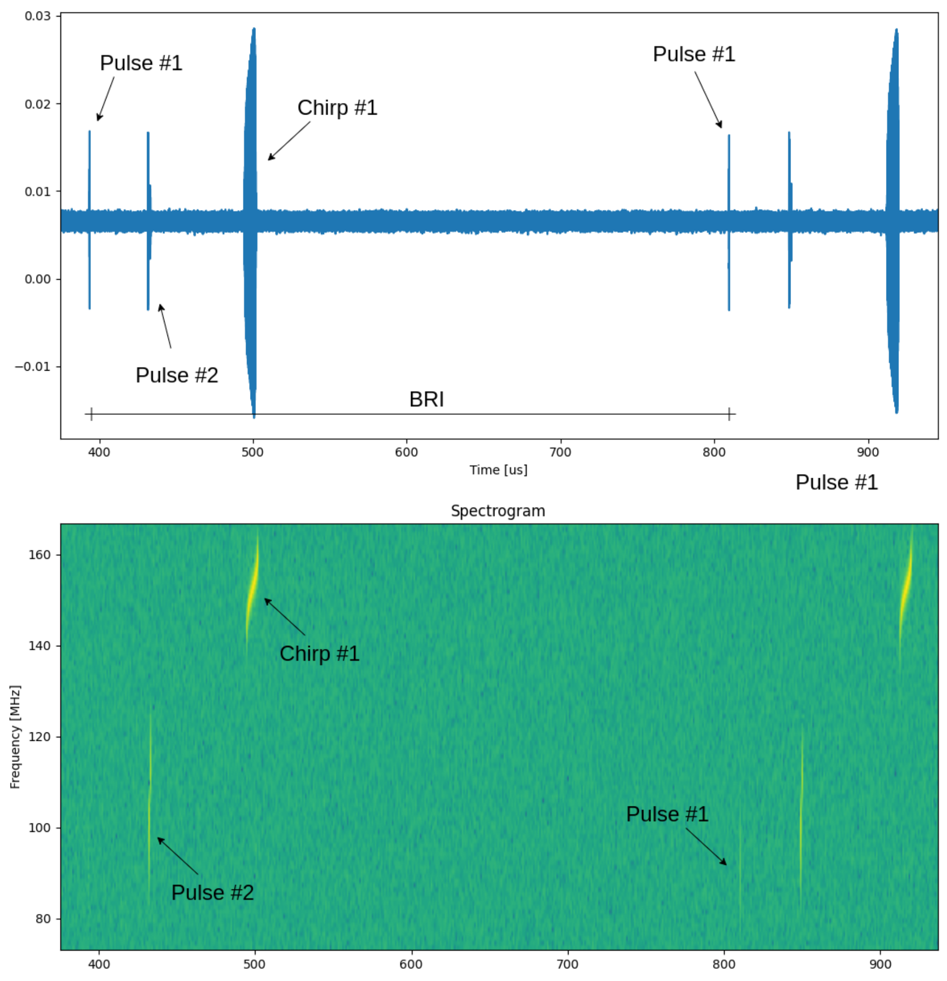


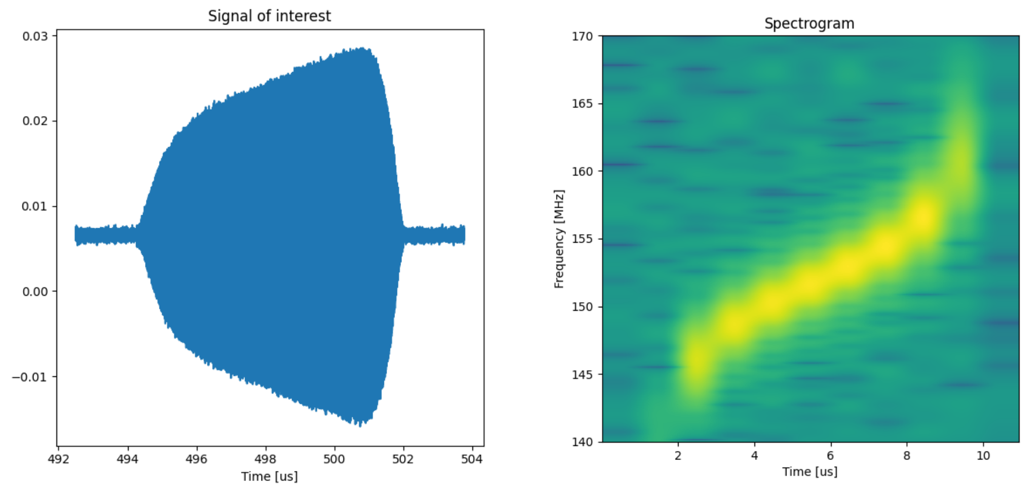
Figure 2.8: Time domain and spectrogram plots of two bursts for the 0.75 NM setting. Two pulses and one chirp can be seen for each burst.

2.3.3 Measurement discussion

The length of the pulses and chirps increases with the range, likely to improve the SNR for long distances. Pulse #2 is 20 times longer than Pulse #1 and only used for the longer ranges. The pulses seem to have a fixed length. Chirp #1 however increases its length for longer ranges, which is also true for Chirp #2. Chirp #3 is only used for ranges above 6 NM and goes up to 60 µs.

It was not possible to measure the burst repetition interval for all settings. The three measurements that were made indicate that the BRI increases with range. The BRI can however be used to give some indications of the radar time budget. It can be seen that the BRI is some ten times longer than the TOF for the various ranges. For instance, for the 12 NM range the BRI is 1020 µs compared to a TOF of only 148 µs. This is likely to add guard time in case the radar signal bounces of a strong reflector beyond the intended range, which would look like a large vessel on a short-range to the radar. The large BRI can also indicate that the radar uses some time to transfer or process the incoming data.

As shown in Table 2.2, the radar switches frequencies for the chirps. This is likely done to solve



(a) Time domain plot **(b)** Spectrogram

Figure 2.9: Time domain and spectrogram plots of a single chirp for the 0.75 NM setting.

some ambiguity functions or make the radar more resistant to interference. Table 2.2 further reveals that the radar only performs up-chirps. That is, the chirps only increase in frequency. There are no down-chirps that decrease in frequency, which is a common way to resolve the ambiguity function for pulse compression systems [33].

Furthermore, the radar transmits one or two pulses before transmitting the chirps. This may have been done to trigger radar beacons which may not work with chirps. The fact that it switches between transmitting one or two pulses depending on the setting, along with the radar lacking down-chirps, suggests that the pulses may be used to solve the radar ambiguity function by creating estimates of position and velocity.

Figure 2.9 also suggests that the chirp may be non-linear, as seen by the rapid changes in frequency near the edges of the chirp. Non-linear chirps are a technique used by radars to suppress sidelobes [33]. New measurements must be performed to confirm the non-linearity of the chirp.

Chapter 3

System architecture

This chapter presents the initial project specifications. Estimates for required sampling frequencies and data rates are presented, based on measurements of radar waveforms from a reference radar. The proposed system architecture is presented, along with hardware selection and a proposed FPGA fabric design.

3.1 System specification

The remote site system shall sample and transmit a stream of IQ-samples from the various radar sites to a central processing hub. The data shall be sampled at the radars IF stage using a data acquisition system, as shown in Figure 3.1. As the reference radar is a pulse compression radar, the data acquisition system must also perform dechirping of the received signal, as explained i Section 2.1. Due to the function of the pulses being currently unknown, only the chirps will be considered at this time. The transfer of data shall be done via a mobile broadband connection to avoid the need for cables. 5G should preferably be used due to strategic partnerships with Telia Norge AS during the roll out of their new infrastructure in Trondheim. As the radar network is part of a safety-related system, there should be emphasis on minimizing latency as well. It is desired that the overall system provides less that a one second delay between receiving the radar data to broadcasting the radar tracks to subscribing systems.

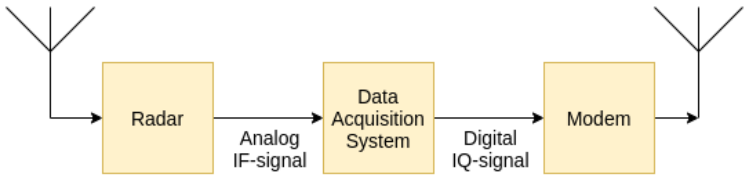


Figure 3.1: Overview of the system.

As the sites are remote, the system shall also have some debug and configuration options that can be accessed remotely.

Since the project is at an early development stage, commercially available off-the-shelf (COTS) parts are preferred to create a minimum viable product (MVP).

The following list summarizes the requirements set for the system at this stage:

- Sample an intermediate frequency (IF) radar signal at 70 MHz.
- Perform dechirping on the reflected radar signal.
- Precisely timestamp IQ-samples for correlation with the data from other sites.
- Transmit data from site to the central processing hub via mobile broadband.
- Offer remote access and debug capabilities for each site.
- Hardware should be COTS

The specs presume a stream of IQ-data are to be transmitted from the radar site. It is therefore assumed that no further processing than the dechirping is desired on-site at this point.

3.2 System proposal

The proposed system for the remote site data acquisition is shown in Figure 3.2.

The data acquisition system is split into an ADC (Analog to Digital Converter) and FPGA (Field-Programmable Gate Array). The analog radar waveform is sampled by an ADC at the radar IF-stage. Both the transmitted and received signals are sampled in order to perform the dechirping, as discussed in Section 3.3. The digitized waveforms are transmitted to an FPGA for processing. An FPGA is chosen as it is well suited for fast, deterministic signal processing at the order of a few 100 MHz. In addition, FPGAs add flexibility due to being re-programmable, so the design can be updated and expanded as needed in the future. The FPGA can also be used to do signal conditioning and preprocessing before it is sent to the central processing hub, to ease the computational load and latency constraints of the latter.

A modem is used to transfer the data to the central processing hub. The User Datagram Protocol (UDP) is preferred for the uplink of data due to its simplicity and low latency. It is also well suited for streaming data in real time systems, as UDP is an unreliable protocol that will drop lost packets rather than request a time consuming re-transmission.

The FPGA must know the exact time of arrival of the radar waveforms. This is necessary to compare data from different sites. While a operating system or processor with a network stack can request the current time trough an internet connection, sub-second precision must be provided trough the use of a GPS (Global Positioning System) disciplined oscillator (GPSDO). The GPSDO uses the accurate clocks in the GPS system as a time source and provides a pulse per second (PPS or 1PPS) signal.

The antenna position must also be transmitted along with the radar data, so the direction of the radar targets can be known. This will however not be considered at this point, due to the radar antenna interface being unknown.

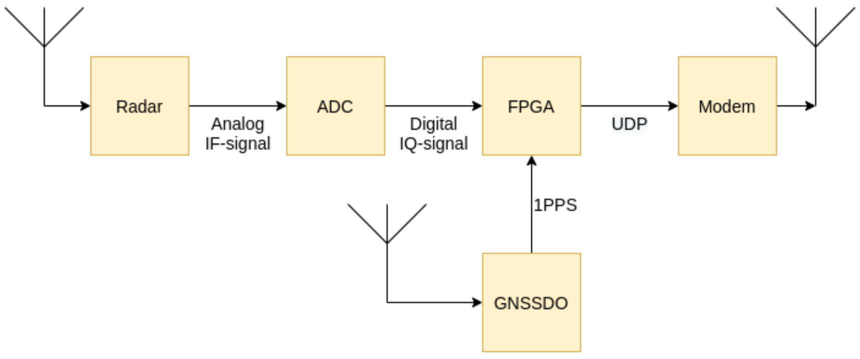


Figure 3.2: Proposed system architecture.

3.3 Proposed FPGA architecture

Figure 3.3 shows the proposed FPGA architecture. SoC (System on Chip) FPGAs, such as the proposed solution, contain a Hard Processing System (HPS) in addition to the usual programmable fabric. The HPS consists of a microprocessor, which can run a Linux-based operating system or be programmed bare metal. In either case, the HPS can be used to implement functionality that is better suited for software, such as using a network stack. The namesake programmable fabric of the FPGA consists of generic logic units, which the compiler uses to implement behaviour specified by a hardware description language (HDL), such as VHDL (Very High Speed Integrated Circuit Hardware Description Language). The fabric additionally contains RAM blocks and special DSP blocks with multipliers for signal processing.

An ADC interface in the FPGA fabric provides a stream of IQ-samples from the ADC. Desired signal processing is performed on the IQ-samples, including dechirping of the radar signal. This is proceeded by a threshold detector to remove the samples between the received signals, in order to reduce the amount of data. The processed IQ-samples is given to a framer, which packs the IQ-data with metadata such as the antenna position and a timestamp from the GNSSDO.

While using Ethernet PHY is possible directly from the FPGA fabric, the use of a MAC-layer (Media Access Control layer) and the rest of the network stack is inherently a software job. Therefore the transfer of the data to the modem should be handled by a HPS rather than the FPGA fabric. The same applies to providing an UDP/IP stack and handling socket traffic to transfer the data to the central processing hub. FPGAs not including an HPS can still initialize a processor in the fabric through the use of an IP core. These are however inherently slower, as they typically use a fabric oscillator at only hundreds of MHz, rather than some GHz. A HPS also works out of the box, requires no extra licenses and can be used to program the FPGA fabric of necessary.

The frames with IQ-samples, antenna position and timestamps are transferred from the FPGA fabric to the HPS via direct memory access (DMA). DMA lets the FPGA fabric directly access the system memory without interrupting the processor, while the processor is busy transferring the frames to the modem via Ethernet and UDP.

The use of a HPS also allows the system to run a Linux-based operating system. As the modem provides the system with an Internet connection, the system can be reached through an SSH (Secure Shell) session for debugging and configuration. As Figure 3.3 indicates, the SSH session can access the FPGA fabric via UART or similar interconnects. Furthermore, the FPGA fabric can also be completely reprogrammed by the HPS if necessary or fall back to a safe version of the firmware if a critical error is detected [15].

3.4 Data rate calculation and link budget

While the Nyquist theorem states that it is theoretically sufficient to sample the radar signal at its Nyquist rate, oversampling the signal increases design flexibility at an early stage in the project. This makes it possible to change the radar, which may also change the signals bandwidth and enter frequency. Furthermore, oversampling has the added benefit of enabling averaging of samples to reduce white noise and therefore increase the effective number of bits (ENOB). For instance, for each additional bit of resolution, the signal must be oversampled by a factor of four [27]. To maximize the design flexibility, using an oversampling factor of four is suggested.

The required ADC resolution is not quantitatively known, as this would require knowledge of the receiver noise level and clutter-to-noise-ratio (CNR), amongst others [33]. As a general rule of thumb, the effective number of bits for an ADC is usually one to three bits less than listed due to the SNR of the device. Qualitatively, this places the ADC in the 12-14 bit range. Opting for a higher number of bits early on also makes for a more flexible option for early development.

The reference radar is according to Section 2.2 stated to use an IF stage of 70 MHz and a chirp bandwidth of 48 MHz. Examining the radar measurements in Table 2.2 show that the bandwidth is at most 30 MHz, but that the center frequencies vary. To find the necessary sampling rate for the ADC, the highest frequency component of the IF signal must be known. Examining the measurements, chirp #1 is revealed to have the highest frequency component at 160 MHz for the longest ranges. However, these measurements are taken at an IF frequency of 100 MHz. Considering

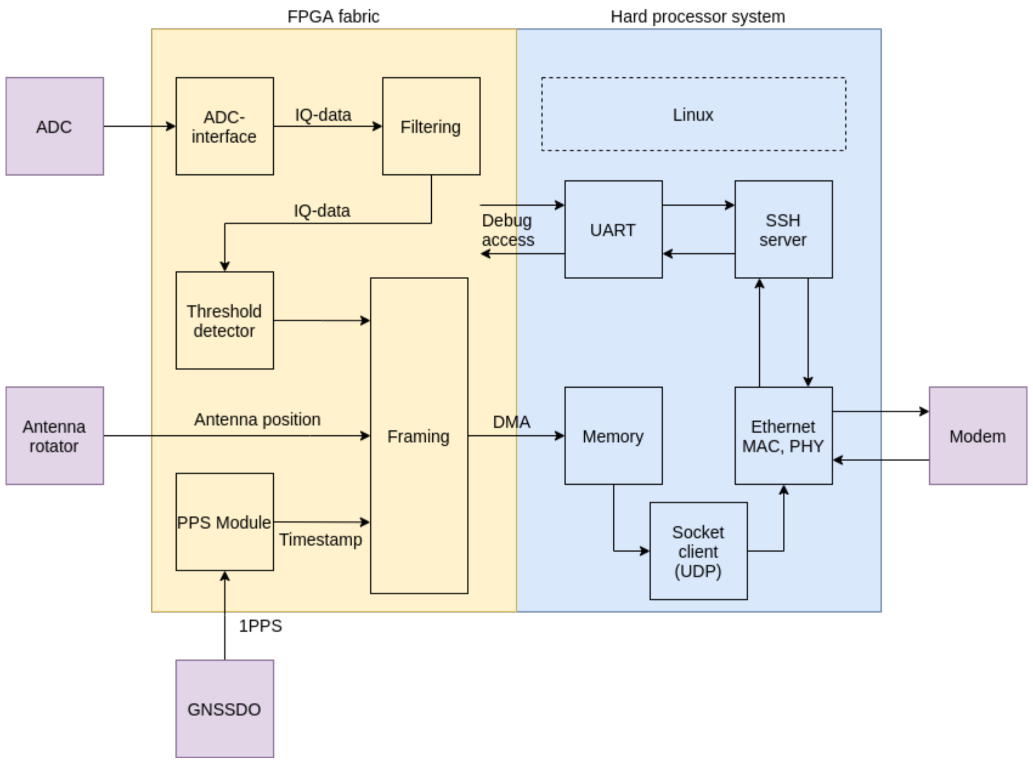


Figure 3.3: Overview over fabric design.

the chirp at an center frequency of 70 MHz yields a highest frequency component of 130 MHz. The data rate calculations for sampling the radar IF signal are thus given in Table 3.1.

Table 3.1: Sample rate and data rate estimates for the IF signal.

Parameter	**Value**
Highest frequency component f_{max}	130 MHz
Nyquist rate $2f_{max}$	260 MHz
Oversampled 4 times	1040 MHz
ADC resolution	14 bit
Data rate at Nyquist rate r_{IF}	3.64 Gb/s
With oversampling $r_{IF,4}$	14.6 Gb/s

The calculations in Table 3.1 reveals that the ADC must at least be able to sample at 260 MHz to satisfy the Nyquist requirement, but up towards 1.0 GHz to add flexibility. This requires the ADC interface to be able to transmit between 3.64 Gb/s and 14.6 Gb/s for a 14 bit resolution.

The data rate requirements for transmitting the radar data to the processing hub varies from the data rate requirements of the ADC, as shown in Table 3.2. Here it is assumed that the varying center frequencies can be ignored by moving the various chirp bandwidths down to complex baseband with digital down conversion (DDC). It is further assumed that the signal is reduced to its Nyquist rate to minimize the data rate. As the listed and measured sweep bandwidth differ, an estimate is given for both. The data rate can be further reduced by neglecting the samples between the bursts, rather than continuously streaming the incoming samples. The burst itself constitutes at most 30 % of the burst repetition interval, seen at the 12 NM range setting in Table 2.2. Therefore the data rate can potentially be reduced by 70 %. More could be removed if the pulses and chirps were isolated.

Table 3.2 shows that the remote site must be able to continuously transmit at least 0.84 Gb/s or 1.34 Gb/s, depending on which estimate is used. If only the bursts are transmitted, this can be reduced all the way down to 0.25 Gb/s for the slightly smaller measured sweep bandwidth. To allow for some overhead and add flexibility, the target uplink rate should be increased towards 1.0 Gb/s, which is a factor of four above the minimum specified.

Table 3.2: Data rate estimates for the complex baseband signal for both the listed and measured sweep bandwidth.

