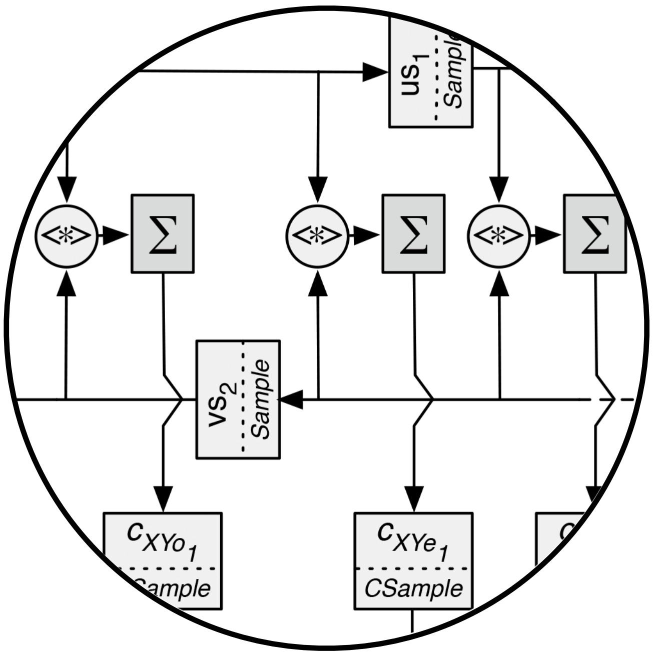
COGNITIVE RADIO: COMPARING CROSS-CORRELATORS

STEFAN ROLINK



Master’s thesis

August 2013

Stefan Rolink: *Cognitive Radio: Comparing Cross-Correlators,* Master's thesis © August 2013

SUPERVISORS:
dr. ir. André B.J. Kokkeler
dr. ir. Jan Kuper
ir. Bert Molenkamp
dr. ir. Mark S. Oude Alink

LOCATION:
Enschede

DATE:
August 2013

Dedicated to the loving memory of Cor van de Capelle.
1984 – 2002

ABSTRACT

Because the free bands in the electromagnetic spectrum are becoming scarce, numerous techniques have been developed throughout the years to increase capacity. But studies have shown that when the activity on the band is measured a lot of the segments are unused. The concept of Cognitive Radio is that it can intelligently detect and utilize the unused segments when available, without interfering primary users.

Sensing the spectrum for spectrum holes is one of the key function of a cognitive radio. The limiting element in this process is the Signal-to-Noise Ratio wall. Cross-correlation was proposed as a solution for noise reduction in spectrum sensing. If two receivers both receive the same wirelessly transmitted data, they each will also contain some random noise. Correlation emphasizes shared signal properties (the transmitted data) and filters the - for each signal unique - noise.

During the graduation period a cross-correlation architecture was created and simulated in CλaSH. This is an at the University of Twente developed Hardware Description language, that can create a formal description of digital logic and electronic circuits, which can be mapped on a FPGA. It was necessary that the resolution and the numbers of lags of the correlator architecture could be adjusted without much effort, so that the circuit could be easily analyzed. The goal of this research was to find a relation between the resolution and number of lags of the correlator and the necessary chip area, the maximum clock frequency the correlation operation could be executed and the sensitivity of the correlator.

ACKNOWLEDGEMENTS

This thesis you are reading now is the product of a year of work, but I couldn't have done it without the help of some people, which I would like to thank here. First off all I would like to thank my supervisors: André, Jan, Bert and Mark. For always being there to answer my questions and giving me a lot of new insights when I needed them, despite of their busy schedule. And of course for reading this thesis multiple times to correct mistakes.

Furthermore I would like to thank all the nice people at the CAES group, for making my stay a very pleasant one and off course for helping me out with problems when needed.

And finally my family and friends who have been awfully patient with my lack of time lately.

CONTENTS

ACRONYMS

ADC	Analog to Digital Converter
ASIC	Application-specific integrated circuit
CAES	Computer Architectures for Embedded Systems
CR	Cognitive Radio
dB	Decibels
DFT	Discrete Fourier Transform
DSAN	Dynamic Spectrum Access Networks
DSP	Digital Signal Processor
FFT	fast Fourier transform
FPGA	Field-programmable gate array
FXC	FX-Correlator
GHC	Glasgow Haskell Compiler
HDL	Hardware Description Language
IC	Integrated Circuit
ISM	Industrial, Scientific and Medical
LE	Logic Element
LUT	Look-up table
LVDS	Low-Voltage Differential Signaling
MAC	Multiply-accumulate
SNR	Signal-to-Noise Ratio
RKRL	Radio Knowledge Representation Language
RTL	Register Transfer Level
SDR	Software Defined Radio
SFDR	Spurious-Free Dynamic Range

SNR	Signal-to-Noise Ratio
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuits
XFC	XF-Correlator
$XXXX_{2's}$	A two's-complement binary number
$XXXX_2$	A unsigned binary number
$XXXX_{10}$	A decimal number

INTRODUCTION

The electromagnetic spectrum is a limited range of frequencies that can be used to exchange data. To avoid interference, this spectrum is segmented into smaller bands. Every band can be used for different services, for instance FM-radio, TV broadcasts, GSM, etc. Since the number of services is ever increasing and every service needs spectrum, the number of free spectrum bands is becoming scarce. A consequence is that spectrum efficiency has become the main parameter for communication systems design. Therefore, throughout the years, several (higher order) modulation schemes have been used. Although these modulation schemes increase the communication capacity, they also increase the power consumption of the device [1]. For devices with limited available power like mobile applications, this can become problematic. Besides, high powered transmitters are also unwanted with respect to global warming. A possible solution for the limits of segmentation of the spectrum is Cognitive Radio (CR) (or Dynamic Spectrum Access Networks (DSAN)).

1.1 COGNITIVE RADIO

As stated above, free spectrum bands are scarce. Access to them is either regulated by means of licenses or free (for instance, the Industrial, Scientific and Medical (ISM) bands). Studies have shown that when the activity on each band is measured, most of the time many spectral segments are unused [2][3], as presented in Figure 1.

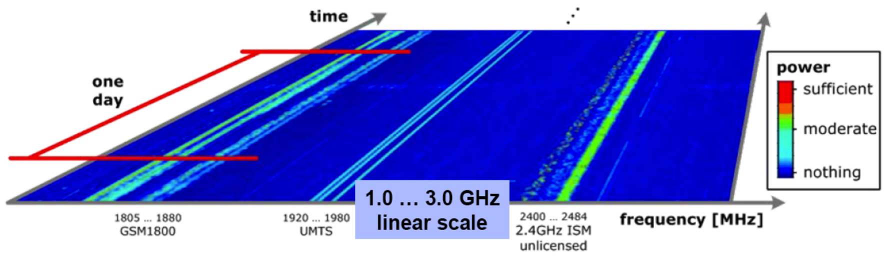


Figure 1: Spectrum measurement performed by Swisscom in Bern [4]

It can be seen that only some frequencies within the GSM1800, UMTS and ISM bands are partly used throughout a whole day, whereas most of the frequency band is unused (denoted by the blue color). A CR is a form of wireless communication which exploits this by using the unused spectrum band (regardless if it is a free band or a licensed one). The CR transceiver can intelligently detect which spectral bands (channels) are in use and which are not. Because of this, it will only use vacant bands and move between these to avoid interfering with occupied ones.

The concept of Cognitive Radio, was coined by Joseph Mitola III in [5], where he describes a self-aware extension to a Software Defined Radio (SDR) [6]. A SDR is a multi-band radio which is capable of supporting multiple wireless interfaces and protocols. Ideally, all aspects of a software radio are defined in software. Figure 2 shows the CR framework proposed in [7]. The radio hardware consists of a set of modules, namely: an antenna, a RF section, a modem, an Information Security (INFOSEC) module, a baseband processor and a user interface (UI). The CR contains an internal model of its own hardware and software structure. The Radio Knowledge Representation Language (RKRL) frames is a language proposed in [8], which defines a framework that offers the CR knowledge of context, radio protocols, air interfaces, networks and user communications states.

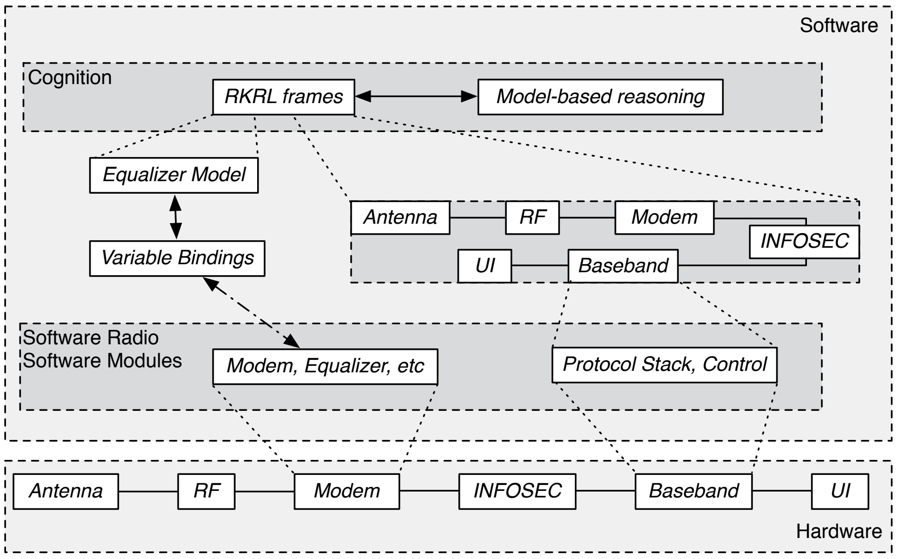


Figure 2: Cognitive radio framework [8]

.

A CR is a SDR that employs model-based reasoning about users, multimedia content and communications context. It senses the outside world to recognize the context of its communications tasks. To use the example presented in [9], the radio may conclude that it is probably going for a taxi ride to a certain location, based upon the knowledge the user just ordered a taxi by voice. It can then tell the network its plan to move from its present location to the new one. The network in turn will know that this user will probably cross certain cells in the network in the next thirty minutes or so.

This example shows that both the radio and the network need a deep understanding of the context of the communication and the resources available. This kind of CR, in which every possible parameter observable by the radio or network is considered, is called a Full Cognitive Radio (or Mitola Radio) [5]. Such a radio is still not realizable due its high technical complexity and is thus considered as goal towards which a SDR should evolve: a fully reconfigurable wireless transceiver which adapts its communication parameters to network and user demands automatically.

A basic realizable form of a CR is a SDR as applied to spread spectrum communication techniques. Its main functions are [10]:

- *Spectrum sensing*, Detect unused spectrum and share the spectrum without harmful interference to other users.

- *Spectrum management*, Capture the best available spectrum to meet the user communication requirements and manage the spectrum allocation among the users.

- *Spectrum mobility*, Maintain seamless communication requirements during the transition to better spectrum

- *Spectrum sharing*, Provide a fair spectrum scheduling method among coexisting CR users.

1.2 SPECTRUM HOLES

Spectrum Sensing is a key physical layer technique for a CR. It is defined as the task to find underutilized subbands of the radio spectrum at a particular instant of time and geographic location (i. e. spectrum holes)[11].

Spectrum holes can be distinguished in three categories, based on their domain, viz.: Spatial, Time and Frequency.

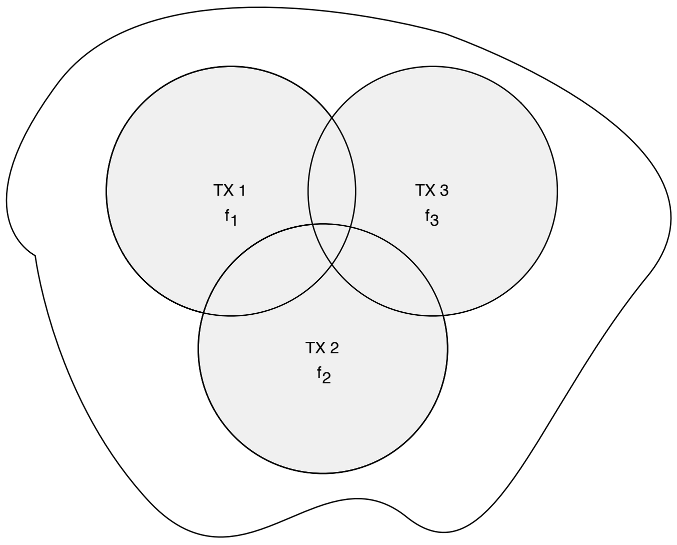


Figure 3: Spectrum holes in space: Three television towers which all transmit signals at their own frequency.

Within its service area, a television tower will always communicate to its users at a certain frequency. Some of the area around this tower will be fully utilized (as can be depicted by the grey circles around the towers *TX 1*, *TX 2* and *TX 3* in Figure 3). Each tower will have its own frequency (f_1, f_2 and f_3). This frequency can therefore not be used by secondary users within transmitting range of the corresponding tower. However, when the secondary user moves outside the transmitting range (in the white area around the towers), it will encounter no problems and can fully reuse the frequency. When the secondary user moves, for example, from the area of *TX 1* to *TX 2*, it becomes possible to reuse the frequency f_1 in most part of *TX 2*, since the frequency f_1 is only used by *TX 1* (As long as the transmitting range of the secondary user doesn't overlap with the transmitting range of the tower). Within such system, recovering spectrum holes in the spatial domain will be the major concern.

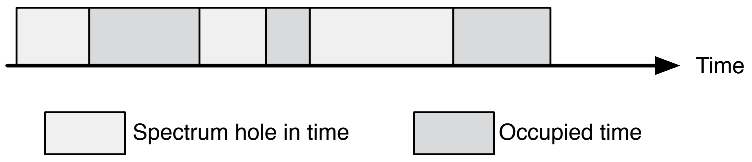


Figure 4: Spectrum holes in time [11].

In contrast to spatial distribution of frequency use, temporal distribution is used in systems that communicate discontinuous, but serve an entire area. Recovering spectrum holes within the time domain will be the major concern for such a system. This can be viewed in Figure 4, where the dark grey color shows that only for certain amounts of time the subband of the spectrum is being used.

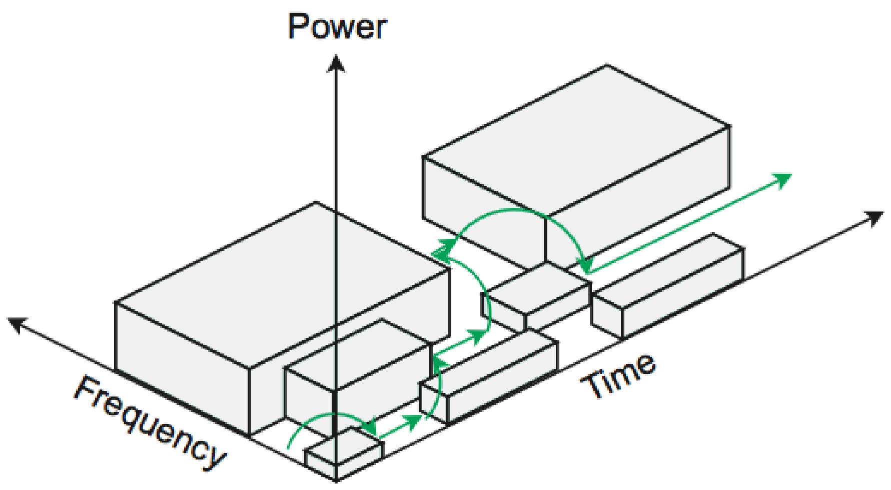


Figure 5: Spectrum holes in time and frequency. The green arrow shows the dynamic spectrum access [11].

A third category of spectrum holes are the ones in the frequency domain. A spectrum hole in frequency is defined as a frequency band in which a transceiver can communicate without interfering with any primary receivers. These kind of spectrum holes can be seen in Figure 1. The dynamic spectrum access exploiting time and frequency holes, is presented as a green arrow In Figure 5. This indicates a path the CR can follow during communication.

In terms of occupancy, spectrum holes for each domain can be categorized as follows:

- *White spaces*, these are free of interferers, with the exception of some natural or artificial noise sources (e.g. thermal noises).
- *Grey spaces*, these are partially occupied by interferers as well as noise.
- *Black spaces*, which are fully occupied by a primary user.

As stated before, the task of spectrum sensing is to find spectrum holes in the radio spectrum. The key limiting element of spectrum sensing is that it suffers from a Signal-to-Noise Ratio (SNR) wall. This is a minimum SNR (the level of a signal to be measured compared to the level of background noise), below which a signal cannot be reliably detected. The SNR wall is caused by the uncertainty in the noise level for energy detection. The decision whether a signal is present or not is based on the difference between the measured power level and the estimated noise power level. The noise level will be composed by the noise from the physical channel and noise from the receiving device. Because the antenna noise varies, for example, due to varying weather conditions, the noise level can only be estimated with limited accuracy. Cross-correlation was proposed as a solution for noise reduction in spectrum sensing [12].

1.3 ORGANIZATION THESIS

This thesis is organized as follows. First, the basic idea behind correlation will be presented in Chapter 2. The mathematical definitions, the usage of and the difference between autocorrelation and cross-correlation, hardware structures for correlation and some optimization techniques are presented here. In Chapter 3 the focus lies on different hardware description languages and will give an introduction to the functional language CλaSH. Chapter 4 will describe how CλaSH was used to create a parametrizable complex multiplier, adder and cross-correlation architecture. In Chapter 5 the created correlator architecture will be analyzed for area utilization and timing constrains. Chapter 6 will contain the conclusions for this research and the subjects that can be further explored by future research is presented in Chapter 7

2 CROSS-CORRELATION

By using cross-correlation, the similarity between two input signals can be measured, while applying a time delay (or time-lag in the digital domain) function to one of them. It can be considered as a kind of template matching. Cross-correlation is used to increase the SNR in received signals, which can be used to detect spectrum holes [13].

2.1 MATHEMATICAL DEFINITIONS

Discrete cross-correlation is quite similar to the dot product, defined by Equation 1. It takes two equal-length sequences of numbers and returns a single value by multiplying corresponding entries and then summing those products.

$$\vec{x} \bullet \vec{y} = \sum_i x_i \cdot y_i \,, \qquad \vec{x} \stackrel{\text{def}}{=} [x_1, x_2, ..., x_n]^\mathsf{T} \tag{1}$$

The cross-correlation is often referred to as the *sliding dot-product* of two inputs signals. Instead of returning a single number, one of the inputs slides over time, which produces a new function. (Eq. 2). The cross-correlation function is represented by the '$\star$' operator,

$$\gamma_{fg} \stackrel{\text{def}}{=} (f \star g)_d \stackrel{\text{def}}{=} \sum_{i=-\infty}^{\infty} f_i^* \cdot g_{d+i} \tag{2}$$

where f^* denotes the complex conjugate of f (a negation of the imaginary part of the complex number). The cross-correlation operation is similar in nature to the convolution (indicated with the '$\ast$' operator in Equation 3) of two functions, but with a time reversal applied to one of the input. Their relation can be seen in Equation 4.

$$(f \ast g)_n \stackrel{\text{def}}{=} \sum_{i=-\infty}^{\infty} f_i^* \cdot g_{d-i} \tag{3}$$

$$(f \star g)_n = f^*(-t) \ast g \tag{4}$$

2.2 AUTOCORRELATION

Autocorrelation is the cross-correlation of a signal with itself. It can be used to find repeating patterns within the signal, for instance the presence of a periodic signal which is buried in noise. The estimated autocorrelation function is defined by Equation 5.

$$\gamma_{xx}[j] = \begin{cases} \dfrac{\sum\limits_{n=1}^{N} x_n^* \cdot x_{n-j}}{\sigma_x^2}, & \text{if } j = 1..M. \\ 0, & \text{otherwise.} \end{cases} \tag{5}$$

$$\sigma_x = \frac{1}{N} \sum_{n=1}^{N} |x_n|^2 \tag{6}$$

By dividing by the variance σ_x^2, which denotes how far a set of numbers is spread out (Eq 6), the resulting γ_{xx} will be in the range of $[-1, 1]$. Where 1 indicates perfect correlation (the signals exactly overlap when time shifted by k) and -1 indicates perfect anti-correlation.

