Implementation of a Software-Defined Radio
Transceiver on High-Speed Digitizer/Generator

SDR14

Examensarbete utfört i Elektroteknik
vid Tekniska högskolan vid Linköpings universitet
av

Daniel Björklund

LiTH-ISY-EX--12/4583--SE

Linköping 2012
Implementation of a Software-Defined Radio Transceiver on High-Speed Digitizer/Generator SDR14

Examsarbete utfört i Elektroteknik vid Tekniska högskolan i Linköping av

Daniel Björklund

LiTH-ISY-EX--12/4583--SE

Handledare: Amir Eghbali

ISY, Linköpings universitet

Jan-Erik Eklund

SP Devices

Examinator: Håkan Johansson

ISY, Linköpings universitet

Linköping, 30 May, 2012
Implementation of a Software-Defined Radio Transceiver on High-Speed Digitizer/Generator SDR14

This thesis describes the specification, design and implementation of a software-defined radio system on a two-channel 14-bit digitizer/generator. The multi-stage interpolations and decimations which are required to operate two analog-to-digital converters at 800 megasamples per second (MSps) and two digital-to-analog converters at 1600 MSps from a 25 MSps software-side interface, were designed and implemented. Quadrature processing was used throughout the system, and a combination of fine-tunable low-rate mixers and coarse high-rate mixers were implemented to allow frequency translation across the entire first Nyquist band of the converters. Various reconstruction filter designs for the transmitter side were investigated and a cheap implementation was done through the use of programmable base-band filters and polynomial approximation.
Abstract

This thesis describes the specification, design and implementation of a software-defined radio system on a two-channel 14-bit digitizer/generator. The multi-stage interpolations and decimations which are required to operate two analog-to-digital converters at 800 megasamples per second (MSps) and two digital-to-analog converters at 1600 MSps from a 25 MSps software-side interface, were designed and implemented. Quadrature processing was used throughout the system, and a combination of fine-tunable low-rate mixers and coarse high-rate mixers were implemented to allow frequency translation across the entire first Nyquist band of the converters. Various reconstruction filter designs for the transmitter side were investigated and a cheap implementation was done through the use of programmable base-band filters and polynomial approximation.
There are a lot of people I would like to thank for helping me through my time at Linköping University and throughout this thesis work.

First, I would like to thank Per Magnusson, who has perhaps done the most of all to inspire and keep alive my interest in electrical engineering. Thank you for starting me on my path to the field of electronics and to Linköping and SP Devices, for always taking time to answer my questions and for all the good ideas and discussions.

I want to thank all of the employees at SP Devices for giving me the best introduction to the hardware industry that I could possibly have hoped for. In particular, I would like to thank Jan-Erik Eklund for accepting the role of my supervisor at the company, and Per Löwenborg for several useful discussions on filter design.

I would like to thank my supervisor, Amir Eghbali, for his enthusiasm regarding my work during the thesis project. I’d also like to thank Mikael and Johan who also did their thesis at SP Devices for keeping me company and for their help in testing the project results.

Thanks go to my classmates, and especially to Thomas, Ludde, and Joakim who have been excellent lab partners and friends during my education at Linköping University.

Last but not least, I would like to thank my girlfriend Jenny, my parents Anders and Ulla, and my brother Johan for supporting me and for keeping me happy during my time here at the university.
# Contents

1 Introduction  
1.1 Background .................................................. 3  
1.2 Project goal .................................................. 3  
1.3 Overview ...................................................... 4  

2 Theory  
2.1 FIR filters ..................................................... 5  
\hspace{1cm} 2.1.1 Half-band FIR filters .................................. 6  
2.2 Interpolation filters ......................................... 8  
\hspace{1cm} 2.2.1 Upsampling ............................................. 8  
\hspace{1cm} 2.2.2 Polyphase decomposition ............................. 9  
\hspace{1cm} 2.2.3 Half-band interpolation ............................. 11  
2.3 Decimation filters ............................................ 13  
\hspace{1cm} 2.3.1 Downsampling ........................................ 13  
\hspace{1cm} 2.3.2 Polyphase decomposition ............................. 13  
\hspace{1cm} 2.3.3 Half-band decimation ................................ 15  
2.4 Number representation ....................................... 15  
\hspace{1cm} 2.4.1 Quantization noise .................................. 16  
2.5 Quadrature signals ........................................... 17  
\hspace{1cm} 2.5.1 Quadrature mixers .................................. 18  
\hspace{1cm} 2.5.2 Complex-modulated filters ........................... 21  
\hspace{1cm} 2.5.3 Amplitude and phase-shift keying .................... 23  
2.6 DAC reconstruction ........................................... 23  

3 Tools, equipment and methodology  
3.1 SDR14 ......................................................... 29  
3.2 Design software ................................................ 30  
\hspace{1cm} 3.2.1 MATLAB ............................................... 30  
\hspace{1cm} 3.2.2 SP Devices ............................................ 30  
\hspace{1cm} 3.2.3 Xilinx .................................................. 30  
\hspace{1cm} 3.2.4 CORE Generator ..................................... 31  
\hspace{1cm} 3.2.5 Verilog ................................................ 31  
3.3 Xilinx Virtex-6 ................................................. 31
4 Problem analysis

4.1 Product specification ................................................. 33
4.2 Evaluation of possible architectures ............................ 35
  4.2.1 Cascaded Integrator-Comb filtering ......................... 36
  4.2.2 Interpolation bandpass filtering ............................ 40
  4.2.3 Half-band interpolation with image selection ............... 42
  4.2.4 Baseband interpolation and multi-DDS mixer ............... 44
  4.2.5 Bandpass interpolation after fine-tuning mixer .......... 48
  4.2.6 Multi-stage mixer architecture ............................ 49
4.3 Chosen architecture .............................................. 53
4.4 DAC reconstruction filter design ................................ 53
  4.4.1 Inverse-sinc FIR filter at system output ................. 54
  4.4.2 Baseband tilt-compensation ................................. 55
  4.4.3 Amplitude-compensation ................................... 58
  4.4.4 Chosen reconstruction filter architecture ............... 60
4.5 MATLAB modeling ................................................ 60

5 Implementation ....................................................... 65

5.1 System block schematics ........................................... 65
5.2 Interpolation and decimation filter implementation .......... 65
5.3 DAC reconstruction filter implementation ..................... 70
5.4 Mixers ............................................................. 73
  5.4.1 Low-rate high-resolution mixers ......................... 73
  5.4.2 High-rate parallelized coarse mixers ..................... 73
5.5 Scaling and wordlengths .......................................... 78

6 Results ............................................................... 81

6.1 Hardware testing .................................................. 81
6.2 Measurements ..................................................... 84
  6.2.1 Scaling ...................................................... 84
  6.2.2 Inverse-sinc characteristic ............................... 87
6.3 Resource usage ................................................... 89
  6.3.1 Evaluation of resource usage ............................... 90
6.4 Deliverables ....................................................... 90
6.5 Conclusions ....................................................... 93
  6.5.1 Comparison to other systems ............................... 93
  6.5.2 Possible improvements .................................... 94

Bibliography .......................................................... 95
List of abbreviations

- A/D - Analog-to-Digital
- ADC - Analog-to-Digital Converter
- D/A - Digital-to-Analog
- DAC - Digital-to-Analog Converter
- dBFS - Decibels, with respect to full scale amplitude (peak or sinusoidal RMS, depends on application)
- DSP - Digital Signal Processing
- FIR - Finite Impulse Response
- FPGA - Field-Programmable Gate Array
- HDL - Hardware Description Language
- IP - Intellectual Property
- I/Q - In-phase/Quadrature-phase
- LO - Local Oscillator
- LSB - Least Significant Bit
- LUT - Look-Up Table (FPGA contains thousands of programmable LUTs)
- MSB - Most Significant Bit
- MSps - MegaSamples per second
- QAM - Quadrature Amplitude Modulation
- QPSK - Quadrature Phase-Shift Keying
- SNR - Signal to Noise Ratio
- SFDR - Spurious Free Dynamic Range
- SDR - Software-Defined Radio
Chapter 1

Introduction

1.1 Background

SP Devices (which stands for Signal Processing Devices) is a company that since its formation in the year 2004 is based in Mjärdevi, Linköping. SP Devices are concerned mainly with creating technology for improving the performance of analog-to-digital (A/D) components. By using algorithms which correct analog-to-digital converter (ADC) linearity errors, and errors which occur when interleaving several ADCs, the converter performance can be improved drastically. Implementations of these algorithms are sold as intellectual property for usage within customer solutions. SP devices also utilize the algorithms in order to implement very compact digitizer and generator\textsuperscript{1} solutions with high sample rates and accuracy, which are sold as complete hardware solutions.

The latest of these hardware platforms is called SDR14, which is intended for radio communication applications, with two analog inputs and two analog outputs. Inside the field-programmable gate array (FPGA) that controls the hardware of the board, there is an area which is reserved for custom user logic, which is located between the host interface and the converter outputs. This allows the user to implement customized signal processing according to their needs, without needing to change anything in hardware. SDR14 is a fairly recent addition to the product line. Therefore, there existed no significant demo applications at the time when this thesis project was started, which showcase what a customer could do using this user logic space\cite{2}.

1.2 Project goal

The goal of this thesis was to make use of the user logic space of SDR14 to construct a system similar to what the average end-user of the system might need. The idea was that the resulting implementation would be used by the company

\textsuperscript{1}Systems which convert analog inputs to digital data are commonly called digitizers, while systems which convert digital data into analog outputs are called generators

3
both for demonstrating the product to potential customers, and for providing the code along with the SDR14 development kit as a modifiable and reusable implementation example. Since the SDR14 platform is intended for radio applications, the system which SP Devices requested to be constructed was that of a digital up-converter for the digital-to-analog converters (DACs) and a digital down-converter for the ADCs. Such a system should act as a link between the host (i.e., the computer which the platform is connected to) which transmits and receives data at a low rate, and the analog inputs/outputs of SDR14 which run at very high sample rates. The system should perform this sample rate conversion without altering or distorting the signal content. By setting a frequency parameter from the user interface, the system should be able to transmit and receive at any given center frequency inside the first Nyquist band of the converters.

The first step of the project was to decide on a specification for the system through discussion with SP Devices, analysis of similar products, and investigation of possible areas of usage for the final system. A system architecture which fulfilled the specification was then to be decided on and modeled in MATLAB for verification. Finally, an implementation in Verilog was to be produced and the result tested on SDR14.

1.3 Overview

Chapter 2 presents some of the theory required to comprehend the design decisions and techniques used within this project. Chapter 3 briefly discusses the software tools that were used for design and implementation, and also examines the hardware which the project is based on. In Chapter 4, an initial system specification is decided on. A variety of different architectures are presented which fulfil the specification either partially, or completely. At the end of the chapter, a decision is made on which architectural approach to use for the implementation of the system.

Chapter 5 discusses the implementation methods for each major block in the system. Some blocks draw inspiration from research articles, while other blocks are designed from scratch. Finally, Chapter 6 presents various measurements and statistics for the final system, and draws some conclusions regarding the final product.
Chapter 2

Theory

In this chapter, the theory behind the most central concepts used in the thesis will be briefly covered. It is assumed that the reader has a decent knowledge of linear systems, transform theory (in particular the discrete fourier transform and the z-transform) and basic digital hardware building blocks. It is also helpful to have at least some experience in digital filter design and radio engineering.

A very large part of the theory used for this thesis is discussed thoroughly in both [15] which deals with digital filter design and implementation and [8] which more generally covers various aspects of digital signal processing. The theory can be found in many other books, articles, and websites and there should be no problem in finding a source for proofs and explanations of the equations and concepts presented in this chapter, should you not be able to acquire the referenced sources.

2.1 FIR filters

finite impulse respons (FIR) filters are described in detail in [15, 8], and will be only briefly summarized here. The output of an FIR filter of order $N$ is given by a weighted sum of the current and $N$ previous samples of the input signal $x[n]$ as

$$y[n] = \sum_{k=0}^{N} h_k x[n - k].$$

(2.1)

This type of weighted sum fits the formula for time-discrete convolution, assuming we imagine that there is an infinite number of zero-valued weights surrounding our actual weights, according to $h_k = 0$, $k \notin [0, N]$. By setting the input to the unit impulse $\delta[n]$, we get the impulse response of the system, which is equal to the weight coefficients in sequence

$$h[n] = \sum_{k=-\infty}^{\infty} h_k \delta[n - k] = h_k, \ k = n.$$  

(2.2)
There will be a total of $N + 1$ coefficients in the filter impulse response. In implementations of FIR filters, each such coefficient is called a “tap”. The impulse response gives a z-transform of

$$H(z) = Z\{h[n]\} = \sum_{k=-\infty}^{\infty} h[k]z^k = \sum_{k=0}^{N} b_k z^{-k}. \quad (2.3)$$

Notably, all poles in the filter characteristic exist at the origin, inside the unit circle, which makes all nonrecursive FIR filters unconditionally stable. This is due to the fact that no feedback from the filter output is present in the summation. Additionally, it can be shown that by making the FIR impulse response symmetric, a linear phase characteristic can be achieved inside the filter passband [15]. This is very useful in many DSP systems since it is usually desirable to preserve the shape of the waveform which is being processed.

In order to calculate the effects of the FIR filter for specific frequencies, we use the substitution $z = e^{j2\pi f}$ to get the filter Fourier transform as

$$H(f) = \sum_{k=0}^{N} b_k e^{-j2\pi f k} \quad (2.4)$$

In this thesis, no actual algebraic design will be done for any FIR filters used. Instead, design tools such as MATLAB are commonly used to produce the filter coefficients. Therefore, in terms of theory, it is enough to accept the fact that FIR filters can be considered linear time-invariant systems which can be used to alter the amplitude and phase characteristics of a signal.

## 2.1.1 Half-band FIR filters

A special class of FIR filters are called half-band filters. These filters always have a frequency characteristic which is symmetric around a quarter of the sampling frequency ($f_s/4$). An intrinsic property of this class of filters is that every other coefficient (except for the center tap) in the impulse response will be zero [8, 15]. This is a very useful property when it comes to saving hardware resources, since FIR filters are commonly implemented in FPGAs using hardware multiplier blocks. Since the zero-valued coefficients do not require a multiplication, the resource usage therefore becomes less intensive. An example of such a filter is shown in Fig. 2.1. Another thing to note about half-band filters is that since they are symmetric about $f_s/4$, we can transform a half-band lowpass filter into a half-band highpass filter simply by using the following equation:

$$H_{HP}(z) = H_{LP}(z) - 1 \quad (2.5)$$

The constant 1 in the frequency domain transforms into a unit impulse in the time domain, which means that the subtraction only affects the center tap of the filter. Since the center tap of a half-band filter is always 0.5, subtracting it with the unit impulse of amplitude 1 yields a negated center tap coefficient at $-0.5$. By negating the center tap of the filter in Fig. 2.1, we get the result in Fig. 2.2.
2.1 FIR filters

Figure 2.1. Impulse response and amplitude characteristic of an 11-tap halfband filter. Note the zero-valued coefficients.

Figure 2.2. Impulse response and amplitude characteristic attained by negating the center tap of an 11-tap lowpass half-band FIR.
It should also be noted that half-band filters by necessity have a passband ripple which is as low as the stopband attenuation [8, 15]. This can be a disadvantage since the passband ripple might not need to be that low for many applications, which means an unnecessary increase in filter order. For some applications which have low requirements on passband ripple, using an ordinary FIR filter might be better than a half-band FIR filter, since the reduction in filter order from not having the equiripple property might outweigh the cost reduction from having zero-valued coefficients.

### 2.2 Interpolation filters

A central concept of digital processing that will be used very much throughout this thesis is that of sample rate conversion, which is the process of modifying the sample rate of a time-discrete signal without distorting the spectral content of the signal itself. This section will deal with the process of *interpolation*, which means increasing the sample rate of a signal. For detailed explanations, refer to [8, 15].