	Listed	Measured
Bandwidth B	48 MHz	30 MHz
Nyquist rate $2B$	96 MHz	60 MHz
ADC resolution	14 bit	14 bit
Data rate at baseband r_{cb}	1.34 Gb/s	0.84 Gb/s
r_{cb} with 30 % duty cycle	0.40 Gb/s	0.25 Gb/s

3.5 Hardware requirements and selection

This section summarizes the hardware requirements set by the design requirements and proposed design. A selection is made and a bill of materials (BOM) is presented at the end.

3.5.1 ADC and FPGA

The ADC and FPGA should be part of a development kit to shorten development time. As such, the selection of these two components are closely related and presented together here. In general, it is desired that both components are part of a development kit and easy to interface with each other. Preferably, they should be part of an evaluation kit consisting of an FPGA and ADC to start development faster. This kit should include the minimum source code to get the ADC and FPGA to work together.

ADC requirements

Sampling rate and data transfer requirements are taken from Section 3.4. The ADC must be part of a development kit or similar that can be readily used, rather than an integrated circuit (IC). It must also employ high-speed interfaces that are easily combined with FPGAs.

The ADC should also provide digital downconversion (DDC) and support complex signals as output to avoid the need to implement it on the FPGA.

To summarize, the ADC requirements are:

- Sampling IF signals at 1 GSps
- Resolution of 12-14 bits
- Support transfer of up to 15 Gb/s of sampled data
- Part of a development kit
- Easy to interface with FPGA
- Should support digital downconversion (DDC)

FPGA requirements

The FPGA is responsible for interfacing with the ADC at high data rates, do real-time signal processing on the digitized IF signal and transmit the data to the processing hub. It also provides a flexible development platform which can be reprogrammed to respond to changing requirements and designs. The FPGA should preferable be part of a development kit with lots of exposed IO, such as PMOD, GPIO, UART/USB-connections for debug, LEDs and various buttons.

Section 3.4 suggests that a Gigabit Ethernet connection is required due to the high data rates. As mentioned in Section 3.3, the FPGA must provide an Ethernet MAC layer which requires a HPS. Thus a SoC FPGA is desired. The FPGA should also be sufficiently large to accommodate future signal processing logic. By experience a mix-range FPGA with around 100.000 logic elements and above a hundred DSP blocks should suffice. These requirements stem from mid-range Cyclone V FPGA the commonly used for such signal processing [9].

To summarize, the FPGA requirements are:

- SoC FPGA
- Gigabit Ethernet connector
- At least 100 000 logic elements
- Above 100 DSP blocks
- High-speed interface for ADC
- Varied IO interfaces such as UART, PMOD, LEDs and buttons

ADC and FPGA selection

Arrow Electronics was chosen as the hardware supplier for the data acquisition system as they are the official vendor for both Intel FPGA and Analog Devices in Norway. They offer both recommendations, advice and support for these products.

iWave's Intel Arria10 SoC/FPGA FMC+ Development Kit (IW-G24D-CU2F-4E002G-S008G-LCM) was recommended by Arrow as a FPGA board for this project. It is part of a reference design for the AD-FMCDAQ2 offered by Analog Devices and Arrow Electronics shown in Figure 3.4. This means that the hardware combination of FPGA and ADC is verified to work, as well as the accompanying HDL. It is also one of the few FPGA boards currently available with an FMC (FPGA Mezzanine Card) HPC (High Pin Count) connector.

(a) iWave Arria 10 SoC Development Board [22].

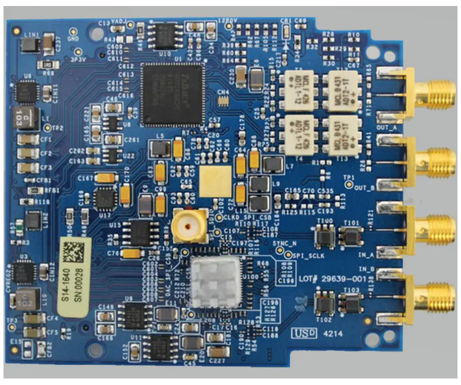

(b) AD-FMCDAQ2 [16]

Figure 3.4: Pictures of the selected FPGA and ADC for the project.

The **Analog Devices AD-FMCDAQ2-EBZ** is a combined ADC and DAQ extension card. It is based on the AD9680 ADC with 14 bit resolution and 1.0 GSPS sampling rate. The card uses a FMC HPC connector and performs data transfer to the FPGA via a JESD204B data interface. It has four FMC lanes running at 10 Gbit/s for the two ADC-channels, which covers the data rate requirement [16]. The ADC can be further configured via SPI (Serial Peripheral Interface) to enable digital down-conversion and signal decimation. Timestamping of the sampled data is also supported for aligning samples from multiple chips [17]. The samples are timestamped relative to a reference signal. This can be considered for the future, if latency between the FPGA and ADC proves to be a problem. While the DAQ is not needed, it can be useful for testing the system by generating test signals. [21].

The iWave systems development kit consists of an Intel Arria 10 SX480 SoC/FPGA system-on-a-module (SOM) and a carrier card containing a FMC HPC (also called FMC+) connector and a Gigabit Ethernet Port. The on-board FMC connector matches the data rate of the ADC, being able to receive a total of 40 Gbit/s of data. This more than covers the data rate requirement from section Section 3.4. The FPGA has 480k logic elements and a speed grade of -2. It features 2736 DSP blocks that can handle 18x19 bit fixed-point multiplication at 438 MHz [12]. The board also contains PCIe lane connectors, USB Blaster-II interface, Pmod connectors, as well as various LEDs, push buttons and switches [22].

The development kit from Arrow was deemed as very good as it more than satisfies the requirements. This combined with the reference design and free support on both FPGA and ADC led to the development kit being selected for the project.

3.5.2 Modem

The modem is responsible to upload the radar data from the remote site. It must do so according to the data rate calculations in Section 3.4. Although a partnership with Telia Norway AS provides early access to the 5G network in Trondheim, the choice will be further elaborated here along with a modem selection.

To satisfy the criteria of up to 1.0 Gbit/s mobile broadband with a high data rate must be used. While the standards are numerous and telecommunication company claims vary, the 3GPP Releases and ITU standards give a good overview. The 3GPP Release 15, which specifies the initial release of 5G, has a target uplink rate of 50 Mbit/s according to the ITU.[1] This number serves as an indicator of the mean data rate, described as the "user experienced data rate" taking protocol overhead and environment variations into account. It should also be noted that this applies only to the eMBB (enhanced Mobile Broadband) part of 5G.

Telia on the other hand only specifies a maximum data rate of at least four times that of their 4G+ networks [5]. This accounts to a maximum data rate of 140 Mbit/s. The maximum data rate is however not suitable to calculate mean operating conditions. It is also not specified weather this is the maximum data rate that can be achieved under the right operating conditions, or the max that can be served in the duration of a short burst of data. Therefore the ITU performance requirements likely give a better and more balanced indication of the data rates.

The Celerway Arctus is a high-end, ruggedized and 5G-ready modem which has been previously tested in the related MilliAmpere project [6]. It has the required gigabit Ethernet interface for the FPGA. The modem itself supports above 1 Gbit/s uplink with load balancing by using six SIM card slots. Using the ITU and Telia data rate estimates, this amounts to between 300 Mbit/s and 840 Mbit/s. The minimum data rate of 0.25 Gbit/s lies somewhere slightly below the presumed mean estimate by ITU, but this leaves little design overhead. It is therefore suggested that measurements must be done to verify the data rate. Furthermore, more signal processing should be applied to decrease the required data rate further.

No further alternatives were found for modems at the time of writing, likely due to the relatively new technology. Teltronika, another modem producer, does not have any 5G ready hardware at the time of writing but expects to release a comparable modem that can be considered in the future [18].

3.5.3 GNSSDO

A GNSSDO (Global Navigation Satellite Systems Disciplined Oscillator) is required for creating accurate timestamps for the radar data. The GNSSDO must have a 1PPS output to help the FPGA determine the exact UTC time. Furthermore, the accuracy of that pulse directly affects the range accuracy of the radar. A 10 ns delay in the system will cause a $c \cdot 10\,\mathrm{ns}/2 = 1.5\,\mathrm{m}$ error in range. Therefore the jitter of the 1PPS signal should be low, but for the development phase some tens of nanoseconds will be sufficient.

As the radar network is a key part of the Ocean Lab Node2 infrastructure, it should also be resistant to jamming and have a high holdover capability should a loss of signal occur. The latter is achieved by featuring the GNSSDO with an extremely stable oscillator. This lets the GNSSDO function with very low frequency drift over an extended time period without being disciplined by GNSS until the signal can be reacquired.

The requirements for a GNSSDO are summarized as follows:

- 1PPS output
- Jitter at only tens of nanoseconds
- Should have holdover capability
- Should be jamming resilient

Furuno supplies high-end GNSSDOs, but for the early development stage the Digilent PMOS GPS suffices as a proof of concept. The PMOS interface is a good fit for the iWave development board as well. It should be noted that the listed jitter of only 10 ns seems a bit low compared to the industrialized Furuno alternatives.

3.5.4 Bill of materials (BOM)

A selection of components is presented in the BOM (bill of materials) in Table 3.3. For the initial development, only the iWave, AD-FMCDAQ2 and PMOD GPS module have been acquired.

Table 3.3: BOM

Name	Price (NOK, exc. VAT)	Vendor
iWave Intel Arria10 SoC/FPGA FMC+	19800	Arrow Electronics [22]
AD-FMCDAQ2-EBZ	13600	Arrow Electronics
Celerway Arctus	0	Borrowed. Price not incquired.
Digilent GPS Expansion Module 410-237	340	RS Components [7]
Sum:	33740	

Chapter 4

Implementation

The following chapter presents an overview of the reference design and how it is altered for the project. It is shown how signals of interest are exported from the **system_bd** entity, which contains the reference design. The new **user_top** entity is introduced to contain the application specific logic, also called the user logic. Furthermore, each module in the user top is presented and documented.

The entire codebase, including the modified reference design, can be found at https://gitlab.com/peteruran/ntnu-coastal-radar.

4.1 Reference design

The reference design is provided by Arrow Electronics and Analog Devices for the evaluation kit containing the iWave Arria 10 SoC board and the AD-FMCDAQ2. It consists of HDL which can be found at GitHub under https://github.com/ArrowElectronics/hdl.git. The reference design also provides a Linux image for the HPS and a software oscilloscope to test the ADC. The following section presents an overview of the design, as well as a manual on how to build and modify it.

4.1.1 Overview of the reference design HDL

The reference design provided by Arrow Electronics is shown in Figure 4.1. The RX path in the fabric is emphasized in Figure 4.2. The figures show that the sampled data is transmitted over FMC to the FPGA, where it is received in the AD9680_JESD204 entity. This entity handles the implementation of the JESD204B protocol, including deserialization of the serial data and synchronization of the four serial channels. The AD9680_CORE entity deframes the data and provides two 64-bit buses for ADC channels ch0 and ch1, as well as a valid signal for each stream.

The samples are then transferred to the HPS via direct memory access (DMA). The DMA is configured to use a 128 bit bus. As such, the AD9680_CPACK entity will combine the ch0 and ch1 samples to a single 128-bit vector, which is buffered up in the AD9680_FIFO before being transferred to the HPS domain with DMA.

The data is further streamed from the HPS via Ethernet using a *libiio* daemon running on a Linux core on the HPS. *libiio* is a protocol developed by Analog Devices for streaming sensor data in an industrial setting [20].

It should also be noted that transferring 14-bit samples using 64-bit signals is under-optimized, especially in regards to the DMA. However, altering the DMA is dependent on the HPS implementation, which is out of scope at this time. Keeping the larger bus also adds flexibility at this point.

4.1.2 Building the reference design

The Arrow Electronics reference design requires the **Intel Quartus Prime Standard version 18.0** software, which can be downloaded from https://fpgasoftware.intel.com/18.0/?edition=

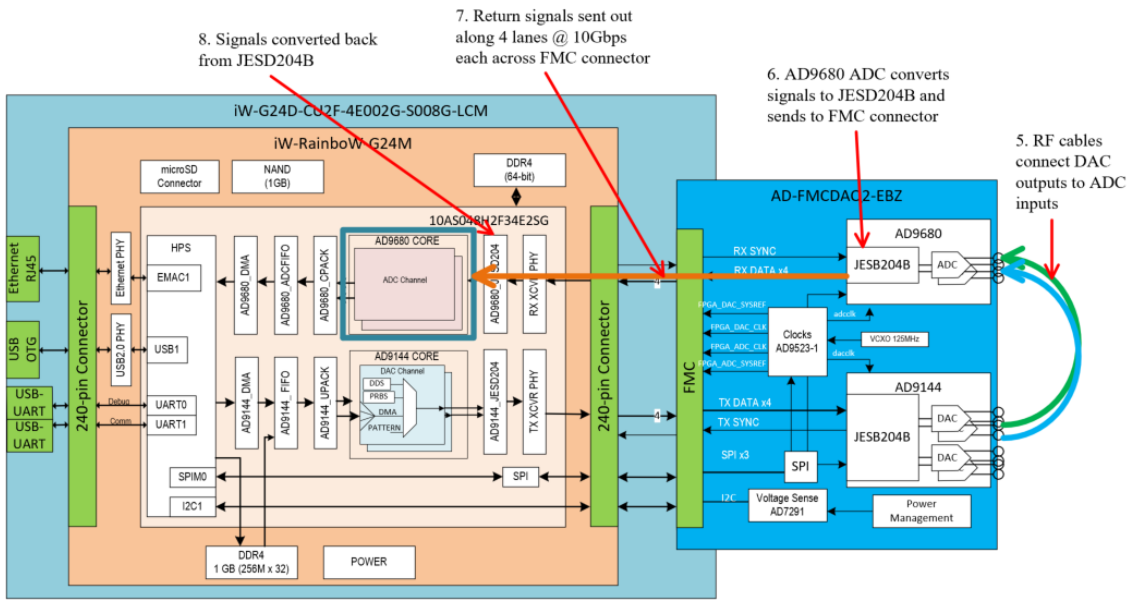

8. Signals converted back from JESD204B
7. Return signals sent out along 4 lanes @ 10Gbps each across FMC connector
6. AD9680 ADC converts signals to JESD204B and sends to FMC connector
5. RF cables connect DAC outputs to ADC inputs
iW-G24D-CU2F-4E002G-S008G-LCM
iW-RainboW-G24M
AD-FMCDAC2-EBZ
240-pin Connector
FMC
AD9680
AD9144
JESD204B
Clocks AD9523-1
SPI
Voltage Sense AD7291
Power Management
DDR4 1 GB (256M x 32)
POWER

Figure 4.1: Overview over the reference design provided by Arrow Electronics, showing a loopback configuration. [20]

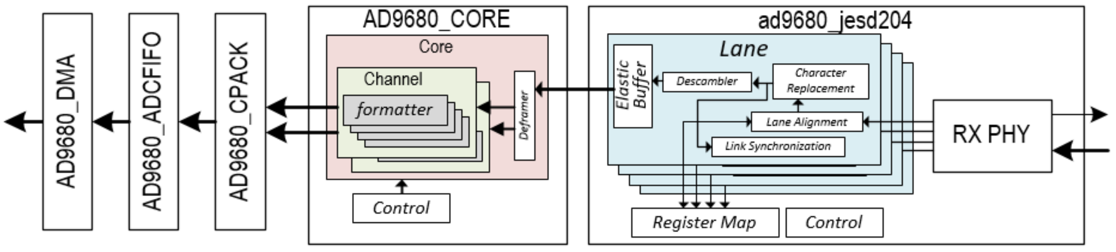


Figure 4.2: The RX path in the FPGA fabric.

standard&platform=linux. The build script will look for Quartus in the system path. The build script will fail if it detects that another version of Quartus is being used.

A floating license can be checked out from the Department for Electronic Systems license server 1716@europa.iet.ntnu.no while being connected to Eduroam. This also works while being connected to Eduroam via the NTNU VPN server (vpn.ntnu.no). Note that the free Quartus Lite version is not sufficient, as does not include support for Arria10 devices.

To build the reference design, first clone the Arrow Electronics HDL repository from Github.

```
git clone https://github.com/ArrowElectronics/hdl.git
cd hdl
```

The repository contains HDL (hardware description language) for various Analog Devices daughter boards and accompanying carrier boards. The latter include both Intel and some Xilinx boards. Checkout the correct branch for the iWave carrier board.

```
git checkout R18.0_IW_CC_2.0
```

To build the HDL, use **make** and target the **daq2** project for the **iwg24d**. The latter is a short form for the iWave board.

```
make daq2.iwg24d
```

The build process can take a significant amount of time to complete, taking up to several hours. The Makefile will run several Tcl-scripts to setup and configure dependencies for the project. The completed project folder will contain a Quartus project file to configure and synthesize the design, a Quartus Qsys (also known as Platform Designer in new versions of Quartus) file to configure peripherals and interconnects. The **system_bd/system_top.v** file contains the design's top level entity **system_bd**.

4.1.3 Modifying the reference design

There are several ways to alter the reference design to include the user logic. One can either

- modify the HDL files directly,
- create a new Qsys component for the user logic and connect it using Qsys,
- or export the **system_bd** signals of interest and manually instantiate and connect the user top.

While modifying the HDL files directly is most straightforward and requires less use of the tools, doing so puts the design changes at risk of being overwritten should the Platform Designer be used to regenerate the HDL wrappers. While creating a new Qsys component mostly consists of selecting the desired HDL source files in the tool, it creates an unnecessary hard dependency on using Qsys to build the design. A decision is made to divide and conquer by exporting the signals of interest from Qsys and manually instantiate an entity for the user logic. This leaves the responsibility of generating the reference design to Qsys, while the designer keeps full control of the changes in the user logic. This also has the added benefit of being able to keep the altered reference design and user top in separate git repositories. This is an advantage as the latter can be updated and simulated independently, in addition to increasing the portability of the design. The altered reference design only is needed for the compilation.

To contain the user logic, a new entity must be inserted into the design. As it contains the user logic and provides a new top-level abstraction created by the user, it shall be called the *user top*. Figure 4.3 shows how the user top is inserted into the reference design by exporting signals of interest from system_bd. The user top must use valid-handshaking on its input and output interfaces to match the proceeding and succeeding entities in the signal processing chain. This is a simple bus protocol enforcing the use of a separate valid signal to indicate whether the data in the associated data bus is valid or not [23].

The design is altered by first opening the **system_bd.qsys** file in Qsys. This will open a large list of Qsys components with ports and interconnects. As it is desired to insert the **user_top** entity between the **axi_ad9680_core** and **util_ad9680_cpack** entities, the **adc_ch_0** and **adc_ch_1** must be exported, along with the **adc_valid** and **adc_clock** signals. The Qsys components with their ports are shown in Figure 4.4 and Figure 4.5. Exporting the signals is done by clicking the "Export" tab in Qsys for both entities. This will break the connection between the entities and instead expose the signals as ports in the **system_bd** entity. These ports can then be connected to the **user_top** instance in the **system_top.v** file.

After exporting the desired signals the HDL must be regenerated. The Qsys generated entity is called **system_bd** and is manually instantiated in **system_top.v** as part of the reference design. The port map of the **system_bd** instance in **system_top.v** must be manually updated to reflect the changes done in Qsys. When this is done, the **user_top** entity can be instantiated beside **system_bd** in **system_top.v**. The user should ensure that the **adc_clock** signal is routed to the **user_top** along with the data to keep everything in the same clock domain.

4.1.4 Pitfalls and other considerations

This section lists some important considerations to successfully set up the project, as well as some pitfalls.