Next, with the help of MATLAB an example of autocorrelation will be presented to discover hidden periodic signal in a noisy signal.

Listing 1: Autocorrelation example in Matlab

```
 1  rng('default');                     % Make the results
        reproducable
 2
 3  N  = 1000;                          % Number of samples to
        generate
 4  f1 = 1;                             % Frequency of the
        sinewave
 5  FS = 200;                           % Sampling frequency
 6  n  = 0:N-1;                         % Sampling index
 7
 8  x = sin(2*pi*f1*n/FS);              % Generate sine
 9  x = x + randn(1,N);                 % Add random noise to sine
10
11  [Yxx, lags] = xcorr(x, 'coeff');    % Calculate
        autocorrelation Yxx
```

In Listing 1 (at line 8 and 9) random noise is added to a sine function (with a frequency of $1Hz$, sampled at $200Hz$). This noisy sine function $x(n)$, can be viewed below in Figure 6.

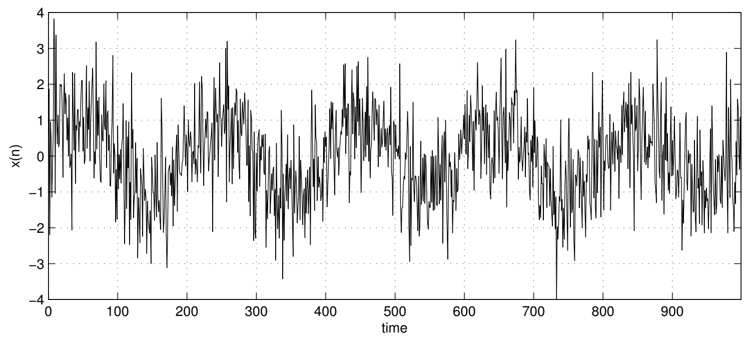


Figure 6: Pure sinewave with added normal distributed noise.

Looking at the autocorrelated signal γ_{xx} in Figure 7, a peak value of 1 at zero lag can been seen, which means that the signals are perfectly correlated. This is always the case with autocorrelation, since the correlation is between two verbatim copies. Furthermore, the periodic signal which was obscured in the noise becomes visible again. At its edges, γ_{xx} is slowly being attenuated by the fact that the number of samples is finite. When shifted in time (cut off at one edge and padded with zero's on the other side) the similarities will thus decrease in time.

Note that this is due the implementation of the *xcorr* function in Matlab, where two N-size array are correlated. An alternative could be to correlate a $2N$-size array with a N-size copy, this results in an N-size γ_{xx} without the attenuated edges.

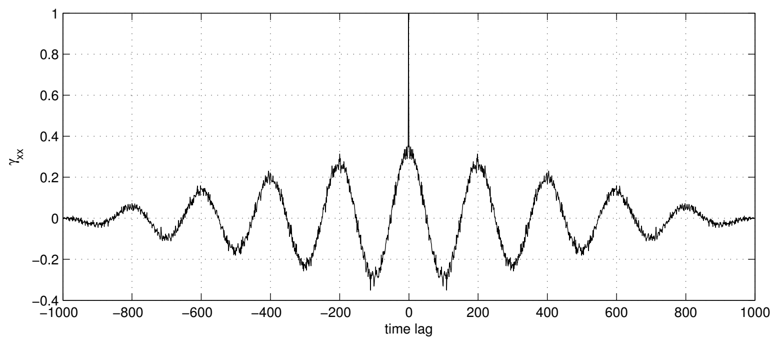


Figure 7: The autocorrelated signal with a 'hidden' sine function visible.

2.3 cross-correlation

As stated before, cross-correlation makes it possible to measure the similarity between two signals. By utilizing this property, it becomes possible to increase the SNR from received (analog or digital) data. The definition of the digital cross-correlation can been seen in Equation 7.

$$\gamma_{xy}[j] = \begin{cases} \frac{\sum\limits_{n=1}^{N} x_n^* \; y_{n-j}}{\sigma_x \sigma_y}, & \text{if } j = 1..M. \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

Imagine two receivers, both receiving the same wirelessly transmitted data. Although they are both physically the same, they will receive different raw data. This is because every antenna will be interfered by some random environmental and thermal noise. This noise can even be several times stronger than the expected data (take for example the very low power signals from outer space that radio telescopes detect). Since receivers don't receive their data at exactly the same time, the peak value of the correlation will be shifted.

Listing 2 shows an example of cross-correlation, in which the same values for x are used as in Figure 6. The $y(n)$ values are generated using the same sine function as the $x(n)$ values, but with different noise added, simulating two receivers that receive the same signal, but added with some unique random noise. Since separate antennas don't receive the same signal at exactly the same time, the $y(n)$ values are shifted 30 units in time with the *lagmatrix* function. When $y(n)$ is shifted, it is cut off at one edge and padded with NaN's at the other side, these are converted to 0's in line 11.

Listing 2: Cross-correlation example in Matlab

```
rng('default');                    % Make the results
    reproducable

N  = 1000;                         % Number of samples to
    generate
f1 = 1;                            % Frequency of the sinewave
FS = 200;                          % Sampling frequency
n  = 0:N-1;                        % Sampling index

x = sin(2*pi*f1*n/FS);             % Generate sine
y = x + randn(1,N);                % Add random noise to sine
y = lagmatrix(y , 30);             % Shift y 30 units in time
y(isnan(y)) = 0;                   % Filter out 'NaN'
x = x + randn(1,N);                % Add random noise to sine

[Yxy, lags] = xcorr(x, y);         % Calculate cross-
    correlation Yxy
```

Looking at the results γ_{xy} in Figure 8, we can see a periodic signal appearing which was first obscured by the random noise. Additionally, note that the highest value of γ_{xy} isn't at 0-lag, but lies at -30, which is the lag that was defined in Listing 2. Furthermore, the peak that became visible with autocorrelation is gone, since the correlation is between 2 signals which aren't exact copies (due to the fact that the added noise signals to two receivers are uncorrelated with each other).

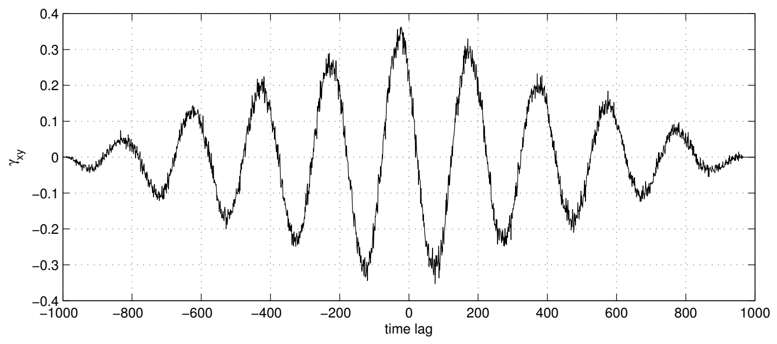


Figure 8: The cross-correlated signal of x and y.

2.4 CROSS-SPECTRUM ANALYSIS

Cross-spectrum analysis provides a statement on how common activity between two processes is distributed across frequency.

The Wiener-Khinchin theorem states that the power spectral density $\Gamma_{xx}[f]$ is the Fourier transform of the corresponding autocorrelation function $\gamma_{xx}[\tau]$ (Eq. 8 and Eq. 9)[14].

$$\Gamma_{xx}[f] = \mathcal{F}\{\gamma_{xx}[\tau]\} = \sum_{\tau=-\infty}^{\infty} \gamma_{xx}[\tau]\, e^{-2\pi i \tau f} \tag{8}$$

$$\gamma_{xx}[\tau] = \mathcal{F}^{-1}\{\Gamma_{xx}[f]\} = \sum_{\tau=-\infty}^{\infty} \Gamma_{xx}[f]\, e^{2\pi i \tau f} \tag{9}$$

$\Gamma_{xx}[f]$ is a complex function of frequency. The same derivations hold for the cross-correlation function $\gamma_{xy}[\tau]$, which forms a Fourier transform pair with the cross power spectrum (or simply cross-spectrum) $\Gamma_{xy}[f]$.

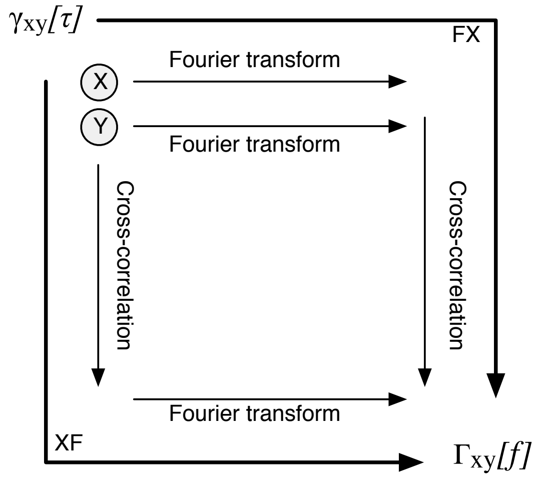


Figure 9: The cross-spectrum $\Gamma_{xy}[f]$ can be calculated, either by first calculating the Fourier transform and then applying cross-correlation, or vice versa.

To acquire the cross-spectrum $\Gamma_{xy}[f]$, a Fourier transform and a cross-correlation are calculated. The order of these operations is interchangeable, as can be seen in Figure 9. There are two main hardware structures to obtain the cross-spectrum by digital cross-correlation, the FX-Correlator (FXC) and the XF-Correlator (XFC). These two differ in the order in which the cross-correlation (X) and the Fourier transform (F) (using the fast Fourier transform (FFT)) are calculated [15].

2.5 FX-CORRELATOR

The principle of an FXC can be viewed in Figure 10. Both receivers (the inputs of the ADCs) receive their data from the same source $S(t)$, but have their own unique noise ($n_1(t)$ and $n_2(t)$) added.

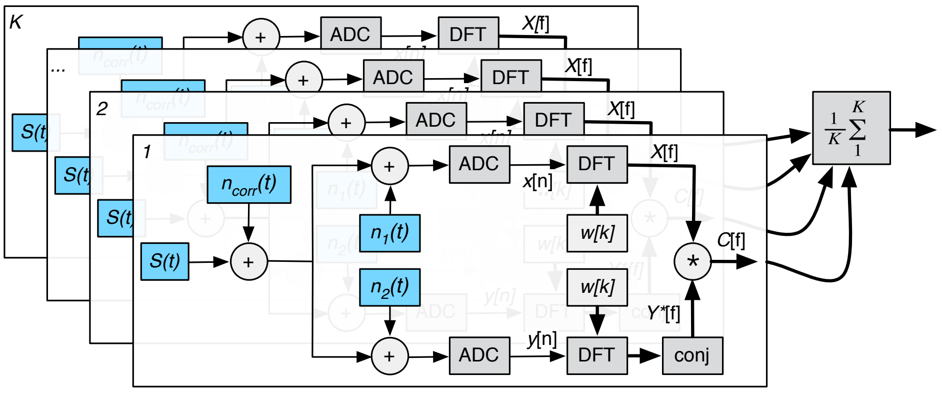


Figure 10: Schematic view of a FXC [12].

The receiver outputs $x[n]$ and $y[n]$ are transformed by a Discrete Fourier Transform (DFT) to obtain $X[f]$ and $Y[f]$. Since this DFT is calculated over N samples, the output of the DFT also contains N samples (indicated by the bold lines). This is in contrast with its input, which receives each sample individually and therefore it is necessary for the DFT to have some internal memory. A window $w[k]$ can be used to reduce the spectral leakage that is caused by the finite time window in the DFT [16]. The vectors $X[f]$ and $Y^*[f]$ are element-wise multiplied to produce the cross-spectrum estimate $C[f]$, also note here that this operation needs N multiplications. To reduce the variance, the calculations are repeated K times with new samples and the results are averaged.

2.6 XF-CORRELATOR

The XF-correlator first estimates the cross-correlation function $\gamma_{xy}[\tau]$, denoted by $c_{xy}[k]$ and estimating the cross-spectrum afterwards by taking the Fourier transform of $c_{xy}[k]$ to arrive at $C[f]$, as can be seen in Figure 11.

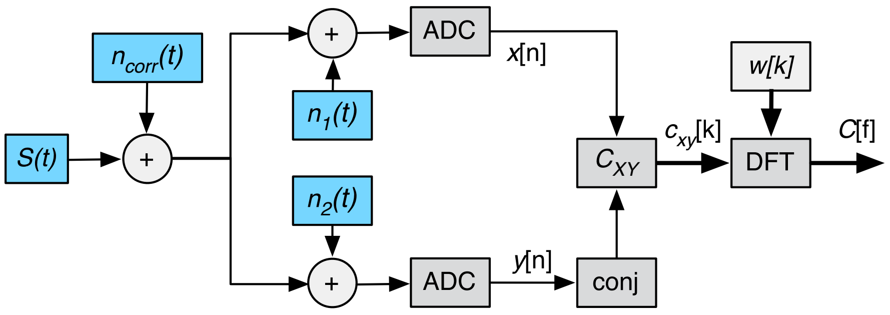


Figure 11: Schematic view of a XFC [12].

The schematic view of a XFC in Figure 11 can be a bit deceptive, since it looks less complex than the FXC in Figure 10. This is because the correlator C_{XY} presented here is only an abstracted view. An implementation of C_{XY} is presented in Figure 12 [12].

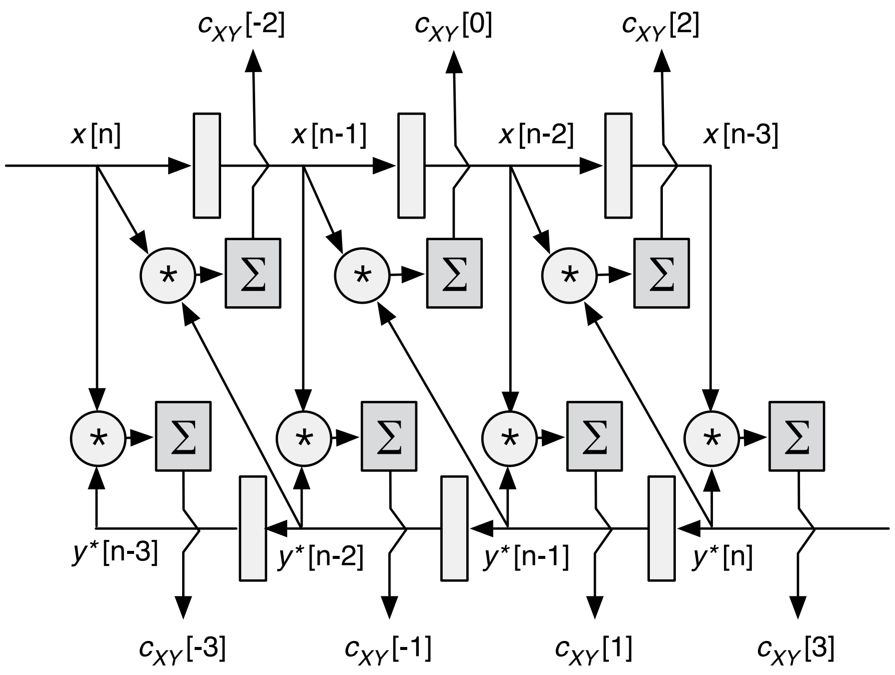


Figure 12: The implementation of C_{XY} [12].

This view might be rather intimidating, but when compared to the definition (Eq. 2), it becomes quite straightforward. The $x[n]$ values at the top can be seen shifted through delays from left to right, whilst the $y[n]$ values are shifted from right to left at the bottom. The values of $x[n]$ and $y^*[n]$ are individually multiplied. The result of this multiplication is then accumulated to form the output $c_{XY}[k]$ (where $c_{XY}[k]$ for odd k's are calculated by the vertical connections and for even k's by the diagonal ones. Both parts need to be merged later on). Note that $c_{XY}[k]$ is a vector, which means the DFT doesn't need memory elements like the FXC.

The difference between the FXC and the XFC is that, within the FXC the DFT is the most complex part and the correlator itself is relatively simple. With the XFC it's the other way around: the DFT is quite simple, while the correlator is more complex.

The FX-correlator has a complexity of $N \cdot log(N)$ in complex multiplications and summations (which is dominated by the DFT), while the XFC has a complexity of N^2 (which is dominated by the correlator). However, when the ADCs have a low resolution (number of bits), the XFC has to perform a lot of very simple Multiply-accumulate (MAC) operations, while in the FXC each stage in an FFT may require more bits. For example, when the resolution of the ADCs is just one bit, a multiplier reduces to a simple XOR-gate. For low power consumption, it may therefore be preferable to use an XFC.

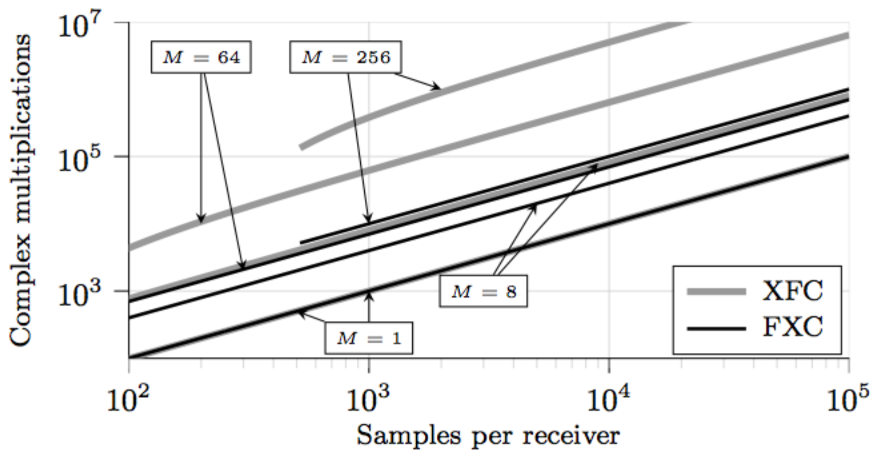


Figure 13: The number of complex multiplications for the FXC and the XFC for an M-point spectral estimate [12].

For an M-point spectral estimate, M points in the cross-correlator need to be calculated. The number of complex multiplications for the FXC and the XFC are presented for $M = 1, 8, 64$ & 256 in Figure 13. The FXC seems to have a large computational advantage with respect to the XFC (on a general purpose processor).

2.7 LOW RESOLUTION OPTIMIZATION

The basic operations that are performed by an XF correlator are multiplication, time delay and integration. In most digital-processing systems, a sufficiently large number of bits for quantization (rounding a signal, to represent it in digital form) and processing is required to allow the digitization effects to be ignored. However, a digital correlation spectrometer, in practical sense, easily becomes too complex, unless a small number of bits is used [17]. There are a couple of options within the quantization process and the correlator which will lead to a much less complex design, but with an acceptable loss in sensitivity.