#### 2.2.1 Upsampling

Interpolation, in theory, is usually described as being done in two stages. First, we increase the sample rate of the signal by a desired factor $M$, and then we filter out any unwanted resulting images. The common way of performing a sample rate increase is to add $M - 1$ zero-valued samples between every two samples of the input signal. This process is called *zero-padding*. To see why this is a good method, let us first describe the padded version of $x[n]$ as

$$x_{up}[m] = \begin{cases} x[n] & \text{for } m = n \cdot M \\ 0 & \text{else.} \end{cases} \quad (2.6)$$

Let us now take the discrete-time Fourier transform of $x[n]$ as

$$X(f) = \sum_{n=-\infty}^{\infty} x[n]e^{-j2\pi fn}, \quad (2.7)$$

and of $x_{up}[m]$

$$Y(f) = \sum_{m=-\infty}^{\infty} x_{up}[m]e^{-j2\pi fm} = \sum_{n=-\infty}^{\infty} x_{up}[Mn]e^{-j2\pi fMn} = \sum_{n=-\infty}^{\infty} x[n]e^{-j2\pi fMn} = X(Mf). \quad (2.8)$$

From this, we can see that the frequency characteristic of the output in the range of 0 to $f_{s,new}$ is simply that of the input in the range of 0 to $Mf_{s,old}$. Since the frequency characteristic in the input signal repeats itself at multiples of $f_s$, what we get at the output is a set of $M$ copies of the original signal band, spread out
2.2 Interpolation filters

over the entire Nyquist band of the new sample rate. For the case of \( M = 2 \), we will have one spectral image at baseband and one image at \( f_s/2 \). Unless a frequency translation is also desired, the baseband image is the one we want to keep, while removing all the other \( M - 1 \) images. An ideal filter which performs this operation would have an amplitude characteristic of

\[
|H_{LP}(f)| = \begin{cases} 
1 & \text{for } |f| < \frac{f_{s,old}}{2M} \\
0 & \text{else} 
\end{cases} 
\] (2.9)

The structure of an upsampling followed by a low-pass filter, which in total implements an interpolation, is shown in Fig. 2.3.

\[
\begin{array}{c}
x[n] \\
\uparrow M \quad H \quad y[m]
\end{array}
\]

**Figure 2.3.** Simple interpolation by \( M \).

### 2.2.2 Polyphase decomposition

A useful implementation technique when doing upsampling and FIR filtering after each other is called **polyphase decomposition**. An upsampled version of the input signal alternates between being zero-valued and having actual signal values. Therefore, not all filter coefficients are actually of interest when calculating each sample, since some coefficients will always be multiplied by zero-valued signal data. As an example, consider a signal \( x[n] \), which has been upsampled by two to produce the signal \( x_{up}[m] \), and imagine that this signal is filtered by an \( N \)-tap FIR filter to produce the output \( y[m] \). We can describe every two consecutive samples of \( y[m] \) as

\[
y[2n] = \sum_{k=0}^{N-1} h[k]x_{up}[2n - k] 
\] (2.10)

and

\[
y[2n - 1] = \sum_{k=0}^{N-1} h[k]x_{up}[2n - 1 - k]. 
\] (2.11)

However, we also know that the upsampling by \( M = 2 \) causes every other input value to be zero. Let us say that \( x_{up}[m] = 0 \) for all odd values of \( m \). This allows us to reduce the equations to

\[
y[2n] = \sum_{k'=0}^{\left\lceil \frac{N-1}{2} \right\rceil} h[2k']x_{up}[2n - 2k'],
\] (2.12)
and

\[ y[2n - 1] = \sum_{k'=0}^{\lfloor \frac{N-1}{2} \rfloor} h[2k' + 1]x_{up}[2n - 1 - (2k' + 1)]. \] (2.13)

We can see that (2.12) indexes different taps from the original filter compared to (2.13). A logical simplification is then to divide the original filter into two new subfilters which only contain the taps that each equation uses. Let us introduce two filters \( H_1(z) \) and \( H_2(z) \), which have impulse responses of

\[
\begin{cases}
  h_1[n] = h[2n] \\
  h_2[n] = h[2n + 1].
\end{cases}
\] (2.14)

Using these, we can reduce our two equations to

\[ y[2n] = \sum_{k'=0}^{\lfloor \frac{N-1}{2} \rfloor} h_1[k']x_{up}[2(n - k')], \] (2.15)

and

\[ y[2n - 1] = \sum_{k'=0}^{\lfloor \frac{N-1}{2} \rfloor} h_2[k']x_{up}[2(n - k')]. \] (2.16)

We can now note that there is a factor of two in the indexing of the \( x_{up} \) signal. This is due to having accounted for the zero-valued parts of the upsampled signal and rewritten the equations in order to not include these. This means that the actual upsampling is not needed in the implementation of a polyphase decomposed interpolation filter. By reverting our equations through the relationship \( x[n] = x_{up}[2n] \), we get

\[ y[2n] = \sum_{k'=0}^{\lfloor \frac{N-1}{2} \rfloor} h_1[k']x[n - k'] \] (2.17)

and

\[ y[2n - 1] = \sum_{k'=0}^{\lfloor \frac{N-1}{2} \rfloor} h_2[k']x[n - k']. \] (2.18)

We can see that (2.17) and (2.18) are convolutions of two different subfilters \( h_1[n] \) and \( h_2[n] \) with the same input signal \( x[n] \). This produces two output samples per input sample, which corresponds to an interpolation with the original filter \( H(z) \). This process can be extended in order to perform any interpolation by \( M \), by deconstructing a filter into \( M \) subfilters and running them in parallel. This structure is shown in Fig. 2.4. There are several benefits to this interpolation technique [8]:

- All computations are performed at the input sample rate, and not at the increased sample rate. This allows for cheaper implementation, since there are less computations done per clock cycle.
The zero-padding of the upsampling is never actually performed, and has instead been accounted for in the subfilter decomposition. This means that we never do any unnecessary multiply-by-zero operations.

2.2.3 Half-band interpolation

Half-band filters are of particular interest when performing interpolation, since an upsampling by $M = 2$ will create an image centered at $f_{s,new}/2$. All the content of the desired signal will exist at $|f| \leq f_{s,new}/4$, and the image content at $|f| \geq f_{s,new}/4$. Since a half-band filter is symmetric around $f_s/4$, this presents an ideal filter construct for attenuating the image while keeping the signal content.

If we recall the results from the previous section on polyphase decomposition, the two new subfilters contained separate sets of coefficients from the original impulse response, corresponding to even and odd-numbered coefficients. In the case of a half-band filter which contains several zero-valued coefficients, the subfilter $h_2[n]$ (which corresponds to all the odd-numbered taps of the original filter) will actually just consist of several zeros surrounding a center coefficient. This is shown in Fig. 2.5.

This polyphase branch is very cheap to implement in hardware, since it just has to delay the center tap by a number of samples. A straightforward design of a half-band filter yealds a center tap of 0.5 which can be implemented as a simple right shift of the corresponding signal data. The filter can also be scaled to a center tap of 1 in order to preserve the signal energy, which makes the second polyphase branch a pure delay. Due to this, interpolating in consecutive stages of $M = 2$ is a fairly attractive method when it comes to low resource usage.
Figure 2.5. Polyphase decomposition of a half-band filter. Note that all zero-valued coefficients end up in branch 2.
2.3 Decimation filters

Decimation is the opposite of interpolation, i.e., decreasing the sample rate of a signal. Many of the concepts from the theory of interpolation are also present when performing decimation, albeit with slight differences. Again, for detailed explanations, refer to [8, 15].

2.3.1 Downsampling

When performing downsampling, we sample a discrete time signal at a lower sample rate (some integer subdivision of the input sample rate). This translates into keeping every $M$th sample, and discarding the $M - 1$ samples which lay in between. What happens to the frequency content is similar to what happens when sampling a continuous signal according to the sampling theorem [8]: any frequency content above half the resulting sampling rate will fold back through aliasing down into the first Nyquist band.

Since we do not want to distort the signal spectrum, it is important that no frequency content is allowed to fold back into the signal band. Therefore, decimation is performed by first low-pass filtering and then downsampling (i.e., the opposite order of what is done during interpolation). This structure is shown in Fig. 2.6. Naturally, if we can be absolutely sure that there is no frequency content in the signal which can fold back, we can allow ourselves to do the downsampling only, but this is not usually the case.

![Diagram of simple decimation by M](image)

Figure 2.6. Simple decimation by M.

2.3.2 Polyphase decomposition

Polyphase decomposition is an important implementation technique for decimation filters as well as for interpolation filters. Since downsampling by $M$ discards $M - 1$ samples, keeping the low-pass filter and the downsampling separate is a very wasteful method, since the filter would be calculating more output samples which are then immediately discarded. Rewriting the filter equations in order to take into account the downsampling is a more attractive alternative.

If we take a decimation by $M = 2$ as an example, to calculate a given output sample $y[n]$ after a low-pass filtering using $h[n]$ of order $N$, the FIR equation (2.1)
gives us

\[ y[n] = \sum_{k=0}^{N} h[k]x[2n - k]. \quad (2.19) \]

We can subdivide this equation into two separate terms of

\[ y[n] = \left\lceil \frac{N-1}{2} \right\rceil \sum_{k'=0}^{\left\lceil \frac{N-1}{2} \right\rceil} h[2k']x[2n - 2k'] + \left\lfloor \frac{N-1}{2} \right\rfloor \sum_{k'=0}^{\left\lfloor \frac{N-1}{2} \right\rfloor} h[2k' + 1]x[2n - 2k' - 1]. \quad (2.20) \]

By mapping the filter coefficients used in each term of the equation to two new subfilters \( H_1(z) \) and \( H_2(z) \), where the impulse response \( h_1[n] \) contains the even-index coefficients from \( h[n] \) and \( h_2[n] \) contains the odd-index ones, we get

\[ y[n] = \sum_{k'=0}^{\left\lceil \frac{N-1}{2} \right\rceil} h_1[k']x[2n - 2k'] + \sum_{k'=0}^{\left\lfloor \frac{N-1}{2} \right\rfloor} h_2[k']x[2n - 2k' - 1]. \quad (2.21) \]

As can be seen, the inputs to the filters will be two downsampled versions of the input signal. Filter \( h_1[n] \) only uses the even-numbered samples of \( x[n] \), while \( h_2[n] \) only uses the odd-numbered ones. In other words, we can implement our decimation as two subfilters running at the lower sample rate, instead of one large filter running at the high input sample rate. The structure which results from these equations is shown in Fig. 2.7. This is very similar to the interpolation case, except here we sum the polyphase branches in order to produce an output sample, while for interpolation, the polyphase branch outputs became separate samples in the output signal.

---

**Figure 2.7.** Polyphase decomposition of a decimation by M.
2.3.3 Half-band decimation

Similarly to interpolation, polyphase decomposition of half-band filters provides a cost-effective method of implementing decimation by $M = 2$. The polyphase branch $h_2[n]$ will have filter taps which only consist of zeros and a center tap. A polyphase implementation will therefore only need one actual FIR filter in order to calculate branch $h_1[n]$, while the other polyphase branch can be handled by just delaying the input a number of samples and then summing it together with the $h_1[n]$ output.

2.4 Number representation

The discretized number representation used in this thesis work is invariably that of signed two’s complement fixed-point numbers. An extensive section describing these is presented in [8].

Two’s complement numbers work by negating the weight of the most significant bit (MSB). This has the useful effect of spreading out the range of values that can be represented almost evenly between negative and positive numbers. This is necessary when dealing with signals which have varying polarity, which is often the case in DSP architectures. For an $N$-bit two’s complement integer of the form $[x_{N-1}, ..., x_1, x_0]$, the corresponding value is given by:

$$x_{int} = (-2^{N-1})x_{N-1} + \sum_{k=0}^{N-2} x_k 2^k$$  \hspace{1cm} (2.22)

Fixed-point numbers work the same way, except that every number also has a decimal point placed inside the bit-pattern. The fixed point value of such a pattern can be calculated by taking the value which the bit pattern would represent if it was an integer, and then dividing it by $2^F$ where $N$ is the amount of fractional bits (i.e, the amount of bits after the decimal points). The value is calculated as

$$x_{fp} = \sum_{k=0}^{N-2} x_k 2^k 2^{-F} + (-2^{N-1})x_{N-1} 2^{-F}.$$  \hspace{1cm} (2.23)

If we take the bit pattern “1101” as an example, this would be the number $-2^3 + 2^2 + 2^0 = -5$ in an integer representation. If we instead consider the bit pattern as representing a fixed point number with two fractional bits, we would have a decimal point in the middle, as “11.01”. By dividing the integer with the scaling factor of $2^2$, we arrive at a fixed point value of $-5/2^2 = -1.25$.

The maximal value which can be represented by a fixed point system is given by setting the negative-weight MSB to 0 and the rest to 1, while the most negative value possible is given by setting the MSB to 1 and the rest to 0. The value range for an $N$-bit fixed-point system with $F$ fractional bits is then given by:

$$-2^{N-1-F} \leq x \leq 2^{N-1-F} - 2^{-F}.$$  \hspace{1cm} (2.24)
A very common practice in digital signal processing is to use $N - 1$ fractional bits, since this corresponds to an intuitively simple number range of roughly $-1 \leq x \leq 1$. If some arithmetic operation causes the resulting value to be outside the possible range of values, an overflow will occur. A number that is slightly above the positive end will wrap around to produce a negative number, and a number just outside the negative range will produce a positive number.

Sometimes, extra integer bits known as “guard bits” are added internally in a DSP block in order to allow the system to detect overflows. Normally, the guard bits will be redundant and will all have the same value as that of the sign bit (as if the signal had been sign-extended). If an overflow has occurred, the signal value will have extended into the guard bits, and they will not all be of the same value anymore. This means that this problem can be detected by comparing the guard bits with the sign bit at the system output. A common technique for making the overflow effects less severe is called saturation, which entails clipping the signal at the most positive/negative values instead of letting it wrap around.

### 2.4.1 Quantization noise

In [7], an excellent discussion on signal to noise ratio (SNR), noise floor level, and FFTs in the context of quantized signals is presented, and some of results will be briefly summarized here. Whenever a continuous signal is quantized in amplitude, e.g., when converting it with an ADC or during round-off when lowering the wordlength of a digital number, a small error is introduced. If we assume that the continuous signal is large enough in amplitude that the value of the bits which are discarded are not time-correlated, we can let the quantization error be represented by a uniform distribution between $-\frac{1}{2}q$ and $\frac{1}{2}q$, where $q = 2^{-N}$ for an $N$-bit digitizer.

It can be shown that the SNR of a full-scale sinusoid when quantized using $N$ bits is given by

$$SNR = 6.02N + 1.76 \text{ dB.}$$

The quantization noise is spread out evenly across the entire frequency spectrum. If our signal of interest has a narrow bandwidth, the noise outside our bandwidth of interest can be filtered out, and the SNR would then be improved as given by [7]

$$SNR = 6.02N + 1.76 + 10 \log_{10} \left[\frac{f_s}{2BW}\right] \text{ dB,}$$

where $BW$ represents the total signal bandwidth. When using an FFT algorithm to visualize the frequency spectrum of a quantized signal, we will see a noise floor. The level of this noise floor is not equal to the SNR value. In order to explain this, note that the energy density of the quantization noise (measured in W/Hz) is constant since it is modeled as white noise. An FFT consists of a discrete set of frequency “bins”, where each bin value is equal to the signal energy present in the range of frequencies which that bin represents. If we double the length of an FFT, each resulting bin will represent a frequency band which is only half as large. Since the energy density of the noise is constant over frequency, this means that the total noise energy present in each bin will be halved.
2.5 Quadrature signals

Our signal on the other hand, will remain constant in amplitude, since it is highly correlated to a certain set of frequencies. This concept is known as process gain and the level of the quantization noise floor for an FFT of length $M$ can be shown to be given by

$$NF = 6.02N + 1.76 + 10 \log_{10} \left[ \frac{M}{2} \right] \text{ dB.} \quad (2.27)$$

2.5 Quadrature signals

The concept of quadrature signal processing comes from the prospect of using complex-valued signals instead of real valued signals. Using a real-valued signal $x[n]$ always imposes the restriction of conjugate symmetry for the signal’s frequency response, according to $X(f) = X^*(-f)$. The real part of the positive-frequency spectrum must be mirrored at negative frequencies, and the imaginary part must be mirrored by a negated value. If we instead allow $x[n]$ to be complex-valued, we are freed from this restriction [8]. The most common example of a quadrature signal is the complex exponential, given by

$$x(t) = e^{j\omega_0 t} = \cos(\omega_0 t) + j \sin(\omega_0 t). \quad (2.28)$$

By using the formula for the inverse Fourier transform and using the distributive definition of the dirac impulse, we can calculate the Fourier transform of the signal as [9]

$$x(t) = e^{j\omega_0 t} = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(\omega)e^{-j\omega t} d\omega \Rightarrow X(\omega) = 2\pi \delta(\omega - \omega_0). \quad (2.29)$$

We see that the resulting signal spectrum is decidedly unsymmetric with regard to positive and negative frequencies. From this we can extrapolate that a signal consisting of a sum of several complex exponentials with different values for the frequency $\omega_0$, can produce a signal band which is present only on one side of the frequency spectrum origin.

The problem, of course, is that there is no such thing as complex signal values in the real world. However, since there seems to be some clear analytic benefits in using them, an analogous result can be achieved by representing a complex signal using two separate real valued signals. Our complex signal is then divided into the following form:

$$x(t) = x_I(t) + jx_Q(t) \quad (2.30)$$

The subscript I in $x_I$ stands for in-phase and Q in $x_Q$ stands for quadrature-phase. When implementing a quadrature signal in a system, two separate real-valued data streams are used to create $x_I$ and $x_Q$. In order to preserve their representation of a single complex signal, any mathematical operation that is done on the two signals must reflect what would happen if said operation was performed on the complex-valued signal $x_I(t) + jx_Q(t)$. 


Let us take complex multiplication as an example. Suppose we have two complex-valued signals $x(t)$ and $m(t)$ that are implemented as quadrature signals, and produce a complex output $y(t)$ through multiplication of $x$ and $m$. The expression for a complex multiplication simplifies into ($t$ has been excluded for all terms in order to shorten the equation)

$$y_I + jy_Q = (x_I + jx_Q)(m_I + jm_Q) = x_I m_I - x_Q m_Q + j(x_I m_Q + x_Q m_I). \quad (2.31)$$

Rewriting this result into separate equations for the in-phase and quadrature-phase parts, we get

$$\begin{cases}
y_I(t) = x_I(t)m_I(t) - x_Q(t)m_Q(t) \\
y_Q(t) = x_I(t)m_Q(t) + x_Q(t)m_I(t).
\end{cases} \quad (2.32)$$

By performing these exact operations on our real-valued signal streams, the result accurately represents that of a complex-valued multiplication.

### 2.5.1 Quadrature mixers

Quadrature signals are of particular interest when it comes to mixers. As an example, consider a band-limited signal $x(t)$, which we wish to mix to a center frequency of $\omega_c$. If this was real-valued processing, we would multiply the signal with a real-valued local oscillator

$$c(t) = \cos(\omega_c t) = \frac{1}{2}(e^{-j\omega_c t} + e^{j\omega_c t}). \quad (2.33)$$

A cosine local oscillator (LO) would have a frequency spectrum of

$$C(\omega) = \pi(\delta(\omega - \omega_c) + \delta(\omega + \omega_c)). \quad (2.34)$$

Since a multiplication in the time domain will result in a convolution in the frequency domain, we will get the resulting spectrum of

$$Y(\omega) = (X * C)(\omega) = \pi(X(\omega - \omega_c) + X(\omega + \omega_c)). \quad (2.35)$$

We can see that the resulting spectrum is mirrored over onto negative frequencies. However, if we instead modulate using a complex exponential as our local oscillator, we can use the result in (2.29), and arrive at the resulting spectrum of

$$Y(\omega) = (X * C)(\omega) = 2\pi X(\omega - \omega_c). \quad (2.36)$$

We can see that this corresponds to a pure translation in the frequency domain, without any mirroring. The real-valued mixer must follow the restriction of conjugate mirroring between positive and negative frequencies, while the complex mixer does not have to.