VHDL vs Verilog wrappers

Qsys has an option to generate the HDL wrappers in either Verilog or VHDL. Generating the system_bd entity in VHDL resulted in numerous syntax errors. While the **user_top** entity is written in VHDL, it can easily be instantiated in Verilog as Quartus provides multi-language

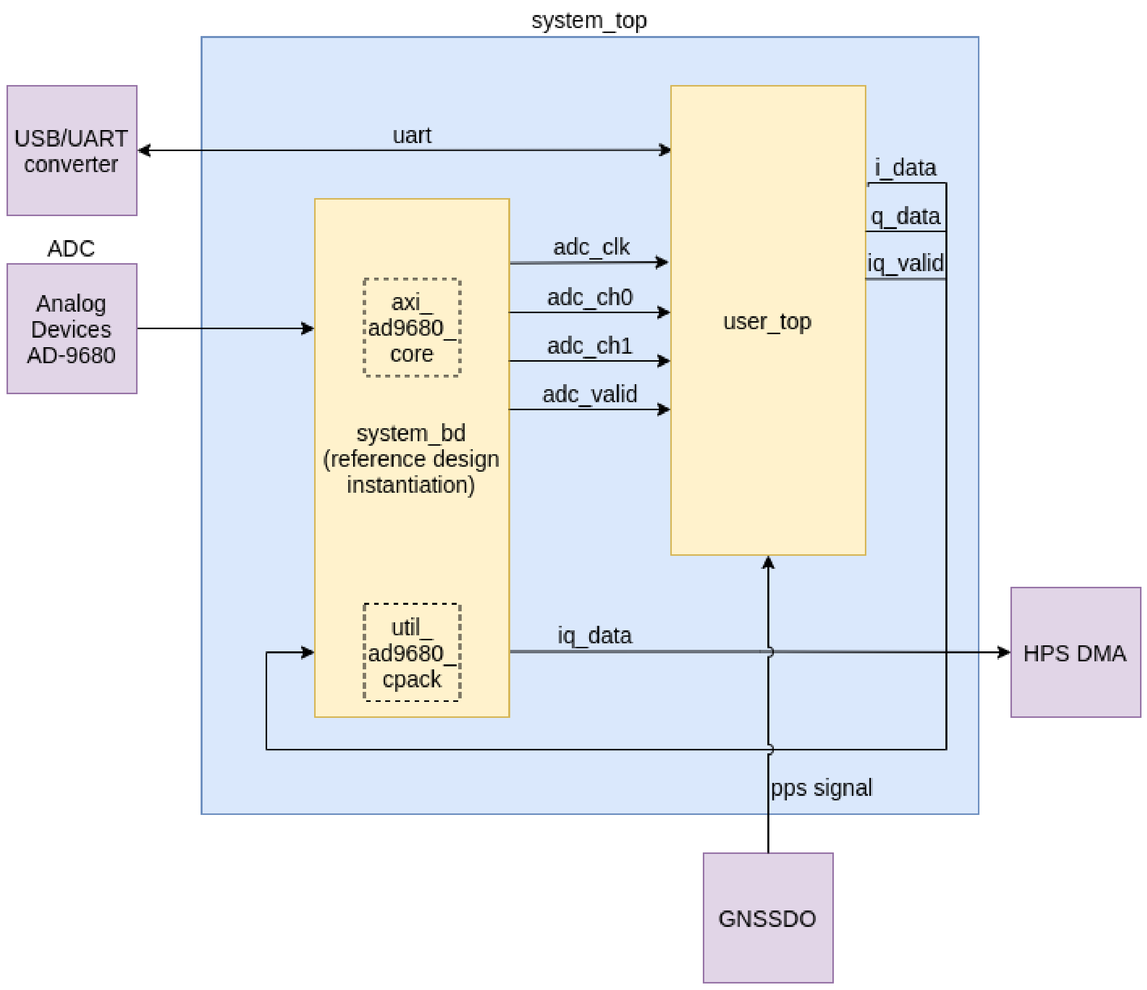


Figure 4.3: Overview of the system top entity, illustrating how the system_bd and user_top entities are connected.

support. This has the added benefit of not needing to translate the **system_top.v** to VHDL, as that is a manually created file supplied as part of the reference design.

Generation of the QIP file

When exiting Qsys, the user is prompted to generate a QIP (Quartus Prime IP) file to store the file paths for all the IP cores. This file must NOT be added to the project, as it is automatically generated and used by **system_qsys.tcl**. Adding the QIP file will cause it to interfere with the **system_qsys.tcl** script and break the synthesis chain.

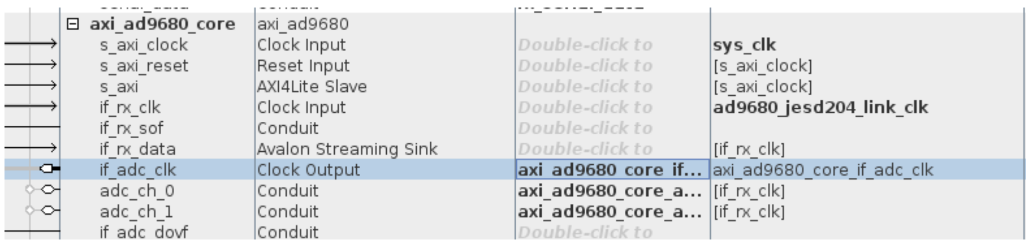
Adding VHDL 2008 support

Quartus Standard only has limited support for the 2008 version of the VHDL standard, compared to Quartus Pro which has full support. While this edition of the standard has few changes for synthesizable code and targets mainly simulation, it brings support for useful constructs such as unconstrained elements in arrays. VHDL 2008 support must be enabled by opening the .qpf (Quartus Project file) file and adding the following line:

```
set_global_assignment -name VHDL_INPUT_VERSION VHDL_2008
```

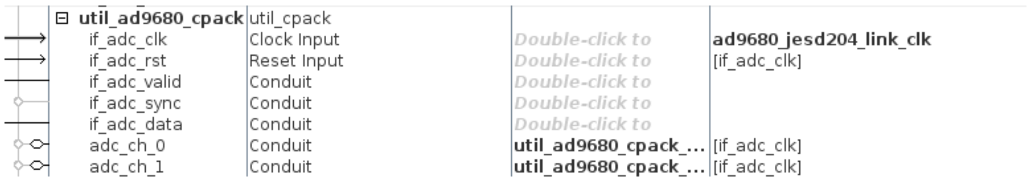
Changing the IP regeneration policy

Quartus will automatically regenerate the whole design, including the Qsys components, when the project is compileded. Since the Qsys components are mostly unchanging and that all frequent



⊟ **axi_ad9680_core**	axi_ad9680		
s_axi_clock	Clock Input	Double-click to	**sys_clk**
s_axi_reset	Reset Input	Double-click to	[s_axi_clock]
s_axi	AXI4Lite Slave	Double-click to	[s_axi_clock]
if_rx_clk	Clock Input	Double-click to	**ad9680_jesd204_link_clk**
if_rx_sof	Conduit	Double-click to	
if_rx_data	Avalon Streaming Sink	Double-click to	[if_rx_clk]
if_adc_clk	Clock Output	**axi_ad9680_core_if...**	axi_ad9680_core_if_adc_clk
adc_ch_0	Conduit	**axi_ad9680_core_a...**	[if_rx_clk]
adc_ch_1	Conduit	**axi_ad9680_core_a...**	[if_rx_clk]
if_adc_dovf	Conduit	Double-click to	

Figure 4.4: The entry for the ad9680_core entity in Qsys, showing the ports, clocks and protocols. Note the exported signals in the right hand side.



⊟ **util_ad9680_cpack**	util_cpack		
if_adc_clk	Clock Input	Double-click to	**ad9680_jesd204_link_clk**
if_adc_rst	Reset Input	Double-click to	[if_adc_clk]
if_adc_valid	Conduit	Double-click to	
if_adc_sync	Conduit	Double-click to	
if_adc_data	Conduit	Double-click to	
adc_ch_0	Conduit	**util_ad9680_cpack_...**	[if_adc_clk]
adc_ch_1	Conduit	**util_ad9680_cpack_...**	[if_adc_clk]

Figure 4.5: The entry for the ad9680_cpack entity in Qsys, packing the ADC ch0 and ch1 into a signle signal.

design updates happen in the **user_top**, Quartus' IP regeneration policy can be changed to only generate IP cores once. This change removes much of the projects compilation time. This setting is found under the **Assignments pane > Settings > IP Settings > IP Regeneration Policy** and is set to "Never regenerate design files for IP cores".

Removal of the DAC code

Compiling the reference design shows multiple setup time violations related to the JESD204 interface of the now unused AD9144 DAC (Digital to Analog Converter). This indicates that the routing algorithm struggles to route the signals in a way ensuring they arrive on time. To fix these, all AD9144 components are removed in Qsys as they played no integral part of the application specific design either way. This has the added benefit of speeding up compilation time, save FPGA resources and relax routing constraints. The removed components are the avl_ad9144_fifo, ad9144_jesd204, axi_ad9144_core, util_ad9144_unpack and axi_ad9144_dma.

Bitstream generation and programming

As the design contains a HPS, the Intel design guidelines discourage programming the FPGA directly with Quartus [13] as this can crash the Linux drivers using the FPGA-HPS interconnect. A restart of the HPS will also cause it to reprogram the FPGA with the bitstream present on the SD card.

To create a new bitstream for the HPS to use when programming the FPGA, the quartus_cpf utility can be used to generate a RBF file (Raw Binary File). This must then be placed on the boot-partition of the SD card in the iWave Arria 10 card.

To generate the RBF file, run:

```
quartus_cpf -c -o bitstream_compression=on daq2_iwg24d.sof
   socfpga.rbf
```

4.2 User top overview

The user top entity contains the application specific design to be inserted into the reference design. This partitioning makes it easier to focus on the core design, as well as increasing the portability. It also makes it simpler and faster to simulate part of the design rather than the system in its entirety.

The user top consists of several debug and convenience modules. The comms module implements a UART interface to the FPGA fabric that can be connected to either the HPS or external ports. Together with the address decoder it makes up a system for register access, where registers can be read and written to. The UART interface and register access are together referred to as the debug interface. This is convenient to provide an entrypoint to inspect the internal workings of the FPGA, as well as adjusting settings in runtime rather than compile time. The address decoder acts as a bridge between the comms module and the various modules. It uses an Avalon MM interface covered in Table 4.3 to access the module registers.

All signal processing is performed inside the DSP Core to provide another abstraction level in the design. It uses a simple valid interface to receive and transmit samples and receives accurate timestamps from the PPS module. The DSP core is covered in Section 4.3.

This section presents the various modules in the user top, along with their testbenches. Testcases and simulation results for the user top and DSP core are presented in Section 5.1. The simulator and test framework selection is also presented there, with an explanation of VUnit.

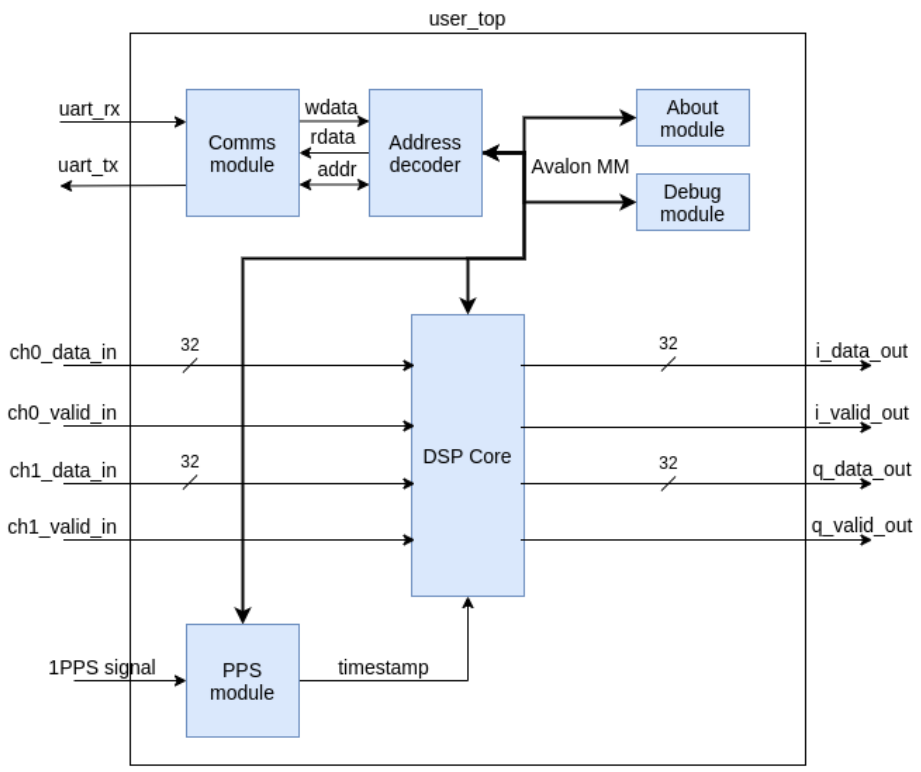


Figure 4.6: Overview of the user_top entity.

4.2.1 Comms module

The comms module is responsible for providing access to the FPGA fabric via UART. It is part of the chain of modules enabling read and write access to FPGA registers together with the address decoder. An overview of the module is shown in Figure 4.7. The UART configuration is shown in Table 4.1.

Table 4.1: Configuration of the the UART serial link.

Baud rate	Data bits	Stop bits	Parity	Flow control	Bit order
115200	8	1	None	None	Least significant bit

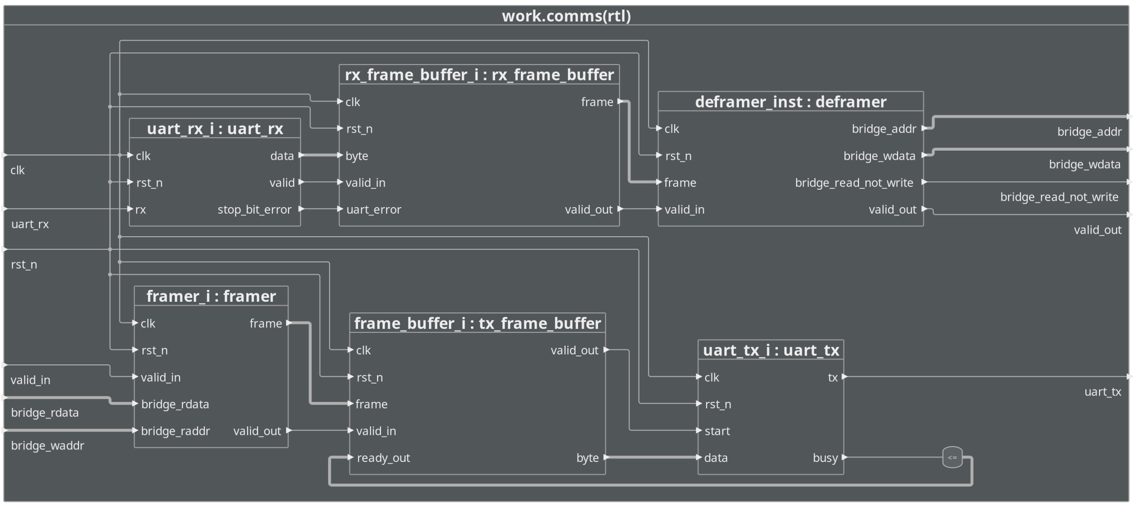


Figure 4.7: Overview over the comms module, showing both the TX- and RX-chain.

Communication is based on frames, containing a header followed by an address and data. The frame is shown in Figure 4.8, consisting of header, address and data fields. The frame is used for both transmit (TX) and receive (RX). The frame header contains flags for specifying write and read, in addition to other system status and error flags. The header flags are listed in Table 4.2. More bits can be reserved in the future for transmitting various status and error flags, such as an unknown address error.

Table 4.2: Table of flags in the frame header.

Bit number	Flag	Purpose
0	Write-not-read	Indicate write or read operation for register access
1	Stop bit error	Indicate possible baud rate mismatch
2	Reserved	Reserved for unknown address error
3	Not used	
4	Not used	
5	Not used	
6	Not used	
7	Not used	

The module consists of an RX and a TX chain. The RX-chain consists of an uart_rx entity and a deframer. The uart_rx entity feeds the deframer with the received bytes, which the deframer uses to assemble the frame and then extract the content. The TX-chain similarly consists of an uart_tx entity and a framer. The framer takes in data and assembles a frame, which is then fed bytewise to the uart_tx entity. The latter exerts backpressure through a simple ready/valid handshake to control when the next byte should be sent. This handshake extens the valid interface with allowing the receiving entity to stall the transfer until it signals that it is ready. Both uart_rx and uart_tx are based on simple state machine implementations.

In the case of the stop bit not being found the uart_rx entity will clear the frame buffer to avoid it

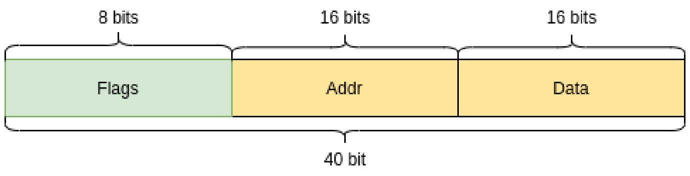


Figure 4.8: Generic frame structure for the UART link.

coming out of sync. It will also initiate the transmission of an empty frame containing the stop bit error flag. This notifies the user of the baud rate most likely being wrong. This is shown in a simulation in Figure 5.7. A stop bit error is trigged by pulling the RX line low for 10 symbols, causing the uart_rx entity to fail to detect the stop bit. This immediately causes it to transmit a frame with the stop bit error flag.

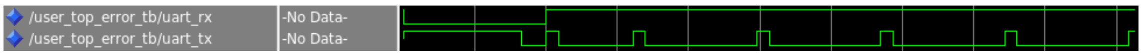


Figure 4.9: Holding the RX line low causes a stop bit error, which triggers the transfer of a frame with the according flag (orange). The stop bit flag can be observed in the second bit of the first byte. Note the reversed bit order.

4.2.2 Address decoder

The purpose of the address decoder is to implement the register access by routing incoming read and write requests to the correct modules. The module is shown in Figure 4.10. It acts as a bridge by translating the incoming address into a module address and a register address. The module address is converted to a chip select signal, whereas the register address is broadcasted to all modules. The activated module in turn either receives the write data or returns the read data, as indicated by the write and read signals (module_wr and module_rd).

The address decoder is based on the Avalon MM standard [8]. It uses single-cycle signals to perform read and write operations. The standard allows for the exact number of signals employed from the standard to be customized to the application, making it a very flexible standard. As such, the pipeline signal rdata_avail has been added from the standard to let the address decoder know that the read data from the module is valid. The subset of signals used from the standard is listed in Table 4.3.

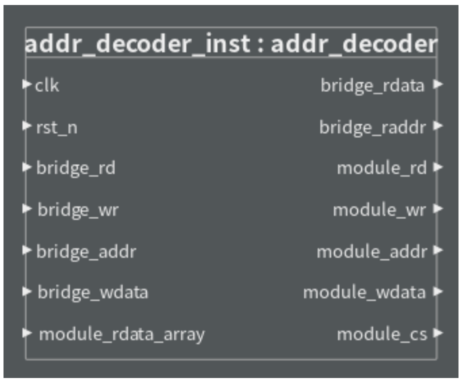


Figure 4.10: An instance of the address decoder, showing the various input and output ports.

Table 4.3: Signals for register access based on Avalon MM [8].

Name	Width	Direction	Description
addr	16	Master → slave	Register address
rdata	16	Slave → master	Read data
wdata	16	Master → slave	Write data
read	1	Master → slave	Indicate read transfer
write	1	Master → slave	Indicate write transfer
chip select	1	Master → slave	Select slave for transfer
rdata_avail	1	Slave → master	Notify master that rdata is valid

4.2.3 About module

The about module serves as a pure convenience module in which information about the current build is stored, such as the git hash and build time. This is useful information for making sure that the correct firmware is used. The values are compiled in and can be accessed through the register access. The address map for the about module is given in Table 4.4.

Table 4.4: Register addresses of the about module.

Register	R/W	Address offset	Description
git_hash	R	0x00	Hash value of current git commit
build_date	R	0x01	DDMMYY
build_time	R	0x02	HHMMSS

4.2.4 Debug module

The purpose of the debug module is to enable the designer to verify basic functionality such as read and write using the register access. It is also used to gather error and status information about the system in one place. As the FPGA is essentially a black box after programming, this module serves as an important entry point to the systems state during runtime. The module also controls the systems LEDs: A blinking status LED to indicate a running system and an error LED to indicate any error. The module's registers and addresses are listed in Table 4.5.