The selection of a correlation scheme is a compromise between complexity and sensitivity. The sensitivity of a digital correlator is proportional to its degradation factor d [18]. This is a measure for the SNR decrease in case of Gaussian signals (Eq. 10), which is caused by the quantization of the input signals and depends on the correlation scheme used.

$$d = \frac{\text{Output signal/noise ratio of digital correlator}}{\text{Output signal/noise ratio of analog correlator}} \tag{10}$$

In [17], Bos presented the difference in degradation factor of 1, 2 and 3-bit correlators for different sample frequencies, with respect to an analog (ideal) correlator. These can be seen in Table 1. Note that the improvement from a 1 to a 2-bit correlator is (relatively spoken) the highest improvement.

N (bits)	f_s	d
1	2B	0.64
	4B	0.74
2	2B	0.88
	4B	0.94
3	2B	0.95
∞ (analog)	-	1

Table 1: The degradation factor d of the performance of a 1, 2 and 3-bit correlator with respect to an analog (ideal) correlator for different sampling frequencies f_s (two and four times the bandwidth B) [17].

In a two-bit quantization scheme, the first of the two bits will be used for sign representation and is assigned to 0 if the input voltage is positive, and 1 when the input is negative.

When the input v lies within two transition levels ($-V_0$ and $+V_0$), the level bit is assigned to 0. It will be assigned to 1, when the input lies outside this range. The four possible output states of the quantizer are assigned to the weighting factors $-n$, -1, $+1$ and $+n$ (Table 2).

Input V:	$V \leqslant -V_0$	$-V_0 < V \leqslant 0$	$0 < V \leqslant V_0$	$V_0 < V$
Sign bit	1	1	0	0
Level bit	1	0	0	1
ω	$-n$	-1	$+1$	$+n$

Table 2: Two-bit quantization scheme, with weighting factors ω [19].

Table 3 gives a view on all possible outcomes of the generated products of two input voltages (which are quantized by the above presented scheme) in the multiplier.

		V_1 State			
		$-n$	-1	1	n
V_2 State	n	$-n^2$	$-n$	n	n^2
	1	$-n$	-1	1	n
	-1	n	1	-1	$-n$
	$-n$	n^2	n	$-n$	$-n^2$

Table 3: Generalized view of a two-bit multiplication table

The value of n can be freely chosen. In Table 4 the multiplication table can be seen for $n = 3$, resulting in the following weighting factors: -3, -1, 1, 3. This will result in a linear multiplier, since the step sizes between the weighting factors are constant, hence 2. Choosing different values for n will not only have an effect on the degradation factor, but also has the side-effect that the multiplier will become non-linear.

The low level products don't have a significant effect on the degradation factor, deleting them (Table 5) will not substantially increase d, while the advantages in the hardware complexity are considerable [19]. Emphasizing the high level products ($\pm n^2$) by assigning higher values to them, doesn't have a significant advantage [20].

		V_1 State			
		−3	−1	1	3
V_2 State	3	−9	−3	3	9
	1	−3	−1	1	3
	−1	3	1	−1	−3
	−3	9	3	−3	−9

Table 4: Two-bit multiplication table, with $n = 3$.

		V_1 State			
		−3	−1	1	3
V_2 State	3	−9	−3	3	9
	1	−3	0	0	3
	−1	3	0	0	−3
	−3	9	3	−3	−9

Table 5: Two-bit multiplication table, with $n = 3$ and low level products deleted.

In Figure 14 the sensitivity of two-bit correlators (with different values for n) with respect to an analog correlator (d) can be seen. Note that the value of the transition voltage V_0 in the quantizator also is a sensitivity factor.

The difference in maximum relative sensitivity between $n = 3$ and $n = 4$ is negligibly small, but the hardware complexity does make a difference. Since multiplying two 2-bit ($n = 3$) numbers, will results in a 4-bit product and doing the same to two 3-bit numbers ($n = 4$), will result in a 6-bit product. As stated before, removing the low level products in the correlation scheme has little influence on the sensitivity.

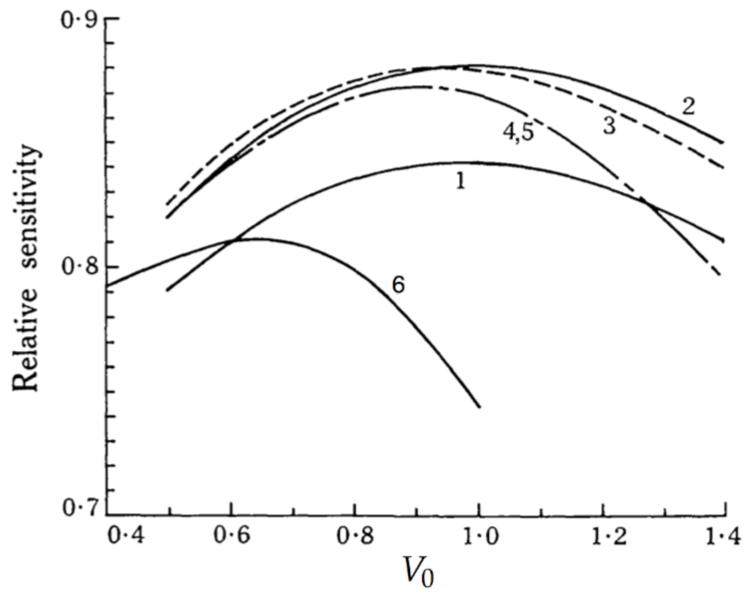


Figure 14: Sensitivities of two-bit correlator as a function of V_0 [19].
1: Full two-bit system, $n = 2$.
2: Full two-bit system, $n = 3$.
3: Full two-bit system, $n = 4$.
4: Low level products deleted, $n = 3$
5: Low level products deleted, $n = 4$
6: Only high level products retained.

These optimization methods presented here may seem dated, as they mostly rely on research done decades ago. Since the introduction of the Digital Signal Processor (DSP) (which are microprocessors specialized in digital signal processing) high resolution correlators became practically possible, leaving these optimization techniques obsolete for the purpose of radio astronomy where power is not the main concern. However, it's becoming interesting again to implement these methods on current technologies. For instance, an antenna which correlates beyond the 100 GHz band will be really small (since the antenna's size is inversely proportional to the frequency), making it possible to implement multiple correlators, complete with antennas on a single chip, provided that these optimization techniques are used.

3 HADWARE DESCRIPTION LANGUAGES

A Hardware Description Language (HDL) is a language for formal description of digital logic and electronic circuits. These languages differ from software programming languages, since they are used to describe the propagation of signals in time and their dependencies. While many software programming languages are procedural, HDLs have the ability to model multiple parallel processes that automatically execute independently of one another. The compiler of a HDL has as goal to transform the code listing into a physically realizable gate netlist, this process is referred to as synthesis. The most well known are VHSIC Hardware Description Language (VHDL) [21] and Verilog [22].

3.1 VHDL

VHDL arose in the early eighties out of the United States Governments's Very-High-Speed Integrated Circuits (VHSIC) program, which was initiated in 1980. The goal of the program was to develop a standard language for describing the structure and function of digital Integrated Circuit (IC)s. VHDL allows designs to be decomposed into sub-designs which can then be interconnected.

VHDL borrows a lot from the Ada programming language [23] in both concepts and syntax. This is due the fact that Ada was also designed within the Department of Defense. Ada had already been thoroughly tested during the design of VHDL. Thus, to avoid re-inventing concepts of Ada, VHDL was based as much as possible on Ada.

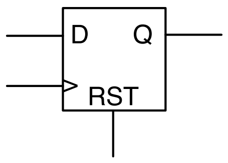


Figure 15: Symbolic view of a D flip-flop

As an introduction to the semantics and syntax of VHDL, a small example of a D flip-flop is presented in Listing 3. The schematics of this D flip-flop can be viewed in Figure 15. It captures the value of the D-input at the rising edge of the clock signal. That value becomes the new Q-output. At other times the output Q does not change unless it is reset by the RST signal.

Listing 3: A VHDL example: D flip-flop

```
 1  library IEEE;
 2  use IEEE.std_logic_1164.all;
 3
 4  -- Input / output definitions
 5  entity d_ff_en is
 6      port ( D, CLK, RST : in  STD_LOGIC;
 7             Q           : out STD_LOGIC );
 8  end d_ff_en;
 9
10  -- Internal behavoir
11  architecture d_ff_ar of d_ff_en is
12  begin
13      process(RST, CLK)
14      begin
15          if RST = '1' then
16              Q <= '0'
17          elsif rising_edge(CLK) then
18              Q <= D;
19          end if;
20      end process;
21  end d_ff_ar;
```

In Listing 3 the separation of the *entity* (line 5 - 8) and the *architecture* (line 11 - 21) can be clearly seen. An *entity* describes a hardware module and declares its input and output signals. The *architecture* describes the internal behavior of the module. An *entity* can have multiple and at least one *architecture*. *Architectures* can be described using a structural, data-flow, behavioral or a mixed style.

3.2 VERILOG

The Verilog HDL was created by Phil Moorby at Gateway Design Automation in 1983 and was initially intended as a simulation language, but was used for synthesis later as well. Verilog became an IEEE standard in 1995. The main difference with VHDL is that Verilog has a syntax which is more similar to C [24]. In contrast to VHDL it features loose typing, and is case sensitive.

Verilog has very simple data types, while VHDL allows user-defined types, which makes it possible for users to create more complex data types.

To give an introduction on the syntax and semantics of Verilog, the D flip-flop example in Listing 3 has been translated to Verilog as can been seen in Listing 4

Listing 4: A Verilog example: D flip-flop

```
module dff_sync_reset (
D      , // Data Input
CLK    , // Clock Input
RST    , // Reset input
Q        // Q output
);

// Input / output definitions
input D, CLK, RST ;
output Q;

reg Q; // Internal Variable

// Internal behavior
always @ ( posedge CLK)
    if (~RST) begin
        Q <= 1'b0; // Assign the 1-bit binary number 0 to Q
    end else begin
        Q <= D;
    end
end

endmodule //End Of Module dff_sync_reset
```

Verilog has built-in predefined net types (like *wire*, *wor*, *wand*, *tri*). Modeling a test bed in Verilog HDL takes relatively little effort, since it was designed with features which are required to model the system's environment (like global variables) [25]. Verilog HDL also has gate and switch level modeling, enabling Application-specific integrated circuit (ASIC) foundries to accurately represent their cell libraries. A result of this is that Verilog HDL is more used for ASIC design, whereas VHDL is more used in Field-programmable gate array (FPGA)s.

3.3 FUNCTIONAL HDL'S

Both VHDL and Verilog have their valuable properties, they do a good job describing hardware properties like timing behavior. However, they lack expressing higher level properties such as parameterization and abstraction. Though concepts like polymorphism were introduced to the 2008 standard of VHDL [26], still no available VHDL synthesis tool can support this. A "functional HDL" does focus on parameterization and abstraction. This leads to a more natural developing experience for a programmer, since most of them are used to higher-order functions, polymorphism, partial application, etc.

Most functional HDLs have the advantage of proving the equivalence of two designs. This makes it possible to prove that a highly optimized design has the same external behavior as the simple design, from which the optimized design was derived. Even though the first functional HDLs are even older than the now most well known HDLs such as VHDL and Verilog, they never achieved the same number of users in the end. The benefits of a functional HDL might soon be widely acknowledged, since today's hardware has an ever increasing design and test complexity and would result in exhaustive effort during test phases, when using imperative HDLs.

3.4 CλASH

The Computer Architectures for Embedded Systems (CAES) research chair at the University of Twente has developed a functional HDL called CλaSH [27], [28] & [29], [30] which is an acronym for 'CAES Language for Synchronous Hardware'. It is a functional HDL that borrows heavily from the functional programming language Haskell [31] in terms of syntax and semantics. Since CλaSH is a functional language, a program will consist of functions and function application. Each function becomes a hardware component. Every argument will be an input port and the computed result is the output port (which can consist of several values). Each function application will become a component instantiation and every result of each argument expression is assigned to a signal. As such, the output port of the function will also be mapped to a signal, which is used as the result of the application. Hardware descriptions written in CλaSH can be translated to a functional net-list (where every component and wire obtains a unique name) in Haskell.

Since there are virtually no tools to process and analyze these functional net-lists (and because it's a huge amount of work to create your own tooling), the net-lists are converted to VHDL code by the CλaSH compiler. These can than be processed by software tools like for instance *Quartus II* by *Altera*.

The basic premise of CλaSH is that all hardware designs can be described as the combinatorial logic and memory of a Mealy machine [32]

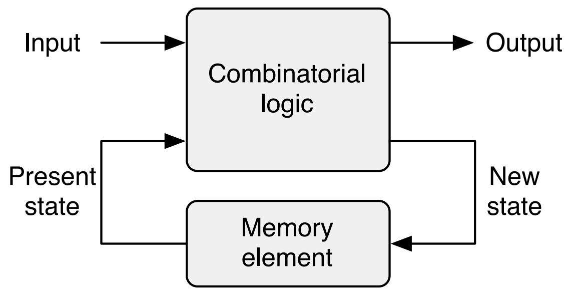


Figure 16: Schematic of a basic Mealy machine

In Figure 16 a graphical representation of a Mealy machine is presented. The Mealy machine is a finite state machine which generates an output and updates its state. The output value is based on the current state and the current input. The basic form of a Mealy machine in CλaSH can be seen in Listing 5. The first line starts with the name of the function *func* followed by the type specification operator *(::)*. After this operator the type of the function is declared. Every input type of the function is separated by a arrow (→), where the last type is always the output type of the function. Thus, in this case the function *func* requires two arguments as input, a State of a arbitrary type *a* (where the keyword *State* indicates that *a* is stored in an register) and another input of type *Input*. As output it generates a tuple, which consist of a new State of the same arbitrary type as its input State *a* and a certain *Output* type.

Listing 5: Logic of Mealy machine in CλaSH.

```
1 func :: State a → Input → (State a, Output)
2
3 func State state input = (State state′, output)
4                            where
5                               state′  = ...
6                               output = ...
```

In Line 3 the types are linked to variable names, a State *state* (which represents the memory element of the Mealy machine) and a certain list of input signals *input* are used as input. The output of *func* will result in a tuple of a new state *state′* and a list of output signals *output*. The function *func* represents the combinatorial logic and is the actual hardware that will be described. At every clock cycle, the input signals and the values from the memory element are fed into the combinatorial circuit, which will result in an output signal and a new value *state′* in the memory element. The state of the design is modeled just as a regular argument of the function.

To simulate such a Mealy machine using CλaSH, a function *simulate* needs to be defined, which is shown in Listing 6. This function maps the input over the combinatorial logic using the state *state*. It simulates the clock, which makes sure that at every clock cycle the function *func* is executed.

Listing 6: Simulation of a Mealy machine in CλaSH.

```
simulate _    _     []     = []
simulate func state (x:xs) = out : simulate func state′ xs
                             where
                               (state′, out) = func state x
```

The *simulate* function has three arguments, which are:

- A function *func*, which determines the functionality of some hardware architecture. This is a formal parameter of the function *simulate*. At every call of *simulate* the parameter *func* will be instantiated to the functionality of a concrete hardware architecture. Since the function *simulate* has another function as a parameter, *simulate* will be referred to as a 'higher order function'.

- A state *state*, that contains all the values in all memory elements in the architecture. Please note that it is allowed to use the apostrophe in a variable name. Thus, *state′* is not the same variable as *state* nor an operation on *state*. Also, *state* may be a structured parameter, consisting of several parts of memory elements instead of just a simple Integer.

- A list of inputs. The function *simulate* consists of 2 clauses; the difference lies in this third argument. The *[]* denotes the empty list, which means that this clause of the function *simulate* will be chosen when it is fed an empty list. When a non-empty list is given it will match the pattern *(x:xs)* and the second clause will be selected. The colon ':'

breaks the list in the first elemens x and the rest of the list xs (which is pronounced as the plural form of x, namely x-es). The function uses x and gives the tail (xs) of the list recursively to the next iteration of itself.

3.5 HARDWARE DESIGN IN CλASH

To give an impression of the design process, a small example of a MAC will be presented. This calculates the dot product of two vectors $\vec{x}$ and $\vec{y}$, which is mathematically represented by Equation 11.

$$z = \sum_{i=0}^{n} x_i \cdot y_i \tag{11}$$

Let's assume that there is only one multiplier and one adder available in the hardware. This clearly means that a memory element is needed here, to store the intermediate results of the addition. This is called the accumulator *acc*, which is initially 0. This is presented schematically in Figure 17.

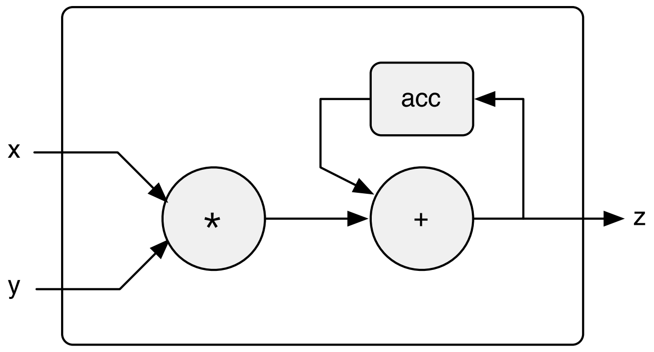


Figure 17: Schematic of MAC architecture.

Each clock cycle the inputs x_i and y_i are multiplied and added to the value of the accumulator. This new value of *acc* will be put back on the accumulator and the output. The function *macc*, which expresses this behavior is presented in Listing 7.

Listing 7: MAC in CλaSH.

```
1 macc :: State Int → (Int, Int) → (State Int, Int)
2
3 macc State acc (x, y) = (State z, z)
4                   where
5                      z = acc + x ∗ y
6
7 simulateMacc = simulate macc 0
```

Let's take a closer look at this function. On the first line the type of the function is defined. *macc* is a function which expects an Int (the previous state of *acc*) then a tuple (Int, Int) and returns a tuple (State Int, Int), in which the first element corresponds with the new state of *acc*. In line 2,3 and 4 the actual MAC is defined.

This function can be simulated using the *simulate* function in Listing 6. Since CλaSH is compatible with Haskell, the simulation can be run with the widespread used Haskell compiler: the Glasgow Haskell Compiler (GHC).

Listing 8: Test variable for Listing 7.

```
-- zip x and y to: [(2,1),(1,3),(4,-2),(-2,3)]
x = [2, 1, 4, -2]
y = [1, 3, -2, 3]
input = zip x y
```

Listing 8 shows the test variables used for simulation. The *zip* function makes a list of tuples, such that each tuple contains elements of both lists occurring at the same position. These can, one tuple at a time, be fed to the *macc* during simulation. Listing 9 shows the results of the simulation in GHC.

Listing 9: Simulation results of Listing 7.

```
> simulateMacc input
[2,5,-3,7]
```

3.6 HARDWARE SYNTHESIS

When the simulation of the algorithm is successful, the next step will be to synthesize the hardware to an FPGA. This is for instance realizable with the *Quartus II* sofware by *Altera*. In the examples presented above the *Int* type were used, but since the size of an *Int* is depending on the hardware it's used (for instance a 32-bit or 64-bit platform) and that it's required to specify the exact size when describing hardware, it will be necessary to change to a fixed size type. To give an example of the synthesis, the smaller 4-bit *D*4 type is more practical.