An example of when this would be useful is when mixing a signal which is already situated at an intermediate frequency, to an even higher frequency. A comparison between quadrature and real-valued processing for this situation is shown in Fig. 2.8 and 2.9. For real-valued mixing, the signal band will appear at both $f_{LO} + f_{IF}$ and $f_{LO} - f_{IF}$. An image-suppression filter is required in order to retain only the desired image, thereby getting a pure frequency translation. If we instead have a quadrature signal and multiply this with a complex-valued LO, no additional images are produced.
2.5 Quadrature signals

Figure 2.8. Example process for mixing a real-valued IF signal to RF, using image-suppression filtering.
Figure 2.9. Example process for quadrature mixing of a complex-valued IF signal to RF, with no image-suppression filtering required.
2.5.2 Complex-modulated filters

The theory in the previous section on quadrature mixing can be used within filter
design as well as representing streams of signal data. Multiplying a signal in the
time domain with a complex exponential will result in a frequency translation. If
we multiply the impulse response of a filter by a complex exponential according
to
\[
h_{new}[n] = h_{old}[n]e^{2\pi j n f_{mod}/f_s} \tag{2.37}
\]
and let the result be the impulse response of a new filter, this will similarly shift
the filter amplitude characteristic by
\[
|H_{new}(f)| = |H_{old}(f + f_{mod})| \tag{2.38}
\]
If we do this to a lowpass reference filter, the result will be a bandpass filter which
filters out a specific signal passband centered around the modulating frequency.
The filter coefficients will of course be complex-valued since we have performed
complex modulation. Because of this, the filter will not necessarily have conjugate
symmetry between positive and negative frequencies. This means that we could,
for example, produce filters which only pass signal spectrums between $-20$ to $-10$
MHz, without also passing signal content between $+10$ and $+20$ MHz.

Let us get a rough idea of the implementation cost of a complex-modulated
filter. One aspect which makes an FIR filter cheaper is a symmetric impulse
response, since this allows taps on opposite sides of the center tap to share mul-
tipliers. An interesting question is then of course if we can retain some measure
of symmetry in a complex-modulated filter. If we place the modulating complex
exponential so that it has zero phase at the center of the impulse response, then
the fact that a sine is an odd function and a cosine an even function will cause the
impulse response to have conjugate symmetry around the center tap. This fact
can be seen in the example in Fig. 2.10.

Let us see if it is possible to implement multiplier sharing for such a conjugate-
symmetric filter. We define two different complex-valued samples $A + jB$ and
$C + jD$ which we imagine sit on opposite sides of the center tap and both should
be multiplied by the same filter coefficient $h_I + jh_Q$. Let us write the expression for
this multiplication and simplify it to as few real-valued multiplications as possible.
We get the result of
\[
(A + jB)(h_I + jh_Q) + (C + jD)(h_I - jh_Q) = h_I(A + C) - h_Q(B - D) + j(h_I(B + D) + h_Q(A - C)). \tag{2.39}
\]
We can see that using two additions and two subtractions, the shared complex-
multiplication will require four real-valued multiplications in total, which is equal
to the cost of a single standard complex multiplication. If we compare this to the
more straightforward implementation of one complex multiplication per sample,
we see that we save one whole complex multiplication (four real-valued multipli-
cations) for every shared multiplication we use.\(^1\)

\(^1\)It should be noted that there exists a more complicated implementation of a complex mul-
Figure 2.10. Example of complex-modulation of an FIR filter by $f_{\text{mod}} = f_s/3.5$, showing conjugate symmetry in the resulting impulse response.
2.5.3 Amplitude and phase-shift keying

In digital communications, quadrature processing is often used to employ a method of symbol transmission which is called amplitude and phase-shift keying. A set of transmission symbols (such as, for example, all possible 2-bit permutations) are mapped to a set of amplitude levels and phase-shift settings. This means that each symbol will occupy a unique point in the complex plane. A sequence of data which is to be transmitted is then encoded into a sequence of such symbols. These are then pulse encoded into a sequence of in-phase and quadrature-phase pulses corresponding to the phase-shift and amplitude of the symbols, and that signal is then sent into the baseband input of an up-converter for transmission. Common forms of amplitude and phase-shift keying includes QPSK (phase-shift only, with four settings) and various QAM-grids (both phase- and amplitude-shift) such as 16-QAM or 64-QAM. Fig. 2.11 shows the arrangement of symbol points on the complex plane for these standards.

![Diagram of symbol points in the complex plane for QPSK, 16-QAM, and 64-QAM.]

Radio waves are quite sensitive to amplitude noise from sources such as thermal noise, interfering radio stations, multipath propagation and more. Transmission using a higher order QAM will typically be more prone to symbol errors due to the fact that the large number of symbol points also puts symbol points closer together in the complex plane.

2.6 DAC reconstruction

At the output of a digital signal processing system, it is often desired to convert the digital data streams into analog signals. A time-discrete signal consists of a series of weighted impulses. Reconstruction of the analog waveform can be done by convolving the impulse train with a reconstruction impulse response. It can be shown that perfect reconstruction which does not distort the signal band at all, the impulse response for the DAC would need to be a sinc waveform. The sinc would have its zero-crossings at multiples of $T_s$ where the surrounding sample impulses would be located.

Multiplication which requires only three real-valued multiplications, at the cost of several additional adders. The possibility of complex-valued multiplier sharing has not been investigated for this structure.
From the Fourier transform, we know that a perfect sinc in the time domain corresponds to a perfectly flat box in the frequency domain [9], which means that frequency content of the analog waveform will match that of the digital one exactly. Recall also that the Fourier transform of the digital impulse train will have a spectrum which repeats itself at multiples of $f_s$. Since the flat box of the perfect reconstruction frequency characteristic ends at $f_s/2$, these additional images will also be completely attenuated so that only the desired baseband signal remains.

This obviously presents a physically impossible structure to implement in an actual DAC. The sinc function is both infinitely long and non-causal. In reality, the technique of choice that is used in DACs is the zero-order hold, which simply holds the output at the value of the time-discrete sample for the duration of the sample length. Figure 2.12 shows a plot of a zero-order hold reconstruction where a 1 Hz sine is reconstructed at an update rate of 10 samples per second.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig2_12.png}
\caption{Zero-order hold reconstruction waveform for a 1 Hz sine at a 10 Hz update rate.}
\end{figure}

Let us define the impulse response of the DAC as $h(t)$. A single unit impulse will cause a value of 1 to be held from $t = 0$ to $t = 1/f_s$, according to

$$h(t) = \begin{cases} 1 & \text{for } 0 \leq x \leq 1/f_s, \\ 0 & \text{else}. \end{cases}$$

(2.40)

Since a convolution in the time domain corresponds to multiplication in the frequency domain, it is of interest to calculate how this non-ideal reconstruction waveform influences our signal passband. The Fourier transform of $h(t)$ can be
shown to be equal to [15]

\[ H(f) = \frac{1}{f_s} e^{-j\pi f/f_s} \text{sinc} \left( \frac{f}{f_s} \right) \]  

(2.41)

The exponential part has a constant amplitude of 1, and so will only contribute to a linear phase shift (this is due to the fact that the midpoint of the convolution waveform is not centered around \( t = 0 \).) The sinc function however, will have detrimental effects for the flatness of the DAC frequency response. A comparison between the frequency characteristics of ideal reconstruction and zero-order hold reconstruction is shown in Fig. 2.13.

![Ideal reconstruction](image1)

![Zero-order hold reconstruction](image2)

**Figure 2.13.** Comparison between the frequency characteristics of ideal reconstruction and zero-order hold reconstruction.

First of all, the sinc’s main lobe will cause the baseband amplitude characteristic to slowly droop until it goes to zero at \( f = f_s \). Since the signal band of interest when using a DAC is the first Nyquist band, let us calculate how far the amplitude has dropped at \( f_s/2 \). By comparing the value at \( f_s/2 \) with the value at DC, we see an amplitude droop of

\[ \frac{|H(f = f_s/2)|}{|H(0)|} = \text{sinc}(0.5) = 0.64 \approx -4\text{dB}. \]  

(2.42)

The amplitude characteristic has dropped by 4 dB at the edge of the Nyquist band, which is a quite significant amount. Also, whereas the perfect reconstruction had a perfectly flat box in the frequency response which completely removed the \( f_s \)-multiple images, the nonideal DAC response does not. The sinc in the frequency characteristic of the nonideal DAC will have zeros at the image locations, but unless the images have an insignificant bandwidth, the imperfect attenuation around the zeros will cause high-frequency content to appear at the output.
This imperfect HF attenuation is easily solved by an analog low-pass filter following the DAC output, which filters out all content above $f_s/2$. Compensating for the passband droop is slightly more complicated, and can be done using either analog filtering at the DAC output or digital filtering preceding the DAC. There are many ways of doing this and such implementation strategies will not be covered in this chapter, but it suffices to say that they all work by approximating the inverse of the reconstruction amplitude characteristic in order to create a system where the total frequency characteristic becomes flat.

The passband droop can be summarized with these significant effects in the context of a radio communications system:

- The signal amplitude will vary depending on transmission frequency
- The wider the signal band is, the more the tilt of the sinc will be visible within the signal band. These effects are investigated in more detail in later chapters that deal with the design and implementation of the reconstruction filter.

It is of interest to see how a tilt error would affect generic digital communication. A MATLAB model was created which modeled the effects of a linear amplitude characteristic tilt across the entire transmission signal band. 64-QAM signals with symbol rates at both $f_s/2$ and $f_s/3$ were tested. The test input was multiplied in the frequency domain with the tilting function, and then inversely FFT:ed back to the time domain, and plotted. The result is presented in Fig. 2.14. It should be noted that very simple symbol encoding was used to create the baseband signal. It is possible that better encoding might make the system more robust, but this possibility was not considered important enough to investigate further within the scope of this thesis.

![Figure 2.14. 64-QAM grid distortion for various tilts. Symbol rate of $f_s/2$ in upper row, and $f_s/3$ in lower row.](image)

We can see that increasing tilt will smudge out the symbol points on the QAM grid. However, the worst case tilt in the sinc function which we are trying to
compensate for does not reach these extreme levels. The tilt inside a 20 MHz signal band at the worst case transmission frequency (near $f_s/2$ where the sinc tilt is at its worst) goes from about $-0.1$ dB to $+0.1$ dB. In Fig. 2.15, a 1024-QAM signal (which puts very high pressure on signal quality) has been tilted at levels similar to that of the reconstruction distortion, and the view has been zoomed in on four symbol points in the QAM grid. We can see that the points get spread out for increasing tilt, and while the error is still very small in relation to the distance between the points, it is not negligible.

Figure 2.15. Zoomed view of 1024-QAM distortion for tilts near $+0.1$ dB, for symbol rate at $f_s/2$. 
Chapter 3

Tools, equipment and methodology

3.1 SDR14

The SDR14 platform is a 14-bit two-channel digitizer and generator, designed and sold by SP Devices [2]. It has two A/D inputs with a sample rate of 800 MSps, and two D/A outputs with an update rate of 1600 MSps. It can be controlled from a host computer through USB 2.0 or a PCIe-port. On-board DRAM modules allow triggered output of stored data to the DAC units, and triggered storage of input data from the ADC units. A photograph of the unit can be seen in Fig. 3.1.

Figure 3.1. Photograph of a rack-mounted version of the SDR14 unit.

The hardware inside SDR14 is controlled by a logic framework which is implemented in a Xilinx Virtex-6 FPGA. In this FPGA, a user logic block exists in the signal path, between the DRAM/PC and the A/D inputs and D/A outputs. Anything which is synthesizeable in FPGA hardware and which fits inside the available system resources can be implemented in the user logic block in order to modify the signal path. The control of the on-board hardware components such as DRAM
and ADC/DACs is invisible to the user logic, and is performed solely by the SP Devices framework which encapsulates the user logic block. This framework is supplied as a precompiled netlist file and is not modifiable.

3.2 Design software

3.2.1 MATLAB

In this thesis, MATLAB R2007b was used to perform the initial mathematical modeling of the system. It was also used extensively during testing, to compare waveforms, perform ideal operations on sampled system output and simulation outputs, among other things. MATLAB provides a large range of computational functions and easily allows user defined functions to be created. This enables all mathematical modeling to be written in a fairly modular way. The Filter Design and Signal Processing toolboxes were used for mathematical design of all of the filters in the system and for evaluation of different filter configurations during the design phase. Most of the graphical figures in this report were also generated through the use of MATLAB.

3.2.2 SP Devices

SP Devices provide a software kit together with their hardware. The parts that have been utilized for this thesis are the following:

- **ADCapture Lab** - A graphical interface that connects to the SDR14 module. It acquires a number of samples and plots both waveforms and FFTs of these, and can also do some minor performance analysis. The sampled data can be saved to an ASCII file and imported into MATLAB for further analysis.

- **SDR14 DevKit** - A development kit for the Xilinx software suite. A script is used to set up an initial SDR14 project with an empty user logic module and a precompiled version of the SP Devices framework. The DevKit also contains a script to run for generating the programming bitfile which is uploaded to the FPGA during reprogramming.

- **ADQ Updater** - A tool used for uploading new configuration bitfiles to the on-board FPGA.

- **MATLAB API utilities** - MATLAB files are provided which allow for functionality such as writing data vectors to the output waveform RAM on SDR14 directly from MATLAB, and for reading data into MATLAB from the digitizer. This was used extensively during testing.

3.2.3 Xilinx

The 12.4 version of the Xilinx software suite was used for all hardware description language (HDL) implementation during the thesis project. In particular, the
Xilinx ISE software tool was used, which provides a project manager and source code editor, along with tools for synthesis and analysis of the design. ISE comes complete with ISim, which is a piece of software used for simulation of the HDL code, which was used frequently during testing to verify correct signal data at all the various points inside the system.

3.2.4 CORE Generator

Xilinx CORE generator, or COREgen as it is usually written, is a tool for automatic generation of various common digital structures that are used in FPGAs. Some examples of cores which can be generated through COREgen and which are of interest for this thesis are FIR filters, complex multipliers, DDS sine/cosine generators and block RAM memory cells.

COREgen allows configuration of several different parameters such as speed requirements, optimization methods, wordlengths and other specifications for each core. Resource cost and performance will vary depending on the setting of these parameters. Generally, COREgen is very good at using implementation structures which have low resource utilization. If we want a 100-tap FIR filter which runs at a very low sample rate compared to the clock, COREgen will automatically select an implementation which requires only a couple of multipliers. COREgen can also utilize things such as coefficient symmetry and half-band zeros in FIR filtering or noise shaping in DDSes to further reduce the resource usage.

3.2.5 Verilog

The hardware-descriptive programming language used for all implementation in this thesis was Verilog, due to the fact that all other HDL code produced at SP Devices is written in Verilog. Xilinx ISE supports most Verilog-2001 constructs and possibly some from Verilog-2005. It does not support SystemVerilog at the moment.

3.3 Xilinx Virtex-6

The FPGA used on SDR14 is the Virtex-6 from Xilinx’s top end family of FPGAs. At the time of writing this report, Xilinx has started rolling out the Xilinx 7 series, but the Virtex-6 remains an extremely potent piece of hardware. The specific Virtex-6 device used on SDR14 is the LX240T, which sports features such as 768 hardware multipliers, 37680 reconfigurable logic slices and 416 block RAM cells at 36 kB each [18].
Chapter 4

Problem analysis

In this chapter, a system specification will be presented which has been produced from a combination of customer requirements and research of similar systems. Various solutions which fulfil the specification will be discussed. A verification of the functionality of the chosen solution will be performed using a MATLAB system model.

4.1 Product specification

A concrete specification was not given for the system from SP Devices, since the primary goal was to develop a demo application for the SDR14 system and not to meet any specific design criteria. However, an example of the kind of signal processing that was desired was given by means of the datasheet for an integrated DAC circuit called AD9776 [1]. This component is manufactured by Analog Devices and consists of both DAC and upconverter in a single component, and SP Devices suggested that the resulting system of this thesis project should be similar to that of this component. By studying the datasheet of the circuit and copying or adapting the specifications to fit SDR14, the following specification was extracted for a software-defined radio (SDR) transmitter:

The transmitter shall

• take two 16-bit data streams as quadrature input.
• interpolate the inputs from 25 MSps up to 1600 MSps.
• output the two 8-parallel data streams to the two on-board 14-bit DACs.
• maintain a passband of at least 80% of the input nyquist bandwidth (-10 to 10 MHz) during up-conversion
• allow for frequency translation across a range covering the entire nyquist band of the DACs (-800 to 800 MHz).
Problem analysis

- contain an inverse sinc compensation filter which compensates for the DAC reconstruction distortion.
- allow for modification of all system parameters (mixer frequencies, sinc filter on/off, chain bypass, etc) from the PC interface.
- have a passband ripple of 0.01 dB or less

Initially, only the transmitter was actually included in the scope of the thesis project. However, after noticing fairly early in the project that there would be time for implementing a receiver as well, such a specification was also created. It has the same kind of requirements as for the transmitter, apart from the fact that the ADCs run at half the sample rate of the DACs:

The receiver shall...

- take two 4-parallel 16-bit quadrature data streams as input.
- decimate the inputs from 800 MSps to 25 MSps.
- output the two resulting quadrature data streams to the software interface.
- maintain a signal passband width of at least 20 MHz (80% of the baseband bandwidth) during down-conversion
- allow for frequency translation down to baseband from a range that covers the entire nyquist band of the ADCs (-400 to 400 MHz).
- allow for modification of all system parameters (mixer frequencies, chain bypass, etc) from the PC interface.
- have a passband ripple of 0.01 dB or less

There was no specification given for how well the system blocks would need to attenuate unwanted spectral content. Any such spurs in the output could for example disturb other channels in the RF spectrum if the system is used for radio broadcasting, or make results less accurate if it is used in a test and measurements lab setup. Following a suggestion in [15] of keeping any stopband ripple in interpolation and decimation filters below half the quantization step, the stopband attenuation was determined by

\[ A_{\text{min}} = -20 \log(Q/2) = -20 \log(2^{14}) \approx 85 \text{dB}. \] (4.1)

This gives a resulting specification, for both transmitter and receiver, of:

- All undesired spectral content from filters, mixers and other system blocks should be attenuated by at least 85 dB.
4.2 Evaluation of possible architectures

We will still be able to see these undesired signals by using long FFTs due to the process gain lowering the noise floor below -85 dB [7], so this calculation is slightly arbitrary. Due to lacking any concrete specification, the equation still seemed like a good enough rule of thumb, and all filters were therefore designed for this level of attenuation.