Table 4.5: Register addresses of the debug module.

Register	R/W	Address offset	Description
debug_value	R	0x00	Fixed value of 0xBEEF
write_reg	R/W	0x01	Empty register to test write operation. Initialized to 0
blink_led	R/W	0x02	Write 0x0001 to blink LED
comms_error	R/W	0x03	Sticky bit indicating that an error has occurred in the comms module

4.2.5 1PPS module

The 1PPS module creates a simple timestamp from a 1PPS signal. It consists of a counter incrementing every 10 ns until it is reset by the rising edge of the 1PPS (1 Pulse-per-second) signal. The counter counts from 0 to $(1\,\text{s}/10\,\text{ns} - 1) = 10^8 - 1$. To avoid nonsensical timestamps in case of jitter the counter will not wrap when reaching its high value and must be reset by the 1PPS signal. The module outputs the timestamps as a $\lceil \log_2 1\,\text{s}/10\,\text{ns} \rceil = 27$ bit standard logic vector, which is rounded up to 32 bits for convenience.

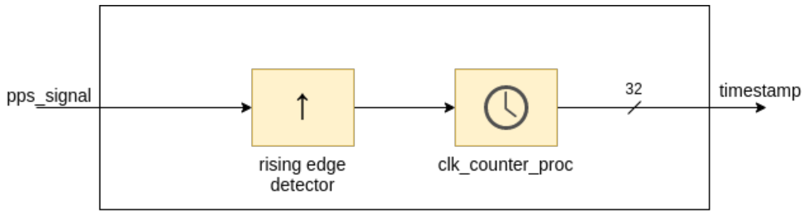


Figure 4.11: Overview over the PPS module.

4.3 DSP core

The purpose of the DSP core is to handle all signal processing in the design. The block diagram of the module is shown in Figure 4.12.

The DSP core performs IQ demodulation on the real TX and RX signals to convert them to complex signals and move their bandwidth down to complex baseband. This reduces the signals Nyquist rate, which makes it possible to reduce the clock frequency needed to drive the signal processing logic, greatly relaxing design constraints. See subsection 4.3.2. It is currently deemed faster to implement the IQ demodulation in the FPGA fabric rather than using the digital downconversion built into the ADC, due to the SPI interface of the ADC interface being tightly connected to HPS drivers.

Clock domain crossing is performed to move the downmixed data to a slower clock domain. Two FIFO (first in, first out) buffers are used to perform the clock domain crossing, which is presented in subsection 4.3.1.

Finally, the pulse compression is performed in the dechirper entity. See subsection 4.3.4.

The DSP core takes in 14 bit wide signals. To comply with the input width of the multipliers in the DSP cores, the IQ demodulator allows the width to increase to 18 bits. This yields an output width of 36 bits from the multipliers in the dechirper, which is also the maximum output from a multiplier. See subsection 4.3.4 for further details about the DSP blocks and signal widths. Note that the output from the DSP core is scaled back to 32 bits to comply with the FIFO buffers used to interface with the HPS for now.

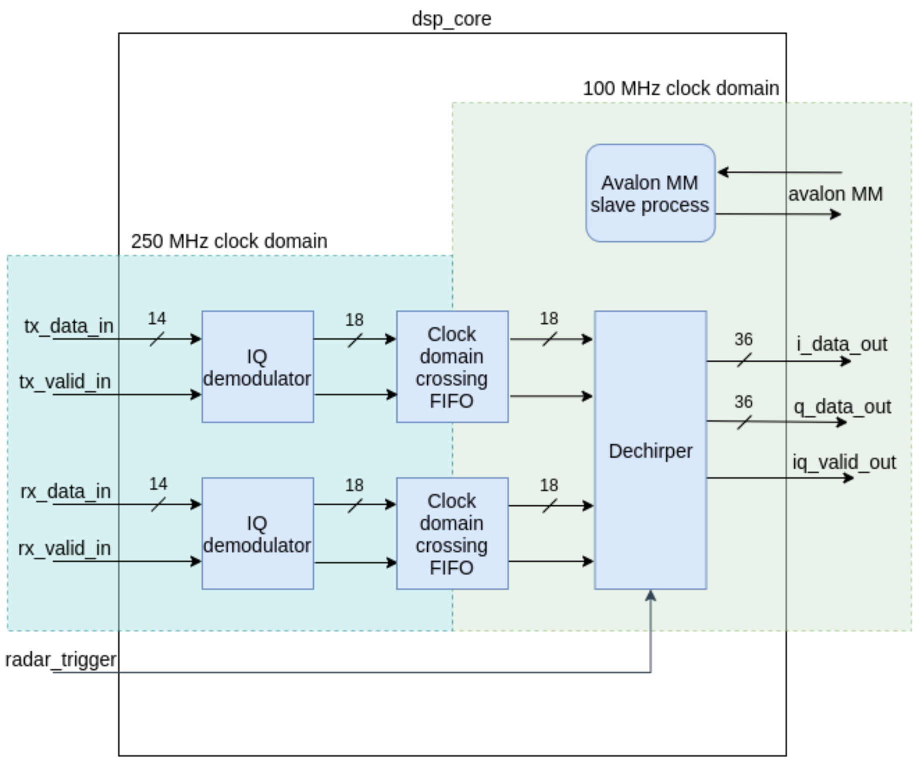


Figure 4.12: Block diagram of the DSP core module. Samples from the RX and TX channels are moved down complex baseband with IQ demodulation before being used for pulse compression in the dechirper. Two FIFO (first in, first out) buffers are used to cross the two clock domains highlighted in the figure.

4.3.1 Clock domain crossing

Theory of operation

Clock domain crossing is necessary to move data from one clock domain to another while avoiding problems such as setup and hold time violations at the registers. In this case, it is necessary to move the input signals from their 250 MHz input clock to a slower 100 MHz clock to relax design constraints. The latter is chosen as it is a commonly employed clock frequency in FPGA design, offering some flexibility while being neither especially fast nor slow. It is also readily available from the design top layer.

Running the entire design at the data input clock of 250 MHz essentially requires that data must be manipulated, routed and finally stored in registers within a clock cycle of only 4 ns. As the design grows, the place and route algorithms will struggle and fail more due to the increased area used on the FPGA. This in turn increases the distance the signals must potentially traverse in the duration of one clock cycle, which increases the likelihood of setup time violations.

Clock domain crossing is necessary due to the clock cycles being shorter in the fast domain than in the slow domain. Since the data is linked to the clocks, the data in the fast domain may already be gone before the slow domain registers that it is there. Essentially, the data must persist longer in the register to enable the slower clock domain to register it. One simple way to achieve this is by using some of the FPGA fabric's block RAM as a FIFO (first-in, first-out) buffer. The block RAM can be initialized as a two-port RAM using different clocks on either port. One port is assigned to writing data from the fast clock domain, while the other port is used to read the data to the slower clock domain. Although this can be expensive in terms of consuming block RAM resources, it is more guaranteed to work as the toolchain can ensure its functionality. Additionally, since the data rate on either side of the FIFO is the same, it should never underflow or overflow. [30]

Implementation

The clock domain crosser is shown in Figure 4.13. It is based around the Intel DCFIFO IP core, which can be readily configured using Quartus. The DCFIFO is based on using FPGA block

RAM as a circular buffer with two ports and some timing optimizations [30]. Using the IP core ensures the correct implementation of this FPGA-specific hardware functionality. It is also faster to implement, with the only drawback being that it makes the design dependent on Verilog source code and simulation models. The simulation of this entity therefore requires cross-language support. See subsection 5.1.1.

The DCFIFO (Dual-Clock FIFO) is configured to use 36-bit wide words, which is equal to the length of one IQ-sample. The memory depth is set to 128 words. This is large enough to hold a considerable amount of samples, while not too large as to waste RAM. The Arria 10 SX 480 contains 28620 kbit of block RAM, so memory usage is not a concern at this point. Note that the IP core requires the memory depth to be a power of two.

As shown in Figure 4.13, one process using the faster write clock feeds incoming samples to the DCFIFO by packing I- and Q-samples to a single array. It is submitted to the DCFIFO along with setting the **wrreq** port to **'1'**, but only if the **wrfull** flag indicates that the FIFO is not full. A similar output process monitors the state of the **rdempty** port. The process will request samples from the DCFIFO by asserting the **rdreq** port as soon as samples are available to minimize latency. The DCFIFO uses one clock cycle to return the samples after **rdreq** is asserted, on which the output **iq_valid_out** port is set.

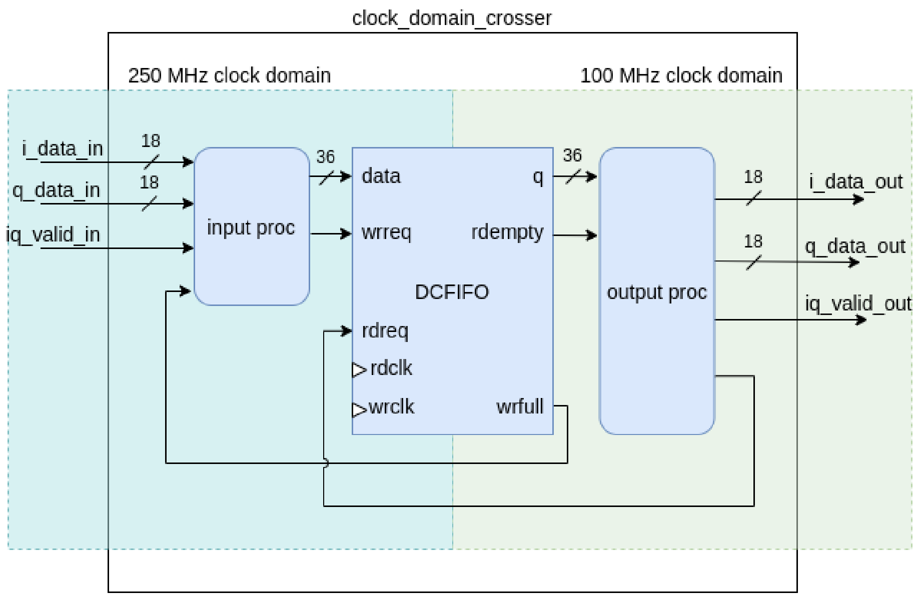


Figure 4.13: Clock domain crosser based on the DCFIFO (Dual-clock FIFO) IP Core, going from a fast clock domain of 250 MHz to 100 MHz. The input process feeds samples to the FIFO, while the output process retrieves them as soon as they are available.

Simulation

The simulation for the cross domain crosser is shown in Figure 4.14. The main process generates random IQ-samples, which are applied to the DUT. The fifo_slave process record the samples at the output of the DUT and places them in a queue. The main process then compares the input values with the values stored in the queue. If they are equal, the simulation passes. Overrun of the DCFIFO inside the DUT is not tested at this point.

4.3.2 IQ demodulator

Theory of operation

The IQ-demodulator is important for coherent processing, as well as reducing the signal's Nyquist rate. The entity converts a real input signal to a complex signal and moves the signals bandwidth down to a complex baseband around the 0 Hz frequency. This reduces the highest frequency component of the signal and lowers the Nyquist rate, while preserving the signal's bandwidth. This makes it possible to reduce the signals sample rate, which makes it possible to clock the signal with

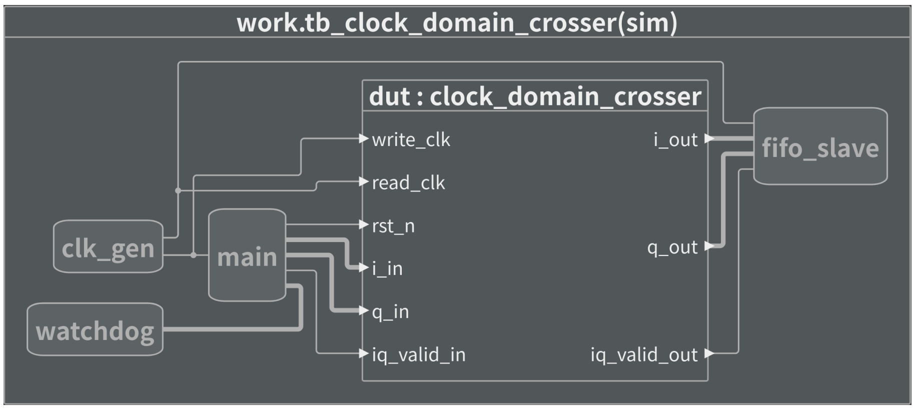


Figure 4.14: Testbench the clock domain crosser. The main process generates random IQ-samples and compares the input samples with the output samples stored in the fifo_slave queue.

a slower clock frequency on the FPGA. The latter has multiple benefits, such as simpler routing in regards to relaxed timing constraints and possible re-use of multipliers.

An IQ-demodulation is performed by multiplying with a complex sinusoid, or the hardware equivalent of using sine and cosine in parallel as shown in Figure 2.2. This moves the positive frequency components of the real input signal down to baseband, while at the same time causing the negative frequency components to wrap around to the positive side of the frequency axis. These high-frequency components are then filtered away using a lowpass filter, preserving only the signals around 0 Hz. The process results in IQ-samples forming a complex baseband containing both positive and negative frequency components.

Sample sorting

Constructing such sinusoids becomes infeasible as the data arrives at such high clock speeds. It is however possible to create a square wave with the same frequency as the input signal. The square wave acts as a 2 bit sinusoid alternating between 1, 0 and -1. This can be viewed as sorting the real input samples around the unit circle in the complex plane using two 4-to-1 muxes, as illustrated in Figure 4.15. The upper mux sorts the samples into the in-phase (I) channel and the lower mux into the 90 degree delayed quadrature (Q) channel. Muxes are inferred by using if-statements in VHDL, as demonstrated in Appendix B. A wrapping sample counter is used to determine which input is to be selected. This implementation requires the signal to be oversampled by a factor of twice the Nyquist rate of the signal to correctly move the spectrum down to a complex baseband.

Filter design

A low pass filter is required to suppress the frequencies stemming from the negative part of the real input signal spectrum, keeping only the complex baseband components. A half-band filter is an efficient solution for this problem. It consists of a filter where every even coefficient, except 0, is 0. The filter requires only half the number of multipliers as the multiplications with zero are constant and can be optimized away. A halfband filter has its transition band around $^1/_4$ of the sampling frequency, which is also called the halfband frequency [32].

The filter coefficients for a half-band filter can easily be generated using the remez function in the scipy.signal package for Python [36]. This is a free option and good for creating models, this does not provide the necessary HDL to implement the filter on an FPGA. As shown in subsection 4.3.4, implementing it by hand requires a lot of work and can end up as under-optimized designs. The Intel FIR II IP Core provides the option to import filter coefficients from file and likely the most optimized design for Intel FPGAs. It is however licensed, which creates a hard dependency for the project [10]. The third alternative is to use Matlab to design the filter and generate the HDL using the DSP System Toolbox [29]. This is chosen because it gives the fastest implementation time and generates unlicensed VHDL code that can be used freely.

pyFDA should also be mentioned as a free option to generate filters. It provides a graphical user interface for generating filters, using the scipy.signal package as a backend. It was not chosen due

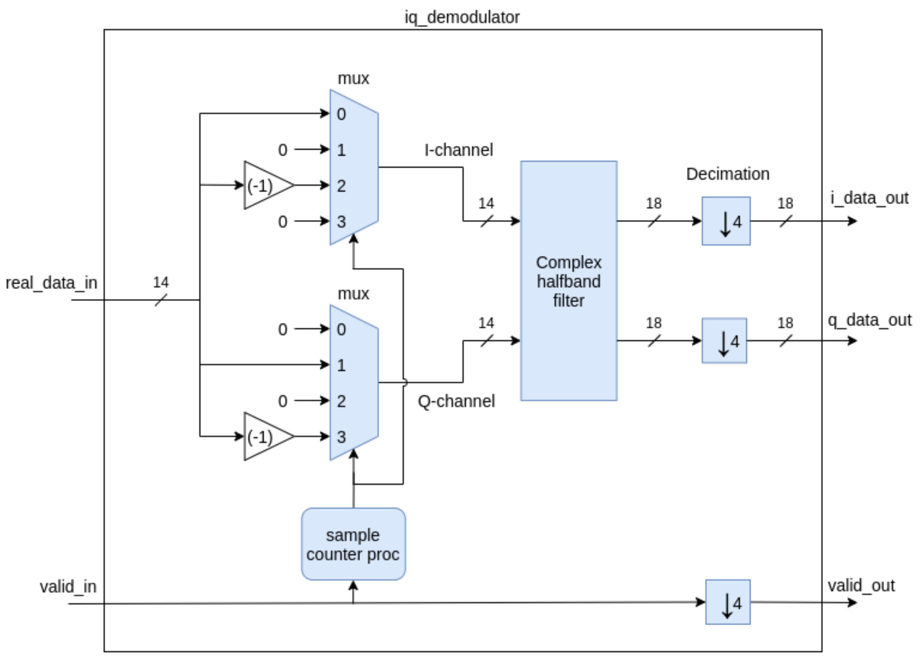


Figure 4.15: Block diagram of the IQ-demodulator. The signal is mixed down by sorting the real input samples using two 4-to-1 muxes. A complex halfband filter suppresses the mirror frequency. The signal is decimated to remove redundant samples.

to the HDL generation lacking maturity and only supporting Verilog at this point in time [25].

The filter is generated in Matlab by using the DSP System Toolbox. The filter type is set to halfband filter with a linear phase. The arithmetic must be set to fixed-point in order to make Matlab use HDL synthesize data types instead of floating-point. The input width is set to 14 bits, but the filter output is set to 18 to match the output port of the entity. The filter is not allowed to overflow to avoid creating distortions and making overflowing easier to debug later, although it should not occur with a properly constrained design.

4.3.3 Decimation

A complex signal does not have an ambiguity between positive and negative frequencies. This makes it possible to reduce the sampling rate of the complex signal down to the signals Nyquist rate without violating the Nyquist theorem, reducing the data rate to its minimum without loss of information [26]. This is done by decimating the filter output by a factor equal to the oversampling rate. The decimation is done by taking the moving average of four samples and keeping every fourth sample. The moving average makes it possible to slightly increase the SNR by averaging away white noise. Dividing by a power of two enables the compiler to implement the division using an inexpensive right shift operation and throwing away the rightmost bits. This is a matter of simple signal routing and does not consume any special hardware on the FPGA.

Simulation results

A simulation for the module is made using Python, Scipy and VUnit. A real-valued chirp with a 5 kHz bandwidth is generated around a 40 kHz center frequency using the scipy.signal.chirp function [36]. The input signal is shown in Figure 4.16. The samples representing the chirp are written to file and accessed in the simulation with the help of VUnit's integer_array_t type [38]. The resulting complex output signal is logged to an array of the same type and similarly written back to file, before being plotted in Python. The output signal is shown in Figure 4.17. The power spectral density plot clearly shows that the bandwidth has been moved down to complex baseband around 0 Hz. The time domain plot illustrates the 90 degree offset between the I- and Q-channels. Furthermore, the magnitude of the complex signal lies at a constant level as expected.

The various signals in the module can also be seen in the Modelsim simulation waveforms shown in

Figure 4.18. Here the real-valued signal is first sorted into the I- and Q-channels. Then the filter smoothes out the signal, removing all high-frequency components from the signal.

Resource usage

The Fitter Resource Utilization Report by Entity lists that each filter channel used 14 DSP blocks, in addition to 1200 ALMs (Adaptive Logic Modules). The latter contains generic logic, such a look up tables, adders, muxes and registers. The effect of using the halfband filter is clearly seen, as the number of DSP blocks consumed is half the filter order due to every other coefficient being zero. As there are 185.000 ALMs on the Arria10, this is a modest contribution. 1368 DSP blocks are more precious, where the IQ demodulator uses approximately 2 % of the available resource. This is an important consideration for a smaller FPGA, such as the Cyclone V, which has between 25 and 342 DSP blocks depending on the version [9].