Listing 10: 4-bit MAC in CλaSH.

```
 1 {-# LANGUAGE Arrows #-}
 2 module MAC where
 3
 4 import CLasH.HardwareTypes
 5
 6 type Int4 = Signed D4
 7
 8
 9 macc :: State Int4 → (Int4, Int4) → (State Int4, Int4)
10
11 macc (State acc) (x, y) = (State z, z)
12                         where
13                            z = x ∗ y + acc
14
15 maccL = macc ^^^ 0
16
17 {-# ANN maccL TopEntity #-}
```

In Listing 10 the final version of the MAC can be viewed. The module *CLaSH.HardwareTypes* is loaded in line 4, which defines a number of operations that can be used to build circuits. Then the type *Int*4 is defined as a signed 4-bit value (In line 6. This type is then used in the type declaration of the *macc* function (Line 9), leaving the body of *macc* (since Listing 7) unchanged (Line 11 - 13). In line 15 the initial value of the accumulator is defined as 0 by the (^^^) operator (this operator is part of the *Arrows* abstraction, which loaded in line 1). Finally, the *TopEntity* annotation pragma is added to the description to indicate that the *maccL* (the 'lifted' version of *macc*) circuit is at the top of the hierarchy.

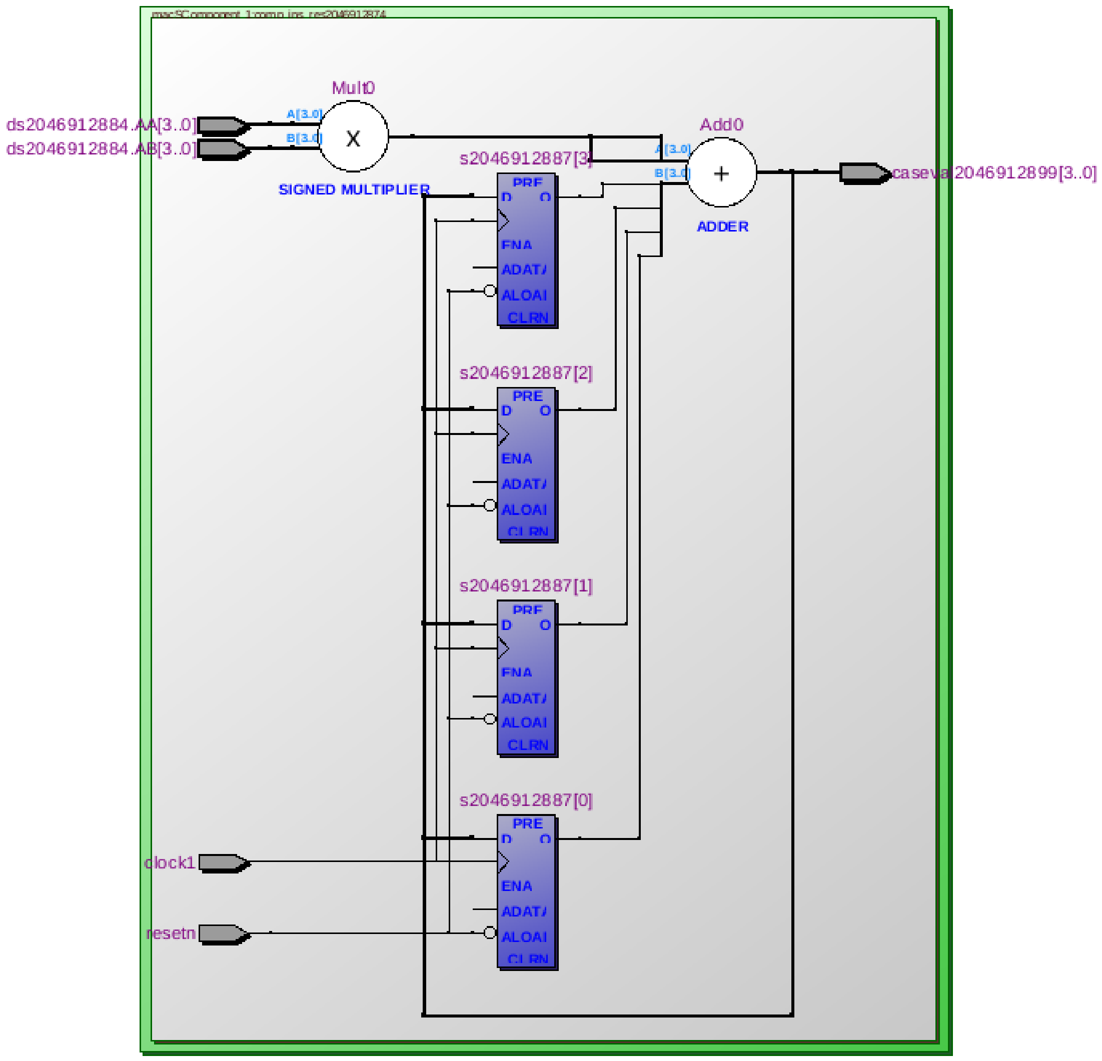


Figure 18: Register transfer level view in Quartus II of the MAC architecture.

The final step is now to generate the VHDL code from CλaSH and load it into Quartus II. Figure 18 shows the Register Transfer Level (RTL) view (a graphical representation of the synthesis results) of the design in Quartus II. The 4-bit signed multiplier and the 4-bit adder are clearly visible as well the 4 flip-flops to store the 4-bit accumulator. The output of the adder is connected to the inputs of the registers and vice versa. Note the similarities with Figure 17.

4 CROSS-CORRELATOR ARCHITECTURE

In this chapter is described how CλaSH was used to construct and simulate a complete correlator architecture. This correlator basically consist out of multipliers and adders (or accumulators), as can been seen from Equation 2. One of the goals of this research is to find a relationship between the necessary chip area and input size used in the correlator. It is therefor required to build the correlator and all its building-blocks in a flexible way, in which it is easy to change the resolution size.

4.1 SIGNED NUMBER REPRESENTATION

The input data of a correlator implemented in a CR comes from an antenna which generally uses an complex I/Q mixer to improve the received data by canceling the unwanted (or image) sideband. A schematic view of an I/Q mixer can be seen in Figure 19.

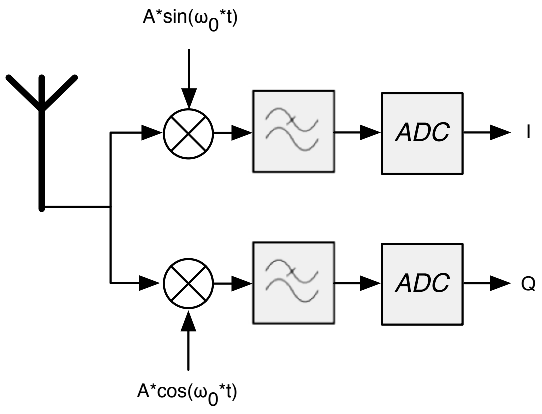


Figure 19: Schematic view of an I/Q mixer.

The in-phase I and quadrature Q output will be fed to the correlator and can be interpreted as complex data with a real and imaginary part. Since a cross-correlator uses multiplication and complex data samples, the output can result in negative products (e.g. $j^2 = -1$), it is therefore necessary to use a signed number representation. One of the most used signed number encodings is the two's-complement number system.

In a two's-complement number system, the value of each bit is a power of two, but unlike unsigned numbers, the most significant bit's value is the negative of the corresponding power of two. The definition of a two's-complement number system can be seen in Equation 12, where v is the value of an n-bit integer, consisting of
$a_{n-1}, a_{n-2}, ..., a_1, a_0$, with $a_i \in \{0, 1\}$.

$$v = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i \tag{12}$$

The most significant bit has a value of -2^{n-1} which also determines the sign of the number. Using n-bits, all integers from -2^{n-1} to $2^{n-1} - 1$ can be represented. Note that, to represent 2^{n-1}, a $n + 1$-bit number is needed. For instance, 8_{10} cannot be represented in a 4-bit two's-complement number (hence, $1000_{2's} = -8_{10}$). One extra bit is then required: $01000_{2's} = 8_{10}$.

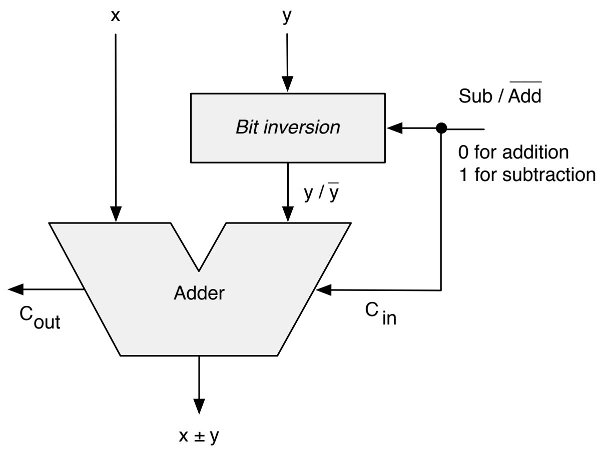


Figure 20: Adder/subtractor architecture for two's-complement numbers. [33]

The main advantage of a two's-complement number system is that basic arithmetic operations like addition and subtraction are identical to those for unsigned numbers. Signed multipliers needs some modification, but they will also be able to handle unsigned numbers. This will be explained later on.

Normally, complementing is performed for converting subtraction to addition. Complementing a two's-complement number consist of inverting each single bit, and incrementing the result with 1. Which means that every time a complementation is performed, an adder is needed. This can be quite some overhead, luckily, it is possible to combine an adder and subtractor in a single circuit, where the desired operation can be simply selected by an extra input line, mitigating the complementation disadvantage. Figure 20 shows the required elements for a two's-complement adder/subtracter.

4.2 IMPLEMENTING ADDITION / SUBTRACTION

In CλaSH, a set of n bits will be presented as a *Vector* of type *Bit* with size n, (denoted by $\langle Bit \rangle_n$[1]) where the least significant bit is the last bit in the vector.

To add two binary numbers, many different implementations are known. The most simple form is the ripple carry adder, which is shown in Figure 21.

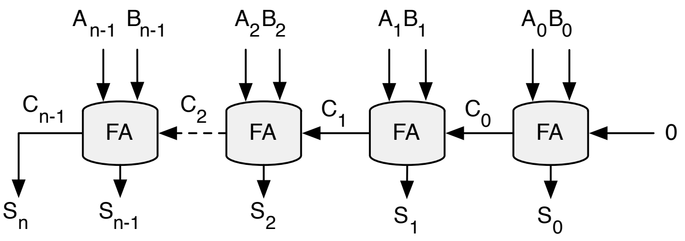


Figure 21: A ripple carry adder.

Note that the output vector S is one bit larger then the input vectors A and B. With a small adjustment, the standard ripple carry adder can be converted to an adder/subtraction circuit. Shown in Figure 22. When subtraction is selected, the XOR elements are used to complement all bits in vector B. Since complementing a two's-complement number also requires incrementation, the selection bit (which is *High* when selecting subtraction) is fed to the ($Carry_{\text{in}}$) input of the adder.

1 Which is actually notated in CλaSH as: Vector *Dn Bit*

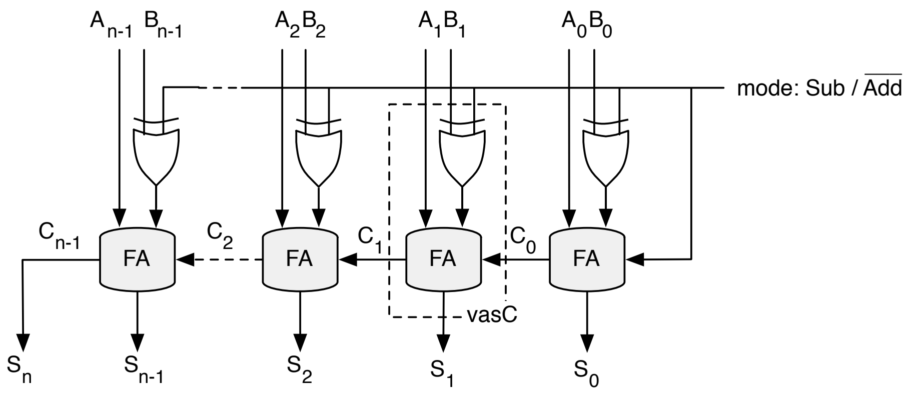


Figure 22: Ripple carry adder/subtractor.

Figures 21 and 22 clearly show a repeated pattern of components. This property can be utilized in CλaSH, where this component can be defined. Using higher order functions, these components can be chained together to form the full circuit. The declaration of this adder/subtracter-component (*vasC*) (which was represented by the components in the dotted box in Figure 22) can be viewed in Listing 11.

Listing 11: Implementation of *vaddsub_component* in CλaSH.

```
vasC :: Bit → (Bit, (Bit, Bit)) → (Bit, Bit)
vasC mode C (A, B) = (C′, S)
  where
    (C′, S) = fullAdd C (A, hwxor mode B)


fullAdd :: Bit → (Bit, Bit) → (Bit, Bit)
fullAdd C (A, B) = (C′, S)
  where
    C′       = hwxor C1 C2
    (C2, S)  = halfAdd S1 C
    (C1, S1) = halfAdd A B


halfAdd :: Bit → Bit → (Bit, Bit)
halfAdd A B = (C, S)
  where
    S = hwxor A B
    C = hwand A B
```

The *vasC* function expects a *mode* bit, which determines the addition or subtraction operation, a carry-in bit C and a tuple of two bits (A, B) as its input. The result will be a tuple of the carry-out bit with the calculated sum (C', S). The body of the function (Line 4) is pretty straightforward as its consist of a full-adder and an XOR port. For completeness, the declarations of the *fullAdd* and *halfAdd* are also given in Listing 11.

The adder/subtraction component can now be chained by the *vchain* function. The *vchain* function is more or less a combination of the standard *vzipWith* and *vfoldl/r* functions. It performs a function f on multiple input vectors and produces an output vector (like *vzipWith*), but the calculation of the next element of the output vector depends upon the result from the previous constituent part of the output vector (like *vfoldl/r*). Figure 23 shows a schematic representation of the *vchain* function.

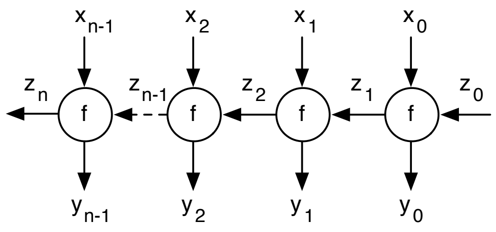


Figure 23: Schematic representation of the *vchain* function.

The implementation of the *vchain* function can be found in Listing 12. This looks a bit deceptive since a *vzipWith* with a certain function f is used on the vector xs and a seemingly empty vector zs. However, the zs vector is filled with the input argument z_0 on the first instantiation, with z_1 on next instantiation of the chain and so on. Remembering that Haskell uses lazy evaluation (which delays the evaluation of an expression until its value is needed), which makes this usage of *vzipWith* completely valid. The output vector ys gets filled with each instantiation of the chain. The *vreverse* function is used here to correct the order of the vector, since the least significant bit (the one on which function f first needs to be performed) is the last element of the vector.

Listing 12: Implementation of *vchain* function in CλaSH.

```
vchain f z xs = (z', (vreverse ys))
  where
    res      = vzipWith f (z +> zs)[2] $[3] vreverse[4] xs
    (zs, ys) = vunzip res
    z'       = vlast zs
```

Although the *vchain* function can be a bit tricky to understand, using it is quite straightforward. The function f in Figure 23 can simple be substituted by the vaddsub-component function (*vasC*). This way the input of the chain (the *xs* in Figure 23) corresponds to the inputs of the *vasC* function: A, B and *mode*. The *zs* of Figure 23 matches with the carry bit C of *vasC* and the output *ys* will corresponds to the sum S.

Listing 13 shows the implementation of the *vaddsub* function. The type of the function *vaddsub* is given in Line 1, the first argument is of the type *Bit*, the second and third argument are a *Vector* of type *Bit* with a length n. The output is a *Vector* of type *Bit* with length $n + 1$.

These three input arguments represent: a mode bit to select between addition or subtraction and two bitvectors (*as* and *bs*). The result is a vector *cs* (Line 3).

In Line 5 the essence of the function can be seen, where the function *vasC* together with the *mode*-bit are 'chained'. The *mode* is also used as initial value (the second argument of the *vchain* function), which would correspond to z_0 in Figure 23. The *input* of the chain consist of the *vzip* of the bitvectors *as* and *bs*, which is described in Line 6.

2 Add an element at the start of the vector

3 The $ operator is used to avoid parenthesis. Anything appearing after will be used as input to the function left.

4 Reverse the order of the Vector

Listing 13: Implementation of *vaddsub* function in CλaSH.

```
1  vaddsub :: Bit → ⟨Bit⟩_n[5] → ⟨Bit⟩_n → ⟨Bit⟩_n+1
2
3  vaddsub mode as bs = cs
4    where
5      (_, cs) = vchain (vasC mode) mode input
6      input   = vzip (sE as) (sE bs)
7
8  sE as = (vhead as) +> as
9
10 (<+>) = vaddsub Low
11 (<->) = vaddsub High
```

In Line 6 the two input vectors *as* and *bs* are sign-extended by the function *sE* (which puts the first element of *as* in front of it, as can been seen in Line 8), to fix overflow errors. These occur when the magnitude of the addition exceeds the representation limits. For example, when the sum of two positive numbers yields a negative result, or if the sum of two negative numbers results in a positive result. The overflow errors can be anticipated by a circuit, but this will be larger then just adding another full adder to the initial circuit.

In lines 10 and 11 two operators are defined, deducting the addition of two vectors from '*vaddsub Low as bs*' to: '*as <+> bs*', omitting the *mode* bit. This is possible in Haskell, because:

$$\text{func}_1 \;\; arg_1 \;\; arg_2 \;\; arg_3 = \text{func}_2 \;\; arg_1 \;\; arg_2 \;\; arg_3 \tag{13}$$

is equivalent to:

$$\text{func}_1 = \text{func}_2 \tag{14}$$

The created adder/subtracter architecture will be used as a component for a MAC, which will in turn be used as a building block for the eventual correlator design.

5 Which will be declared in CλaSH as: *Vector Dn Bit*.

4.3 BAUGH-WOOLEY TWO'S COMPLIMENT MULTIPLIER

This section will describe how a complex multiplier was created using CλaSH. During this thesis the following notation will be used:

A	Multiplicand	$a_{n-1}, a_{n-2}, ...a_1, a_0$
B	Multiplier	$b_{n-1}, b_{n-2}, ...b_1, b_0$
P	Product ($A \times B$)	$p_{2n-1}, p_{2n-2}, ...p_1, p_0$

The product of two numbers can be calculated by summing up their partial products. Every ith partial product bit is formed by the AND of each ith multiplier bit with each multiplicand bit ($b_i \cdot A \cdot 2^i$). A 4-bit example can been seen below:

$$
\begin{array}{ccccccccc}
 & & & & a_3 & a_2 & a_1 & a_0 & \\
 & & & & b_3 & b_2 & b_1 & b_0 & \times \\
\hline
 & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 & \\
 & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & 0 & \\
 & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & 0 & 0 & \\
 & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & 0 & 0 & 0 & + \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 &
\end{array}
$$

When multiplying two's-complement numbers, there lies a difficulty in the signs of the multiplicand and the multiplier. Consider two n-bit numbers A and B, both encoded as a two's-complement number (Equation 12). The product $P = A \times B$ can be given by Equation 15.

$$
\begin{aligned}
P = &- p_{m+n-1}2^{m+n-1} + \sum_{i=0}^{m+n-2} p_i 2^i = A \times B \\
= &\left(-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i\right) \times \left(-b_{n-1}2^{n-1} + \sum_{j=0}^{n-2} b_j 2^j\right) \\
= &a_{n-1}b_{n-1}2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j} - \\
&2^{n-1}\sum_{i=0}^{n-2} a_i b_{n-1} 2^i - 2^{n-1}\sum_{j=0}^{n-2} a_{n-1} b_j 2^j
\end{aligned}
\tag{15}
$$

This indicates that the final product is obtained by subtracting the last two positive terms from the first two terms. But, instead of using subtraction, addition can be used if the two's-complement of the two last terms is taken. This can be viewed in Equation 16.

$$P = \left(a_{n-1}b_{n-1}2^{2n-2}\right)_{\mathbf{I}} + \left(\sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j}\right)_{\mathbf{II}} + \left(2^{n-1}\sum_{i=0}^{n-2}\overline{a_i b_{n-1}}2^i\right)_{\mathbf{III}} + \left(2^{n-1}\sum_{j=0}^{n-2}\overline{a_{n-1}b_j}2^j\right)_{\mathbf{IV}} \tag{16}$$

The final product $P = A \times B$ can then be obtained by simply adding all the terms [34].