The frequency resolution was also not rigidly specified. In conversation with my supervisor at SP Devices, we concluded that it was generally a good idea to let the system have an extremely high resolution. This makes the system less limiting in terms of applications, and a very high resolution can be accomplished easily using DDSes.

Some research was still done on what kind of relaxations could be made on frequency resolution while still having the system be widely usable in communication systems. Most digital communication standards that are widespread in mobile phone and TV broadcasting use a 200 kHz channel spacing, but some go as low as 50, 30 or 12.5 kHz [13]. Since this system should be as flexible as possible to allow use by a wide variety of end-users, it seems reasonable that all these different channel spacings should be supported, at the very least. In one article, SDR technology is presented as very useful for VHF-band public safety radio [6]. The P25-technology which is described in the article uses a 6.25 kHz channel spacing in the most recent technology standards. Yet another application might be aviation control radio, which in Europe uses a 8.33 kHz spacing [3].

A system that supports all these communications standards must have a frequency step size which is a common divisor of all the different channel spacings. For most of these standards (apart from the aviation radio spacing), a step size of 6.25 kHz would solve the problem. However, since most DDS implementations operate at a step size of a power-of-two subdivision of the clock frequency, getting an exact step size of 6.25 kHz might not be possible on this hardware platform. To support all types of channel spacing with low error, it is therefore probably a good idea to have step sizes in range of 1 Hz, since this also allows for any exotic frequency allocation standards to be used with the system without modification.

From this discussion, we can add the following point to the specification:

- The architecture should be able to handle a channel spacing of 6.25 kHz, and if possible have a minimum frequency step size as low as 1 Hz.

4.2 Evaluation of possible architectures

There are various strategies that can be used to meet the specification, especially since some points were not really set in stone. One thing to note is that the Virtex-6 FPGA which the system was to be implemented on contains 768 hardware multipliers, which for most applications is a very large number. While most DSP designs are all about being as cheap in resources as possible, design decisions in this project were sometimes made in order to have better performance at a slightly higher cost, since resources were readily available anyway.

Any resource usage estimations in this section are done at a fairly high level. While consideration is made of implementation strategies and possible cost re-
ductions, these have not actually been tested in code and the estimations might therefore not be completely accurate. All architectural evaluations in this chapter were done only on the transmitter side in order to make the section less cluttered with content. Since the receiver and transmitter will perform roughly the same kind of operations albeit in opposite order, conclusions regarding a certain transmitter architecture are valid when implementing the receiver as well.

In order to make cost estimations for the system, it is necessary to know the clock rate which the filters are running at. In the SDR14 user logic block, two main clock sources are available, at 200 and 400 MHz respectively. Using as high a clock rate as possible is of course good for keeping the filter costs down, which would suggest that using the 400 MHz clock is best. However, in order to make it possible to place and route the synthesized HDL netlists in the FPGA without breaking any timing constraints, using the 200 MHz clock is probably the best option since experience has shown that not much logic at all can be fit inside the allowable critical path of a 400 MHz clock system. Therefore, all cost estimations were made with a 200 MHz clock in mind.

Three good starting points for getting some basic ideas of how up- and down-converters are usually implemented are found in [17, 16, 10]. Some recurring themes in these articles regarding construction of DDCs and DUCs are:

- Frequency translation is usually performed using mixers and DDS:es
- CIC filters are often used, especially when implementing large-scale sample rate change (large being in the range of about 32 or higher)
- For small-scale sample rate changes, halfband FIR filters are often used

### 4.2.1 Cascaded Integrator-Comb filtering

Before starting on the topic of CIC-filters, it should be noted that no CIC filters were actually used for the project. This section is included purely in order to explain why CIC filters were discarded in favor of other filter solutions despite the fact that the references above all state that CIC filters are an excellent way of implementing sample rate conversion for DUCs and DDCs.

CIC filters were first proposed by Hogenauer in [5], where CIC stands for Cascaded Integrator-Comb. These filters use no multipliers in their implementation, and consist instead of elements known as combs and integrators. Each comb or integrator consists only of a delay element and an addition or subtraction. The blocks have output equations of

\[ y_{comb}[n] = x[n] - x[n - 1] \]  \hspace{1cm} (4.2)

and

\[ y_{int}[n] = x[n] + y_{int}[n - 1]. \]  \hspace{1cm} (4.3)

We can see that a comb performs a differentiation, while an integrator performs an integration. The block diagram of a CIC interpolation filter is shown in Fig. 4.1. Only a single comb and a single integrator are drawn in the figure, but as we
shall soon see, larger numbers of them can be used as well. With a cascade of $N$ combs, followed by upsampling by $R$, followed by a cascade of $N$ integrators, the resulting amplitude characteristic is given by [5]

$$|H(f)| = \left|\frac{\sin(\pi f)}{\sin(\pi f/R)}\right|^N \tag{4.4}$$

If we look at this in the frequency domain, we will see zero locations present at the points where imaging occurs from the upsampling. This means that the resulting output signal will be an interpolated version of the input signal. Figure 4.2 shows the frequency characteristic of a CIC for upsampling by $R = 8$, using both $N = 1$ and $N = 3$ cascading stages. CIC filters present a very attractive alternative to performing interpolations using polyphase FIR or IIR filters, since there are no coefficient multiplications used (only additions and subtractions). One problem with CIC filters is that in order to increase the stopband attenuation, we must use more cascade stages (thereby increasing the number $N$.) However, by doing this, the attenuation present in the passband is also increased. This phenomenon can be clearly seen in Fig. 4.2. Additionally, if we look at the amplitude characteristic, the attenuation is very quickly lowered as we move outwards from the zero locations. This means that both the worst case image attenuation and the worst case passband droop are drastically worsened if we increase the signal bandwidth. Due to this, CIC filters are best used when the signal of interest has a narrow bandwidth.

One common way of dealing with the passband droop is to use an optimized FIR filter which straightens out the passband characteristic without amplifying the stopband content [5]. The compensation filter is usually placed before the CIC, since this means lower sample rates and cheaper implementation. Having it before the CIC also allows for an additional improvement, by also making the compensation filter an interpolating filter as well. That way, we double the sample rate to signal bandwidth ratio, thereby making the signal more narrowband. Since the passband droop is highly dependent on the signal bandwidth, this allows for much better results.

Suppose we might wish to use a CIC filter in order to interpolate a baseband signal with a 20 MHz bandwidth (frequency content between -10 and +10 MHz)
Problem analysis

Figure 4.2. Plot of a CIC filter characteristic over -fs/2 to fs/2 in the new sample rate, for interpolation by eight, using both N=1 and N=3 cascade stages.

from 50 MSps to 1600 MSps. This presents a realistic situation for this system since we want a total interpolation from 25 to 1600 MSps, and the compensation filter is commonly made to deal with the first step (from 25 to 50 MSps.) The CIC frequency characteristic has its worst image attenuation at close end of the image which appears nearest to the baseband, which for this example is at 50 MHz. The 20 MHz bandwidth required for the system in this thesis means that we find the worst case passband droop at 10 MHz, and the worst case image attenuation at $50 - 10 = 40$ MHz. Using equation 4.4 for $R = 1600/25 = 32$, we arrive at the results in Table 4.1 for various values of $N$.

We can see that for $N = 7$ we have an adequate attenuation for matching the specification of having at least 85 dB attenuation in the system. For $N = 7$, the passband droop at the signal band edge is at $-4.6$ dB.

At the end of the transmitter, we are running at a sample rate which is eight times that of the clock rate. Although CIC filtering is very popular in DUC/DDC applications [17], this type of high-rate situation probably is not very common-place. In order to see why CIC filters are problematic at high sample rates, we need to look closer at the integrator block.

The integrator of the CIC contains a recursive loop, feeding a delayed version of the integrator output back into the integrator adder. In order to be able to handle eight parallel samples per cycle like we have at the output sample rate, we would need to unroll this recursive loop eight times. The unrolled integrator is shown in Fig. 4.3.
4.2 Evaluation of possible architectures

<table>
<thead>
<tr>
<th>N</th>
<th>Passband droop (dB)</th>
<th>Min. stopband att. (dB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.58</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>-1.2</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>-1.7</td>
<td>38</td>
</tr>
<tr>
<td>4</td>
<td>-2.3</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>-2.9</td>
<td>63</td>
</tr>
<tr>
<td>6</td>
<td>-3.5</td>
<td>76</td>
</tr>
<tr>
<td>7</td>
<td>-4.1</td>
<td>88</td>
</tr>
<tr>
<td>8</td>
<td>-4.6</td>
<td>100</td>
</tr>
</tbody>
</table>

**Table 4.1.** Droop and attenuation of CIC interpolation by 32, for varying N.

**Figure 4.3.** Block diagram of a loop-unrolled CIC integrator for eight parallel samples.
The problem now becomes more visible. We can see that we have a recursive loop which contains eight adders and only a single delay element. There is no way to pipeline the data path inside a recursive loop, and that loop will put a bound on the critical path of the system [8]. It is quite impossible to meet the timing constraints if we have eight cascaded adders which all need to finish within one clock cycle.

Due to this problem, CIC filters were not used in the project. This conclusion can of course be extended to say that any filters which involve recursive loops (IIR filters) are not a good idea for implementation at these high sample rates. FIR filters are much more easily scaleable, due to the fact that the eight parallel samples only need to interact through a summation after the coefficient multiplication, and a summation can easily be pipelined in the form of an adder tree. For this reason, only FIR filters are used throughout the rest of the architectural discussions.

4.2.2 Interpolation bandpass filtering

In total, a sample rate change from 25 Msps to 1600 MSps is desired within the transmitter. If the input signal was upsampled (i.e., zero-padded) by 64, there would be 64 copies of the input spectrum spread out at intervals of 25 MHz across the entire range of $-800$ to $800$ MHz. The standard procedure in interpolation would be to use a low-pass filter to retain the baseband image. However, by utilizing a bank of bandpass filters, this could be extended into keeping any single one of the 64 images. By doing this, we would achieve both frequency translation and interpolation at the same time, since we can keep the image that is located at the frequency we want to transmit on. The transmitter would then consist of a single programmable filter and a coefficient memory. Frequency resolution for this architecture would be very coarse at steps of 25 MHz, which does not meet the specification regarding frequency resolution, and so this evaluation is for cost comparison purposes only.

Say we want to transmit at a frequency of 175 MHz using this architecture, we would pick a filter that retains that image and filters out all the other ones. The steps of this procedure are shown in Fig. 4.4.

In another type of DSP construct called a transmultiplexer, similar operations are performed except that the filters are instead used to divide a broadcast spectrum into several small channels. The theory behind transmultiplexers can be used as a source of inspiration for this architecture. The standard methodology for TMUX design is to first design a lowpass reference filter which selects the image which is centered at 0 Hz [14]. In order to design the rest of the filters, that reference filter is complex-modulated in order to put its passband at other frequencies. This would be done 63 times in order to get a total of 64 filters (one for every possible image location,) and all these 64 coefficient sets could then be stored in a memory on the FPGA.

The MATLAB function `firpmord` was used to estimate the filter order required to create a reference filter which passes the -10 to 10 MHz baseband bandwidth with 0.01 dB ripple and has a stopband which attenuates the nearby images starting at $25 - 10 = 15$ MHz, by 85 dB:
Figure 4.4. Plot showing a bandpass interpolation from 25 to 1600 MSps, with an image selection filter picking out the image at 175 MHz
>> firpmord([10 15], [1 0], [(1 - 10^(-0.01/20)) 10^(-85/20)], 1600)
ans =
 1284

Since the filter input is an upsampled signal, it makes sense to implement the filter as a polyphase decomposition. Such a solution would have 64 filters running in parallel, at a sample rate of 25 MSps. In the discussion on complex-modulated filters in section 2.5.2, it was noted that the complex modulation still retained conjugate symmetry around the center tap in the impulse response, which can be utilized for the polyphase decomposed filters as well (although the distribution of the multiplication results will quite complicated as compared to a simple non-decomposed symmetric FIR filter). Using a 200 MHz system clock, we get a theoretical total multiplier cost for the filter of

\[
N_{\text{taps}} \cdot 4_{\text{complex}} \cdot \frac{1}{2_{\text{symmetric}}} \cdot \frac{f_s}{f_{\text{clk}}} = 1284 \cdot 4 \cdot \frac{1}{2} \cdot \frac{25}{200} = 321. \tag{4.5}
\]

This cost is quite extreme and results from the high filter order required to produce the very narrow transition region, the fact that the filter is complex valued and also that we cannot utilize the symmetry in the impulse response due to the polyphase decomposition.

### 4.2.3 Half-band interpolation with image selection

An alternative to the single polyphase-implemented FIR filter in section 4.2.2 is to draw inspiration from the frequency masking designs as discussed in [15]. By cascading several filters at different sample rates, a total filter response with a narrow transition region can be achieved. Since we are interpolating by a power of two, one intuitive approach is to use a cascade of half-band filters. Since there is a fairly small set number of possible image locations in each interpolation step (two after the first upsampling, sixty-four after the final upsampling), a good idea could be to have one specific filter for every possible image location. This would mean having a programmable complex-valued half-band filter at each stage, and selecting a complex-modulated filter coefficient set to program it with to retain the correct image. For the first stage, there would be two different filters to choose from and for the last there would be sixty-four. This process is shown in Fig. 4.5, for the first two interpolations. As shown in the plots, we get an interpolation by 4 and a frequency translation of +25 MHz by doing the steps shown, and this can be extended to any 25 MHz multiple and an interpolation by 64 for the entire system. Since a complex-modulation of a half-band filter would entail a time-domain multiplication with the impulse response, the zero-valued coefficients would be preserved as zeroes. This means that polyphase decomposition can still be used regardless of whether the filter has been complex-modulated or not.

For estimating the implementation cost, it was first noted that the first filter is only either highpass or lowpass since there are only two possible image locations, and that this filter will therefore be real-valued. This filter can therefore be implemented according to the discussion on highpass/lowpass half-band filters.
Figure 4.5. Half-band image selection using complex-modulated filters, for the first two interpolation stages.
in section 2.1.1, by adding an on/off-switchable negation in the second polyphase branch. This would invert the center tap, thereby turning into a highpass half-band filter. Later stages will have more possible image locations and will require complex-valued filtering. Note also that a real valued filter will still need to operate on both I and Q channels, at a cost of two multiplications per tap, while a complex-valued filter operates on both at once using complex multiplications, for a cost of four multiplications per tap.

Unlike in the previous section, we can use polyphase decomposition to good effect here by polyphase-decomposing each of the six filter stages. This does not ruin the impulse response symmetry, which means that the conjugate symmetry as discussed in section 2.5.2 can be used to make the implementation cheaper. Assuming that we have normalized the center tap to $1 + j0$ so that it does not require a multiplication, the cost for a specific complex-modulated half-band polyphase filter can be calculated as

$$\left[\frac{4}{(\text{compl. mult})} \cdot \left\lceil \frac{N_{\text{taps}}}{2(\text{halfband})} \cdot \frac{1}{2(\text{symmetry})} \right\rceil \cdot \frac{f_s}{f_{\text{clk}}} \right].$$

The cost of each filter has been calculated using this formula and is presented in Table 4.2. Apart from the multiplier cost, we also have some added costs in the form of filter bank memories and adders/subtractors for the complex-valued filters. Still, this appears to be a much cheaper way of performing bandpass interpolation compared to the previous section. However, we still have the problem of not meeting the system specification in terms of frequency resolution. We are forced to use a channel spacing of 25 MHz, which is way too large for the majority of applications, and therefore this solution is also only for comparison purposes.

### 4.2.4 Baseband interpolation and multi-DDS mixer

If the signal is kept at baseband throughout all interpolation, the interpolation filters would be cheaper to implement. The filters would be real-valued, which means that each tap would cost two multiplications (one for I, one for Q.)
would be no need for any filter banks with multiple coefficient sets. The corre-
sponding resource cost for interpolating a +/-10 MHz bandwidth at baseband using
half-band filtering is shown in 4.3. This is significantly cheaper than the previous

<table>
<thead>
<tr>
<th>Stage</th>
<th>Filter taps</th>
<th>HW multipliers (I+Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>51</td>
<td>[2\lceil51/4\rceil \cdot \frac{1}{2}\rfloor = 4 ]</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>[2\lceil15/4\rceil \cdot \frac{1}{2}\rfloor = 2 ]</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>[2\lceil11/4\rceil \cdot \frac{1}{2}\rfloor = 3 ]</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>[2\lceil11/4\rceil \cdot \frac{1}{2}\rfloor = 6 ]</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>[2\lceil7/4\rceil \cdot \frac{1}{2}\rfloor = 8 ]</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>[2\lceil7/4\rceil \cdot \frac{1}{2}\rfloor = 16 ]</td>
</tr>
</tbody>
</table>

**Table 4.3.** Estimated hardware cost for pure interpolation from 25 MSp to 1600 MSp, for a +/-10 MHz baseband bandwidth.

implementation, but does not include any frequency translation. One possiblity is
for frequency translation to be performed through the use of mixers instead of select-
ing an interpolation image (see section 2.5.1 for the theory behind quadrature
mixers). If interpolation is performed using the method above, the mixer stage
would then be located at a sample rate of 1600 MSp, at eight times the clock
rate. Since there would be eight output samples per clock cycles, we would also
need eight DDS values per clock cycle. Figure 4.6 shows the block schematic of
such an architecture.