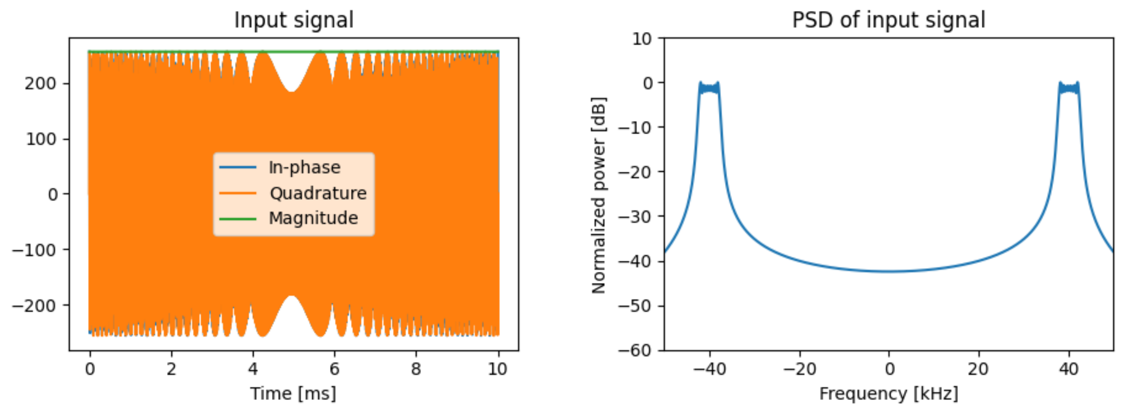


Figure 4.16: Input data for the IQ-demodulator simulation with a 5 kHz chirp around a 40 kHz center frequency. The power spectral density (PSD) is mirrored around 0 Hz due to the input signal being real.

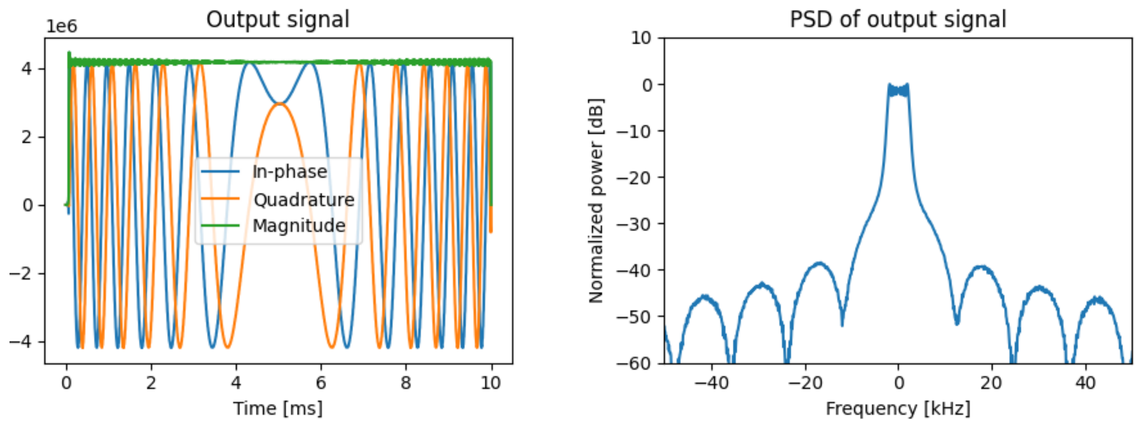


Figure 4.17: Output data from the IQ-demodulator simulation. The chirp has been moved down to the complex baseband and the mirror frequency has been suppressed with a filter.

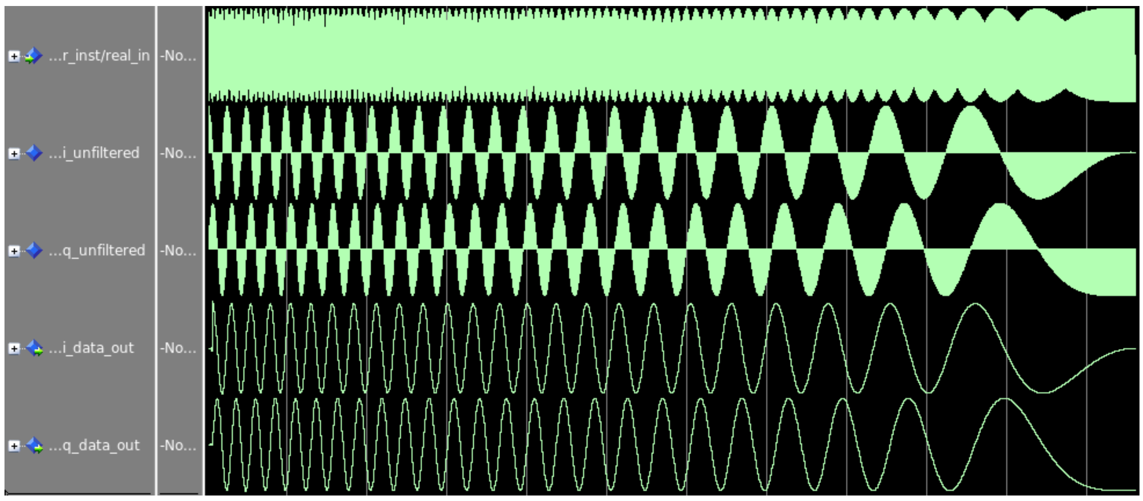


Figure 4.18: Modelsim displaying the real chirp signal after simulation, as well as the unfiltered and filtered I- and Q-channels.

4.3.4 Dechirper

Theory of operation

The dechirper is part of the DSP core and is responsible for converting the received radar chirp to a pulse by performing pulse compression. As specified in Section 3.3, the dechirper must create a matched filter using the TX signal at the rising edge of the radar trigger pulse. The module must then convolute the reflected RX signal with the filter, which dechirps the signal.

A matched filter maximizes the SNR of the system by convolving the reflected signal $s_{RX}(x)$ by convolving $s_{TX}(t)$ with the complex-conjugated and time-reversed transmitted signal $s^*_{TX}(-t)$ [33],

$$f(t) = s_{RX}(t) * s^*_{TX}(-t), \tag{4.1}$$

where both signals are assumed to be complex valued.

The dechirper can be implemented as shown in Figure 4.19. The filter creator process implements a finite state machine (FSM) that samples the TX signal at the rising edge of the radar trigger pulse. The FSM is illustrated in Figure 4.20. The process will take in samples until either the constant MAX_CHIRP_LENGTH or the input port *chirp_length* is exceeded. The latter enables the DSP core to specify the length of the chirp. This is necessary as the chirp lengths vary and some may be shorter than the maximum specified by the constant. When the TX signal is sampled, the FSM goes into its last state and creates the filter coefficients by taking the complex-conjugate and time-reversed of the signal. The former is as simple as adding a sign-bit to the quadrature channel to make it negative. The latter is done by reversing the array containing the samples, which the compiler performs by simple routing. Neither is very resource-intensive for the FPGA. Finally, the FSM goes back to the initial state, listening for the trigger signal.

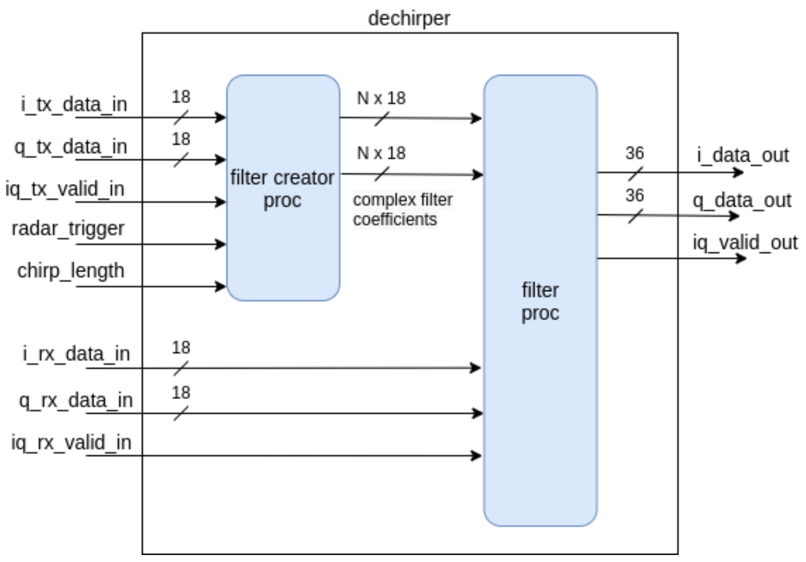


Figure 4.19: Block diagram of the dechirper entity. One process samples the TX signal and creates the matched filter coefficients, which the filter process uses to perform the corrolation with the RX signal.

The filter process performs the correlation between the filter coefficients created by the filter creator process and the RX signal. To design the filter, the filter length given by the compile-time constant MAX_CHIRP_LENGTH, must first be determined. To find the maximum number of samples the chirp with the highest time-bandwidth product is considered. This equals to a 20 MHz wide chirp over 15 µs for a range of 12 nautical miles Table 2.2. Sampling this signal at baseband at its Nyquist rate with an ADC equates to

$$N = 2 \cdot 20\,\text{MHz} \cdot 15\,\mu\text{s} = 600 \text{ samples}, \tag{4.2}$$

which equals to a filter length of 600 samples. Further considering that the multiplication of two complex signals, x and y, equals to

$$x \cdot y = (x_I + jx_Q) \cdot (y_I + jy_Q) = (x_I y_I - x_Q y_Q) + j(x_I y_Q + x_Q y_I), \tag{4.3}$$

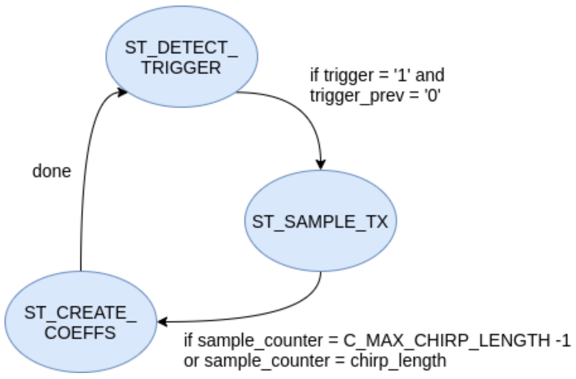

ST_DETECT_ TRIGGER
if trigger = '1' and trigger_prev = '0'
ST_SAMPLE_TX
if sample_counter = C_MAX_CHIRP_LENGTH -1 or sample_counter = chirp_length
ST_CREATE_ COEFFS
done

Figure 4.20: Flow chart illustrating the FSM of the filter creator process, which samples the TX signal and creates the matched filter coefficients for the dechirper.

where it can be seen that two multipliers are needed per sample. This requires a total of $2 \cdot 600 = 1200$ multipliers. As the Arria 10 DSP blocks contain two multipliers each, the filter will consume 600 of about 1300 DSP blocks available. This consumes a large number of FPGA resources, but can nevertheless be done as the resources are there. See subsection 6.2.2 for a discussion on optimizing the DSP block usage.

Two design attempts for the filter process are presented: A simple design approach and a slightly more optimized one. This section serves to illustrate why careful consideration of the design and resource usage is necessary for FPGA design.

Simple filter design approach

Listing 4.1 shows a slightly simplified version of the filter process in VHDL. It consists of a for-loop correlating the filter coefficients with the RX signal and summing the contributions to the **v_i_data_out** and **v_q_data_out** variables, which are applied to the output of the dechirper entity at the next clock cycle. Notice that the product of two equal length signals has twice the length of the originals.

```vhdl
filter_proc : process(clk) is

    variable v_i_data_out           : signed(2*C_CHANNEL_SIZE - 1 downto 0);
    variable v_q_data_out           : signed(2*C_CHANNEL_SIZE - 1 downto 0);
    variable v_iq_valid_out         : std_logic;

begin

if rising_edge(clk) then
    if rst_n = '0' then
        i_data_out => (others => '0');
        q_data_out => (others => '0');
        iq_valid_out => '0';
    else

        -- Correlation of complex signals
        for i in 0 to MAX_CHIRP_LENGTH - 1 loop
            v_i_data_out := v_i_data_out + (i_rx(i) * i_filter(i) - q_rx(i)
                * q_filter(i));
            v_q_data_out := v_q_data_out + (i_rx(i) * q_filter(i) + q_rx(i)
                * i_filter(i));
        end loop;
        v_iq_valid_out <= '1';

    end if;

    -- Set output signals
    i_data_out   <= v_i_data_out;
    q_data_out    <= v_q_data_out;
    iq_valid_out <= v_iq_valid_out;

end if;
end process;
```

Listing 4.1: Simplified process for implementing the matched filter.

This design works perfectly under simulations and provides the matched filter output as desired. The for-loop will perform the convolution of the two complex signals and accumulate the filter response in the **v_i_data_out** and **v_q_data_out** variables, before storing the final value in the **i_data_out** and **q_data_out** signals. However, due to the concurrent nature of FPGAs, compiling the design will cause the synthesizer to implement the content of the for-loop in parallel. In hardware terms, a chirp of $MAX\_CHIRP\_LENGTH = 64$ number of samples will need $2 \cdot 64$ multipliers for a single channel, equaling to 128 multipliers for both channels. While this could work for a few samples, the compiler will not be able to implement the design for any significant number of samples.

This design works perfectly under simulation and provides the matched filter output as desired. However, due to the concurrent nature of FPGAs, compiling this design will cause the synthesizer to implement the content of the for-loop in parallel. This means that all 1200 multiplications are expected to be performed in parallel. As the DSP blocks are placed in groups around the hardware, this places heavy constraints on the routing algorithm as the physical distance the signals must traverse in a single clock cycle is relatively large.

In addition to high consumption of DSP block for multiplication, the dechirper also suffers from timing violations due to long combinatorial paths being inferred when summing the filter contributions in the for-loop. The graph generated by the Quartus Timing Analysis tool in Figure 4.21 for one of the dechirper output signals reveals that the data arrived more than seven full clock cycles too late, corresponding to $-28.56\,\text{ns}$ in negative slack. Ultimately, the compiler is not able to synthesize the design as intended by the designer.

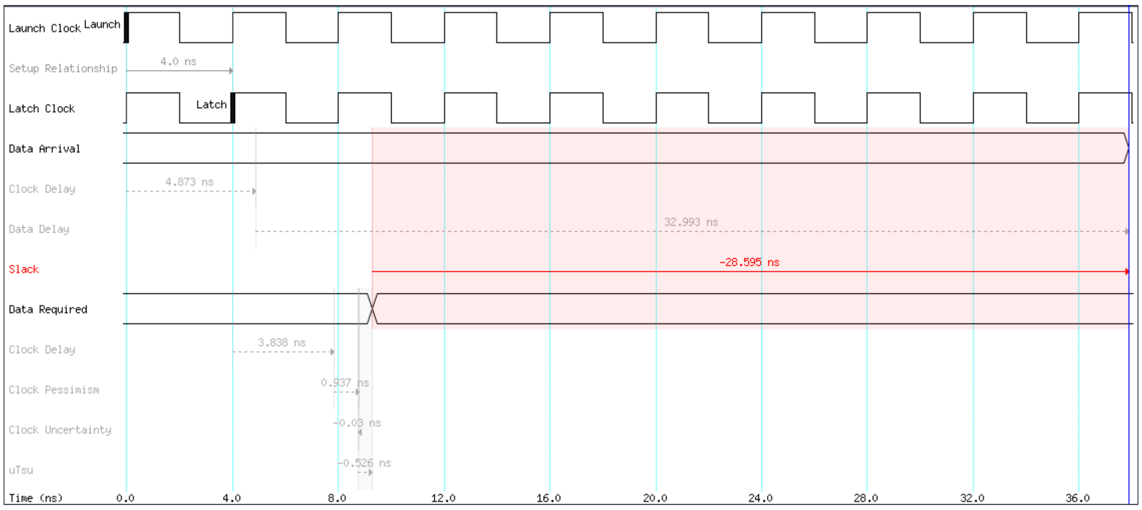


Figure 4.21: Quartus Timing Analysis output for the dechirper initial design without pipelining, showing a large setup time violation over at least seven clock cycles.

Improved design using pipelining

To avoid the timing errors encountered in the simple design attempt, the filter behavior model must be more closely specified using HDL.

Setup time errors generally indicate that a process must be executed over multiple clock cycles. This can generally be done in one of two ways. The process can be changed to an asynchronous process and declared a multicycle path in the timing constraints. This allows the signal to propagate through an asynchronous logical circuit using more than a single clock cycle. Multicycle paths are however infeasible if data arrives at the input every clock cycle, as the asynchronous process is busy propagating the signals for several clock cycles. The other option is to create a pipeline by splitting the large process into multiple smaller processes. For every clock cycle, the smaller processes perform their calculation and hand over the result to the next process in the pipeline. This retains fine-grained control of the processes by clocking them, as well as allowing data to be transferred to the process at every clock cycle.

A pipelined version of the filter is shown in Figure 4.22. Note that the figure is simplified and does not take multiplication by the complex conjugated filter coefficients into account, as specified in Equation 4.3. The idea still holds but must be extended to include complex signals. Convolution between the signal and the filter coefficients is implemented by exploiting the parallel nature of FPGAs. Therefore the convolution can be implemented by multiplying each sample in the shift register by the filter coefficients. The resulting products are then added together over multiple clock cycles using a pipelined binary adder tree. After all filter contributions are summed, the resulting filter response will be made available at the output of the process.

To ease the design constraints somewhat due to exceedingly long compilation times, the chirp length is set to 256 samples.

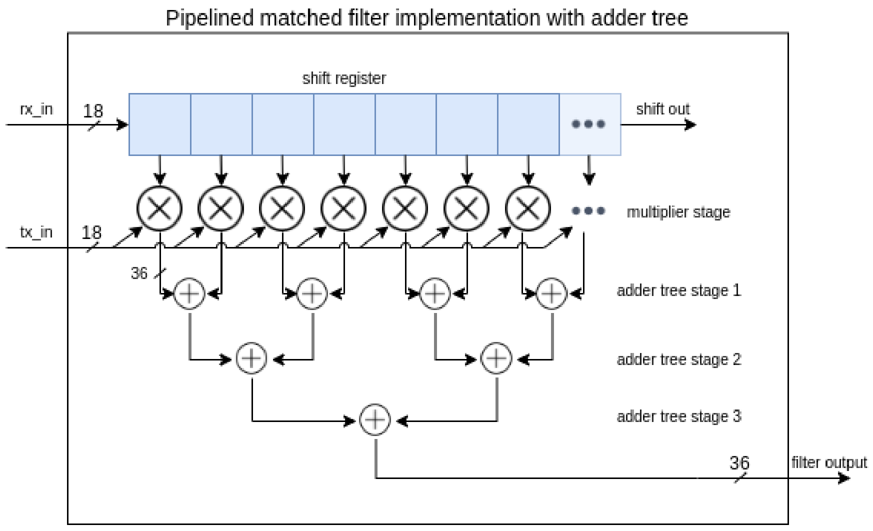


Figure 4.22: Simplified architecture for a pipelined matched filter implemented using shift register based convolution and a binary adder tree.

Using pipelining has the consequence of introduces latency into the design. Due to the adders binary tree structure, the number of pipeline stages can be found using the base 2 logarithmic function. For instance, adding the response from 256 filter coefficients results in $log_2(256) = 8$ pipeline stages. Using a 10 ns clock to drive the adder, this introduces 80 ns of latency to the system. Furthermore, 256 filter coefficients result in $4 \cdot 256 = 1024$ multipliers consumed due to multiplying two complex numbers. This consumes around a quarter the available multipliers in the Arria 10 FPGA. As each DSP block contains two multipliers, this translates to half the DSP blocks on the FPGA.

The dechirper is simulated along with the overall design in Section 5.4.

Chapter 5

Design verification

This chapter presents the process of verifying the FPGA design through simulation, as well as physical testing. Verification is a central part of digital design to ensure that the design works in hardware. Unlike software, it is much harder to test the design ad-hoc as it requires both time-consuming compilation to hardware and as well as a physical test setup. Furthermore, insight and debugging capabilities are reduced considerably when the design has been synthesized and programmed to the FPGA. It is therefore important to have good testbench coverage of the design.