A schematic representation of this Baugh-Wooley multiplier can be seen in Figure 24. Where two n-bit two's-complement numbers A and B are multiplied to produce the $2n$-bit product P.

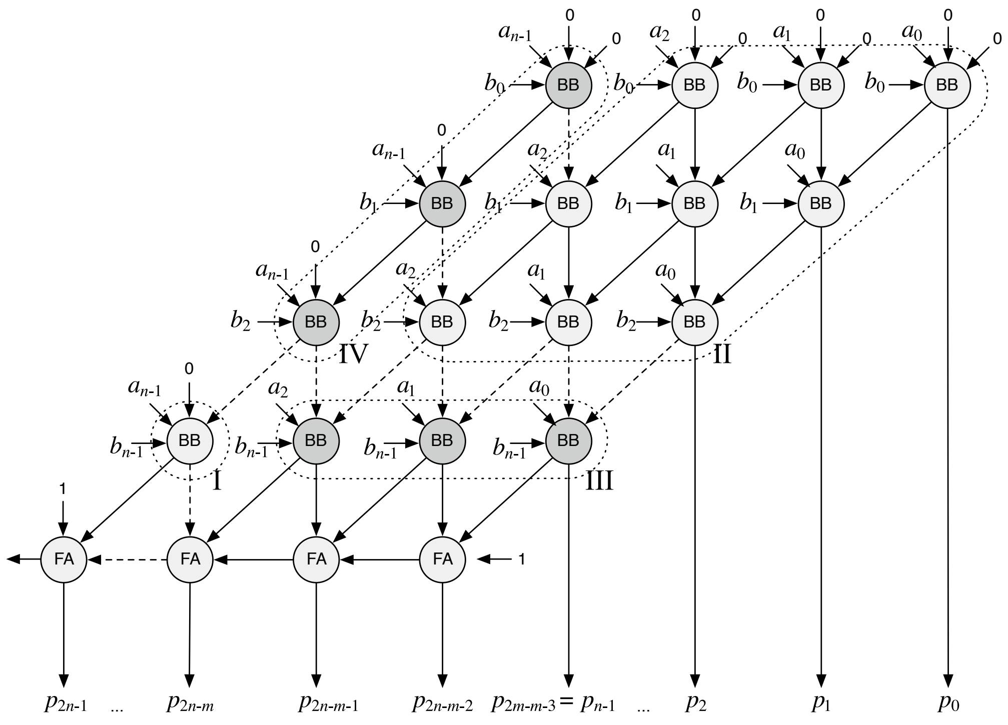

BB
FA
0
1
$a_{n\text{-}1}$
a_2
a_1
a_0
b_0
b_1
b_2
$b_{n\text{-}1}$
I
II
III
IV
$p_{2n\text{-}1}$ ... $p_{2n\text{-}m}$ $p_{2n\text{-}m\text{-}1}$ $p_{2n\text{-}m\text{-}2}$ $p_{2m\text{-}m\text{-}3} = p_{n\text{-}1}$ ... p_2 p_1 p_0

Figure 24: Schematic representation of a Baugh-Wooley multiplier.

Each cell of the Baugh-Wooley multiplier can be seen as a chainable 1-bit multiplier and will all have the same architecture, as can been seen in Figure 25. They basically consist out of a full adder, an XOR gate and an AND gate. The result of the AND gate can be inverted (to a NAND gate) with the XOR gate using the *Config* input to realize the complement operation in

Equation 16. The *Config* input is not shown in Figure 24, but the dark-grey cells depict a high *Config* input, whilst the light-grey cells mark a low *Config* input. This *Config* bit is fixed for every cell.

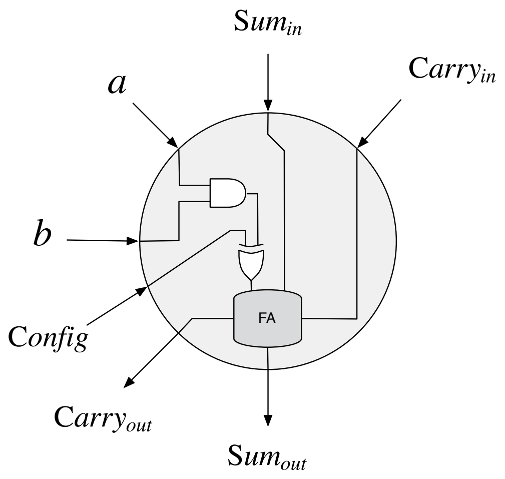


Figure 25: Schematic representation of a Baugh-Wooley cell.

The CλaSH implementation of a Baugh-Wooley cell (*bw_cell*) from Figure 25 is rather straightforward and can be viewed in Listing 14. In Line 1 can be seen that the function needs 5 inputs of type *Bit* (corresponding to *Config*, *a*, *b*, $Carry_{\text{in}}$ (c) and Sum_{in} (s) in Figure 25) and its result will be a 2-tuple of bits ($Carry_{\text{out}}$ (c') and Sum_{out} (s')).

Depending on the *Config* input (*cfg*), the output of the the AND gate (1-bit multiplication) will be complemented (Line 5).

Listing 14: Implementation of a Baugh-Wooley cell in CλaSH.

```
1 bw_cell :: Bit → Bit → Bit → Bit → Bit → (Bit, Bit)
2
3 bw_cell cfg a b c s = (c', s')
4   where
5     (c', s') = fullAdd c (s, hwxor cfg (hwand a b))
```

From a schematic point of view, the Baugh-Wooley multiplier is a two dimensional structure (note, Figure 24) consisting of identical cells. A row of these cells corresponds to a partial product, as can been seen in Figure 26.

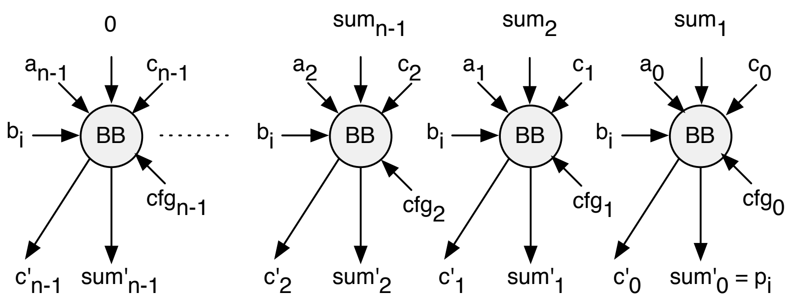


Figure 26: Schematic representation the partial product (a row of Baugh-Wooley cells).

By looking at the above schematic, the input types of the function *bw_row* can be defined. A row of *bw_cell*s needs 5 input vectors of length n:

- One for representing the multiplicand (as)
- One for representing every $Carry_{\text{in}}$ (cs)
- One for every Sum_{in} ($sums$) (notice in Figure 24 that the $sums$ bits are propagated vertically, which means that for every row, the most significant bit of the sums is discarded and a 0 is put in front of the vector), a vector of config bits, which tells each *bw_cell* if it needs to complement its output or not (with the exception of the last partial product, this vector will thus contain: $< 1, 0, ..., 0, 0 >$)
- And finally, a vector which contains n copies of b_i ($b_i s$), where i is the row number.

The output is a 2-tuple (since each cell in Figure 26 has 2 outputs) of bitvectors with length n, consisting out:

- The $carry_{\text{out}}$ vector (cs')
- The $sums_{\text{out}}$ vector ($sums'$).

The row of cells also output a single Bit, which is the partial product (p_i) of row i. These inputs and output corresponds to the type declaration of *bw_row* in Lines 1 to 2 in Listing 15.

Listing 15: Implementation of *bw_row* in CλaSH.

```
1 bw_row :: (⟨Bit⟩n, ⟨Bit⟩n, ⟨Bit⟩n) → (⟨Bit⟩n, ⟨Bit⟩n) →
2           ((⟨Bit⟩n, ⟨Bit⟩n, ⟨Bit⟩n), Bit)
3
4 bw_row (as, cs, sums) (cfg, bis) = ((as, cs', sums'), pi)
5  where
6    lsums        = Low +>>6 sums
7    res          = vzipWith57 bw_cell cfg as bis cs lsums
8    (cs', sums') = unzip res
9    pi           = vlast sums'
```

The body of the function *bw_row* is shown in Lines 6 to 9 in the above Listing. The 5 input vectors (*cfg*, *as*, $b_i s$, *cs* and *lsums*, with the last being *sums* with a *Low* in front of it (Line 6)) are zipped using the function *bw_cell* (Line 7). This results in the vector of 2-tuples *res*, which can be unzipped to the $carry_{\text{out}}$ vector (cs') and the sum_{out} vector ($sums'$). The partial product (p_i) is equivalent to the most significant bit of the sum output ($sums'$) (Line 9).

Since every partial product requires a 3-tuple of $(as, cs, sums)$ and also generates a 3-tuple $(as, cs', sums')$ (in addition to an element of the final product P), the partial product can be chained. This can be viewed in Figure 27, notice the similarities with Figure 24.

The chaining is done within the *vmult* function, which describes a complete Baugh-Wooley multiplier, this can be seen in Listing 16. As expected the function requires two bitvectors of length n (*as* and *bs*) and produces a $2n$-bit vector (*ps*) (Line 1).

First some preparations are executed, In Line 4 the initial value for the $Carry_{\text{in}}$ and Sum_{in} are generated, which is a vector with n elements of *Low* (named *ls*). This is done by the *vcopyn* function, which copies a certain value n times. In Line 5 the *vcopyn* function is mapped on vector *bs*, resulting in a n-by-n vector $b_i s$, where each ith row consist of n copies of b_i. Every row of $b_i s$ (along with *cfg*) will be used as input for the *bw_row*, as can been seen in Figure 27. Line 6 shows a function *cell_config* which generates a n-by-n bit vector, which contains a *Config* bit for every cell.

6 Add an element at the start of the vector, while leaving the length unchanged (i.e. the last element will be discarded).

7 Makes a list, which its elements are calculated from the function and the elements of the 5 input lists.

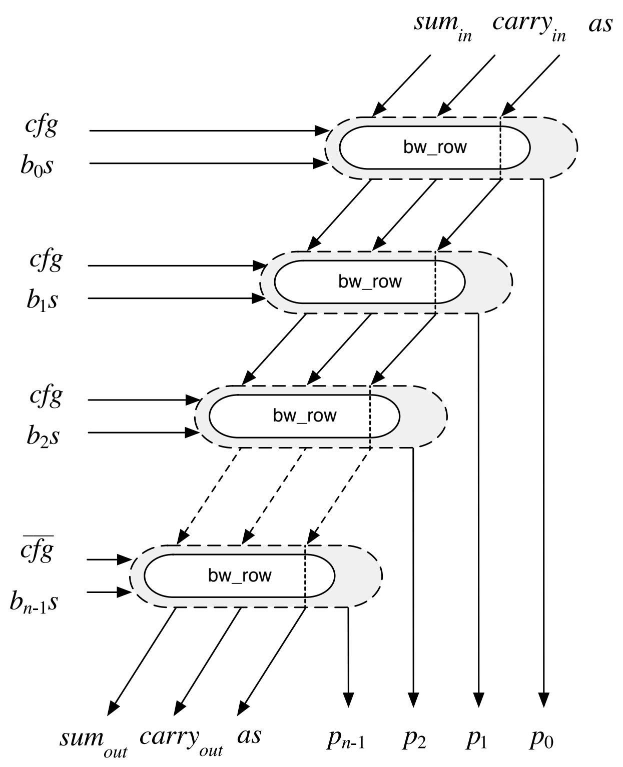


Figure 27: Partial products being chained.

After these preparations, the partial products (*bw_row*) are chained together in Lines 7 and 8, according to Figure 27. Using (as, ls, ls) as initial value and iterating on the zipped values of *cfg* and b_is. This chain results in a vector P_0, which corresponds with the values p_{n-1} to p_0 in Figure 24.

Listing 16: Implementation of a signed multiplier in CλaSH.

```
 1  vmult :: ⟨Bit⟩n → ⟨Bit⟩n → ⟨Bit⟩2n
 2  vmult as bs = ps
 3    where
 4      ls                = vcopyn n Low
 5      bis               = vmap (vcopyn n) bs
 6      cfg               = cell_config as
 7      ((_, cs, ss), P0) = vchain bw_row (as, ls, ls)
 8                            (vzip cfg bis)
 9      (_, P1)           = vchain fullAdd High
10                            (vzip cs (High +>> ss))
11      ps                = P1 <++>8 P0
12
13  (·) = vmult
```

8 concatenates two vectors.

Next, the elements of the last $Carry_{\text{out}}$ vector (the one on the bottom of Figure 27) will be added (by chaining it with the *fullAdd* function) to the last Sum_{out} vector (again, shown at the bottom of Figure 27), in lines 9 and 10. Resulting in a vector P_1 (p_{2n-1} to p_{2n-m-2} in Figure 24). Finally, P_1 is concatenated with P_0, (Line 11) to result in vector P, the product of A and B.

Finally, in Line 13 the function *vmult* is deducted to the operator '·', which lets the syntax of multiplying two *Vector*s be as simple as:

```
ps = as · bs
```

4.3.1 *Complex multiplier architecture*

There are basically two substitution algorithms for calculating the product of two complex numbers. The first one is shown in Equation 17. This algorithm requires four multiplications, one addition and a subtraction. (Please note that the notation of complex numbers also contains a plus symbol, but this is not an actual addition. Complex numbers are stored as a tuple of two *Vector*s of type *Bit*.)

$$\begin{aligned} P + jQ &= (A + jB) \times (C + jD) \\ P + jQ &= AC + jAD + jBC + j^2BD \\ P + jQ &= (AC - BD) + j(AD + BC) \end{aligned} \tag{17}$$

The implementation in CλaSH is really straightforward (Listing 17). The function *vcmult* takes two tuples consisting of two bitvectors of length n and produces a tuple of two bit vectors of length $2n + 1$ (Line 1 and 2). The function which describes the real part of product (P) is shown in Line 6, the imaginary part (Q) is shown in Line 7.

Listing 17: Implementation of a signed complex multiplier in CλaSH.

```
1  vcmult :: (⟨Bit⟩_n, ⟨Bit⟩_n) → (⟨Bit⟩_n, ⟨Bit⟩_n) →
2            (⟨Bit⟩_2n+1, ⟨Bit⟩_2n+1)
3
4  vcmult (as, bjs) (cs, djs) = (p, qjs)
5    where
6      p   = (as · cs)  <-> (bjs · djs)
7      qjs = (as · djs) <+> (bjs · cs)
```

The second algorithm for calculating the product of two complex numbers was discovered by Gauss in 1805 [35] and can be viewed in Equation 18 and requires only three multiplications, but three additions and two subtractions.

$$\begin{aligned} P + jQ &= (A + jB) \times (C + jD) \\ k_1 &= C \cdot (A + B) \\ k_2 &= A \cdot (D - C) \\ k_3 &= B \cdot (C + D) \\ P + jQ &= (k_1 - k_3) + j(k_1 + k_2) \end{aligned} \tag{18}$$

Every addition increments the length of the result and every multiplication doubles the length. Therefore, the implementation of the function *vcmult2* (Listing 18) shows some syntactic overhead to produce a result of length $2n + 1$.

Listing 18: Implementation of a signed complex multiplier in CλaSH.

```
1  vcmult2 :: ⟨Bit⟩_n, ⟨Bit⟩_n) → (⟨Bit⟩_n, ⟨Bit⟩_n) →
2            (⟨Bit⟩_2n+1, ⟨Bit⟩_2n+1)
3
4  vcmult2 (as, bjs) (cs, djs) = (vtail p, vtail qjs)
5    where
6      k_1  = (sE cs) · (as <+> bjs)
7      k_2  = (sE as) · (djs <-> cs)
8      k_3  = (sE bjs) · (cs <+> djs)
9      p    = (vtail k_1) <-> (vtail k_3)
10     qjs  = (vtail k_1) <+> (vtail k_2)
```

Just as in the other algorithm in Listing 17, does the Gauss algorithm takes two 2-tuples of bitvectors with a size n to create the result, which is a 2-tuple bitvector of size $2n + 1$. The helper variables k_1, k_2 and k_3 are defined in Lines 6 to 8. Note that the left argument of the multiplication is sign extended to a size of $n + 1$ by the function *sE* (which was defined in Listing 13). This is necessary since the right argument (the sum of two vectors) of the multiplication has length $n + 1$ and the multiplication architecture needs two vectors of the same size. In Line 9 the real part of the product (p) is described and in Line 10 the imaginary part (qjs).

Since the complexity of the multiplication architecture increases exponentially when increasing the length (resolution) of the vectors, the second algorithm seems the best choice when working with larger vectors, because it uses one multiplier less. However, adding two n-bit numbers, will result in an $n + 1$-bit number. Since the $'\cdot'$-operator requires two vectors of the same length, one of the inputs needs to be sign extended, which leads to more complex hardware, because not an n-bit multiplier is required here, but an $n+1$-bit one.

The focus of this research is particularly on lower resolution systems. So further investigation how the architecture scales on lower resolutions, with regards to the necessary chip area and the maximum frequency is welcome.

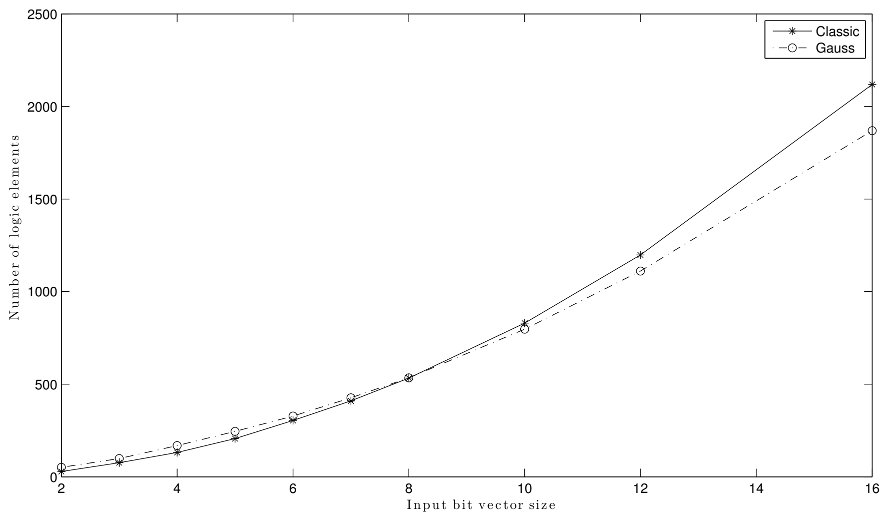


Figure 28: Comparison between the necessary area of both complex multiplication methods

In Figure 28 the comparison between the two algorithms, with respect to the necessary area can be seen. The two algorithms are compared by mapping them on an FPGA using the *Quartus II* software. The chosen FPGA architecture was the Altera *Cyclone IV* series, because of its widespread use and relatively large size.