![Block schematic of a transmitter architecture performing frequency translation using eight DDSes after the 25-1600 MSp interpolation.](image)

**Figure 4.6.** Block schematic of a transmitter architecture performing frequency translation using eight DDSes after the 25-1600 MSp interpolation.
Problem analysis

A DDS sine/cosine generator can be instantiated through COREgen with various phase and data widths. A DDS is essentially just a phase counter which increments by a set value every clock cycle corresponding to the desired frequency, combined with a phase-to-magnitude look-up table implemented in BRAM memory. The increment value which should be used in order to produce a specific output frequency is calculated as

$$\omega_{\text{norm}} = 2\pi f T_s = \frac{2\pi f}{f_s}. \quad (4.7)$$

This means that its resource usage will consist primarily of BRAM memory cells. Table 4.4 shows the BRAM cost for various data width settings, along with the frequency resolution corresponding to the selected phase width. Experimenting with COREgen made it clear that it is mostly the data width and not the phase width which determines the resource cost, which is why all the DDSes in the table were set to 32 bits phase width. Table 4.2 also shows the spurious-free dynamic range (SFDR) of the resulting sine/cosine outputs (taken directly from COREgen). Since we want to keep an 85 dB attenuation everywhere in the system for unwanted frequency content, the DDS should do the same. We can probably get away with using 14 bit data-width for an 84 dB attenuation for a cost of 8 BRAM memory cells in total.

We want to simulate a single DDS at a sample rate of 1600 MSps with some phase increment setting $\delta\phi$. If we want to map this to a set of eight DDS:es, this $\delta\phi$ value will instead correspond to a phase offset between consecutive DDS:es. The first DDS will have an offset of $0\delta\phi$, the second $1\delta\phi$, and so on until the last DDS at $7\delta\phi$. In order to preserve this phase incrementation for the next clock cycle, the eight DDS:es should therefore all have a phase increment setting of $8\delta\phi$ per clock cycle. An example of what this would look like over a period of three clock cycles is shown in Table 4.5.

Calculating correct phase increment and offset values for each DDS for every time the user changes $\delta\phi$ value in order to select a new transmit frequency can be done with some control logic and an accumulator, every time the user changes mixer frequency. On top of this, we would need to perform eight complex multiplications every clock cycle in order to mix the DDS outputs with the eight I/Q data samples. Each multiplication would require four hardware multipliers, for a total cost of 32 multipliers.

The estimated hardware cost of the entire solution is presented in Table 4.6.

<table>
<thead>
<tr>
<th>Data width</th>
<th>Phase width</th>
<th>Freq. step (Hz)</th>
<th>SFDR</th>
<th>BRAMs</th>
<th>BRAM tot.</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>32</td>
<td>0.4</td>
<td>96 dB</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>15</td>
<td>32</td>
<td>0.4</td>
<td>90 dB</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>14</td>
<td>32</td>
<td>0.4</td>
<td>84 dB</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 4.4. Evaluation of performance and cost for various DDS configurations.
Table 4.5. Overview of DDS phase values over three clock cycles, showing constant offsets and a constant increment of $8\delta\phi$.

<table>
<thead>
<tr>
<th>Clock cycle:</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS</td>
<td>Phase</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$0\delta\phi$</td>
<td>$0\delta\phi + 1 \cdot 8\delta\phi$</td>
<td>$0\delta\phi + 2 \cdot 8\delta\phi$</td>
</tr>
<tr>
<td>2</td>
<td>$1\delta\phi$</td>
<td>$1\delta\phi + 1 \cdot 8\delta\phi$</td>
<td>$1\delta\phi + 2 \cdot 8\delta\phi$</td>
</tr>
<tr>
<td>3</td>
<td>$2\delta\phi$</td>
<td>$2\delta\phi + 1 \cdot 8\delta\phi$</td>
<td>$2\delta\phi + 2 \cdot 8\delta\phi$</td>
</tr>
<tr>
<td>4</td>
<td>$3\delta\phi$</td>
<td>$3\delta\phi + 1 \cdot 8\delta\phi$</td>
<td>$3\delta\phi + 2 \cdot 8\delta\phi$</td>
</tr>
<tr>
<td>5</td>
<td>$4\delta\phi$</td>
<td>$4\delta\phi + 1 \cdot 8\delta\phi$</td>
<td>$4\delta\phi + 2 \cdot 8\delta\phi$</td>
</tr>
<tr>
<td>6</td>
<td>$5\delta\phi$</td>
<td>$5\delta\phi + 1 \cdot 8\delta\phi$</td>
<td>$5\delta\phi + 2 \cdot 8\delta\phi$</td>
</tr>
<tr>
<td>7</td>
<td>$6\delta\phi$</td>
<td>$6\delta\phi + 1 \cdot 8\delta\phi$</td>
<td>$6\delta\phi + 2 \cdot 8\delta\phi$</td>
</tr>
<tr>
<td>8</td>
<td>$7\delta\phi$</td>
<td>$7\delta\phi + 1 \cdot 8\delta\phi$</td>
<td>$7\delta\phi + 2 \cdot 8\delta\phi$</td>
</tr>
</tbody>
</table>

Table 4.6. Total estimated hardware cost of baseband interpolator and multi-DDS mixer architecture.

<table>
<thead>
<tr>
<th>Block</th>
<th>Multipliers</th>
<th>BRAM cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpolation filters</td>
<td>39</td>
<td>0</td>
</tr>
<tr>
<td>DDS</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Mixer</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>71</strong></td>
<td><strong>8</strong></td>
</tr>
</tbody>
</table>
4.2.5 Bandpass interpolation after fine-tuning mixer

One possible hybrid solution would be to combine the baseband and bandpass interpolator architectures. If we do baseband interpolation for the first three interpolation stages and add one DDS-based mixer at 200 MSps in order to achieve fine-grained frequency translation, we could then try to perform the last interpolation from 200 MSps to 1600 MSps through a bandpass interpolator. This would involve first upsampling the 200 MSps signal by $M = 8$, thereby producing eight spectral images at multiples of 200 MHz. Second, a bank of several bandpass filters would be used to filter out the one spectral image which is situated at the frequency we are interested in transmitting at, while attenuating all other images. The filter implementation would be a polyphase structure, with a set of eight sub-filters running at 200 MSps, producing eight output samples per cycle. Figure 4.7 shows a block schematic of this architecture.

![Figure 4.7. Block schematic of a transmitter architecture using a fine-tunable mixer at 200 MSps followed by a polyphase bandpass interpolator.](image)

Assuming we want to be able to transmit at all frequencies, we would be required to allow a range of -100 to +100 MHz for the fine-tunable mixer, in order to cover all the frequencies around the coarse steps of the bandpass interpolator. The problem with this approach is that the signal will not be centered at baseband as it enters the bandpass interpolator, which means that the eight images will not be centered exactly at multiples of 200 MHz anymore. We would either need filters with wider pass- and stopbands to account for several mixer settings at once, or a greater number of filters in the filter bank to be able to select an appropriate one for a given fine mixer setting. The realistic alternative is probably to use a hybrid of the two, by having multiple sets of filter coefficients per image location which also have wide passbands to allow a broad range of mixer frequencies.

We can consider the solution of using one set of filter coefficients for every possible fine mixer setting. This would mean an impossibly huge number of rows in the coefficient bank, but for evaluation purposes we can still consider the multiplier cost of this strategy. If we have one filter for each fine mixer setting, the filter requirements will be relaxed. The signal bandwidth is $\pm 10$ MHz, at an image
4.2 Evaluation of possible architectures

spacing of 200 Mhz, which means that the filter would have a transition band of 180 MHz. The common way of designing the image selection filters is to design one prototype filter which is then complex-modulated to the correct position for each image location [14]. The prototype filter in this case would be a low-pass filter with a passband edge at 10 MHz, and stopband edge at 190 MHz, for a sample rate of 1600 MHz. To match the specification it would also need 85 dB attenuation in the stopband and 85 dB attenuated ripple in the passband.

Using the MATLAB function `firpmord`, it was determined that a filter matching the specification of this prototype filter would require 44 taps. Since the input to this filter would be an 8-interpolated signal, the reasonable method of implementation would be to use polyphase-decomposition. The cost would then be equal to a 44-tap filter running at a sample rate equal to the clock rate.

We want to be able to pick out an image that exists purely within positive or negative frequencies in order to have the same capabilities as that of a quadrature mixer. This means that we require the filter coefficients to be complex-valued, for a cost of four real-valued multiplications per tap. All in all, this means that we have a hardware cost of $44 \times 4 = 176$ multipliers. Add to this the 9 multipliers for the first three baseband interpolators (see Table 4.3, along with four multipliers and a BRAM cell for the fine-tunable mixer, and we get a total estimated cost as shown in Table 4.7.

<table>
<thead>
<tr>
<th>Block</th>
<th>Multipliers</th>
<th>BRAM cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseband interp.</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>DDS</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Mixer</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Bandpass interp.</td>
<td>176</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>189</strong></td>
<td><strong>1</strong></td>
</tr>
</tbody>
</table>

Table 4.7. Total estimated hardware cost of baseband/bandpass interpolation hybrid with fine-tunable mixer.

4.2.6 Multi-stage mixer architecture

One design methodology which is always central in interpolating and decimating architectures is to move all operations as far down towards low sample rates as possible, in order to save computational power. This is also applicable in the case of mixers. In the architecture in section 4.2.4, eight DDS generators were used to create a fine-tunable mixer running at the high output sample rate. If we instead choose to do the fine-tunable frequency translation at lower sample rates, the resource cost of the mixer would be lower. The problem then is that running the mixer at a low sample rate limits the range of frequency translation, since the

1In practice, using a filter with an even number of taps is often not desirable, since such a filter will contain a half-sample delay. Here, it is of no importance since the filter design was purely for the purpose of cost comparison.
highest frequencies will not be available until after the final interpolation stages. 
In order to still cover the entire output range while also running the fine mixer 
at low sample rates, we must therefore also add a coarse mixer at the end of the 
system. The coarse mixer can be allowed to have only a few possible translation 
settings, since the fine mixer can then be used to fine tune the transmit frequency 
around these coarse frequency settings.

One problem with this approach is that frequency translation at lower sample 
rates means that the system must allow for a broader spectrum of frequency con-
tent during interpolation, which puts more strain on the interpolation filter orders 
and increases hardware cost. It is then of interest to see if the simplification of 
the mixer/DDS hardware is worth the increase in filter hardware cost.

The actual implementation details of coarse mixers are described later in this 
report, in section 5.4. We can summarize the details by stating that when mixing 
with a sine/cosine of a frequency which is a subdivision of the sample rate, the 
sine/cosine values will be very regular. The mixer therefore won’t need an entire 
DDS and complex multiplier, since all the output results can be implemented using 
simpler operations such as negations, additions and constant multiplications along 
with some control logic. A rule of thumb for now is to use mixers with frequency 
steps down to at most $f_s/16$ and preferably only to $f_s/8$, and that we can then 
consider these coarse mixers free in terms of multipliers/BRAM cells.

There are several different configurations of interpolations and mixers which 
can fulfill the system specification. By calculating their effects on interpolation 
filter orders and the sample rate at which the fine mixer along with its complex 
multiplier will be operating at, we can evaluate their resource usage. Whenever 
we use a mixer in the interpolation chain, the effective bandwidth of the signal 
will be increased. If we for example mix a ±10 MHz baseband with a mixer range 
of ±50 MHz, the next interpolation filter must have a passband that encompasses 
±60 MHz to account for all mixer settings. The placement of a mixer and the limit 
on its range of LO frequencies must therefore always be carefully considered, since 
it can have a huge effect on the following interpolation filter orders.

Some configurations of fine and coarse mixers are shown in Table 4.8. Note 
that since a 14-bit DDS with 32-bit phase width costs a single BRAM cell when 
running at 200 MSps, we do not have much to gain from moving the DDS to lower 
sample rates than that, as is done in the first row. The single BRAM cannot be 
subdivided further, and the only gain is that the complex multiplier used with the 
DDS costs half as much. This is weighed up by the fact that we require several 
coarse mixers in order to cover all frequencies since the fine mixer is running at 
lower sample rates and cannot cover as much in its range. The coarse mixers will 
not cost multipliers to implement, but do require a fair amount of logic slices.

In row two, a decent basic configuration is shown. It uses two $f_s/8$ mixers 
and the DDS is running at 200 MSps. Rows three to five show various ways of 
lowering the cost of this configuration, by using either $f_s/16$ mixers (which are 
slightly harder to implement) or by using a double DDS configuration running 
at 400 MSps. Note that moving the DDS to 400 MSps doubles the cost of the 
complex multiplier which is used as mixer. In row six, we see that moving the 
DDS even further does not actually lower the cost anymore, since the quadruple
<table>
<thead>
<tr>
<th>Description</th>
<th>Intp. orders</th>
<th>Estim. cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_s/8$-mixers at 400 MSps, 800 MSps and at output, fine mixer at 100 MSps</td>
<td>51 15 35 15 15 15</td>
<td>74 1</td>
</tr>
<tr>
<td><em>Multiplicative cost:</em></td>
<td>4 2 10 8 16 32</td>
<td></td>
</tr>
<tr>
<td>$f_s/8$-mixers at 800 MSps and at output, fine mixer at 200 MSps</td>
<td>51 15 11 27 15 15</td>
<td>76 1</td>
</tr>
<tr>
<td><em>Multiplicative cost:</em></td>
<td>4 2 4 14 16 32</td>
<td></td>
</tr>
<tr>
<td>$f_s/8$-mixers at 800 MSps and at output, fine mixer at 400 MSps</td>
<td>51 15 11 11 15 15</td>
<td>72 1</td>
</tr>
<tr>
<td><em>Multiplicative cost:</em></td>
<td>4 2 4 6 16 32</td>
<td></td>
</tr>
<tr>
<td>$f_s/16$-mixer at output, fine mixer at 200 MSps</td>
<td>51 15 11 27 15 11</td>
<td>68 1</td>
</tr>
<tr>
<td><em>Multiplicative cost:</em></td>
<td>4 2 4 14 16 24</td>
<td></td>
</tr>
<tr>
<td>$f_s/16$-mixer at output, fine mixer at 400 MSps</td>
<td>51 15 11 11 15 11</td>
<td>64 2</td>
</tr>
<tr>
<td><em>Multiplicative cost:</em></td>
<td>4 2 4 6 16 24</td>
<td></td>
</tr>
<tr>
<td>$f_s/16$-mixer at output, fine mixer at 800 MSps</td>
<td>51 15 11 11 7 11</td>
<td>64 4</td>
</tr>
<tr>
<td><em>Multiplicative cost:</em></td>
<td>4 2 4 6 8 24</td>
<td></td>
</tr>
</tbody>
</table>

*Table 4.8.* Cost estimation for various mixer orderings. Estimations for filter costs were calculated in the same way as in Table 4.3. Base cost for the fine mixer is 4 multipliers and 1 BRAM, which is then multiplied by the sample/clock ratio at the DDS location.
cost of the complex multiplier is not weighed up by the lower filter orders.
4.3 Chosen architecture

The architecture which was ultimately chosen for implementation in this thesis was that of row four in Table 4.8 of the multi-stage mixer section. It avoided the slightly inelegant problems of synchronizing multiple DDS:es which some other architectures had, and still had similarly low multiplier usage (and much lower BRAM usage).

The same type of architecture is immediately applicable for designing the receiver. Since the ADC:s run at a lower sample rate of 800 MSps, this architecture will require only five decimation stages, and we can use an $f_s/8$-mixer at the high-rate end of the receiver in order to get the same coarse frequency step and same fine mixer setting as that of the transmitter. The costs of the filters will then be equal to the first five in the transmitter architecture, as seen in Table 4.9.

<table>
<thead>
<tr>
<th>Description</th>
<th>Intp. orders</th>
<th>Estim. cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_s/8$-mixer at 800 MSps input, fine mixer at 200 MSps</td>
<td>$D_1$</td>
<td>$D_2$</td>
</tr>
<tr>
<td></td>
<td>51</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 4.9. Cost estimation for a receiver architecture, designed similarly to that of the 4th row transmitter architecture in Table 4.8.

4.4 DAC reconstruction filter design

In section 2.6, the theory behind DAC reconstruction distortion was presented. To summarize, when compensating for the reconstruction we reshape the system frequency response using a compensation filter, which should have a frequency response that approximates the inverse of the reconstruction sinc. Since the signal bandwidth at the output of the system is limited to between $-f_s/2$ and $f_s/2$, this is the only part of the inverse sinc that needs to be approximated in the compensation filter (the analog frontend of SDR14 will handle the analog low-pass filtering needed to remove the HF harmonics caused by the reconstruction).

Much thought and time were put into deciding on a good implementation strategy for the compensation filter. There are two aspects to the compensation that can be considered separately if we wish to. First, we want to correct for the in-band frequency domain tilting. Second, we want to compensate for the frequency-dependent amplitude level caused by the drooping amplitude characteristic, so that a frequency sweep with a constant amplitude input signal will produce a constant amplitude at the DAC output. Since the amplitude compensation cannot go over full scale at the system output, the actual implementation will instead attenuate lower frequencies, so that the attenuation remains constant over frequencies.
4.4.1 Inverse-sinc FIR filter at system output

The first solution that was investigated was that of an FIR filter at the system output which approximates an inverse sinc. A linear programming MATLAB script was used which produced an inverse-sinc approximating filter for a given number of taps. An example of the sinc distortion, compensation filter response, and compensated response for a 5 tap compensation filter is shown in Fig. 4.8.

\[ c = 2I + Q \cdot \frac{f_s}{f_{clk}} \cdot \frac{N - 1}{2^{symmetry}}. \]  

(4.8)

Figure 4.8. Frequency response of compensation filter up top, the DAC distortion at the bottom, and the complete system response in the middle.

In Table 4.10, the maximum errors over the Nyquist band are shown for varying filter lengths. The tilt error was calculated as the maximum deviation from the mean amplitude level inside the 20 MHz transmission bandwidth, and the amplitude deviation was calculated as the difference between the maximum and minimum values of the amplitude characteristic over the whole Nyquist band.

Note that the FIR filter does not do much to fix the tilt error regardless of the number of taps used. The amplitude deviation throughout the Nyquist band also remains quite large.