An overview and selection of verification tools and simulators is presented, along with a selection of these for the project. Justification for a continuous integration (CI) workflow for FPGA is given, in addition to the tool configuration.

The testbenches for the user top and the DSP Core are presented, along with their simulation results. A final physical test is also performed to verify that the design works as intended in hardware.

5.1 Verification toolchain overview and selection

5.1.1 Simulators

Simulators are used to make cycle-correct emulations of the design to ensure that the logic is valid. Two simulators are considered for this project: Modelsim and GHDL. High-end simulators like Rivera Pro are considered out of scope for this project and not part of the selection.

Siemens/Mentor Modelsim

Modelsim is a proprietary and commercially available simulator made by Mentor Graphics (now Siemens EDA). It comes bundled with Intel Quartus and is therefore a widely used simulator for Intel FPGA designs. It also supports cross-language simulation with VHDL and Verilog, as well as support for encrypted IP cores. Both are often a requirement when working with IP cores, for instance to interface with a particular part of the FPGA hardware. Intel also provides libraries with bus-functional modules (BFMs) for Modelsim, which are non-synthesizable behavioural models of integrated circuits and similar hardware. It also has a GUI (graphical user interface) which can be very useful for debugging. Modelsim is a licensed product and requires one license per processor core if used in a parallel processing setting.

GHDL (Open source)

GHDL is an open-source simulator. Since its version 1.0 release in 2021 the simulator has fairly good support for VHDL 2008 constructs, to the point where doesn't restrict the designer. While it lacks a dedicated GUI, it is completely free and can therefore be run on multiple processor cores at once. This can significantly speedup testbench execution by running multiple tests in parallel, better utilizing modern multi-core processors. It also provides a significant speedup over Modelsim. This makes GHDL very good for deploying in continuous integration (CI) settings. The drawback of using GHDL is that the design must be pure VHDL in addition to lacking support for encrypted IPs. Consequently, GHDL is best for performing unit tests on a module basis rather than top-level simulations of entire designs.

Simulator selection

Both simulators are used in this project, due to different requirements. Modelsim is necessary to run simulations on parts of the design using Verilog, such as the DCFIFO component used in the clock domain crosser in subsection 4.3.1. It also provides a GUI, which is an advantage while debugging. GHDL is however used for running the continuous integration pipelines, described in Section 5.2. While it cannot simulate the entire design due to it only supporting VHDL, it can be used to execute most of the module testbenches. It being a free, open source simulator also makes it possible to run the simulator in the cloud, which is further described in Section 5.2.

5.1.2 Verification library

While VHDL can be used for both design and verification, mostly simple **assert *boolean*** statements are provided for verifying conditions in testbenches. It is therefore lacking in features increasing both testing capabilities and usability, such as asserting conditions within time limits or checking for signal stability. This can be added by using verification libraries, such as UVVM and VUnit. Along with providing extended checking functionality, they provide BFMs (bus-functional modules) to emulate and interface with different bus standards, such as UART and SPI. As OSVVM (Open Source VHDL Verification Methodology) is included as a dependency in both VUnit and UVVM it will not be considered as a stand-alone verification library.

UVVM

UVVM (Universal VHDL Verification Mythology) and its subset **UVVM Light** are open-source verification frameworks provided by Inventas (formely Bitvis AS) [24]. The light version consists of a feature subset including a checking library and BFMs for commonly used interfaces like UART, SPI, SBI and Avalon MM.

VUnit

VUnit is an open-source unit testing framework offering commonly used BFMs and an extended checking library. It differs from UVVM by being more community-driven and openly developed than UVVM, albeit currently with a less mature selection of BFMs. It also differs from UVVM by providing a Python-based run-library. The run-library of VUnit offers scanning directories of source files, in addition to simulator backends for most simulators. The run-library also functions as an API between the testbenches and Python. This enables easier interfacing between Python and VHDL. This allows designers to use Pythons more extensive language and powerful libraries to generate and validate simulation data using a high-level programming language, rather than VHDL. VUnit further enables designers to make use of continuous integration toolchains, which is further discussed in Section 5.2.

Selection

UVVM is considered too extensive for a project of this scale, making it's subset UVVM Light a better choice. While VUnit and UVVM Light provide similar capabilities on checking and logging, UVVM Light has a larger and more mature selection of BFM's. VUnit however provides a testbench compilation and execution environment, as well as a simple and powerful Python interface. The capability to create and validate testbench stimulus in Python outweighs the slightly more immature BFM selection, which makes VUnit the verification library of choice for the project. Although it is possible to include both VUnit and UVVM, it is deemed better to take on as few dependencies as possible for stability.

5.2 Continuous integration for design verification

Continuous integration (CI) is a technique where changes to the source code are frequently merged into the main codebase to avoid diverging codebases and conflicts. It is combined with automatic execution of tests to ensure that functionality is not accidentally broken. While inherently a software development technique, it is well suited for FPGA design. This is especially true for large designs where testbench executing becomes a time-consuming task.

Continuous integration for FPGA design is one of the advantages VUnit advocates. VUnit advises to test early and often by automating test execution of unit tests [38]. Unit tests are tests of individual modules in the design, rather than testing the whole design at once. VUnit offers hooks for CI tools so it can monitor the status of the unit tests and report if any of them are failing, thus immediately altering the designer when changes break existing functionality.

This project employs git as a version control system with the main codebase stored at GitLab. GitLab is chosen over Github and Bitbucket as it is the only git server that supports local runners. As the name indicates, local runners are programs that run on machines set up by the user, rather than in the GitLab cloud. The local runners can automatically execute a set of instructions after new code is pushed to the server, such as running the FPGA synthesis and programmer chain.

CI is performed in the GitLab cloud using VUnit and Docker. Whenever code is pushed to the server, the GitLab CI service uses Docker to fetch a minimalistic Linux-based image with the latest GHDL and VUnit versions. It then uses this image to run the tests by executing the VUnit *run.py* file with a Python3 interpreter. The *run.py* file scans the project for source and testbench files, compiles them and executes all tests. VUnit then reports back to the GitLab CI service, which either accepts the new code or alerts the designer of failing tests. The relatively simple YAML file used to configure the service is shown below.

```yaml
default:
  image: ghdl/vunit:mcode

vunit-tests:
  stage: test
  script:
    - echo "Installing requirements..."
    - python3 -m pip install -r ./requirements.txt
    - echo "Running VUnit run.py script..."
    - python3 ./run.py
```

Listing 5.1: Process for providing a module with Avalon MM register access.

5.3 User top testbench

The user top testbench is intended for simulating how the various modules work together, as opposed to the unit tests written for the individual modules - see chapter 4 for the unit tests. This excludes testing the ADC and HPS interfaces, as they are part of the reference design and outside the abstraction level of the user top. Testing the signal processing is performed in the testbench for the DSP core. This is done to narrow down the scope, as well as reduce testbench execution time. The user top testbench is intended to expand as more modules are added to the design, which will increase the way the modules interact with each other.

The testbench is currently used for verifying the register access for reading and writing to the registers in the various modules. These tests are well suited for the user top testbench, as they rely on both the comms module and the address decoder. It also tests the register access process in the various modules. The testbench is also used for verifying that errors, such as the stop bit error in the comms module, propagate correctly to the debug module, which in turn enables the error LED. The various tests are listed in Table 5.1.

Testbench setup

An overview of the testbench is shown in Figure 5.1. The main process contains the testbench's test cases. It instructs other processes to apply stimulus and then checks the output of the DUT

Table 5.1: Test cases for the user top

Testcase	Name	Description	Pass/Fail
TCS-1	Read from about module	Register access read	Pass
TCS-2	Read from debug module	Register access read	Pass
TCS-3	Read from DSP core	Register access read	Pass
TCS-4	Write to debug module	Register access write	Pass
TCS-5	Status LED	Status LED is enabled after reset is deasserted	Pass
TCS-6	Error LED	Error LED is enabled due to baud rate mismatch	Pass

(Device Under Test). Interfacing with the debug interface to access the design's registers is done by using the UART master and slave BFMs, which are connected to the DUT. These BFMs are part of VUnit. They enable the main process to push frames to a queue, which is transmitted to the DUT via UART. The UART slave concurrently reads any output from the DUT and puts it in a queue of bytes, which the main process can pop. Additional functions have been developed for assembling the bytes to frames, as well as decoding the frames into data per Figure 4.8. A watchdog process, also part of VUnit, terminates the testbench if it gets stuck and the execution time exceeds 1 ms. This is important for automatic testbench execution in CI.

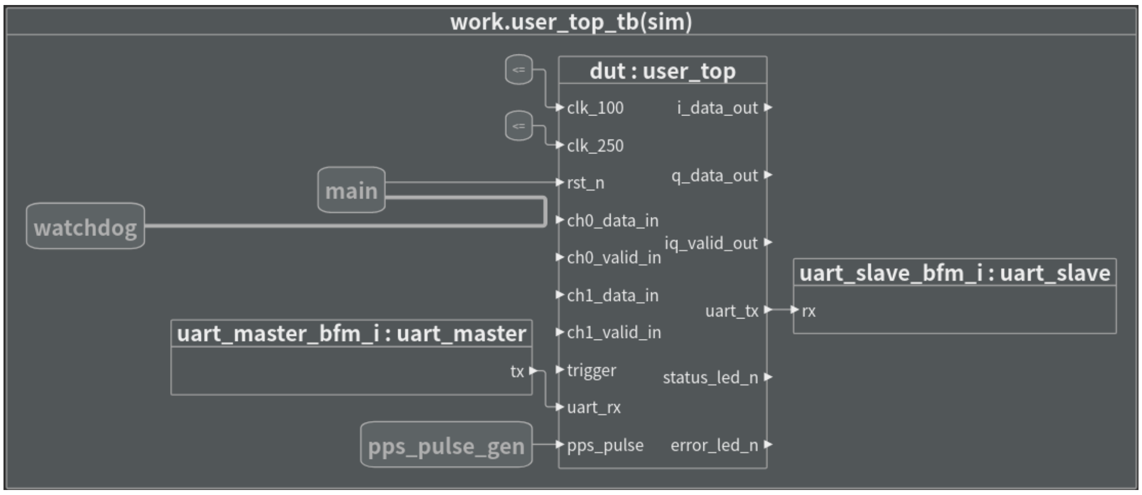


Figure 5.1: Diagram of the user top testbench.

5.4 DSP core testbench

The DSP core testbench is used to verify that the signal processing chain performs as expected. The testbench differs from the user top testbench by having a more narrow scope, only focusing on testing the signal processing. The testbench also differs from the user top testbench by relying heavily on external scripts in Python to do the testing, rather than standard VHDL checking. These scripts generate radar data and validate the simulation results in Python, by using VUnit to pass data to and from the testbench. This leverages the fact that Python is a programming language with a higher abstraction level and better suited for handling math than VHDL, which is ultimately a hardware description language (HDL).

Testbench setup

An overview of the testbench is shown in Figure 5.2. It consists of the main process controlling the test execution, as well as the DSP core acting as the DUT (Device Under Test). It also contains several other concurrent processes to apply and read stimuli. These are controlled by the main process. Two avalon_mm_master and avalon_mm_slave processes also provide the main process with access to the DSP core's registers. The testbench uses VUnit's integer_array_t for reading simulation data from files and writing the results back. The type has built-in functions to load and write data to and from files with comma-separated values (CSV). The VHDL simulation can be broken into the following steps:

1. The design and testbench are loaded into the simulator. CSV data for the RX and TX signal is loaded from file using the integer_array_t type.
2. The main process starts. It asserts the reset signal for several clock cycles to put the DUT in a known state.
3. The main process uses the avalon_mm_master process to provide the DUT with the chirp length. This is necessary when the applied chirp is shorter than the maximum chirp length, to prevent the dechirper from expecting too long of a chirp.
4. The radar trigger signal is asserted by main. This signals the dechirper to start sampling the TX signal for constructing the matched filter.
5. The main process then signals the stimuli_tx and stimuli_rx processes to concurrently apply the data on the DUTs input ports, mimicking the transmitted and received radar signal.
6. Simultaneously, the main process signals the save_output process to concurrently monitor the **iq_valid_out** signal and record the output from the **i_data_out** and **q_data_out** ports. These are converted from bit vectors to integer types and stored in a 2D integer_array_t.
7. The main process waits until the stimuli_tx and stimuli_rx processes signal that they are done applying stimuli. The main process then stops the save_output process.
8. Finally, main saves the output data to CSV file using the integer_array_t type built-in function and ends the simulation.

In the event of the testbench becoming stuck, the testbench execution will be terminated by the watchdog process after 1 ms of simulated time.

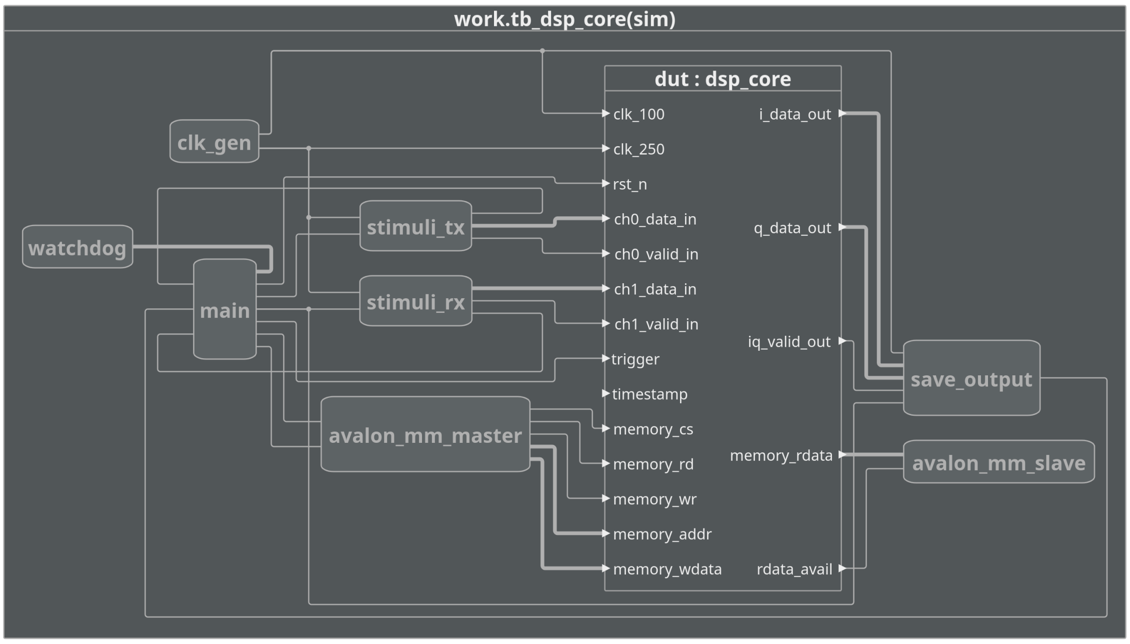


Figure 5.2: Diagram of the DSP core testbench.

In order to provide the simulation with data, as well as extract the result, VUnit allows pre-config and post-check functions to be registered for a testbench. These are callback functions that are automatically called by VUnit before and after the simulation, respectively. Part of the VUnit run.py file is given in Listing 5.2. The complete file is given in Appendix C.

```
# Add pre_config and post_check to testbenches
tb_dsp_core = lib.test_bench("tb_dsp_core")
tb_dsp_core.set_pre_config(dsp_core_sim.pre_config)
```

```python
import numpy as np
import scipy.signal as signal
import matplotlib.pyplot as plt

def pre_config(output_path):
    # Create TX signal
    chirp = signal.chirp(t=t, f0=f0, t1=t[-1], f1=f1, method='linear')
    tx_signal = chirp * (2 ** 5) / chirp.max() # Scale chirp to 5 bits

    # Create RX signal
    channel = signal.unit_impulse(10000, 'mid') # 10 km @ 1 m resolution
    rx_signal = signal.fftconvolve(channel, tx_signal, mode='same')

    # Save data to file
    np.savetxt(output_path + 'tx_signal.csv', tx_signal.as_type(np.int32))
    np.savetxt(output_path + 'rx_signal.csv', rx_signal.as_type(np.int32))

    return True

def post_check(output_path):
    # Load sim data from CSV file
    csv_data = np.loadtxt(os.path.join(output_path,file_name),
        delimiter=",")
    # Convert data to np.complex128
    complex_sim_output = csv_data[:,0] + 1j*csv_data[:,1]

    # Plot results
    plt.plot()
    plt.title('Expected␣correlation')
    plt.plot(complex_sim_output.real, label='I-channel')
    plt.plot(complex_sim_output.imag, label='Q-channel')
    plt.plot(np.abs(complex_sim_output), label='Magnitude')
    plt.show()

    return True
```

Listing 5.3: Simplified callback functions for the DSP core testbench

```python
tb_dsp_core.set_post_check(dsp_core_sim.post_check)
tb_dsp_core.set_sim_option('modelsim.init_file.gui', './dsp_core/tb/wave.do')
```

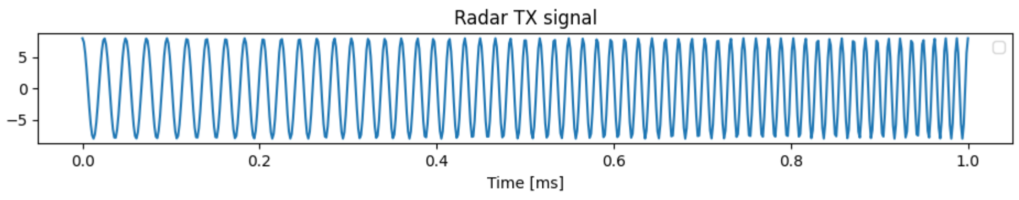
Listing 5.2: Using VUnit to add pre- and post-simulation callback functions for the DSP Core testbench.

The callback function simply utilizes the scipy.signal library's chirp() function to create the radar TX signal. The RX signal is created by convolving the signal with a Dirac pulse, representing a single scatterer. Some white Gaussian noise can be added to the RX signal as well to test the performance of the pulse compression. The post-sim callback function only plots the result using Matplotlib. A simplified version is given in Listing 5.3.

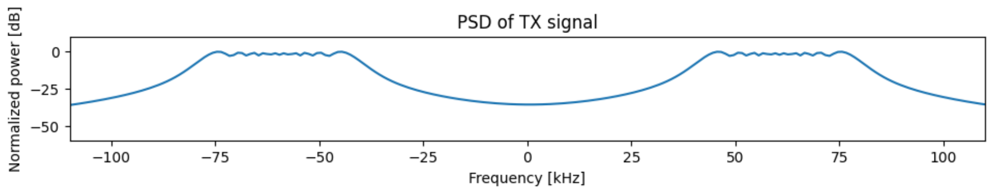
For the test a 1 ms chirp with bandwidth 40 kHz and center frequency 60 kHz is generated with Python, which approximately matches the compiled in filter length of 256 samples after decimation. The chirp is shown in Figure 5.3. Convolving the simple channel model with the TX signal yields the RX signal shown in Figure 5.4.

5.4.1 Simulation results

Figure 5.5 shows the simulation output. The peak in the simulated output is slightly skewed to the left. This is likely due to the filter in the dechirper having to be filled before starting to output samples. The output is a complex valued signal, but the magnitude shows a distinct peak where the vessel is. The peak is not as distinct in the simulation, likely because of slight filter mismatch and quantization effects.

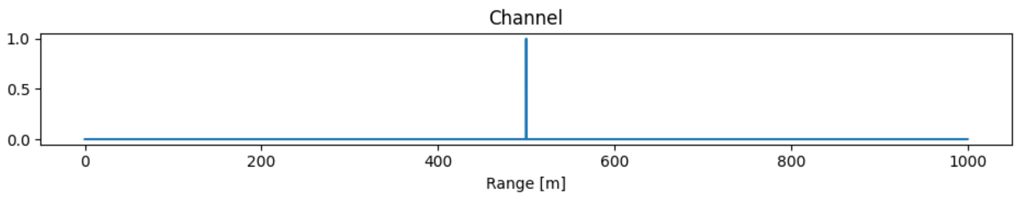


(a) Time domain plot of the simulated TX signal.