Surprisingly the first algorithm seems to have a slight advantage over Gauss's algorithm, up to a bit vector size of 8 bits. After that, the Gauss algorithm has an increasing advantage over the first algorithm with respect to the required area. This is probably due the fact that the Gauss's algorithm uses a multiplier with 1-bit more resolution, which can have a bigger impact on the complexity then using an extra multiplier (when applied to lower bit vector sizes), like the first algorithm does.

Next, the maximum frequency feasible to compute the complex multiplication in a single cycle is a valuable criteria, which can be seen in Figure 29.

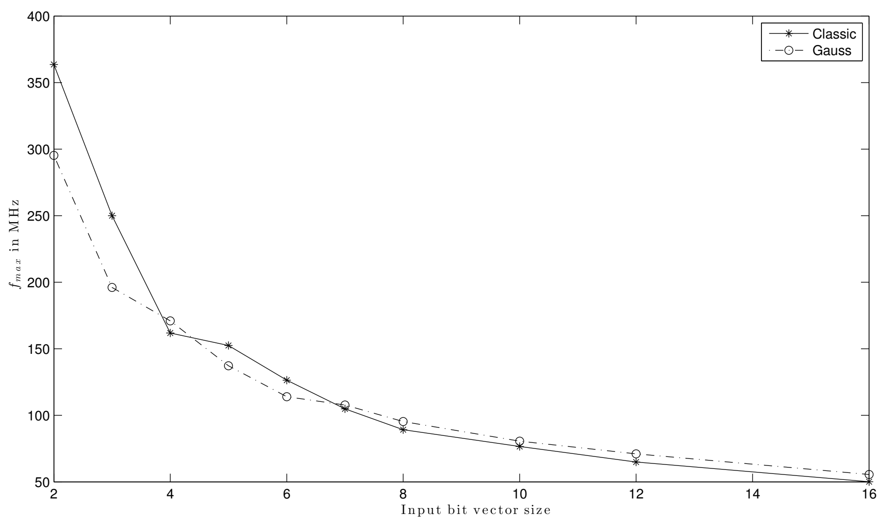


Figure 29: Comparison between the maximum frequency of both complex multiplication methods

Again, the first algorithm has a slight advantage on Gauss's algorithm, up to 6 bits. After that the differences become marginal. Although the advantages are not that significant, the first algorithm seems to be the best choice.

A operator symbol will be defined for the first algorithm, as can be shown in Listing 19. This will deduct the complex multiplication of two tuples to: $'(as, bs)$ <∗> $(cs, ds)'$.

Listing 19: Defining a operator symbol for complex multiplication in CλaSH.

```
(<*>) = vcmult
```

4.4 IMPLEMENTING CROSS-CORRELATION

The adder and multiplication circuits described in the previous paragraphs will be used as building blocks for the eventual cross-correlation algorithm. In Figure 30 can be seen how the complete cross-correlation circuit is constructed. A complex data sample *xs* is fed from the top left to the chain of *uss* (which consists of $\{us_0, us_1, ..., us_{n-1}\}$). The *ys* sample is fed to the registers *vss* registerbank (consisting out $\{vs_0, vs_1, ..., vs_{n-1}\}$) in the opposite direction at the bottom right.

The behavior of the correlator corresponds to Equation 2. Each opposing element of *uss* and *vss* is multiplied using the complex multiplication function '<∗>' (Listing 19).

The result of this multiplication is accumulated (Σ) and stored in the C_{XY} register bank. C_{XY} thus holds the result of the correlation operation. Internally it consists of two parts, the even values (C_{XYe}, which consists out of $\{C_{XYe_0}, C_{XYe_1}, ..., C_{XYe_{n-1}}\}$) and the odd values ($C_{XYo}$ consisting out $\{C_{XYo_0}, C_{XYo_1}, ..., C_{XYo_n}\}$).

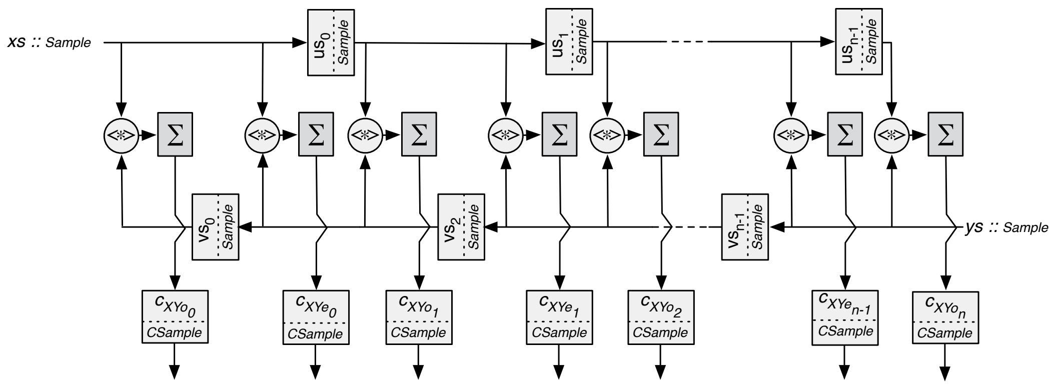


Figure 30: Schematic view of the corralator architecture.

The data types used in this circuit are defined in Listing 20. The input type of *xs* and *ys* is a *Sample* (Line 6) which consist of a tuple of two *Vector*s of type *Bit* with a size *Size*. The type *Lag* (Line 4) represents the number of lags (which is basically, the number of odd C_{XY} outputs). Every us_i and vs_i register is of the same *Sample* type as the input. This means that the register-banks *uss* and *vss* are both of type $\langle Sample \rangle_{Lag}$. The output of the complex multiplication (<∗>) is a tuple of bit-vectors of size $2n + 1$, which corresponds with the type *MSample* (Line 7). The *CSize* type describes the output size of the accumulator and thus each element of C_{XY} and the output of the correlator. The register-banks C_{XYe} and C_{XYo} are of type $\langle CSample \rangle_{Lag}$ and $\langle CSample \rangle_{Lag+1}$ respectively.

Listing 20: Types used in the cross-correlator architecture.

```
type Size  = n
type MSize = 2*n + 1[9]
type CSize = c
type Lag   = L

type Sample   = (⟨Bit⟩_Size, ⟨Bit⟩_Size)
type MSample  = (⟨Bit⟩_MSize, ⟨Bit⟩_MSize)
type CSample  = (⟨Bit⟩_CSize, ⟨Bit⟩_CSize)
```

4.4.1 *Accumulator*

As described earlier, in order to correctly represent the sum, every addition increases the size of the output vector. Which means that when a number gets accumulated n times, its vector size also increases with n bits. This is a very unpractical property for an accumulator to have, because our hardware architecture size cannot grow and thus has to be fixed. Therefor a complex adder which leaves the size of the bit-vectors unaffected was created. This can be viewed in Listing 21. Note that the type of the two input arguments is the same as the output argument (Line 1).

Listing 21: Accumulator function in CλaSH.

```
1 (Σ) :: (⟨Bit⟩_n, ⟨Bit⟩_n) → (⟨Bit⟩_n, ⟨Bit⟩_n) → (⟨Bit⟩_n, ⟨Bit⟩_n)
2
3 (Σ) (as, bs) (cs, ds) = (ps, qs)
4   where
5     ps = vtail (as <+> cs)
6     qs = vtail (bs <+> ds)
```

The accumulator function (Σ) uses the addition operator (<+>), which was described in Listing 13. It discards the first element by using the *vtail* function (in Lines 5 and 6), this way the output remains the same size as the input. To still have the necessary accuracy, the samples from the multiplier of type *MSample* are sign extended to the much bigger *CSample* type by using the *ms2cs* function, as can been seen in Figure 31.

9 In CλaSH this would be notated as: *Succ (n :+: n)*

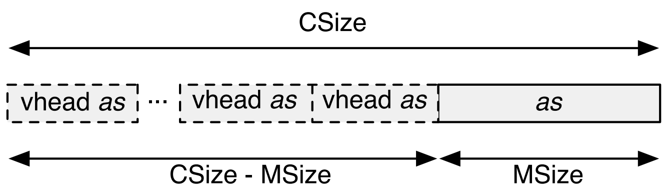


Figure 31: Sign extending a *MSample* to a *CSample*.

The CλaSH description of the *ms2cs* function is presented in Listing 22. It takes a argument of type *MSample* and puts out a *CSample* type (as can be viewed in the type declaration in Line 1).

Listing 22: Converting a *MSample* to *CSample* by sign extension.

```
1 ms2cs :: MSample → CSample
2
3 ms2cs (as, bs) = (as′, bs′)
4   where
5     t   = vcopyn (CSize - MSize) (vhead as)
6     t′  = vcopyn (CSize - MSize) (vhead bs)
7     as′ = t <++> as
8     bs′ = t′ <++> bs
```

In lines 5 and 6 two vectors (t and t') are created. They will only contain several copies of the first element of the inputs *as* and *bs*, and therefore the actual sign extension. Their size is defined by $CSize - MSize$. When they are concatenated (using the (<++>) function) with *as* and *bs* in Lines 7 and 8 the size of the output vectors as' and bs' results in: $CSize - MSize + MSize = CSize$.

4.4.2 *Cross-correlator architecture typing*

In Listing 23 the type declaration of the function *ccc* can be seen, which describes the input and output arguments of the cross-correlation architecture. The first argument (a 4-tuple of *Vector*s) of the cross-correlation function is annotated with the *State* keyword (Line 2 to 5), which indicates that those four *Vector*s will be connected to memory elements. The first two *Vector*s are the *uss* and *vss* register-banks, the other two together represent the C_{XY} register-bank (Fig 30). The second argument is the actual input, the two *Sample*s *xs* and *ys* (Line 6).

Listing 23: Type of the cross-correlator function in CλaSH.

```
1  ccc ::
2       State (
3          ⟨Sample⟩_Lag, ⟨Sample⟩_Lag,       -- (uss, vss)
4          ⟨CSample⟩_Lag+1, ⟨CSample⟩_Lag    -- (CXYe, CXYo)
5          ) →
6     (Sample, Sample) →                    -- (xs, ys)
7     (State (
8          ⟨Sample⟩_Lag, ⟨Sample⟩_Lag,       -- (uss', vss')
9          ⟨CSample⟩_Lag+1, ⟨CSample⟩_Lag    -- (CXYe', CXYo')
10         ),

11    (⟨CSample⟩_Lag+1, ⟨CSample⟩_Lag))
```

The output consist of a tuple, where the first part (Line 7 to 10) is identical to the first input argument. This contains all the updated content for the memory elements. The second part is the actual output of the cross-correlation (Line 11), i. e. the updated values of the C_{XY} registers.

4.4.3 *Cross-correlator architecture description*

The description of the cross-correlation function *ccc* can be viewed in Listing 24.The new content of register-banks *uss* and *vss* are described in Lines 4 and 5. The *xs* value is added to the beginning of *uss* and *ys* to the end of *vss*. The *afss* and *agss* in Lines 6 and 7 describe the output of the complex multiplier. Note that, *afss* is the complex multiplication between *xs* in front of (note the '+>'-operator) *uss* and *vss* in front of *ys* (making both vectors one element larger than *uss'* and *vss'*). *agss* is on the other hand the complex product between *uss'* and *vss'* (using the '+»'-operator), this way *afss* represents the odd number of complex multiplications in Figure 30 and *vss'* the even ones.

Listing 24: Architecture of a cross-correlator in CλaSH.

```
1   ccc (State (uss,  vss, c_xye, c_xyo)) (xs, ys) =
2       (State (uss', vss', c'_xye, c'_xyo), (c'_xye, c'_xyo))
3     where
4       uss'  = xs +>> uss
5       vss'  = vss <<+ ys
6       afss  = vzipWith (<*>) (xs +> uss) (vss <+ ys)
7       agss  = vzipWith (<*>) uss' vss'
8       afss' = vmap ms2cs afss
9       agss' = vmap ms2cs agss
10      c'_xye  = vzipWith (Σ) afss' c_xye
11      c'_xyo  = vzipWith (Σ) agss' c_xyo
12
13  cccL = ccc ^^^ (uss_init, uss_init, c_xye_init, c_xyo_init)
14
15  {-# ANN cccL TopEntity #-}
```

First every *MSample* in *afss* and *agss* gets upscaled to a *CSample*, by mapping the *ms2cs* function on them (Line 8 and 9). Both upscaled complex products ($afss'$ and $agss'$) are then added to the *CSample* registers c'_{xye} and c'_{xyo} by 'zipping' them with the previous value of the output state (c_{xye} and c_{xyo}), using the accumulator function (Σ) (Line 10 and 11).

Line 15 tells the compiler that the function *cccL* is the *TopEntity*, which indicates that it's at the top of the circuit hierarchy. In Line 13 the memory elements of *ccc* are filled with their initial values, using the lift function (^^^). These values are declared in Listing 25.

Listing 25: Generating the initial values of the registerers of a cross-correlator circuit.

```
 1  uss_init = vcopyn Lag (t, t)
 2    where
 3      t = (vcopyn Size Low)
 4
 5  c_xye_init = vcopyn (Lag + 1) (t, t)
 6    where
 7      t = (vcopyn CSize Low)
 8
 9  c_xyo_init = vcopyn Lag (t, t)
10    where
11      t = (vcopyn CSize Low)
```

To construct the initial values of the registers, the *vcopyn* function copies the complex number (t, t) (Line 1, 5 and 9)) *'Lag'*-times (Line 1 and 9) or *'Lag+1'*-times (Line 5). The tuple (t, t) describes a *Sample* type (Line 3) or a *CSample* type (Line 7 and 11) with value $0 + j0$.

4.4.4 *Non-complex multiplier*

Since not every cross-correlator will need the ability to handle complex data, it is also sensible to analyze a non-complex (integer) cross-corelator. The correlator architecture of Listing 24 can be fairly easily adapted for this. First, the input data needs to be converted to a non-complex type. This can bee seen in Listing 26. Note that the size of the output of the multiplier is now $2n$ instead of $2n + 1$.

Listing 26: Types used in the cross-correlator architecture.

```
type MSize   = 2*Size[10]

type Sample  = ⟨Bit⟩_Size
type MSample = ⟨Bit⟩_MSize
type CSample = ⟨Bit⟩_CSize
```

The description of the cross-correlator barely changes. Because of the benefits of using higher order functions, the complex multiplication operator '<∗>' can be interchanged with the normal multiplication operator '·'. The accumulator function is switched to the more simple '‡'-operator, which is presented in Listing 27.

10 In CλaSH this would be notated as: Size :+: Size

Listing 27: Accumulator function in CλaSH.

```
(‡) as bs = cs
  where
    cs = vtail (as <+> bs)
```

The function *ms2cs′* (Listing 28) can be used to convert a real *MSample* type to a real *CSample* type (Line 1). Its body is basically the same as *ms2cs* in Listing 22, but instead it handles only one vector (*as*) instead of 2-tuple.

Listing 28: Converting a real *MSample* to a real *CSample* by sign extension.

```
1  ms2cs' :: MSample → CSample
2
3  ms2cs' as = as′
4    where
5      t   = vcopyn (CSize - MSize) (vhead as)
6      as′ = t <++> as
```

4.4.5 *Simulation*

Because the two vectors C_{XY_e} and C_{XY_o} represent the odd and even correlated samples in the circuit, they need to be merged together. This is done during simulation of the circuit, which is done in Haskell, outside the CλaSH environment, where the limitations of CλaSH don't apply. This means that it is possible to utilize techniques such as lists and recursiveness during simulation. Before the C_{XY} vectors will be merged, they will be converted to two lists which both holds signed integers (using the standard function *bv2s* and *fromVector*). They will then be merged by the *merge* function, displayed below in Listing 29.

Listing 29: *merge* function.

```
1  merge []     bs     = bs
2  merge as     []     = as
3  merge (a:as) (b:bs) = a : b : merge as bs
```

merge requires two lists (which can differ in size) as input and puts the first element of both lists (*a* and *b* in Line 3) next to each other in a new list and then recursively calls itself with the remaining list. If one of the lists becomes empty, the elements of the non-empty are just concatenated to rest (Lines 1 and 2), resulting in a list with size that is equal to the sum of the sizes of the two input arguments ($2 \cdot Lag + 1$).

The final step in designing the cross-correlator architecture is to verify if it works as expected by checking if it generates the right data. The code that was used to simulate the architecture is presented in Listing 30.

Listing 30: Simulating the correlator.

```
1  re = [..., ...]
2  im = [..., ...]
3
4  input = zip re im -- Complex Sample
5
6  C_XY = simulate cccL $ zip input input
7
8  (C_XYe, C_XYo) = unzip $ toSigned C_XY
9
10 C'_XYe = map fromVector C_XYe
11 C'_XYo = map fromVector C_XYo
12
13 outputdata = zipWith merge C'_XYe C'_XYo
```

The first two lines present the real and imaginary part of the input signal, of which a complex *Sample* is constructed by zipping both (Line 4). In Line 6 two copies of the *input* sample will be fed to the correlator architecture (*cccL*) using the *simulate* function. Thus, the architecture functions as an auto-correlation circuit here. The output of the simulation is stored in C_{XY} and casted to a signed Integer in Line 8. Both parts of C_{XY} are then converted from a vector to a list (Lines 10 and 11) by mapping the standard *fromVector* function over them. The output data consists of both parts of the C_{XY} merged together by zipping C_{XYe} and C_{XYe} with the *merge* function (Line 13). Note that, every time a sample is fed to the correlator, the output register C_{XY} is updated. This results in the list *outputdata*, in which every element chronically corresponds to a value these C_{XY} registers have had.

The outputted data was manually verified for correctness, which showed that the architecture works properly. The next step is generating the VHDL-files and do a post-simulation in *Quartus II* by *Altera*. The results will be presented in the following chapter.

5 ARCHITECTURE ANALYSIS

The analysis of cross-correlator architecture will be done by mapping it on a *Cyclone IV* FPGA by *Altera*, using the *Quartus II v11.1* software. In this chapter the results for the area utilization analysis, timing analysis and the case study for the sensitivity are presented, for both the complex correlator and the non-complex variant.

5.1 AREA UTILIZATION

The utilization of the area on an FPGA is measured in the number of registers and Logic Element (LE)s used. LEs are the smallest logic units in FPGAs. Each LE of the *Cyclone IV* consist of a four-input Look-up table (LUT) which can implement any function of four variables [36]. Next to a LUT, every LE also features a programmable register to store data and some components to chain multiple LEs together in different directions, as can be seen in Figure 32.