A big problem with the output filter approach is also that while it is quite simple in theory, the high data rate at the system output makes implementing the filter extremely expensive in system resources. Since there are eight parallel samples going out at the output for every clock cycle, an FIR filter implemented at that location would need to calculate eight output samples every cycle. Assuming that the filter is symmetric and the center tap normalized to 1, the multiplicative cost of the filter implementation can be calculated as
### Table 4.10

<table>
<thead>
<tr>
<th>Filter taps</th>
<th>Max. tilt error (dB)</th>
<th>Amplitude deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(unfiltered)</td>
<td>0.13</td>
<td>3.9 dB</td>
</tr>
<tr>
<td>3</td>
<td>0.107</td>
<td>1.45 dB</td>
</tr>
<tr>
<td>5</td>
<td>0.105</td>
<td>0.82 dB</td>
</tr>
<tr>
<td>7</td>
<td>0.103</td>
<td>0.56 dB</td>
</tr>
<tr>
<td>9</td>
<td>0.100</td>
<td>0.42 dB</td>
</tr>
<tr>
<td>11</td>
<td>0.094</td>
<td>0.34 dB</td>
</tr>
</tbody>
</table>

To take the filter shown in Fig. 4.8 as an example, the cost would be 32 multipliers.

#### 4.4.2 Baseband tilt-compensation

One thing to note about the compensation filter in the previous section is that we are only using a narrow portion of it at a given time instance. The input signal bandwidth of 20 MHz will occupy less than one 64th of the entire frequency spectrum from -800 to 800 MHz, and we are not really interested in what the filter does at other parts of the spectrum.

With this in mind, the prospect of implementing the sinc filter at lower sample rates seemed interesting. Since the system does not modify the signal bandwidth contents in any significant way, any frequency-shaping done at baseband will pass through unmodified to the end of the system. It is therefore possible to process the signal at baseband by trying to match what an ideal reconstruction filter would do at the frequency range which the signal band is going to end up at after mixing. This would achieve the same effect as a compensation filter at the output, with the difference being that the actual processing is done at much lower sample rates and therefore uses a much lower amount of hardware resources.

In order to implement this, the baseband compensation filter would need to have a parameterized frequency characteristic according to the transmitter frequency which the system is set to, since the filter needs to do different things for different broadcast frequencies. A logical way of doing this is to subdivide the frequency range into a number of subsections, and generate a set of baseband filters which match an ideal compensation filter in each of these subsections. The system could then look at the current mixer settings and select the appropriate filter for that frequency from a filter bank. Two alternatives were considered for the baseband programmable filter. The first was putting the filter directly at the system input, before any interpolation has been done. The system data rate is at its lowest in this part of the system, which allows for very cheap filter implementation.

If we look at Fig. 4.9 as an example, we can see that the mapping of a section of the output inverse-sinc response to a baseband filter requires a non-symmetric filter response between positive and negative frequencies. We know from transform theory that the only way to create such a filter would be to have a complex-
Figure 4.9. A section (300 to 325 MHz) of the sinc response, mapped to baseband. The bottom plot shows a corresponding inverse sinc filter at baseband.
4.4 DAC reconstruction filter design

valued impulse response. A complex-valued filter would encompass both data streams and have a cost of four multipliers per tap. A problem with this approach is that of actually designing a complex-valued filter. MATLAB does have the \textit{cfirpm} function, but this function is still not ideal as it works by defining pass- and stopbands of various amplitudes instead of approximating a given frequency response.

Another more simple solution is to instead place the programmable inverse-sinc filtering \textit{one} step into the interpolation chain. We have twice the bandwidth of that of the signal band available there, which allows us to use a DSP trick to simplify the filtering. By modulating the signal by $f_s/4$, the signal is shifted entirely to positive frequencies. By doing this, we can allow ourselves to use real-valued filtering, which design tools are readily available for. The filters should then be optimized to approximate a section of the inverse-sinc mapped to the frequency range $[0, f_s/2]$. Note also that since the system is specified to only allow an 80\% input bandwidth, these filters only need to accurately approximate the inverse-sinc over the range of $[0.1 f_s/2, 0.9 f_s/2]$.

A logical subdivision of output spectrum is to divide it by 64, since a 64th of the spectrum from -800 to 800 MHz is 25 MHz, which can be mapped directly to the baseband filter. Note that any transmission frequency inside one such section would be mapped to the same compensation filter, so the error would be worst for transmission frequencies near the edges of the subdivision since the compensation filter will be optimized for a transmission frequency which is at the section center.

A modification of the MATLAB script which was used in the previous section to optimize the output compensation filter was used to generate compensation filters for the 64 subdivision ranges. Worst case errors for transmission at both the section centers and the section edges were calculated and are summarized in Table 4.11.

<table>
<thead>
<tr>
<th>Filter taps</th>
<th>Maximum error (centered)</th>
<th>Max. error (near edge)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(unfiltered)</td>
<td>0.107 dB</td>
<td>0.107</td>
</tr>
<tr>
<td>3</td>
<td>0.0107 dB</td>
<td>0.0112 dB</td>
</tr>
<tr>
<td>5</td>
<td>0.0105 dB</td>
<td>0.013 dB</td>
</tr>
<tr>
<td>7</td>
<td>0.00403 dB</td>
<td>0.00597 dB</td>
</tr>
<tr>
<td>9</td>
<td>0.00404 dB</td>
<td>0.00649 dB</td>
</tr>
<tr>
<td>11</td>
<td>0.00245 dB</td>
<td>0.00479 dB</td>
</tr>
</tbody>
</table>

\textit{Table 4.11.} Maximum tilt error for ideally placed signal bands (centered on the subdivisions) and for worst case placement of signal bands (near the subdivision edges)

From the results in Table 4.11, we can see that there isn’t even any reason to go above 3 taps in filter length, since the worst case error is down to a hundredth of a decibel. We can compare this to the previously considered implementation in Table 4.10 where the tilt error remained at 0.1 dB all the way up to 9 taps. This is due to the fact that each set of coefficients for the programmable FIR needs only to approximate a very small region of the inverse sinc, while the implementation
A symmetric 3-tap filter that is scaled to unity for the mid coefficient will require one multiplication per output sample. For the programmable filter running at only 50 MSps, we could get away with time-multiplexing a single multiplier between the two channels, since the sample rate is only one fourth of the clock rate. We will also require a coefficient bank of 64 sets of filter coefficients. For a 3-tap filter with a unity-scaled mid coefficient, we need only store one actual coefficient per set. This corresponds to a memory with 64 rows of 18-bit values, which is not that bad. There is also the added cost of the \( f_s/4 \) mixers on either side of the programmable FIR. Since these are quite easy to implement as simple finite state machines with only negation and multiplexing operations, the costs of these are negligible.

4.4.3 Amplitude-compensation

In the previous section, the focus was primarily on correcting the tilting of the transmission band caused by the reconstruction. However, it is not enough to just attain a flat transmission band. It is also desirable that the output amplitude for a constant amplitude input does not depend on transmission frequency. This means that we must not just compensate for the transmission band tilt, but we must also scale the entire transmitted signal in order to compensate for the frequency-dependent amplitude of the reconstruction.

One possible solution would be to scale the three-tap filters so that they both flatten the signal band and set the correct amplitude level. However, there is a problem with this approach. Recall that every transmission frequency inside one of the 64 sections will map to the same FIR filter. Suppose we let the transmitter input be a sinusoid at a constant frequency, and perform a sweep of the transmission frequency across the entire Nyquist band. This means that during a sweep, the sinusoid would be scaled by a constant value for all transmission frequencies within each 25 MHz section.

If we looked at the system output prior to the DAC for such a configuration, the amplitude vs frequency plot would then look like a sort of staircase which follows the curve of an inverse sinc. During a frequency sweep, the amplitude level at the output would be something similar to a sawtooth wave, since every time the sweep enters a new subdivision range, the amplitude of the sine will jump to the value of the filter in the new region. Figure 4.10 shows the output level of such an implementation when keeping a sinusoid in the center of the input signal band and sweeping the system transmission frequency. We can see that the amplitude jumps by about 0.25 dB for the worst case. Since the system could very well be used by a customer in a measurement setup which performs frequency sweeps, this is undesirable behaviour.

A better solution would be to let the tilt correction filters keep the signal level unchanged and to add one scaling multiplication per channel instead. The scaling factor could be made either modifiable from the software API, or calculated in hardware based on the mixer settings. That way, during a frequency sweep, this scaling factor could be made to follow the inverse sinc to a very exact degree by
Figure 4.10. Frequency sweep amplitude over frequency, when scaling the 64 FIR filter coefficient sets as amplitude compensation.
setting it to the correct value every time the mixer setting is changed.

Having the amplitude scaling separate from the tilt correction also allows for the option of turning the scaling on and off, depending on if the user is interested in a constant amplitude over all frequencies or not. If the user wishes to just transmit at a single frequency, the scaling should probably be turned off in order to avoid lowering the SNR by attenuating the signal.

4.4.4 Chosen reconstruction filter architecture

We can do a cost analysis of the programmable FIR. Using a single hardware multiplier, we could potentially perform four multiplications for every pair of I and Q samples, since the sample rate of 50 MSps is only one fourth of the clock rate. We need to filter both I and Q and multiply them by a scaling coefficient. If we scale the 3-tap filter so that the center tap is normalized to 1, and let the two other taps share multiplier (since they have the same coefficient due to symmetry), only two multiplications per I/Q pair would be needed. The scaling would also require two multiplications, which means that a single hardware multiplier with its four multiplications per sample pair would theoretically be enough for implementing both the FIR filters and the level scalers.

If the amplitude scaling coefficient is to be calculated in hardware, this would require additional arithmetic operations as well. Since the coefficient calculation can be allowed to take some time from setting the mixer frequency to the calculation being finished, this part could be done using a single hardware multiplier which would do the computations iteratively. A cost comparison between this architecture and a comparable output FIR implementation is shown in Table 4.12. The programmable FIR can be seen to be better in all aspects. For obvious reasons,

<table>
<thead>
<tr>
<th>Tilt error</th>
<th>Ampl. error</th>
<th>Mult cost</th>
<th>Memory cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-tap FIR at output</td>
<td>0.105 dB</td>
<td>0.82 dB</td>
<td>32</td>
</tr>
<tr>
<td>Programmable 3-tap FIR at 50 MSps with polynomial scaling</td>
<td>0.01 dB</td>
<td>0.01 dB</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 4.12. Comparison between FIR filter at the output, and programmable filter at baseband for reconstruction filtering.

the programmable FIR with polynomial amplitude scaling was therefore chosen for implementation, due to superior performance and cost compared to that of an approximating FIR at the system output.

4.5 MATLAB modeling

A model of the entire system was created in MATLAB before any implementation in Verilog was done. This had the very useful effect of ironing out some design
issues early in the project without having to deal with the lengthy process of implementation in hardware. In particular, quantization effects throughout the system could be analyzed by adding quantizations after filters, multiplications and other operations. This also allowed for reviewing filter scaling factors, signal levels and overflow risks in the system.

Some details such as internal quantization inside the FIR filters, quantization method and noise shaping for the DDS and the coefficient quantization method for the constant multiplication in the coarse mixers were not taken into account in the model, since this was too early in the project to know for sure what structures would be used.

MATLAB scripts were also written to generate test data input for the system. The scripts were designed to produce data both for use in MATLAB and for importing to the memory storage in the Verilog simulator and finalized hardware. This makes testing more accurate since hardware and HDL simulator tests can be run with the exact same data as the MATLAB model, and any problems can quickly be isolated by comparing the outputs. Some examples of input data which was added to the script were complex exponential signals at various baseband frequencies, unit impulses for measuring the system impulse response, and random QPSK- and QAM-encoded data at various symbol rates.

Figure 4.11 shows a test of both the transmitter and receiver MATLAB models, with the transmission frequency set at 335 MHz. The signal from the transmitter model is fed back into the receiver module, and the baseband signal is reacquired without visible changes in the signal spectrum. The inverse-sinc reconstruction filter was also modeled in MATLAB, complete with both the coefficient set selection and level scaling. This was tested by using an impulse as the transmitter input, and then simulating the output for transmission at 64 different frequencies across the Nyquist band. A plot with the superimposed resulting spectrums for all the transmission frequencies is shown in Fig. 4.12.
Figure 4.11. Simulation of both the transmitter and receiver in MATLAB, for transmission at 335 MHz. Transmitter output has been looped back into the receiver.
Figure 4.12. Superimposed amplitude characteristics of the transmitter system for various transmission frequencies, showing the inverse-sinc reconstruction filter characteristic.
Chapter 5

Implementation

5.1 System block schematics

Block schematics were drawn up to match the architecture which was decided upon in the previous chapter. The transmitter block schematic is shown in Fig. 5.1 and the block schematic of the receiver is shown in Fig. 5.2.

5.2 Interpolation and decimation filter implementation

The firhalfband() function from the MATLAB filter design toolbox was used to design equiripple halfband filters that matched the specifications that were decided on in Chapter 4. The six interpolation filter impulse responses can be seen in Fig. 5.5. The first five filters are also the exact same filters used in the decimations for the receiver.

When the implementation of these filters was started, it became obvious that Xilinx CORE Generator was not equipped to generate filters that could handle sample rates above the clock rate. Therefore, some time was spent on designing a generic FIR filter module that would work for parallel data. There are some main differences between the standard-form implementation of an FIR filter and a parallelized implementation:

- The delay line will store several samples per clock cycle
- For an $M$-parallel filter, $M$ output samples must be calculated for every cycle.
- When a calculation is to use several consecutive samples, that selection of samples must wrap around to the following delay line slot when it has reached the end of the current slot’s set of samples.

An example of how the delay line is indexed in a parallelized FIR filter is shown in Fig. 5.3 and 5.4. These demonstrate how the first two output samples of a
Figure 5.1. Final transmitter block schematic.

Figure 5.2. Final receiver block schematic.
4-parallel 5-tap symmetric FIR filter are calculated. The calculation of sample $y[4n]$ uses samples $x[4n]$ to $x[4n-4]$, and the calculation of $y[4n-1]$ uses samples $x[4n-1]$ to $x[4n-5]$. In the same way, we calculate $y[4n-2]$ and $y[4n-3]$ for a total of four output samples per clock. Any samples of the input beyond the first four (which are available directly, at the filter input) have to be taken from the delay line. Samples $x[4n-4]$ through $x[4n-7]$ are located in the first delay line slot, $x[4n-8]$ through $x[4n-11]$ in the second slot, and so on.

The Verilog implementation was made easier by mapping the parallelized delay line onto another data vector in the code. This virtual data vector pointed at the exact same samples as in the parallel delay line, but indexed a single sample at a time instead of several parallel ones. Generate-statements were used to generate $M$ identical standard FIR filters (with sample rate equal to the clock rate), one for every output sample. These were provided with input samples taken from the single-sample data vector, with the first subfilter starting at index $x[4n]$ and the rest having increasing offsets, up to filter $M$ starting $M-1$ samples into the delay line at $x[4n-(M-1)]$. In order to meet timing requirements, generate-statements were also used to create a pipelined adder tree for each filter, since the summation of all the multiplier outputs cannot be performed in one clock cycle. By using adders with two inputs and one output, the summation tree ends up having a depth of $\lceil \log_2(N_{\text{multipliers}}) \rceil$.

The module was parameterized to allow instantiation of any $N$-tap, $M$-parallel symmetric FIR filter. Both even and odd filter lengths were allowed for by setting instantiation parameters. One instantiation flag was also added for filters with a normalized center tap. When the flag is set, the mid coefficient multiplier is replaced with a feed-forward. This is useful for making the implementation cheaper in filters with a center tap equal to 1, since we save one multiplier per output sample in this way.

In order to set the coefficients of the filter, an input port was added where the coefficient values were inputted at 18 bits each. This filter module was used to create all the interpolation and decimation filters which used sample rates above the system clock. It could be used to instantiate filters at sample rates below the clock rate as well, but since it contains no code for time-multiplexing the multipliers, this is not a good idea. The filter was also used for the inverse-sinc implementation in section 5.3, due to the coefficient input port making it very easy to reprogram the filter coefficients on the fly.
Figure 5.3. Calculation of $y[4n]$ in a 5-tap 4-parallel symmetric FIR.

Figure 5.4. Calculation of $y[4n - 1]$ in a 5-tap 4-parallel symmetric FIR.
Figure 5.5. Impulse responses and amplitude characteristics of the six interpolation half-band filters. The first five are also the same filters used for decimation in the receiver.
5.3 DAC reconstruction filter implementation

In section 4.4.2, the programmable 3-tap filter was picked as the architecture of choice for implementation. The filter which was created for the parallel interpolation and decimation FIR filters in section 5.2 had a coefficient input port, making it very easy to reprogram on the fly. Because of this, this filter module was used for the compensation filter as well, since we want to be able to reprogram the filter based on the mixer settings. For practical reasons and in order to make the code more readable, the I and Q filters used one hardware multiplier each.

In order to select the appropriate filter, a 6-bit indexing number (since $2^6 = 64$) was calculated by sign-extending and adding together the four bits from the coarse mixer’s frequency setting and the two top bits of the fine-tunable mixer. This results in any transmission frequency between 0 and 25 MHz being mapped to filter $h_0[n]$, any frequency between 25 and 50 MHz to $h_1[n]$, and so on. The filter bank is two’s-complement indexed, so a transmission at $-25$ to 0 MHz maps to filter $h_{63}[n]$, $-50$ to $-25$ MHz to $h_{62}[n]$, etc.

In section 4.4.2, one possible compensation scheme was to have the system automatically calculate a scaling factor for the current mixer settings. This would be done in order to compensate for the frequency-dependent signal level of the reconstruction. Such a system would have to calculate a scaling factor using the current mixer settings as input parameters.

![Polynomial compensation curve and reconstruction droop](image1)

![Resulting amplitude curve](image2)

**Figure 5.6.** Plot of the inverse-sinc amplitude polynomial evaluated over the entire nyquist frequency range, and the resulting amplitude curve after reconstruction.