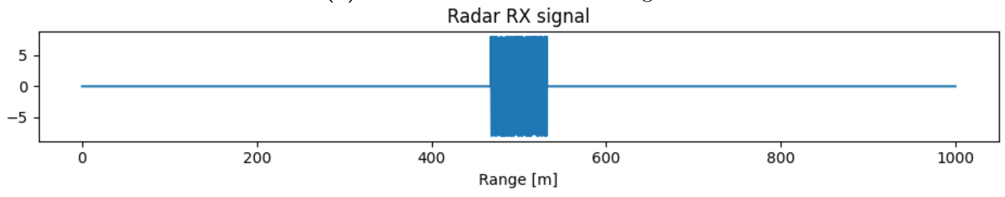


(b) PSD of the simulated TX signal.

Figure 5.3: Time domain and spectrogram plots of a 1 ms chirp with bandwidth 40 kHz and center frequency 60 kHz used for testing the design.



(a) PSD of the simulated TX signal.



(b) Time domain plot of the simulated TX signal.

Figure 5.4: Simple channel model and resulting RX signal generated by convolving the TX signal with the channel.

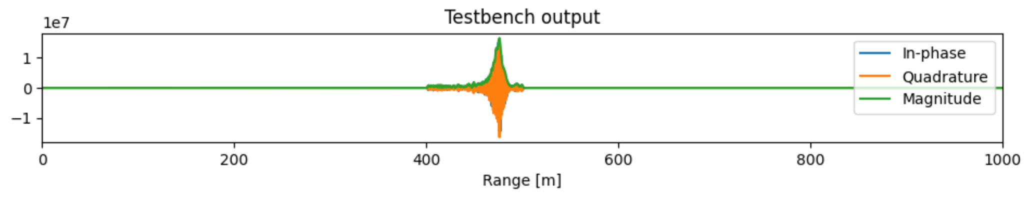
5.5 Physical test

A physical testbench is needed to verify whether the design works as intended in hardware or not. Simulation is the most important verification tool since it gives the designer the most information. It also has the fastest turnover time, as simulation does not involve the often lengthy process of compiling the design for the FPGA. Simulation models do however have limited coverage, operate under conditions and assumptions set by the designer and cannot accurately recreate the operating conditions. For instance, both the ADC and HPS present a considerate complexity in the system and are not part of the simulation. Physical tests can therefore be used to verify these, and indicate to the designer which testbenches need to be further developed to fix a bug.

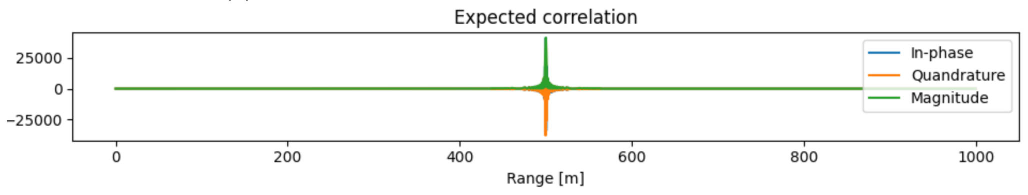
The goal of the physical test is to verify whether the reference design and user top entity work together. It is also desired to demonstrate that both the dechirping and debug interface work as intended.

5.5.1 Test setup

The test setup is shown in Figure 5.6. The iWave Arria 10 development kit is set up as the DUT. It is connected to a test computer, which runs a virtual machine with a digital oscilloscope provided by Arrow and Analog Devices as part of the reference design [20]. The HPS will stream samples from the FPGA fabric to the scope via Ethernet. The test computer also runs a simple UART interface written in Python, parsing frames to enable register access for debugging. It can be used



(a) Time domain plot of complex simulation output signal.



(b) Expected simulation output.

Figure 5.5: Expected and simulated output of the DSP Core under simulation.

to read and write frames as described in subsection 4.2.1. The interface is provided in Appendix D. A Digilent PMOD USBUART module is used to provide the fabric with UART.

To mimic the functionality of the radar, a TX, RX and trigger signal is needed. These are provided with a 2.4 GS/s Siglent SDG6022X Arbitrary Waveform Generator. This is capable of generating the chirp waveforms with its 200 MHz bandwidth. The radar trigger must be pulsed at the same time as the TX signal is transmitted. Because the waveform generator only has two outputs, one channel will be used to create the chirp signal, while the other channel will pulse the radar trigger for every second chirp. That way the first chirp will be registered by the DUT as the TX signal, while the second chirp will be registered as the reflected signal.

It is expected that a correlation peak can be observed briefly in the signal scope when the RX signal overlaps with the matched filter in the dechirper.

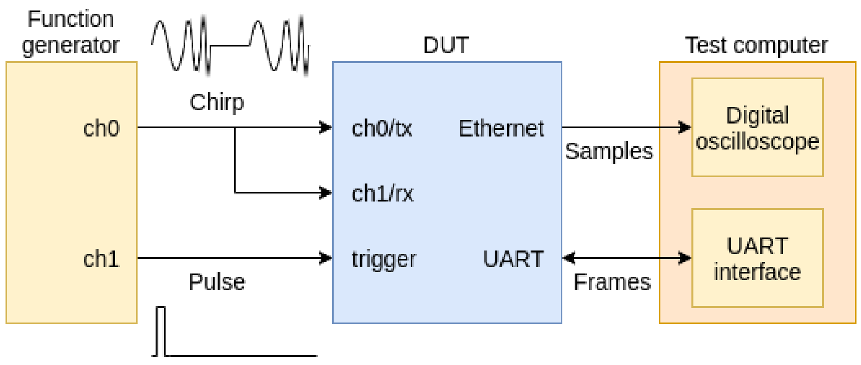


Figure 5.6: Block diagram of the physical test setup, using a function generator to create radar waveforms.

5.5.2 Register access test

The UART interface is tested by transmitting a frame to the FPGA via the PMOD USBUART component, requesting a read from the debug value register at address 0x0000. This returns the debug value, 0xBEEF, as expected. Write is tested by writing 0xCAFE to the write register in the debug module, located at address 0x0001. The value can be successfully read back.

Trying to change the baud rate to an incorrect one causes the DUT to respond with an empty frame containing the stop bit error flag, as seen in Figure 5.7. The stop bit flag can be observed in the second bit from the left of the first byte. Note the reversed bit order employed in the UART connection. The stop bit also causes the error LED to turn on, as can be seen in Figure 5.8.

These tests successfully verify that the UART debug interface works as intended for accessing the FPGA registers by performing both read and write operations. Furthermore, it is proven that the stop bit error causes a response frame and causes the error LED indicator to turn off.

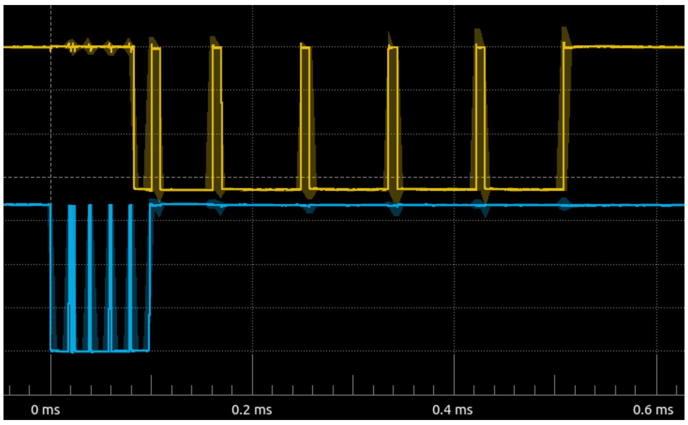


Figure 5.7: Mismatching baudrates causing a stop bit error, triggering the transfer of a frame with the according flag (orange). The stop bit flag can be observed in the second bit of the first byte. Note the reversed bit order.

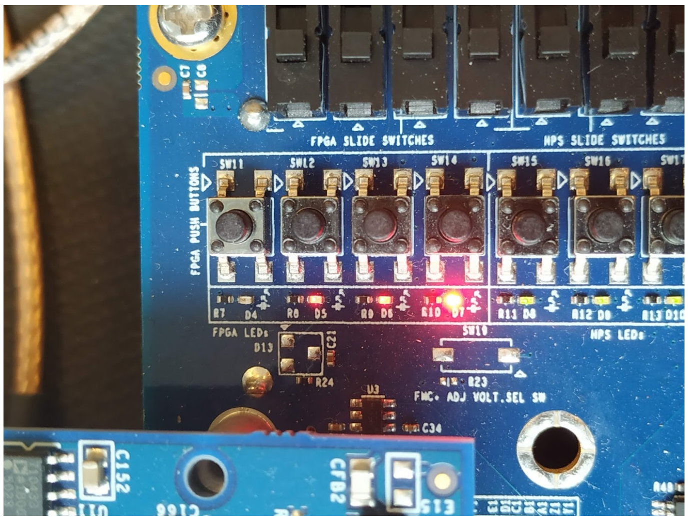


Figure 5.8: Mismatching baudrates causing a stop bit error, turning off the error LED routed to diode D7 on the iWave carrier card (rightmost diode).

5.5.3 Dechirping test

Stimulus is applied using the waveform generator as described above. The software oscilloscope connects to the HPS without errors, but does not display any signals. Using the UART debug interface it is revealed that the ADC valid signals have never been active. To further debug the design, the Intel Signal Tap IP is compiled in. This provides a logic analyzer that can read signals directly from the FPGA fabric via the on-chip USB Blaster. This further reveals that the 250 MHz clock delivered by the ADC interface does not tick, indicating an error with the reference design.

The system log in the Linux kernel running on the HPS reveals that the HPS driver has trouble accessing the fabric via the HPS-FPGA interconnect. This probably makes the HPS driver to unable to properly set up the ADC, causing the ADC interface to stop its output clock.

Due to the described error, it remains inconclusive whether the signal processing performs as intended in hardware.

Chapter 6

Discussion

This chapter provides a brief evaluation of various key elements from the design process. A brief overview over future work is also given.

6.1 Evaluation

6.1.1 Radar selection

The Simrad Halo 20+ pulse compression radar considered for the project has proven itself to be source of uncertainty for the development. Measuring and characterizing the waveforms showed that the radars actual sweep bandwidth was 30 MHz, in contrast to the 48 MHz listed in the manual. Furthermore, it is unknown why the radar only performs upchirps and not downchirps, as discussed in Section 2.3. As both the technical specifications and internal workings of the radar are unknown, it has been hard to reverse engineer the radar for the use in this project.

Making use of a simpler radar and extracting clearer requirements would most likely have made the implementation of the system easier. Alternatively, one could consider to implement a simple radar using the DAC on the AD-FMCDAQ2 component in addition to the ADC. The FPGA could handle the needed signal processing to create the radar waveforms.

6.1.2 Hardware selection

The FPGA and ADC selected for the project offered a flexible development platform. The reference design provided with the evaluation kit significantly shortened the development time by providing complete ADC interfaces and a working HPS implementation. The JESD204 interface for the ADC was however not completely documented by Analog Devices on the Arrow Wiki [20]. This was however complemented by Arrow Electronics offering support on the development kit.

Having such a large FPGA provides a good base that should cover all future development needs. Due to the low quantity of FPGAs necessary the cost of using a high-end FPGA should be negligible, especially as it is compatible with the ADC.

6.1.3 Verification toolchain and continuous integration

VUnit has shown itself to be a powerful tool, both for traditional checking, but also for leveraging the simulation workflow. While it is somewhat less mature than UVVM in the selection of BFMs, as discussed in Section 5.1, the selection was good enough for this project, requiring only UART, Avalon and a FIFO.

Using continuous integration (CI) for automatically executing the testbenches upon project changes proved to be very helpful. Testing often made it possible to quickly detect and fix errors that appeared after doing small design changes, such as changing signal widths. The full benefit of using CI could not be utilized, as some testbenches were not self checking. This was the case for the DSP core testbench, which relied on the designer manually reviewing the output plots. As the CI pipeline relied on using GHDL some testbenches could not be ran due to Verilog dependencies.

This can however be solved by setting up a GitLab local runner to execute the testbenches on a local computer with Modelsim, rather than relying on the GitLab cloud.

The use of CI can be further leverages by using formal verification. This can be used to automatically verify that the system as a black box performs according to formal requirements, but also requires that the system is more strictly specified.

6.1.4 Physical test

The physical test of the signal processing was unsuccessful due to a driver issue. However, due to the extensive simulation and passing compilation the FPGA design is expected to work as intended. This is further proved by the debug interface working as intended.

6.1.5 Data rates

The reqired uplink data rate of the system was estimated to a minimum of 0.25 Gbit/s in Section 3.4. However, to add design overhead and flexibility this estimate was increased towards 1.0 Gbit/s. As the ITU listed only the minimal requirement and Telia only listed their minimum data rate, measurements must be done to verify what rates can be expected under various conditions. This is however expected to be difficult to achieve, as the proposed solution in subsection 3.5.2 was estimated to achieve between 300 Mbit/s and 840 Mbit/s. The estimated data rates also don't take the increased dynamic range of the radar signal into consideration, which is currently not part of the systems specification.

This suggests that it will be difficult to get the system to perform as intended, and that more signal processing must be done to reduce the data rate.

6.2 Future work

6.2.1 Timing analysis and timing violations

Proper timing analysis and optimizations were largely skipped to create a minimum viable product in time. As the design grows, place and route will struggle and fail more due to the occupied area on the FPGA increasing. This in turn increases the distance the signals potentially have to cross ion one clock cycle, increasing the change of setup time violations on the registers. The Quartus Timing Analysis already identifies the design to be unstable in its multicorner timing analysis. The design frequently fails the Slow 0 °C Timing Model. This particular corner case is identified by excepting the slowest silicon performance for the specific FPGA's speed grade, combined with slower delays caused by temperature inversion of the transistors due to low temperature. It also expects the lowest possible core voltage. While failing one of the corner cases doesn't stop the design from working, it provides another uncertainty to both debugging and operation of the system. [11]

6.2.2 DSP block optimization

The design ended up consuming most of the available DSP blocks on the FPGA. This not only makes it hard to add more signal processing to the design, but also requires the routing tool to utilize DSP blocks that are spread far apart. The main consumers of DSP blocks are the IQ demodulator and the dechirper through construction of filters. As these entities perform processing on both I- and Q-channels of complex signals, any optimization of the number of DSP blocks will have twice the payoff, which doubles the amount of multipliers available. As the DSP blocks are optimized FIR filters, a more careful implementation could also save adders by making use of the multiply-accumulate functionality of the DSP blocks, shown in Figure 6.1. This is called the 18x18 Sum 2 Mode, which in addition to calculating $result_a$ and $result_b$ also adds them in the same clock cycle.

The IQ demodulator uses DSP blocks for it's internal halfband filter. The demodulator is used for both the TX and RX signal, which additionally contains one half band filter for both the I- and Q-channels of these signals. Removing one DSP block in the filter therefore results in freeing four DSP blocks in the design. The halfband filter already uses half the number of multipliers than a conventional lowpass filter, as discussed inn subsection 4.3.2. It is however possible to reduce the

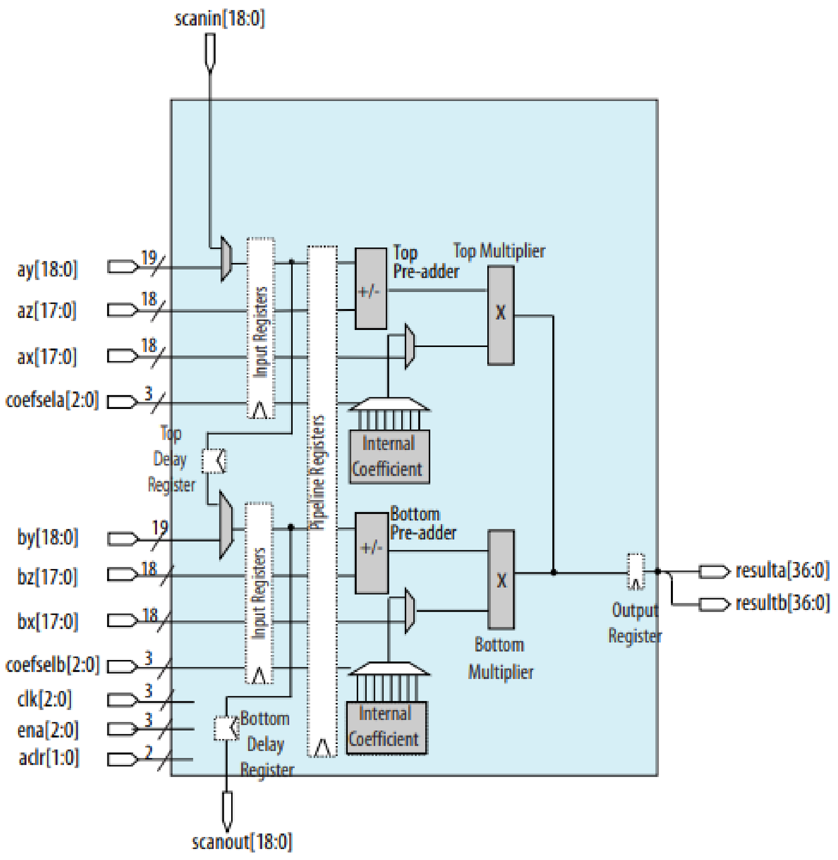


Figure 6.1: Block diagram of an Arria 10 SoC DSP block.

filter order by carefully examining how much dampening is necessary to avoid aliasing. Furthermore, the Matlab implementation of the filter is somewhat underoptimized by not being targeted at a specific FPGA. It also runs at a fairly high 250 MHz, placing it higher at risk for timing errors.

The dechirper is by far the largest consumer of DSP blocks due to the high time-bandwidth product of the chirp. Due to the clock domain crossing described in subsection 4.3.1, the data is clocked at a rate much slower than the DSP blocks can be ran. Figure 6.2 shows a solution where the DSP block performs the multiply and accumulate (MAC) at a much higher clock speed than the surrounding logic. Two multiplexers are used to feed samples and filter coefficients to the DSP block. Clocking the data at 100 MHz while running the DSP block at 400 MHz uses four times less multipliers. Clocking the data at 50 MHz could use eight times less, but at the expense of setting up and managing yet another clock domain. The DSP block can take in multiple clock sources and perform the clock domain crossing from fast to slow at the output register [14]. It should be noted that the fabric surrounding the DSP block must be able to run a 3-bit counter and multiplexers at 400 MHz without timing violations. The Arria 10 FPGA fabric is however rated for this [12].

6.2.3 HPS driver

The development has been mostly focused on creating and verifying the FPGA fabric design. Therefore the HPS has been neglected and only used as part of the reference design build. The project lacks a HPS driver that reads data from the FPGA fabric and transfers it to the processing hub, preferably with a UDP socket via the onboard Ethernet connector. This not only constitutes creating a Linux kernel driver, but also to potentially modify the fabric-HPS interconnect and build a Linux kernel with an updated device tree.