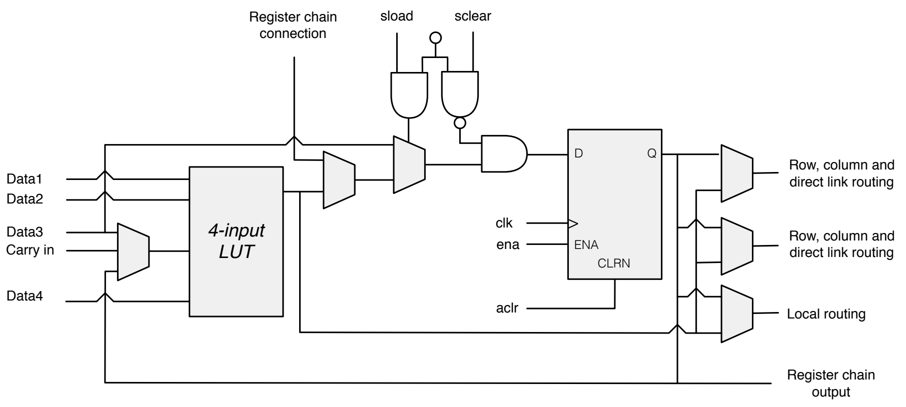


Figure 32: Schematic view of a *Cyclone IV* LE in normal mode [36].

The goal of the area utilization analysis is to find what the relation is between the number of LEs needed for the architecture for different number of lags and input vector size in the correlator.

5.1.1 *Area utilization for complex correlator*

The number of LEs will probably increase linearly with each lag. Since with every lag, two input registers (*us* and *vs*), two complex multipliers (<∗>), two accumulators (Σ) and two output registers (C_{XY}) are added to the correlator architecture (Figure 33).

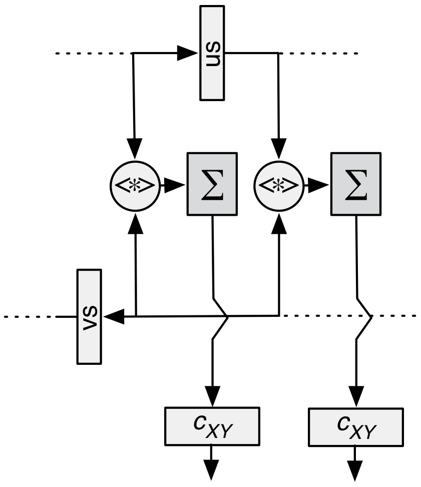


Figure 33: With every increase of lag, the architecture increases.

The estimate of the increase in size of the architecture ($\Delta\Theta$) with every added lag can therefore be formulated as:

$$\Delta\Theta_{nc} \approx 2\delta_n^{\mathrm{uv}} + 2X_n + 2\sigma_c + 2\delta_c^{C_{XY}} \tag{19}$$

Where n indicates the number of bits (or resolution) of the input vector (*Size* in Listing 20) and c correspondes to the resolution of the output vector of the correlator (*CSize* in Listing 20). δ_n^{uv} represents the number of LEs which are needed for the input register *us* or *vs*, X_n stands for the size of the complex multiplier (in LEs), σ_c depicts the number of LEs needed for the correlator output and $\delta_c^{C_{XY}}$ indicates the size of the output register C_{XY} (in LEs).

For every part of Equation 19 the estimation for the necessary number of LEs can now be formulated. The input registers *us* and *vs* are both of the same size, which corresponds to:

$$\delta_n^{\mathrm{uv}} = \delta_n^{\mathrm{us}} = \delta_n^{\mathrm{vs}} = 2n \tag{20}$$

This will be $2n$, since both registers store a complex sample, which needs a real and an imaginary part. The estimation of the size of the output registers $\delta_c^{C_{XY}}$ is given by:

$$\delta_c^{C_{XY}} = 2c \tag{21}$$

This will also be a complex sample, but it depends on the size of the output vector c.

To find the relationship between the number of input bits used against the size of the accumulator σ, some post-simulation was done in *Quartus II*. The results can be seen in Figure 34.

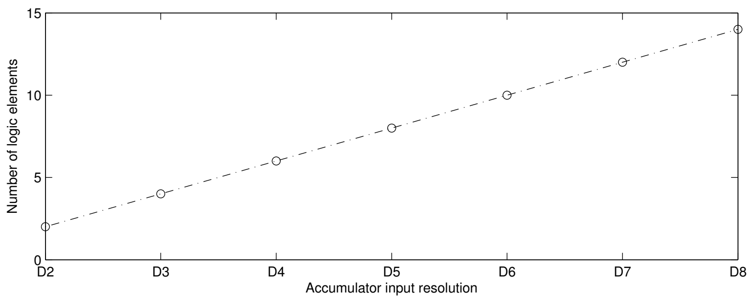


Figure 34: Necessary area for accumulator architecture.

Where the $\circ$ shows the measured data and the dotted line shows the linear estimation. From this the following relation can be extracted:

$$\sigma_c = 4(c-1) \tag{22}$$

Tests have shown that Equation 22 holds up beyond $c > D256$.

The number of LEs that are needed for the complex multiplier will depend on the the size of its internal components: its multipliers (χ), adders (τ^+) and subtracters (τ^-) (note that, the adder and subtracter will have a size depending on $2n$, since they add up the multiplier results, as can be seen in Listing 17):

$$X_n = 4\chi_n + \tau^+_{2n} + \tau^-_{2n} \tag{23}$$

Therefore it is necessary to analyze each internal component to give an estimated relationship on the total size. The relation between the adder/subtracter architecture and the used bit vector size, can be seen in Figure 35. The $\circ$ shows the measured data, while the dotted line shows the linear estimation.

The dotted line shows a linear relation, which results in the following estimation:

$$\tau^+_n = \tau^-_n \approx \lceil 2.2n - 2 \rceil \tag{24}$$

This estimation holds for a big n ($n > D256$), but occasionally has - for unclear reasons - a small deviation (as can be seen at $n = D4$ in Figure 35), but these are always within a reasonable margin of only a couple LEs.

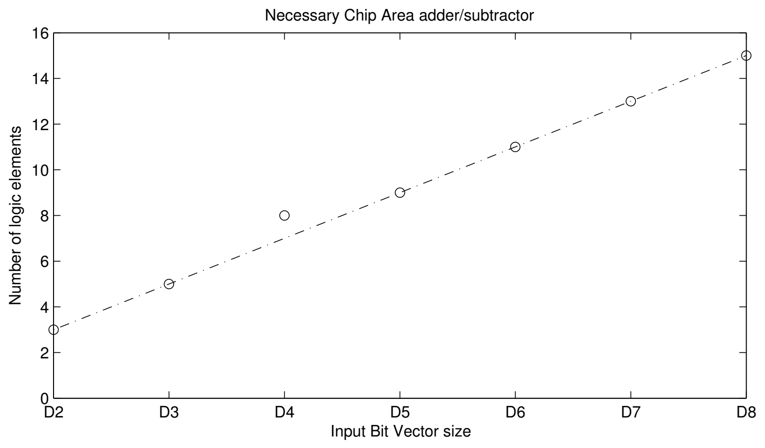


Figure 35: Necessary area for the adder/subtracter architecture.

And finally, the impact which the input vector has on the size of the multiplier (χ) is shown in Figure 36. The $\circ$ shows the measured data and the dotted line the quadratic interpolation.

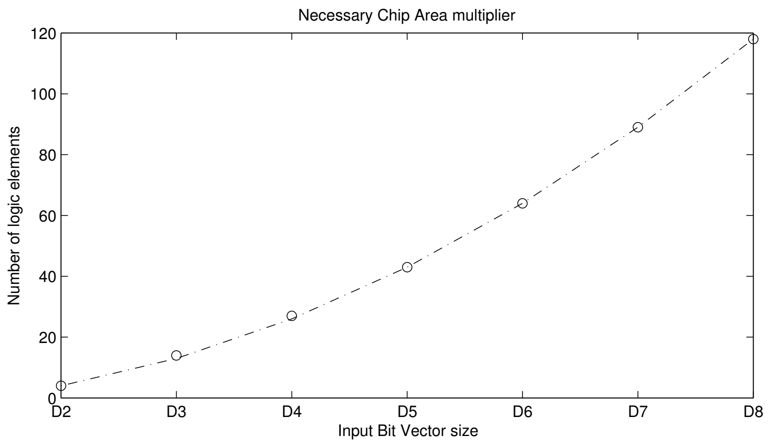


Figure 36: Necessary area for the multiplier architecture.

From the estimation that was used for the dotted line, the following relation can be derived:

$$\chi_n \approx \lceil 2n^2 - 0.75n - 1 \rceil \tag{25}$$

This relation holds for vectors above $n > D256$ (with an estimation error of $-3,98\%$).

Based on the previous relations, it will be possible to derive a relation for the complete complex correlator architecture size against it's number of lags and input vector size. Which will results in:

$$\Theta_{Lnc} \approx (2L+1) \cdot (\delta_n^{\text{uv}} + X_n + \sigma_c + \delta_c^{C_{\text{XY}}}) \tag{26}$$

Where L represents the number of lags in the correlator (which corresponds with *Lag* in Listing 20), meaning that $2L + 1$ will indicate the number of complex multipliers, accumulators or input & output registers in the architecture. Equation 26 thus not only gives a relation between the necessary LEs and the input vector size, but also between the number of lags and the required area.

To verify the relations, the actual required area for the complex correlator architecture was produced by a fitter report of *Quartus II*. The graph of Figure 37 shows the synthesis results for the number of necessary LEs against the number of lags for different input resolutions.

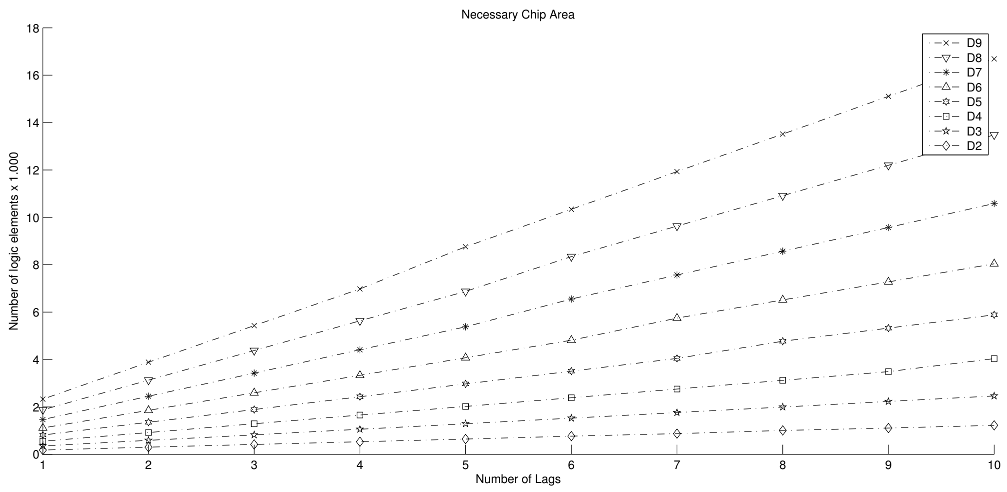


Figure 37: Necessary area for the complex correlator architecture for different input bit vector sizes and nummber of lags.

The results in the graph don't go beyond an input resolution of $D9$ and 10 lags, because around this point the synthesizing procedure became too time consuming. The results needed up to a day of calculation time on a modern desktop PC, while the relation found between the necessary LEs and the number of lags used was clearly linear.

Tests have shown that the estimated error (ϵ) of the calculated results with respect to the measured synthesis results is between the -2% and 3.3% (and on average 0.61%). The estimation of Θ_{Lnc} was tested for a large architectures with a $L = D256$ and an input resolution of $D9$, which still seems to hold.

$$\epsilon = \frac{\Theta_{Lnc_{\text{estimated}}} - \Theta_{Lnc_{\text{measured}}}}{\Theta_{Lnc_{\text{measured}}}} \cdot 100\% \tag{27}$$

Surprisingly, while this error margin holds for even very large correlator architectures, the estimated necessary area for $n = D2$ is a lot bigger then actual necessary area calculated by *Quartus II* ($\epsilon \approx -19\%$). This is probably due the fact that *Quartus II* can utilize a lot of optimization algorithms when such small vectors are used.

The FPGA that was used for during synthesizing was the *Cyclone IV* (type: *EP4CGX150DF31C7* by *Altera*) features 149.760 LEs and 508 I/O pins. By using Equation 26, we can calculate that a correlator architecture with 10 lags, an input resolution n of $D8$ and a output resolution c of $D19$ needs:

$$\Theta_{10,8,19} \approx (2 \cdot 10 + 1) \cdot (\delta_8^{\text{uv}} + X_8 + \sigma_{19} + \delta_{19}^{C_{\text{XY}}}) = 14.238 \text{ LEs} \quad (28)$$

Which is roughly around 10% of the available LEs. But when looking at the number of I/O pins that are necessary:

$$P_{Lnc} \approx 2 \cdot \delta_n^{\text{uv}} + (2L+1) \cdot \delta_c^{C_{\text{XY}}} + r \quad (29)$$

The number of necessary pins is depended on the 2 input pins δ_n^{uv}, the output pins $\delta_c^{C_{\text{XY}}}$ (which depend on the number of lags) along with some non-data pins (needed for power-supply and clocks), which is indicated by r. The number of pins that are necessary for δ_n^{uv} and $\delta_c^{C_{\text{XY}}}$ are equal to the amount of LEs needed in Equation 20 and 21. Unfortunately $P_{10,8,19}$ would be around 830 pins, which is more than the chosen FPGA has. In Figure 38 the needed pins for different input resolutions and number of lags can be seen. The horizontal line at 508 pins shows the number of pins available.

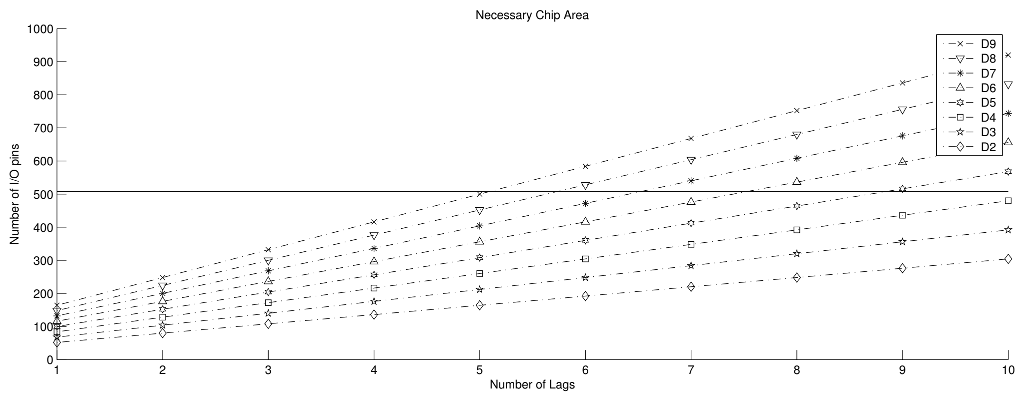


Figure 38: Necessary pins for the complex correlator architecture for different input resolutions and nummber of lags.

5.1.2 *Area utilization for non-complex correlator*

The area utilization relationship for the correlator architecture which uses only real bit vectors, can be derived from Equation 26. The complex multipliers (X_n) will be interchanged for non-complex ones (χ_n). The area usage of χ was analyzed in Equation 25.

$$\Theta_{Lnc_{\mathbf{R}}} \approx (2L+1) \cdot (\delta^{\mathrm{uv}}_{n_{\mathbf{R}}} + \chi_n + \sigma_{c_{\mathbf{R}}} + \delta^{C_{XY}}_{c_{\mathbf{R}}}) \tag{30}$$

The size of the input registers will be halved with respect to their complex peers, due to the fact that they only store the real part:

$$\delta^{\mathrm{uv}}_{n_{\mathbf{R}}} = \delta^{\mathrm{us}}_{n_{\mathbf{R}}} = \delta^{\mathrm{vs}}_{n_{\mathbf{R}}} = n \tag{31}$$

The same reduction holds for the output registers $\delta^{C_{XY}}_{c_{\mathbf{R}}}$, which depends on the size of the output vector c.

$$\delta^{C_{XY}}_{c_{\mathbf{R}}} = c \tag{32}$$

Finally the relationship between the output vector size and the number of necessary LEs of the accumulator ($\sigma_{c_{\mathbf{R}}}$) needs to be analyzed. The measured results are presented in Figure 39, where the $\circ$ shows the measured data and the dotted line the linear interpolation as usually.

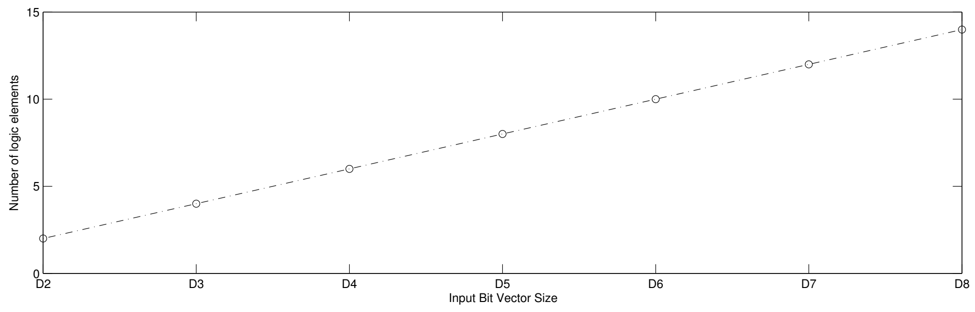


Figure 39: Necessary area of the accumulator $\sigma_{c_\mathbf{R}}$ for different input bit vector sizes.

The linear estimation that was used in the above plot was:

$$\sigma_{c_\mathbf{R}} = 2(c-1) \tag{33}$$

This relation was tested to hold up exactly, even to very big input vector sizes (up to $D256$). Again, to see how every component congregates in the whole of Equation 30, the *Quartus II* software was used to generate the actual amount of necessary chip-area. The results can be seen in Figure 40.

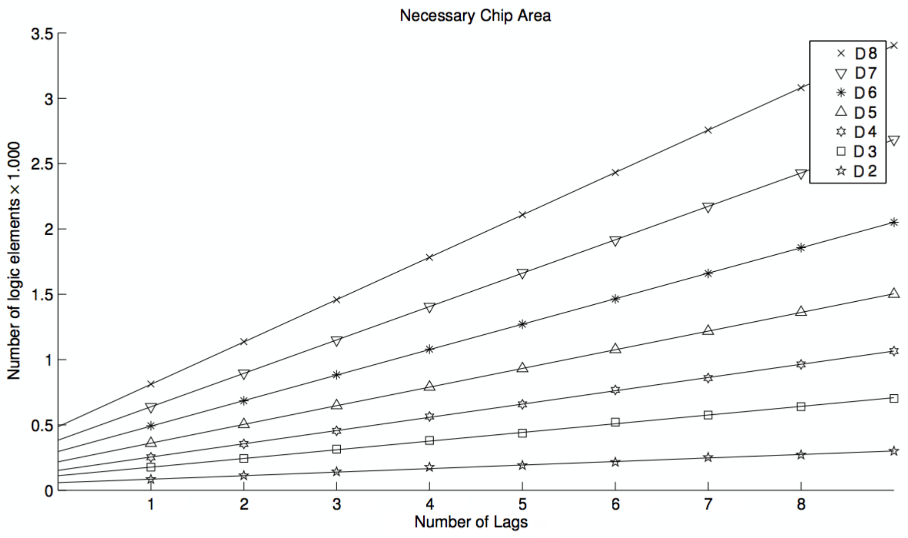


Figure 40: Necessary area for the non-complex correlator architecture for different input bit vector sizes and nummber of lags.