MATLAB was used to fit polynomial equations to match the inverse sinc amplitude curve as a function of the transmit frequency. It could be seen that an fourth-order polynomial produced a result with only 0.01 dB in error. Since the sinc is symmetric around 0 Hz, the resulting polynomial contains even-numbered
5.3 DAC reconstruction filter implementation

exponential terms only, according to

\[ s = a + bx^2 + cx^4. \]  
(5.1)

Since there isn’t any reason for requiring for the scaling factor to be calculated immediately after the mixer settings change, this calculation can be implemented using a single multiplier, by letting it perform all of the required multiplications iteratively. The following equation shows how the polynomial can be partitioned into singular operations as

\[ s = a + x^2(b + x^2(c + 0)). \]  
(5.2)

This calculation was implemented using a finite state machine, one hardware multiplier and one adder. These are the states in numbered order (note that the hardware multiplier has a latency of 4 cycles), where \( x \) is a number between -1 and 1, corresponding to a transmission frequency between \(-f_s/2\) and \(f_s/2\):

0: If there has been a change in the mixer settings, go to 1, else loop to 0
1: Calculate \( x \times x \)
2-4: (wait for multiplication to complete)
5: Store \( x^2 \) in register
6: Calculate \( c \times x^2 \)
7-9: (wait for multiplication to complete)
10: Calculate \( b + c \times x^2 \)
11: Calculate \( x^2 \times (b + c \times x^2) \)
12-14: (wait for multiplication to complete)
15: Calculate \( a + x^2 \times (b + c \times x^2) \)
16: Output \( y = a + x^2 \times (b + c \times x^2) \)

The default setting of the FSM is to simply increase the FSM counter by 1 per clock cycle unless at state 0 where the FSM needs to wait for a change in transmit frequency.

The polynomial coefficients which were used for the final implementation are shown in Table 5.1. A plot showing the polynomial evaluated over the entire frequency range and the corresponding compound amplitude characteristic after the reconstruction, is shown in Fig. 5.6. We can see that the resulting curve only ripples by about \(\pm 0.01\) dB

<table>
<thead>
<tr>
<th>Coeff</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.637298583984375</td>
</tr>
<tr>
<td>b</td>
<td>0.248367309570313</td>
</tr>
<tr>
<td>c</td>
<td>0.111877441406250</td>
</tr>
</tbody>
</table>

Table 5.1. Polynomial coefficient values for fourth order inverse-sinc amplitude approximation.
Figure 5.7. Block schematic of the entire final inverse-sinc implementation.

Figure 5.7 shows a block schematic of the entire compensation filter. An enable input was added to both the tilt- and amplitude-compensation blocks, allowing the user to bypass them if desired.
5.4 Mixers

Frequency translation is an integral operation in any up- or downconverter. A transmission/reception range from \(-f_s/2\) to \(f_s/2\) for the A/D and D/A converters was one of the goals of the thesis project, and while the theory behind frequency translation is quite simple, it is more of a challenge to find an implementation that performs according to specification while at the same time having a low resource usage. During the modeling phase, it was decided that a good mixer configuration would be one fine-tunable mixer at a lower sample rate, and two coarse mixers at higher sample rates. The coarse mixers should each be capable of mixing by multiples of \(f_s/8\) and should, of course, use quadrature processing. The coarse mixers should also utilize the fact that all multiplications will be with a few known constants, which should be possible to implement without using hardware multipliers.

5.4.1 Low-rate high-resolution mixers

Fine-tuneable mixers at sample rates equal to or lower than the clock rate, can be implemented using Xilinx CORE Generator. As we saw in the previous chapter in Table 4.4, 14-bit data is probably enough to maintain an 85 dB SFDR. 15-bit data was used in the implementation in order to be on the safe side, since the BRAM cost is still extremely small. A DDS sine and cosine generator was created with a 32-bit phase width, and 15-bit output data. A streaming port was set up for the DDS so that the oscillator frequency could be changed on the fly. A DDS running at a 200 MHz clock with 32-bit phase resolution gives a frequency resolution of

\[ f_{\text{step}} = \frac{f_{\text{clk}}}{2^{32}} \approx 0.05 \text{ Hz}. \]

(5.3)

The complex multiplier for the fine mixer was also instantiated using COREgen. Judging from its resource usage of four DSP slices, it appears to be a straightforward pipelined implementation of equation (5.4).

\[(A + jB)(C + jD) = (AC - BD) + j(AD + BC).\]

(5.4)

This setup was used for the low-rate mixer in both the receiver and transmitter, for the final system implementation. There is also one slight modification to the fine-tunable mixer for the receiver side implementation. In communication schemes such as QPSK or QAM, the received signal has an unknown phase-shift, which means that the received symbol grid will be rotated by some unknown angle. The decoder must use some kind of algorithm to find the phase offset and apply the corresponding inverse rotation, in order to correctly align the symbol points. This offset could be added in the decoder, but it is more resource-efficient to let the decoder control a phase-offset input to the DDS in fine-tunable mixer instead. Such an input port was therefore added to the receiver mixer.

5.4.2 High-rate parallelized coarse mixers

When mixing with sine waves that are subdivisions of the sample rate it is of interest to see what the actual modulating sine-wave will look like when sampled
at \( f_s \). In figure 5.8, we can see (from top to bottom,) one cosine with a frequency of \( f_s/2 \), one with a frequency of \( f_s/4 \), and one with a frequency of \( f_s/8 \). We can see where the sampling points end up at the sine. It is obvious that there is no need at all for a large DDS look-up table and a complex multiplier at these frequencies, since the modulating waveform will have a very small set of possible values which appear at regular intervals. When implementing a coarse mixer, we want it be able to multiply a signal sample by any value that the local oscillator signal might have. For a LO frequency at a sample rate subdivision of \( f_{LO} = f_s/K \), the LO signal can be written as

\[
x_{LO}[n] = e^{2\pi j n f_{LO}/f_s} = e^{\pi K/2 j n}
\] (5.5)

We can see that the complex exponential in the LO signal can have any angle \( \phi = n \cdot \frac{\pi}{K/2} \), for all integers \( n \). Our mixer must therefore be able to rotate the input in the complex plane by all such angles. From the problem analysis, it was clear that mixers at \( f_s/8 \) and \( f_s/16 \) were desirable, which then corresponds to angles of \( n \cdot \frac{\pi}{4} \) and \( n \cdot \frac{\pi}{8} \) respectively.

There are no cores in COREgen that would take advantage of the inherent simplicity of a coarse mixer. Instantiating a complex multiplier and a DDS would use several multipliers, despite it not being necessary. Therefore, this block had to be created manually. Fortunately, rotations in the complex plane for angles such as \( \pi/4 \) and \( \pi/8 \) are very common in digital algorithms. In particular, FFT algorithms require such operations when performing constant multiplication with
the *twiddle factor*. Because of this, there exists a large amount of literature which deals with optimization of such rotator structures.

In [4], a number of rotator kernels are presented which utilize a very small amount of adders to perform the constant multiplication. The values for the complex exponential at a number of angles are quantized using an optimization method which is aware of the number of additions that would be needed in order to implement a shift-and-add multiplication structure for the resulting numbers. By allowing the optimizer to use any real-valued scaling factor for the quantized numbers as long as the ratio between them is correct, the rotator output will scale the signal level by an amount which is not a power of two. This can easily be compensated for in a nearby filter, and is not a problem. In return for this scaling, the optimizer finds scaling factors which causes all quantized constants to require a low amount of adders in their multiplier implementation.

For this thesis, two different rotators were borrowed from the article, which both achieve a precision which corresponds to 15-bit coefficients. The first rotator uses the numbers 584, 384 + j384 to represent $e^0, e^{\pi/4}$, and requires four adders in its implementation. A quick calculation shows that their ratio appears to give a correct approximation: $(384 + j384)/584 \approx 0.707 + j0.707$. The second allows for even lower angles and uses the numbers 669, 618 + j256, 543 + j543 to represent $e^0, e^{\pi/8}, e^{\pi/4}$, at a slightly higher cost of eight adders.

Block schematics for implementation of the rotators were also provided in the article, and these were used to produce the actual Verilog code for the rotators. The only modification was to add appropriate signal scaling to avoid unnecessary signal gain in the mixer block. Since the number 1 was represented by 584 and 669 respectively in the two rotators, the closest power-of-two scaling factor was 512. This causes the $f_s/8$-mixer to have a signal gain of $20 \log_{10} \frac{584}{512} \approx 0.5$ dB, and the $f_s/16$-mixer to have a gain of $20 \log_{10} \frac{669}{512} \approx 2.3$ dB. These scaling factors can easily be compensated for by scaling one of the interpolation filters to have a corresponding attenuation.

The FFT kernels only deal with a few select angles in the range of $[0, \pi/2]$, but in order to use them as mixers we want to be able to use any multiple of $\pi n/4$ or $\pi n/8$. The values we get from the original rotators and the values which we require for implementing mixers are shown in the plots in Fig. 5.9.

A good first step towards this is to add a stage before the kernel which performs rotation by multiples of $\pi/2$. Since this corresponds to a multiplication by $j$, the output for such an operation is simple to produce. For an input of $x_I + jx_Q$, the corresponding output for rotations of $0, \pi/2, \pi, 3\pi/2$ is $-x_Q + jx_I, -x_I - jx_Q, x_Q - jx_I$, respectively. It is therefore enough to add a multiplexer and a negation stage to each input, along with some control logic.

By doing this we have extended the range from $[0, \pi/2]$ to cover the entire unit circle. In other words, if the basic rotator covers the range of $[0, \pi/2]$ with a spacing that is fine enough for the mixer there, adding a $\pi/2$-multiple rotation before it will allow it to cover the entire unit circle. This is the case for the $f_s/8$ rotator, since it could handle all multiples of $\pi/4$ in the range of $[0, \pi/2]$ from the start. In the second rotator, however, we are still missing the capability for rotation by the angle $3\pi/8$. It would be best if this functionality could be added
Figure 5.9. Possible multiplications using the original rotators, compared with necessary multiplications for implementing fs/8 and fs/16 mixers.
5.4 Mixers

without substantial redesign of the rotator.

Let us define two operations that can be performed on a complex number using simple hardware. First, we have conjugation, which will be denoted \(x^*\). This consists of a negation of the imaginary part, so that \((x_I + jx_Q)^* = (x_I - jx_Q)\). A conjugation causes the angle of the complex number to be negated, so that \((re^\phi)^* = re^{-\phi}\).

The second operation will in this thesis be called a *flip* and denoted as \(\text{flip}(x)\), and consists of exchanging the real and imaginary part: \(\text{flip}(x_I + jx_Q) = x_Q + jx_I\). If we look at this in the complex plane, it would mean that the number exchanges its vertical and horizontal coordinates. This is equivalent to mirroring the number across the angle \(\pi/4\), for an equation of \(\text{flip}(re^\phi) = re^{\pi/2 - \phi}\).

Using these two operations, it is possible to find a way of modifying the rotator inputs and outputs so that we achieve a \(3\pi/8\)-rotation even though the rotator is set to \(\pi/8\). Let us consider a data sample \(x = e^\phi\). We set up an expression for the desired rotation, and start modifying it until we get something that is usable with the \(\pi/8\)-rotator, as given by

\[
xe^{3\pi/8} = e^\phi e^{3\pi/8} = e^{\phi + 3\pi/8} = (-e^\pi)e^{\phi + 3\pi/8} = -e^{\phi + 11\pi/8}
\]

\[
= -e^{\phi - 5\pi/8} = -e^{\phi - \pi/2 - \pi/8} = -(e^{\pi/8 + \pi/2 - \phi})^* = -(\text{flip}(x)e^{\pi/8})^*.
\]

To summarize the result, we get the desired rotation using

\[
xe^{3\pi/8} = -(\text{flip}(x)e^{\pi/8})^*.
\]

By rotating a flipped version of the input and performing a negated conjugation on the rotator output, the result is a rotation of \(3\pi/8\). This final modification allows all multiples of \(\pi/8\) to be reached, which is all that is needed to create an \(fs/16\)-mixer.

By taking the output value for each rotator angle in sequency, an entire period of the modulating waveform is created. The frequency spectrums of these waveforms for our two low-adder rotators are presented in Fig. 5.10. It can be seen that the quantized coefficients cause spurs to appear and that the SFDR is about 85 dB, which is adequate for this system.

The coarse mixers will run at high data rates with several output samples being calculated per clock cycle, and every output sample only depends on a single input sample. The logical method was then to implement the rotators as submodules, which could then be instantiated a number of times corresponding to the amount of output samples per cycle for the mixer.

In order to make the submodules multiply by the correct sequency of local oscillator values together, one phase increment input and one sample number input was added. The sample number tells the unit which sample in the local oscillator period cycle (eight samples for \(fs/8\), sixteen for \(fs/16\)) that it should multiply with. The phase increment port selects the modulating frequency as given by

\[
f = \frac{d\phi}{2\pi dt} = \frac{N_{\text{inc}}\pi}{2\pi T_s (K/2)} = \frac{N_{\text{inc}}fs}{K},
\]
where the base increment angle of $\frac{\pi}{K/2}$ is taken from (5.5). A phase increment value of $N$ for an $f_s/K$-mixer will therefore set the LO frequency to $Nf_s/K$.

For a mixer where the local oscillator cycle stretches out over several clock cycles, the sample number inputs of the submodules should alternate correspondingly. An example would be a 4-parallel $f_s/8$-mixer, where the eight-sample cycle of the local oscillator would extend across two clock cycles. There would be four $f_s/8$ submodules instantiated in the mixer, and the sample settings would alternate between (0,1,2,3) and (4,5,6,7) in order to get a correctly ordered output sequence.

5.5 Scaling and wordlengths

At the PC-interface side of the transceiver, 16-bit ports were used according to specification. A 24 bit signal data wordlength was used internally throughout the system. The wordlength was chosen because of the filters generated through COREgen, where an input width of 24 bits or lower should be used if the resource usage is to be kept down. Symmetric FIR filter implementations utilize multiplier sharing in order to keep resource costs down. Since this means that an adder is placed before each multiplier to sum two data values together, COREgen will assume that this addition causes the wordlength to increase by one bit. Since a single multiplier can only handle a 25-bit number at most, a symmetric filter using wordlengths above 24 bits will therefore require more than one hardware multiplier in order to perform a single multiplication.

A 24-bit wordlength is not really necessary for performing according to the
5.5 Scaling and wordlengths

specification. Just adding two guard bits and a couple of fractional bits would be enough to ensure that no significant spurs will occur from the quantization errors. Using a lower wordlength of 20 bits would not decrease the multiplicative cost of the system, since a single multiplier can handle the 24-bit wordlength as well. What it would do, however, is decrease the amount of logic slices used in adders, pipeline registers and other similar blocks. Since the blocks from the system might be reused under varying circumstances by the end-user, a long wordlength might at some point be desired, which is why the 24-bit wordlength was kept for the final implementation.

At the system outputs, the additional fractional bits were truncated using rounding, and the guard bits were removed and used to perform saturation clipping of the output waveform. A test of the saturation block can be seen in the results chapter, in Fig. 6.3.
Chapter 6

Results

In this chapter, data on the final hardware implementation will be discussed. Various measurements which show that the system is performing as specified will be shown and resource costs and module descriptions will be presented.

6.1 Hardware testing

Concurrently with this thesis project, two other students were doing their thesis project at SP Devices. The title of their project was “Live Demonstration of ADC Interleaving Post-correction Performance”, and had the goal of demonstrating the performance of one of SP Devices ADC error correction algorithms. This was done by transmitting various digital communication signals such as QPSK or QAM modulated data out on the analog outputs. A strong blocking signal from a signal generator was then added at a higher frequency, selected so that the ADC interleaving error would alias the blocker down into the transmitted signal band. The signal would then be fed back through the analog inputs and through a receiver, and finally plotted on the complex plane. The QPSK/QAM symbol points could then be seen, and the error correction could then be turned on and off in order to visualise the performance improvement [11].

In order to construct such a measurement system, a transceiver such as the one presented in this thesis is required. Therefore, their thesis project dealt with the baseband processing required for QPSK/QAM encoding and decoding, and then relied on the results of this thesis for the actual up- and downconversion of the transmitted signal channels. During hardware testing, their thesis project turned out successfully, which proves the functionality of both their system and that of this thesis.

In Fig. 6.1 we can see a plot of the frequency spectrum at the transmitter output of their system. Four W-CDMA data channels are being sent within the 20 MHz signal band of the transmitter. Each channel has a 5 MHz bandwidth and a $\approx 3.84$ MSps symbol rate. This transmission setup was successfully used to communicate with a "ROHDE & SCHWARTZ SMIQ03B" signal generator, which gives additional proof of the system functionality.
Figure 6.1. Frequency plot of the transmitter output signal from the other thesis project at SP Devices, which utilized the transceiver from this thesis. The data is from an actual hardware run of the system.

A plot of one of the transmitted 64-QAM grids used in their system is shown in Fig. 6.2.
Figure 6.2. Complex plane plot of received symbols from the other thesis project.
6.2 Measurements

6.2.1 Scaling

The system was designed to have near-constant amplitude. The following printout shows peak signal levels measured in the Verilog simulator for a signal that has been sent through transmitter and back through the receiver to baseband. The signal amplitude in this measurement was defined as the magnitude of the complex number $x_I + jx_Q$. During the measurement, inverse-sinc compensation was turned off in order to not disturb the signal level. The input signal was a complex exponential, scaled to have a peak magnitude of 90% full-scale in order to test for overflow.