6.2.4 More processing on the FPGA

The FPGA provides a powerful platform to do real-time processing of high bandwidth signals. Depending on future specifications of the overall system, the FPGA can implement more signal

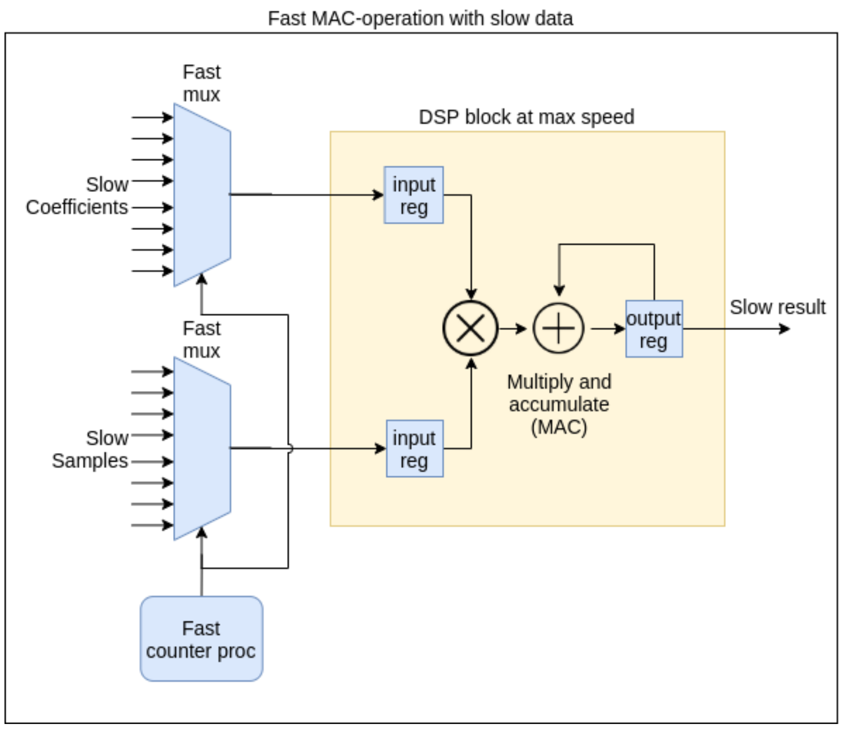


Figure 6.2: Proposed optimization to reduce the number of DSP blocks used in the design, by having the DSP block perform multiply and accumulate (MAC) at a much higher clock speed than the surrounding logic. A counter and two multiplexers are used to feed samples and filter coefficients to the DSP block running at 400 MHz. The data is clocked by a much slower 50 MHz.

processing to relax the computational load of the processing hub and the data rate of the communication link. For instance, target detection could be performed on-site and sent to the processing hub for comparison with the other sites. Such optimizations could include pulse integration of the radar signal to increase the SNR and decrease the output data rate [33], or the use of compression algorithms of the processed signal.

6.2.5 Build tools

The FPGA part of the project already employs continuous integration (CI) for verifying the design. This could easily be expanded to include continuous deployment (DO) by expanding the pipeline to compile the design and program the FPGA. This saves time for the designer by automatically using the latest gateware for testing, as well as providing the latest resource and timing reports. This can be further expanded to automatic hardware tests, as well a building the Linux kernel.

6.2.6 Python model

While the Python models employed by VUnit test the functionality reasonably well, they can be further improved to give performance indications such as SNR. They can also be used for formal verification by asserting whether the design has the necessary performance, provided that such specifications are made. Formal verification could be done on the system level by adding channel effects such as noise, clutter and fading and measuring output SNR or probability of detection.

Chapter 7

Conclusion

A design was proposed where radar data was sampled from a radars IF stage. Measuring and characterizing the radar signals of the Simrad Halo 20+ Pulse Compression radar revealed that the radar transmitted bursts of pulses and chirps. The number of pulses and chirps, as well as the latter's length, bandwidth and frequency varied based on the radars range current setting. A design proposal was made to use an FPGA to sample both the radars RX and TX signals. The TX signal could then be used to implement a matched filter, which performs the pulse compression on the received RX signal. Sampling the TX signal takes into account the varying waveform characteristics for each transmission, which made it possible to update the matched filter for each transmission.

The iWave systems Arria 10 SoC development kit along with the Analog Devices FMCDAQ2 evaluation board were selected for this project. These have proven to be a capable and flexible development platform, providing the ability to sample and process up to 1 GS/s in real-time. Basing the remote site data acquisition system on an FPGA also has the benefit of making it possible to alter and update the signal processing. A suitable 5G modem was also found, but initial data rate calculations revealed that the data rate is currently too high to be transmitted. More signal processing should be added to the system to reduce the data rate further.

The proposed design was implemented in the FPGA fabric using VHDL. Simulations show that the design works, successfully performing IQ demodulation on the RX and TX signals before performing the dechirping the transmitted signal. The chirp length was reduced to 256 samples due to consuming all available DSP cores in the FPGA. It was not possible to verify this in hardware due to an error occurring in a driver related to the FPGA-HPS interconnect.

The project also successfully made use of a continuous integration (CI) pipeline to automatically run all testbenches on design changes. This was done by using the VUnit testing framework and the open source GHDL simulator. The project also made use of VUnit to transfer simulation data to and from the simulation. This successfully enabled rapid verification of signal processing using the high-level programming language Python rather than VHDL.

The project still lacks customization of the HPS to move the resulting data from the FGPA to the HPS and transmit it to the processing hub. Further optimizations should also be done on the fabric design to reduce the consumption of DSP blocks. Specifications of the overall system should be done before proceeding with further development.

Bibliography

[1] International Telecommunication Union (ITU). *Minimum requirements related to technical performance for IMT-2020 radio interface.* URL: https://www.itu.int/dms_pub/itu-r/opb/rep/R-REP-M.2410-2017-PDF-E.pdf.

[2] The Norwegian Coastal Administration. *Allows for testing of autonomous ships.* Sept. 30, 2016. URL: https://www.kystverket.no/Nyheter/2016/september/apner-for-test-av-autonome-skip/.

[3] Adresseavisen. *Til sommeren vil denne verdensnyheten bli fast inventar i Trondheim.* URL: https://www.adressa.no/pluss/okonomi/2020/11/28/Til-sommeren-vil-denne-verdensnyheten-bli-fast-inventar-i-Trondheim-23063011.ece.

[4] Navico Holding AS. *Halo Pulse Compression Radar Installation Manual.* 2015. URL: simrad-yachting.com.

[5] Telia Norge AS. *5G bridrar til utviklingen av autonome skip.* URL: https://www.telia.no/magasinet/5g-fra-telia-gjor-at-denne-fergen-kan-seile-uten-kaptein/.

[6] Celerway. *Celerway Arctus.* URL: https://www.celerway.com/product/celerway-arcus.

[7] RS Components. *Digilent GPS Expansion Module 410-237.* URL: https://no.rs-online.com/web/p/sensor-development-tools/1346455/.

[8] Intel Corporation. *Avalon Interface Specifications.* May 26, 2020. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf?language=en_US.

[9] Intel Corporation. *Cyclone V FPGAs and SoC FPGAs.* URL: https://www.intel.com/content/www/us/en/products/details/fpga/cyclone/v.html.

[10] Intel Corporation. *FIR II IP Core: User Guide.* URL: https://www.intel.com/content/www/us/en/programmable/documentation/hco1421694595728.html#mwh1409958274502.

[11] Intel Corporation. *Guaranteeing Silicon Performance with FPGA Timing Models.* URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01139-timing-model.pdf.

[12] Intel Corporation. *Intel Arria 10 Device Datasheet.* Oct. 20, 2020. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/a10_datasheet.pdf.

[13] Intel Corporation. *Intel Arria 10 Device Datasheet.* Mar. 29, 2021. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-qpp-design-recommendations.pdf.

[14] Intel Corporation. *Intel Arria 10 Native Fixed Point DSP IP Core User Guide.* Oct. 20, 2020. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_nfp_dsp.pdf.

[15] Intel Corporation. *Intel Arria 10 SoC Device Design Guidelines.* URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an-a10-soc-device-design-guidelines.pdf.

[16] Analog Devices. *AD-FMCDAQ2-EBZ HDL Reference Design.* URL: https://wiki.analog.com/resources/eval/user-guides/ad-fmcdaq2-ebz/reference_hdl.

[17] Analog Devices. *AD9680 Datasheet.* URL: https://www.analog.com/media/en/technical-documentation/data-sheets/AD9680.pdf.

[18] Dustin. *Teltronika RUTX12.* URL: https://www.dustin.no/product/5011188022/.

[19] Norwegian University of Science Egil Eide and Technology. "Radar network in the Trondheim Fjord (proposal)". In: (2020).

[20] Arrow Electronics. *AD FMCDAQ2 Platform FPGA Architecture.* URL: https://github.com/ArrowElectronics/arrow-adi-intel-psg/wiki/AD-FMCDAQ2-Platform-FPGA-Architecture.

[21] Arrow Electronics. *Analog Devices AD-FMCDAQ2-EBZ*. URL: https://www.arrow.com/en/products/ad-fmcdaq2-ebz/analog-devices?q=AD-FMCDAQ2-EBZ.

[22] Arrow Electronics. *iWave Systems IW-G24D-CU2F-4E002G-S008G-LCM*. URL: https://www.arrow.com/en/products/iw-g24d-cu2f-4e002g-s008g-lcm/iwave-systems-technologies-pvt-ltd.

[23] UC Berkeley College of Engineering. *EECS150: Interfaces: "FIFO" (a.k.a. Ready/Valid)*. URL: https://inst.eecs.berkeley.edu/~cs150/Documents/Interfaces.pdf.

[24] GitHub. *UVVM Light*. URL: https://github.com/UVVM/UVVM_Light.

[25] Github.com. *pyFDA*. URL: https://github.com/sp1nelson/pyFDA.

[26] John Kirkhorn. *Introduction to IQ demodulation of RF-data*. URL: https://folk.ntnu.no/htorp/Undervisning/TTK10/IQdemodulation.pdf.

[27] Silicon Labs. *AN118 IMPROVING ADC RESOLUTION BY OVERSAMPLING AND AVERAGING*. URL: https://www.silabs.com/documents/public/application-notes/an118.pdf.

[28] Kongsberg Maritime. *SITUATIONAL AWARENESS*. URL: https://www.kongsberg.com/no/maritime/about-us/news-and-media/our-stories/intelligent-awareness/.

[29] MathWorks. *FIR Halfband Filter Design*. URL: https://se.mathworks.com/help/dsp/ug/fir-halfband-filter-design.html.

[30] Nandland.com. *Crossing Clock Domains in an FPGA*. URL: https://www.nandland.com/articles/crossing-clock-domains-in-an-fpga.html.

[31] The Research Council of Norway. *Norsk veikart for forskningsinfrastruktur*. URL: https://www.forskningsradet.no/sok-om-finansiering/midler-fra-forskningsradet/infrastruktur/norsk-veikart-for-forskningsinfrastruktur/.

[32] DSP Related. *HALFBAND FILTER DESIGN WITH PYTHON/SCIPY*. URL: https://www.dsprelated.com/showcode/270.php.

[33] Mark A. Richards. *Principles of Modern Radar: Basic Principles*. SciTech Publishing Inc, May 1, 2010.

[34] Norwegian University of Science and Technology. *Autoferry*. URL: https://www.ntnu.edu/autoferry.

[35] Norwegian University of Science and Technology. *OceanLab Node2*. URL: https://www.ntnu.edu/oceanlab/node2.

[36] SciPy.org. *scipy.signal.remez*. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.remez.html.

[37] Francesco Sermi et al. "Analysis of the radar coverage provided by a maritime Radar Network of Co-operative Vessels based on real AIS data". In: *2013 European Radar Conference*. 2013, pp. 251–254.

[38] VUnit. *VUnit Open Source Documentation*. URL: https://vunit.github.io/.

Appendix A

Avalon MM slave process

Example of the Avalon MM slave process giving the address decoder read and write access to module registers. This excerpt is taken from the debug module.

```vhdl
register_access : process(clk)
    variable v_memory_rdata : std_logic_vector(DATA_W - 1 downto 0) := (others => '0');
    variable v_rdata_avail  : std_logic;

begin
    if rising_edge(clk) then

        if rst_n = '0' then
            memory_rdata <= (others => '0');
            rdata_avail  <= '0';

            -- Assign default values
            debug_value <= C_DEBUG_VALUE;
            write_reg   <= C_WRITE_REG;
            blink_led   <= '1';
        else

            -- Always reset variable
            v_memory_rdata := (others => '0');
            v_rdata_avail  := '0';

            -- Read from register
            if (memory_cs and memory_rd) = '1' then
                v_rdata_avail := '1';
                case to_integer(unsigned(memory_addr)) is
                    when C_ADDR_DEBUG_VALUE => v_memory_rdata := debug_value;
                    when C_ADDR_WRITE_REG   => v_memory_rdata := write_reg;
                    when C_ADDR_BLINK_LED   => v_memory_rdata(v_memory_rdata'low) :=
                        blink_led;
                    when C_ADDR_COMMS_ERROR => v_memory_rdata(v_memory_rdata'low) :=
                        comms_error_sticky;
                    when others => -- Do nothing
                        v_rdata_avail := '0';
                end case;

            -- Write to registers
            elsif (memory_cs and memory_wr) = '1' then
                case to_integer(unsigned(memory_addr)) is
                    when C_ADDR_DEBUG_VALUE => debug_value <= memory_wdata;
                    when C_ADDR_WRITE_REG   => write_reg <= memory_wdata;
                    when C_ADDR_BLINK_LED   => blink_led <= memory_wdata(memory_wdata'low
                        );
                    when others => -- Do nothing
                end case;

            end if;
            -- Set signal to variable value
            memory_rdata <= v_memory_rdata;
            rdata_avail  <= v_rdata_avail;
        end if;
    end if;
end process;
```

Listing A.1: Process for providing a module with Avalon MM register access.

Appendix B

IQ Demodulator Sample Sorter

```
sample_sorter : process(clk) is
begin
    if rising_edge(clk) then
        if rst_n = '0' then
            sample_counter  <= 0;
            i_unfiltered    <= (others => '0');
            q_unfiltered    <= (others => '0');
            filter_valid_in <= '0';
        else

            -- Single cycle bit
            filter_valid_in <= '0';

            if valid_in = '1' then

                if sample_counter = 0 then
                    i_unfiltered <= real_in;
                    q_unfiltered <= (others => '0');

                elsif sample_counter = 1 then
                    i_unfiltered <= (others => '0');
                    q_unfiltered <= real_in;

                elsif sample_counter = 2 then
                    i_unfiltered <= -real_in;
                    q_unfiltered <= (others => '0');

                elsif sample_counter = 3 then
                    i_unfiltered <= (others => '0');
                    q_unfiltered <= -real_in;

                end if;

                filter_valid_in <= '1';

                -- Increment counter
                if sample_counter = 3 then
                    sample_counter <= 0;
                else
                    sample_counter <= sample_counter + 1;
                end if;
            end if;
        end if;
    end if;
end process;
```

Listing B.1: Simplified process for sorting the input samples.

Appendix C

VUnit run file

```python
import logging
logger = logging.getLogger(__name__)

logging.basicConfig(level=logging.INFO)

from vunit import VUnit

from chirp_radar import ChirpRadar
radar = ChirpRadar()

import dechirper.tb.dechirper_sim as dechirper_sim
import iq_demodulator.tb.iq_demod_sim as iq_demod_sim
import dsp_core.tb.dsp_core_sim as dsp_core_sim


# Create VUnit instance by parsing command line arguments
vu = VUnit.from_argv()
vu.enable_location_preprocessing()
vu.add_verification_components()

# Create library 'lib'
lib = vu.add_library("lib")

# Add all files ending in .vhd in current working directory to library
lib.add_source_files("./*/src/*.vhd")
lib.add_source_files("./*/tb/*.vhd")


# Altera dependencies for DCFIFO
altera = vu.add_library("altera")
altera.add_source_files("/opt/intelFPGA_lite/19.1/quartus/libraries/vhdl/altera/*.vhd")

altera_ver = vu.add_library("altera_ver")
altera_ver.add_source_files("/opt/intelFPGA_lite/19.1/quartus/eda/sim_lib/
    altera_primitives.v")

altera_mf = vu.add_library("altera_mf")
altera_mf.add_source_files("/opt/intelFPGA_lite/19.1/quartus/libraries/vhdl/altera_mf/*.
    vhd")

altera_mf_ver = vu.add_library("altera_mf_ver")
altera_mf_ver.add_source_files("/opt/intelFPGA_lite/19.1/quartus/eda/sim_lib/altera_mf.v"
    )

# Add Intel simulation model for clock domain crossing FIFO (DCFIFO)
cdc_fifo_fifo_180 = vu.add_library("cdc_fifo_fifo_180")
cdc_fifo_fifo_180.add_source_file("./cdc_fifo/fifo_180/sim/cdc_fifo_fifo_180_jn2auoq.v")

cdc_fifo = vu.add_library("cdc_fifo")
cdc_fifo.add_source_file("./cdc_fifo/sim/cdc_fifo.vhd")


# Add pre_config and post_check to testbenches
tb_dechirper = lib.test_bench("tb_dechirper")
tb_dechirper.set_pre_config(dechirper_sim.pre_config)
tb_dechirper.set_post_check(dechirper_sim.post_check)
tb_dechirper.set_sim_option('modelsim.init_file.gui', './dsp_core/tb/wave.do')

tb_iq_demod = lib.test_bench("tb_iq_demod")
tb_iq_demod.set_pre_config(iq_demod_sim.pre_config)
tb_iq_demod.set_post_check(iq_demod_sim.post_check)

tb_dsp_core = lib.test_bench("tb_dsp_core")
```

```python
tb_dsp_core.set_pre_config(dsp_core_sim.pre_config)
tb_dsp_core.set_post_check(dsp_core_sim.post_check)
tb_dsp_core.set_sim_option('modelsim.init_file.gui', './dsp_core/tb/wave.do')

# Run vunit function
vu.main()
```

Appendix D

Python UART interface

```python
import os
import logging
import serial

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s')

logger = logging.getLogger(__name__)

BYTES_PER_FRAME = 5
SERIAL_PORT = '/dev/serial/by-id/usb-FTDI_FT232R_USB_UART_A904DP5X-if00-port0'
BAUDRATE = 115200

register_adresses = {
    # About module
    'git_hash': 0x0000,
    'build_date': 0x0001,
    'build_time': 0x0002,
    # Debug Register
    'debug_register': 0x0100,
    'write_reg': 0x0101,
    'blink_led': 0x0102,
    'comms_error': 0x0103,
    # DSP Module
    'chirp_length': 0x0200,
    'decimation_rate': 0x0201,
    'ch0_data': 0x0202,
    'ch1_data': 0x0203,
    'ch0_was_valid': 0x0204,
    'ch1_was_valid': 0x0205,
    'radar_trigged': 0x0206,
    'ch0_iq_demod_was_valid': 0x0207,
    'ch1_iq_demod_was_valid': 0x0208,
    'dechirper_was_valid': 0x0209,
    'decimator_was_valid': 0x0210,
}

class FpgaInterface:

    def __init__(self):
        self.ser = serial.Serial(port=SERIAL_PORT, baudrate=BAUDRATE, timeout=1)

    def write_register(self, addr, data):
        """Write data to register"""
        tx_frame = _encode_frame(addr=addr, data=data, write=True)
        self.ser.write(tx_frame)

    def read_register(self, addr):
        """Read data from register address."""
        # Create and send a read request
        tx_frame = _encode_frame(addr=addr, data=0, write=False)
        self.ser.write(tx_frame)

        # Get response
        rx_frame = self.ser.read(BYTES_PER_FRAME)

        if rx_frame == b'':
            raise IndexError("Zero bytes received from FPGA.")

        addr, data = _decode_frame(rx_frame)
        return addr, data

    def __del__(self):
        """Close serial connection when instance is deleted."""
        self.ser.close()


def _encode_frame(addr, data, write):
    """Create frame. Returns bytearray."""

    frame = bytearray()

    frame += int(write).to_bytes(length=1, byteorder='big')
    frame += addr.to_bytes(length=2, byteorder='big')
    frame += data.to_bytes(length=2, byteorder='big')

    return frame


def _decode_frame(frame):
    """Decodes frame consisting of bytes to integers."""

    try:
        header = frame[0:1]
        addr = frame[1:3]
        data = frame[3:5]
    except IndexError:
        logger.exception("Frame too short.")

    header = int.from_bytes(header, byteorder='big')
    addr = int.from_bytes(addr, byteorder='big')
    data = int.from_bytes(data, byteorder='big')

    print('header',header)
    print('addr', hex(addr))
    print('data', hex(data))
    if header == 3:
        raise RuntimeError("FPGA responded with stop bit error. Check your baudrate.")

    return addr, data
```