Transmitter
-------------
Peak signal level at txinput: -0.92 dBFS, RMS average at: 2.1 dBFS

Peak signal level at intp1: -3.2 dBFS, RMS average at: -0.21 dBFS
Peak signal level at sinc: -3 dBFS, RMS average at: -0.21 dBFS
Peak signal level at intp2: -3.2 dBFS, RMS average at: -0.21 dBFS
Peak signal level at intp3: -3.2 dBFS, RMS average at: -0.21 dBFS
Peak signal level at intp4: -3.2 dBFS, RMS average at: -0.21 dBFS
Peak signal level at intp5: -3.2 dBFS, RMS average at: -0.21 dBFS
Peak signal level at intp6: -3.2 dBFS, RMS average at: -0.21 dBFS

Peak signal level at dacout/mix2: -0.84 dBFS, RMS average at: 2.1 dBFS

Receiver
-------------
Peak signal level at rxinput: -0.84 dBFS, RMS average at: 2.1 dBFS

Peak signal level at recmix2: -0.33 dBFS, RMS average at: 2.6 dBFS
Peak signal level at dec5: -0.33 dBFS, RMS average at: 2.6 dBFS
Peak signal level at dec4: -0.38 dBFS, RMS average at: 2.6 dBFS
Peak signal level at recmix1: -0.39 dBFS, RMS average at: 2.6 dBFS
Peak signal level at dec3: -0.39 dBFS, RMS average at: 2.6 dBFS
Peak signal level at dec2: -0.39 dBFS, RMS average at: 2.6 dBFS

Peak signal level at recout/dec1: -0.89 dBFS, RMS average at: 2.1 dBFS

Note that the first interpolation filter in the transmitter is scaled to have a $-2.3$ dB gain to compensate for the coarse $f_s/16$-mixer, which (as detailed in the implementation chapter) has a gain of $+2.3$ dB. The mixer is situated between interpolation 6 and the DAC output, and we can see that the signal level goes up
again to where it should be at that point. The same thing is done in the receiver, where the coarse $f_s/8$-mixer adds a gain of $+0.5$ dB after the receiver input, and is then compensated by $-0.5$ dB inside the final decimation filter.

A constant signal level is only maintained if the input signal actually fits the input bandwidth from the start. If there are frequency components outside the 80% bandwidth at the system input, the removal of these might cause peaking and ringing that could raise the peak amplitude.

The following printout shows what can happen if a full-scale signal is sent to the system where there is enough frequency content outside the 80%-bandwidth to cause ringing. A simple unit-step going from 0.98 to 0 was used as baseband input, and the mixer settings were set to 0 Hz to allow us to see only what the band-limiting effects will do to the signal level.

<table>
<thead>
<tr>
<th>Description</th>
<th>Peak Level</th>
<th>RMS Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak signal level at input</td>
<td>-0.18 dBFS</td>
<td>-0.24 dBFS</td>
</tr>
<tr>
<td>Peak signal level at intp1</td>
<td>0.91 dBFS</td>
<td>-0.25 dBFS</td>
</tr>
<tr>
<td>Peak signal level at sinc</td>
<td>0.91 dBFS</td>
<td>-0.24 dBFS</td>
</tr>
<tr>
<td>Peak signal level at intp2</td>
<td>0.91 dBFS</td>
<td>-0.25 dBFS</td>
</tr>
<tr>
<td>Peak signal level at intp3</td>
<td>0.94 dBFS</td>
<td>-0.25 dBFS</td>
</tr>
<tr>
<td>Peak signal level at intp4</td>
<td>0.94 dBFS</td>
<td>-0.25 dBFS</td>
</tr>
<tr>
<td>Peak signal level at intp5</td>
<td>0.95 dBFS</td>
<td>-0.25 dBFS</td>
</tr>
<tr>
<td>Peak signal level at intp6</td>
<td>0.95 dBFS</td>
<td>-0.25 dBFS</td>
</tr>
<tr>
<td>Peak signal level at dacout</td>
<td>0 dBFS</td>
<td>-0.34 dBFS</td>
</tr>
</tbody>
</table>

We can see that in the first interpolation stage, the removal of the high-frequency components causes a 1 dB overshoot in the step response which raises the peak level above 0 dBFS. Since there are extra integer guard bits inside the signal chain, a correct signal waveform is maintained all the way through the transmitter. Finally at the output, the saturation block clips the signal at 0 dBFS. Figure 6.3 shows the output waveform. It should be noted that overflows can still occur despite having an 80%-bandwidth input signal. An example is shown in Fig. 6.4, where the initial signal consists of two full-scale points surrounded by zeroes. Once interpolated by a factor of 2, the resulting signal peaks at a value of $\approx 1.2$. This can be explained by noting that the initial signal is symmetric around the point of 4.5, which is a nonexistant index in the initial signal. Since the signal looks like a symmetric pulse, it is logical that the highest signal peak would be found at the point of symmetry. This point does not appear until we interpolate the signal, causing the 4.5 point to map to a new actual index of 9. Ultimately, it is up to the end-user to make sure that the input signal will not overshoot due to frequency content outside the input bandwidth or from interpolation. Since it is easy to see if there is clipping occurring at the output, a good method is to just turn down the input amplitude until all clipping has disappeared.
Figure 6.3. Overshoot and clipping at system output, for a 98% full-scale step response at the input.

Figure 6.4. Interpolation of a full-scale signal, showing overshoot in the new samples between the middle samples of the initial signal.
6.2 Measurements

6.2.2 Inverse-sinc characteristic

Making sure that the inverse-sinc is operating as it is supposed to in hardware is of course an interesting measurement to perform. This presents a slight problem however, since the ADCs on the board do not have a constant frequency characteristic. This means that by doing simple amplitude measurements for different frequencies, it is unfortunately not possible to see an actual flat/drooping response with/without compensation. The effects of reconstruction distortion and the compensation filter will both be overpowered by the ADC characteristic. While this means that it is not possible to “prove” that the DAC response is made more flat by the filter, we can at least see if the inverse-sinc is doing what it is supposed to, by plotting the amplitude difference between measurements done with and without compensation.

This measurement was done by entering a constant input at baseband to the system (a constant can be considered a complex exponential at 0 Hz), and mixing it to a number of frequencies across the nyquist band. At each frequency, a record of 8192 samples was taken from the ADCs and the RMS average amplitude of the signal was calculated. By plotting this value as a function of transmit frequency, we get the graph in Fig. 6.5. It could be argued that the transceiver should be able to compensate for the ADC characteristic as well. However, equalization of the ADC response is more difficult than the inverse-sinc compensation in that it requires phase compensation as well, and was considered outside the scope of this thesis.
Figure 6.5. Hardware measurements of inverse sinc characteristic. Top plot shows amplitude averages for different frequencies with and without compensation. Bottom plot shows the difference between these measurements which should match the ideal inverse sinc curve (plotted with a dashed line).
6.3 Resource usage

The Virtex-6 FPGA used in this thesis has a total available resource count of 768 hardware multipliers, 416 BRAM cells and 37680 slices. The resource cost for the final transmitter block is shown in Table 6.1.

### Table 6.1. Transmitter resource usage.

<table>
<thead>
<tr>
<th>Block</th>
<th>Mult cost</th>
<th>Slice cost</th>
<th>BRAM cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpolation 1</td>
<td>6</td>
<td>184</td>
<td>0</td>
</tr>
<tr>
<td>Inverse sinc</td>
<td>4</td>
<td>208</td>
<td>0</td>
</tr>
<tr>
<td>Interpolation 2</td>
<td>4</td>
<td>139</td>
<td>0</td>
</tr>
<tr>
<td>Interpolation 3</td>
<td>6</td>
<td>140</td>
<td>0</td>
</tr>
<tr>
<td>Mixer 1 Comp.mult</td>
<td>4</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>Mixer 1 DDS</td>
<td>0</td>
<td>62</td>
<td>1</td>
</tr>
<tr>
<td>Interpolation 4</td>
<td>14</td>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td>Interpolation 5</td>
<td>16</td>
<td>265</td>
<td>0</td>
</tr>
<tr>
<td>Interpolation 6</td>
<td>24</td>
<td>414</td>
<td>0</td>
</tr>
<tr>
<td>Mixer 2</td>
<td>0</td>
<td>1292</td>
<td>0</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>0</td>
<td>96</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>78</strong></td>
<td><strong>2933</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td><strong>% of available</strong></td>
<td><strong>10</strong></td>
<td><strong>7.7</strong></td>
<td><strong>0.24</strong></td>
</tr>
</tbody>
</table>

The resource cost for the final receiver block is shown in Table 6.2.

### Table 6.2. Receiver resource usage.

<table>
<thead>
<tr>
<th>Block</th>
<th>Mult cost</th>
<th>Slice cost</th>
<th>BRAM cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mixer 2</td>
<td>0</td>
<td>319</td>
<td>0</td>
</tr>
<tr>
<td>Decimation 5</td>
<td>16</td>
<td>266</td>
<td>0</td>
</tr>
<tr>
<td>Decimation 4</td>
<td>14</td>
<td>137</td>
<td>0</td>
</tr>
<tr>
<td>Mixer 1 Comp.mult</td>
<td>4</td>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>Mixer 1 DDS</td>
<td>0</td>
<td>72</td>
<td>1</td>
</tr>
<tr>
<td>Decimation 3</td>
<td>6</td>
<td>171</td>
<td>0</td>
</tr>
<tr>
<td>Decimation 2</td>
<td>4</td>
<td>130</td>
<td>0</td>
</tr>
<tr>
<td>Decimation 1</td>
<td>6</td>
<td>185</td>
<td>0</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>0</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>50</strong></td>
<td><strong>1320</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td><strong>% of available</strong></td>
<td><strong>6.5</strong></td>
<td><strong>3.5</strong></td>
<td><strong>0.24</strong></td>
</tr>
</tbody>
</table>

The combined cost for a transceiver using both the transmitter and receiver is shown in Table 6.3.
Results

<table>
<thead>
<tr>
<th></th>
<th>Mult cost</th>
<th>Slice cost</th>
<th>BRAM cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total</strong></td>
<td>128</td>
<td>4253</td>
<td>2</td>
</tr>
<tr>
<td><strong>% of available</strong></td>
<td>16.7</td>
<td>11.3</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 6.3. Combined transceiver resource usage.

6.3.1 Evaluation of resource usage

If we compare the results in the previous section to those of our estimations in Tables 4.8 and 4.9, we can see that both transmitter and receiver exceed our theoretical calculations by 6 multipliers each. This discrepancy can be explained by the fact that the first three filters (which were generated through COREgen during implementation) used a slightly larger number of multipliers than the calculations showed as being the minimum. Since it’s hard to know for sure why COREgen does things this way without delving into datasheets and figuring out the implementation strategy it uses, the extra cost was considered acceptable and the blocks were left this way.

The final resource usage of the system turned out quite well, especially when seen in perspective to the total resources available on the Virtex-6. It is important to note that the standard way to use an SDR transceiver such as the one constructed in this thesis would be to perform any desired extra signal processing such as encoding, pulse shaping, etc. at baseband at a rate of 25 MSps. The transceiver would then interface with the A/D and D/A converters. Since the highest hardware costs are incurred for blocks which operate at high sample rates, such baseband processing will be cheap to implement.

As an example that puts things into perspective, suppose we wanted to do some extreme filtering at baseband, preceding the transceiver. Say we want to run a symmetric 100-tap filter for both the I and Q channels. The multiplicative cost of this would be:

\[
\left\lceil \frac{N_{\text{channels}} \cdot N_{\text{taps}}}{2} \cdot \frac{f_s}{f_{\text{clk}}} \right\rceil = \left\lceil 2 \cdot \frac{100}{2} \cdot \frac{1}{8} \right\rceil = 13
\]

Since we have \(768 - 128 = 640\) multipliers left over after implementing our transceiver, those 13 multipliers amount to only about 2% of the remaining resources. In other words, we could theoretically fit in 50 of these two-channel 100-tap filters at baseband along with our transceiver, in the FPGA. This goes to show that the leftover resources in SDR14 after implementing the transceiver will allow for a lot of powerful baseband processing before running out.

6.4 Deliverables

The resulting source code from the project was delivered in the form of two top module files; sdr14_txchain and sdr14_rxchain. These then use a large number
of submodules and COREgen blocks to perform all of the transceiver operations. The interface of each module will be briefly described below:

<table>
<thead>
<tr>
<th>Port name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idata_i</td>
<td>16</td>
<td>25 MSps in-phase input</td>
</tr>
<tr>
<td>qdata_i</td>
<td>16</td>
<td>25 MSps quad-phase input</td>
</tr>
<tr>
<td>valid_i</td>
<td>1</td>
<td>Valid signal for inputs, high value during a cycle will tell txchain to process the current inputs, while a low value will make txchain ignore them</td>
</tr>
<tr>
<td>idata_o</td>
<td>8 × 14</td>
<td>1600 MSps in-phase output</td>
</tr>
<tr>
<td>qdata_o</td>
<td>8 × 14</td>
<td>1600 MSps quad-phase output</td>
</tr>
<tr>
<td>valid_o</td>
<td>1</td>
<td>Valid signal for outputs</td>
</tr>
<tr>
<td>m1freq_i</td>
<td>32</td>
<td>Frequency setting for fine mixer, signed number representing range of -100 to 100 MHz (only -50 to 50 allowed)</td>
</tr>
<tr>
<td>m2freq_i</td>
<td>4</td>
<td>Mixer 2 frequency, signed number representing discrete multiples of 100 MHz</td>
</tr>
<tr>
<td>isinc_on</td>
<td>1</td>
<td>Active-high enable signal for inverse-sinc tilt compensation</td>
</tr>
<tr>
<td>isincamp_on</td>
<td>1</td>
<td>Active-high enable signal for inverse-sinc amplitude compensation</td>
</tr>
<tr>
<td>clk_1_4</td>
<td>1</td>
<td>200 MHz clock input</td>
</tr>
<tr>
<td>rst_i</td>
<td>1</td>
<td>System reset input</td>
</tr>
</tbody>
</table>

Table 6.4. Interface for sdr14\_txchain.
<table>
<thead>
<tr>
<th>Port name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idata_i</td>
<td>4 × 16</td>
<td>800 MSps in-phase input</td>
</tr>
<tr>
<td>qdata_i</td>
<td>4 × 16</td>
<td>800 MSps quad-phase input</td>
</tr>
<tr>
<td>valid_i</td>
<td>1</td>
<td>Valid signal for inputs, high value during a cycle will tell rxchain to process the current inputs, while a low value will make txchain ignore them</td>
</tr>
<tr>
<td>idata_o</td>
<td>16</td>
<td>25 MSps in-phase output</td>
</tr>
<tr>
<td>qdata_o</td>
<td>16</td>
<td>25 MSps quad-phase output</td>
</tr>
<tr>
<td>valid_o</td>
<td>1</td>
<td>Valid signal for outputs</td>
</tr>
<tr>
<td>m1phaseoffset_i</td>
<td>32</td>
<td>Phase offset setting for fine mixer, 0 to 2π.</td>
</tr>
<tr>
<td>m1freq_i</td>
<td>32</td>
<td>Frequency setting for fine mixer, signed number representing range of -100 to 100 MHz (only -50 to 50 allowed)</td>
</tr>
<tr>
<td>m2freq_i</td>
<td>3</td>
<td>Mixer 2 frequency, signed number representing discrete multiples of 100 MHz</td>
</tr>
<tr>
<td>clk_1_4</td>
<td>1</td>
<td>200 MHz clock input</td>
</tr>
<tr>
<td>rst_i</td>
<td>1</td>
<td>System reset input</td>
</tr>
</tbody>
</table>

*Table 6.5.* Interface for sdr14_rxchain.
6.5 Conclusions

In this report, a high rate up- and downconverter architecture was designed and implemented. The up-converter was fitted with a compensation filter for the on-board DACs. A low-cost, high-performance system architecture was decided upon after analysis of several different methodologies. Results from low-adder rotator research were used in section 5.4 to construct low-cost coarse mixers. Interpolation filters were created using polyphase-decomposed halfband filters. New implementational structures had to be created for the FIR filters in order to cope with the high sample rates, which required parallel processing.

The entire system was verified firstly using a MATLAB model, secondly through simulation of Verilog code, and finally through hardware testing in the form of the work done by Mikael Rothin and Johan Nilsson as described in section 6.1.

The resulting source code was delivered to SP Devices, and the project supervisor was satisfied with the result. Work will continue on the system in order to integrate it into the end-user development kits, and to add control functionality for the system to the SDR14 API.

6.5.1 Comparison to other systems

With a finished system, it is always interesting to compare the results to other similar projects and figure out whether the design decisions were sound or not. However, finding such similar projects proved surprisingly difficult. The majority of up- and down-converter systems operate at sample rates lower than, or equal to, the clock. In the few articles where the sample rates were high enough to require parallelization, there are still many design parameters left which completely reshape the resource cost and which can make a comparison impossible.

As an example, we can take the article referenced in [12], in which a quite similar system to that of this thesis is presented. An interpolation by 16 (from 75 to 1200 MSps) is performed and the final rate of 1200 MSps is 8:1 parallel, just like the final sample rate in the transmitter presented in our system. Judging from the plots in the article, their filter stopband attenuation is slightly lower than our system, at \( \approx 80 \) dB. The total multiplicative cost of the system in the article comes out at 112 multipliers, as compared to the 74 (we don’t count the reconstruction filter) of this thesis.

However, as we saw in chapter 4 of this report, the transmission bandwidth will have an immense effect on the resource cost of the interpolation filters. Since the input to their system is a 75 MSps signal which is three times wider than the 25 MSps input in this thesis, it is probable that they also maintain a wider passband during the up-conversion than the 20 MHz bandwidth in our system. Unfortunately, the article is not clear upon this point, and therefore a comparison is hard to make. It appears from the system block diagram that they have not realized that a polyphase implementation could be useful for their low-rate halfband filter as well, which could present a sizeable cost reduction since the first filter usually has the highest order in order to handle the transmission bandwidth.
6.5.2 Possible improvements

There are several ways in which the system could be improved. Since the goal is to let the system be a very flexible tool for the users of SDR14, I would say that the best way of improving it would be to make it more parameterizable. Some of the following points would make the code more complex, but would in return cause the system to be much more flexible:

- Make it possible to select different input sample rates. An example would be if the user already has a 50 MSps signal and does not wish to utilize the first interpolation in the system. This would cause problems for the current reconstruction filter implementation, however, since it has to be placed at 50 MSps and assumes a 20 MHz signal bandwidth.

- Allow the filter coefficients to be reprogrammable from the user interface.

- Allow for changing FIR filter lengths by editing a few variables, in case users require higher attenuation, wider transmission bandwidth or other similar things. This is already semi-implemented in the high-rate filters, but could use some fine tuning and simplification.

Compressing the reconstruction filter implementation down to its theoretical minimum cost of two multipliers instead of using four, would be a direct (but fairly small) improvement for the system resource cost.
Bibliography