Tests have shown that the estimated error is between the -1.5% and 1.5%:

$$\epsilon_\mathbf{R} = \frac{\Theta_{Lnc_{\mathbf{R}_\text{estimated}}} - \Theta_{Lnc_{\mathbf{R}_\text{measured}}}}{\Theta_{Lnc_{\mathbf{R}_\text{measured}}}} \cdot 100\% \tag{34}$$

As with the complex correlator architecture, this error margin holds for quite large circuits, but not for $n = D2$. Where $\epsilon_{\mathbf{R}} \approx -17.2\%$ on average.

The number of necessary pins for the non-complex correlator is given by:

$$P_{Lnc_{\mathbf{R}}} \approx 2 \cdot \delta_{n_{\mathbf{R}}}^{\text{uv}} + (2L+1) \cdot \delta_{c_{\mathbf{R}}}^{C_{\text{XY}}} + r \approx \frac{P_{Lnc}}{2} \tag{35}$$

Figure 41 shows the necessary pins for the non-complex correlator architecture for different input resolutions and number of lags. The horizontal line indicates the number of available pins in the used *Cyclone IV*. As can be seen, all result shown here will fit.

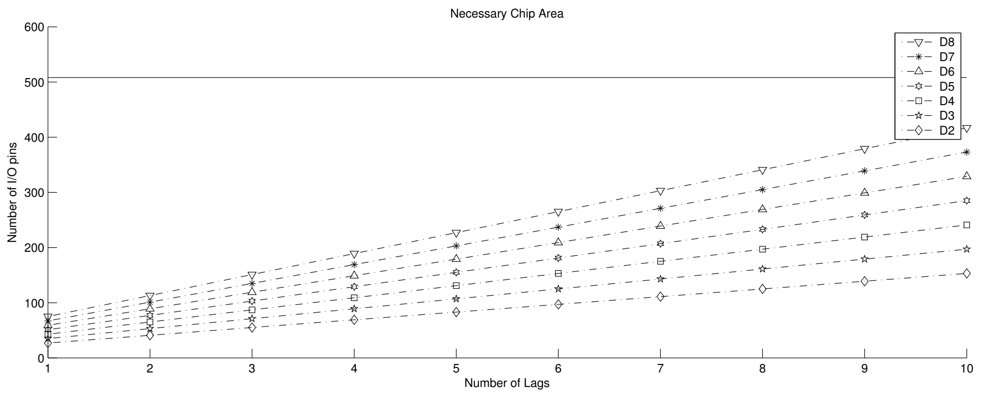


Figure 41: Necessary pins for the non-complex correlator architecture for different input resolutions and nummber of lags.

5.2 TIMING ANALYSIS

Critical path analysis involves the analysis of the longest register-to-register paths in the architecture. It was performed with the *TimeQuest Timing Analyzer* in *Quartus II*. A testbench was created which embedded the correlator architecture within a 'bed' of registers. This way the longest register-to-register path is from the input till the output. This is schematically shown in Figure 42.

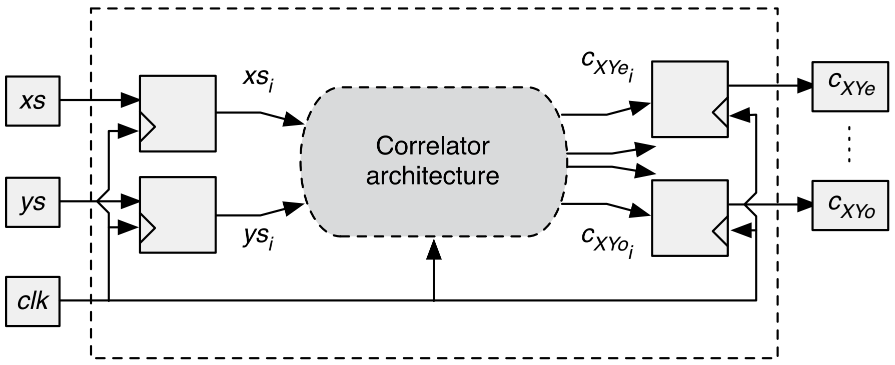


Figure 42: Schematic view of a testbench for a timing analysis.

5.2.1 *Critical path analysis for complex correlator*

The results of complex correlator architecture analysis can be viewed below in Figure 43.

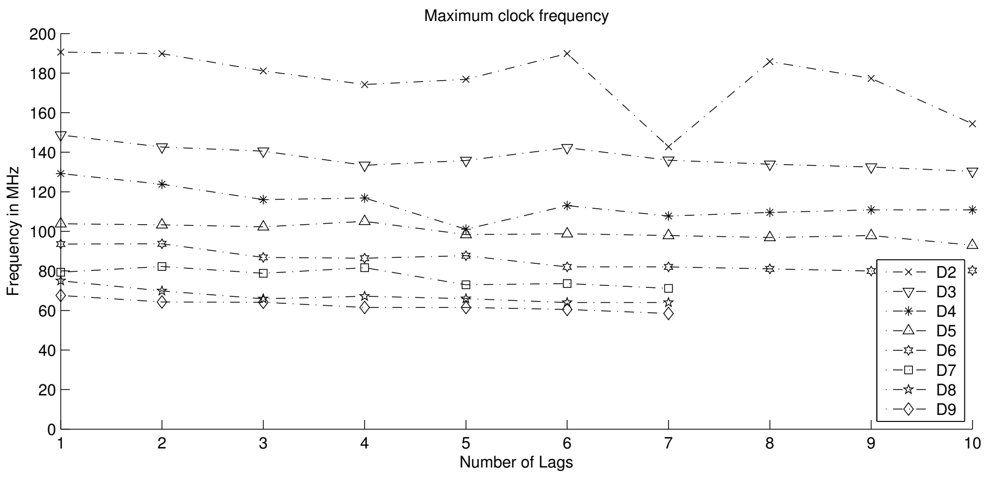


Figure 43: Maximum clock rate for input vector sizes *D2* to *D9*, for a different number of lags.

There are a couple of things that can be noticed from the graph. First, increasing the lag size of the correlator seems to have little effect on the maximum clock frequency that correlator operation can be executed. This is due to the fact that every MAC operation can be executed in parallel, since it does not depend on results from other MACs. For some unclear, reasons the plot of *D2* also show a rather great drop at 7 lags. The plots for input sizes *D7*, *D8* and *D9* stops after a lagsize of 7, this is because the design doesn't fit on the FPGA anymore (Although there is still area available, there are no more pins left). The tooling can still give information about the number of LEs necessary, but is unable to do a timing analysis, because it uses a model of an actual FPGA.

If we take the average of each input size we can show the decrease in clock rate for every input vector size. These results can be viewed in Figure 44, where the ○ shows the measured data and the line the interpolation.

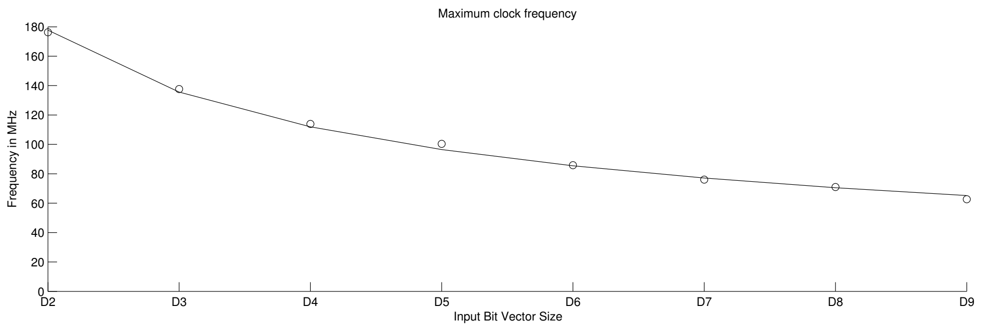


Figure 44: Average maximum clock rate for different input vector sizes.

The estimated interpolation function that was used in the above plot was[1]:

$$\Phi_n \approx min(282 \cdot n^{-\frac{2}{3}},\ f_{\max}) \tag{36}$$

Where Φ_n is the maximum clock rate possible at input vector size n. The clock rate is specific for specific FPGAs, indicated by $f_{\max}$ (the maximum clock rate of an *Cyclone IV* is 250 MHz). This estimation was confirmed to be correct up to a input size of *D32*.

1 This estimation was calculated with the help of the *EzFit* toolbox for *Matlab*.

5.2.2 *Critical path analysis for non-complex correlator*

The same analysis was done for the non-complex correlator. The results can be viewed below, in Figure 45. Again, the de-

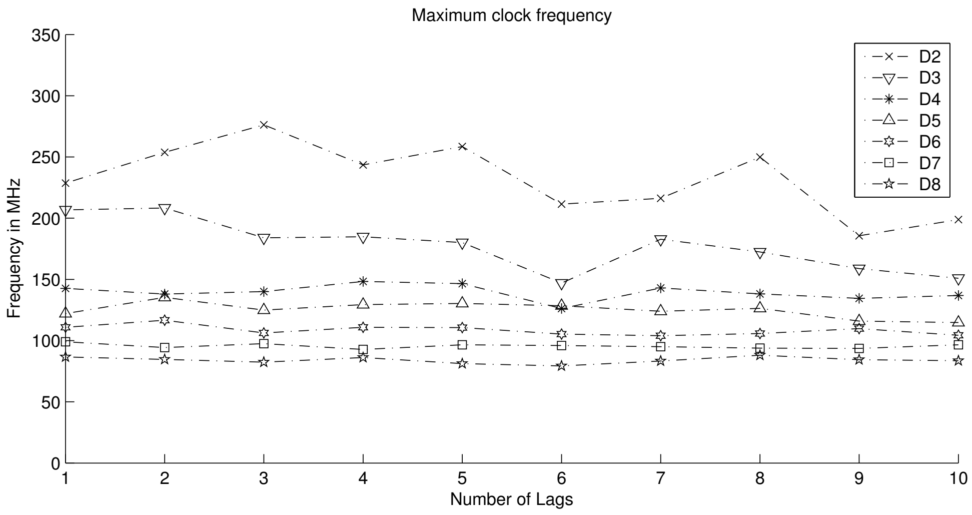


Figure 45: Maximum clock rate for input vector sizes *D2* to *D8*, for a different number of lags.

viation is rather small when increasing the number of lags. The two smallest input vector sizes *D3* and especially *D2* do follow a somewhat uneven course. But when considering that the maximum frequency is cut off at $f_{\max} = 250MHz$, the derivation don't seem so worse. To show the decrease in maximum clock rate for every input vector size, the average of each input size is taken. The results are shown in Figure 46. The non-complex correlator seems to have some advantage over the complex one, this is off course not surprising since the MAC (which is by far the most significant factor in the maximum clock frequency) is a lot smaller in the non-complex circuit.

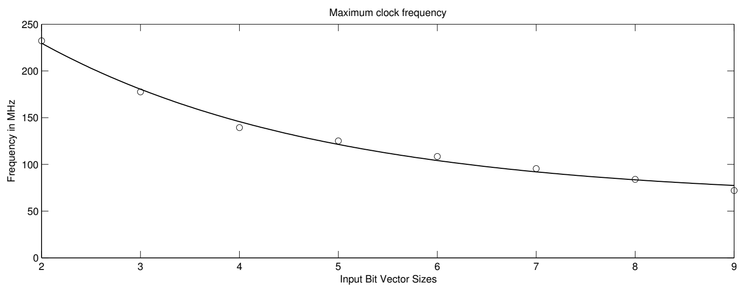


Figure 46: Average maximum clock rate for different input vector sizes.

As usual, the ○ shows the averaged measured data, and the line shows the exponential interpolation, for which the following estimation was used[2]:

$$\Phi_{n_R} \approx min(382 \cdot n^{-\frac{7}{10}},\ f_{\max}) \tag{37}$$

The results seem to hold up to a correlator circuit with an input vector size of *D32*.

5.3 CASE STUDY: SPURIOUS FREE DYNAMIC RANGE

To give an extensive analysis of the sensitivity of the correlator architecture would be a time consuming task. The sensitivity depends on a lot of factors such as the certain types of noise factors, the distribution of the spectral density function of these noise factors, the bandwidth of the input data, the accuracy of the Analog to Digital Converter (ADC), among many others. Given the time limitations during this research, this analysis was limited to a case study, where the Spurious-Free Dynamic Range (SFDR) of the architecture for various input and lag sizes were calculated.

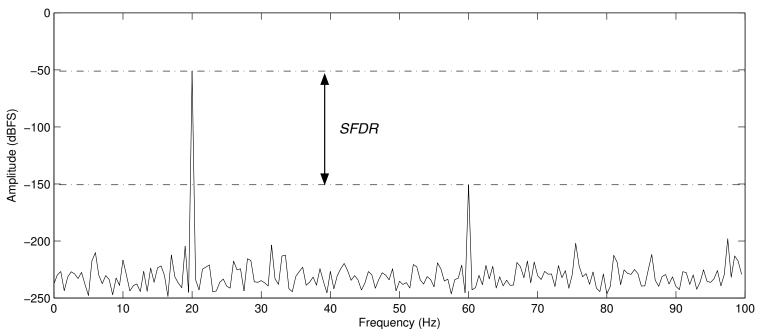


Figure 47: The SFDR example viewed on a single-sided amplitude spectrum.

The SFDR is the strength ratio of the fundamental signal to the strongest spurious signal in the output. An example is showed in Figure 47. The SFDR is measured in Decibels (dB).

2 This estimation was calculated with the help of the *EzFit* toolbox for *Matlab*.

During this case study the SFDR of the cross-correlator architecture for different resolution and lag sizes was calculated. First two signals were created in Matlab, a noisy sine (*s*) and a noisy cosine (*c*):

```
rng('default');                     % Reproducable results

v   = ...                           % Input resolution
N   = 512;                          % Number of samples
f1  = 32;                           % Frequency of the sinewave
FS  = 1024;                         % Sampling frequency
n   = 0:N-1;                        % Sampling index

bound = 2^(v-1)-1;                  % Maximal number
                                    % presentable by v bits

s = bound * sin(2*pi*f1*n/FS);      % Generate sine
s = s + bound/20 * randn(1,N);      % Add random noise

c = bound * cos(2*pi*f1*n/FS);      % Generate cosine
c = c + bound/20 * randn(1, N);     % Add random noise
```

Both signals are multiplied with *bound*, which indicates the greatest number that can be represented by a *v*-bit two's-complement number. This way the eventual signal spans the complete amplitude band of the input vector and can be easier quantized to a lower bit resolution. The amount of noise that is added to both signals is depended on *bound*, which results in a constant SNR (of 13 dB). The quantization is then simply done by:

```
rs = round(s);                      % Quantize Sine
rc = round(c);                      % Quantize Cosine
```

The arrays *rs* and *rc* will contain the newly quantized signals with a resolution of *v* bits. These can then be converted to bit vectors (*bs* and *bc*) with a resolution of *v*:

```
bs = dec2bit(rs, v);
bc = dec2bit(rc, v);
```

These are converted to a Haskell compatible type, which can then be fed to the architecture during simulation:

```
input  = zip ss cs                      -- Complex Sample
output = simulate cccL $ zip input input
```

Zipping the vectors *ss* and *cs* results in a complex vector *input*. Which is used for both inputs of the correlator architecture (*cccL*), thus letting the design serve as a autocorrelation circuit. To see how the frequency activities are distributed across the frequency band, the Fourier transform of the output of the correlator architecture will be taken, which results in the power spectral density function or spetrum (As stated by the Wiener Khinchin theorem, Eq. 8). The results of this casestudy can be found in Appendix A.

6 CONCLUSIONS

I started this graduation project by a literature research and getting familair with functional programming, especially the with the CλaSH language. I first described a simple ripple-carry adder/subtracter circuit. The gained experience was then used to build a flexible signed multiplier architecture. Flexible in the sense that it works for vectors of any given length (as long as they are both the same size). This multiplier architecture was used (along the adder/subtracter circuit) to create a signed compmlex multiplier. It was shown (in Section 4.3.1) that the *classic* complex multiplication algorithm had a slight advantage on the *Gauss* algorithm at smaller bit vector sizes.

The complex multiplier architecture was incorporated in the MAC circuit alongside a accumulator. The MAC was used as a fundamental building block of the correlator architecture. Which was also created to be as flexible as possible, meaning that the size of the input data will determine the size of the complete correlator architecture. This way the architecture can be easily analyzed for different input sizes and number of lags.

In Chapter 5 the created complex and non-complex correlator architectures were analyzed. The relation between the necessary number of LEs with respect to the input resolution and number of lags were given by Equations 26 and 30. But the number necessary LEs didn't seem to be the biggest bottleneck in the architecture, this was the number of needed I/O pins. The relation between number of needed pins with respect to input resolution and number of lags was presented by Equations 29 and 35. And a relation between input resolution and the number of lags with respect to the maximal frequency at with the correlation operation can be performed, was given by Equations 36 and 36.

7 FUTURE WORKS

As is common with most master theses, the work done during this graduation period is not fully finished and thus leaves room for future work. This chapter holds some suggestions on which further research can be based.

7.1 CASE STUDY: SPURIOUS FREE DYNAMIC RANGE

As is presented in Appendix A, the simulation results of the case study for the SFDR contained some errors, which I was unable to solve within the given time. This leaves the SFDR case study open for future research. The case study should result in definition of the relation between the SFDR with respect to the correlator input resolution and the number of lags used. Which can then be used (by - for instance - developers working on a SDR) to choose the right configuration for the corralator, based on the available LEs and the required speed and sensitivity.

7.2 POWER ANALYSIS

Analyzing the power consumption of the correlator architecture can be categorized in two types of analysis: Passive power consumption and Dynamic power consumption analysis. The first one will give an indication about the power consumption of the architecture when it's idle. This most likely will just be a certain constant value, which is depended on the number of LEs used.

The dynamic power dissipation on the other hand, is the total power that is consumed with every toggle of internal bits. Giving insight in the power consumption on active usage. This is of course dependent on the input data, which needs to be a realistic set. The data used in the Case-study which analyzed the SFDR value for different configurations (Section 5.3), could be used for instance.

After successful synthesizing the architecture, *Quartus II* generates a post-synthesis description of the design in VHDL and a *SFD* file, which holds information on the delays in circuit. This data can then be used in a post-synthesis simulation in *Mod-*

elSim by *Altera*. The inputdata for the case-study needs to be converted to a *ModelSim* compatible data-type, that can be used as input for the post-synthesis simulation. The simulation will result in a Value-Change Dump (VCD) file, which contains all the signal toggles that occurred during simulation. This file can in turn be imported in *Quartus II*, which can process the switching activities for each signal and produce a dynamic power consumption report.

7.3 NON-LINEAR MULTIPLIER

In Section 2.7 some ideas for low resolution optimization where coined. By using a non-standard quantization scheme in combination with the assignment of weighting factors to the possible output states of the quantizer, a non-linear multiplier can be created. This multiplier won't have the characteristics of a normal multiplier and thus will have some peculiar, but still deterministic output. An advantage this multiplier will have over an ordinary one, is that it's much smaller area-wise and probably much faster, but this comes at a cost of the accuracy of the output [19]. Future research can be done by looking what the influence is on the number of LEs used, the maximum frequency of which the correlator operation can be performed, the accuracy (degradation factor) of the output and the power consumption:

- When changing the weighting factors in the quantization scheme.
- When choosing a different boundary value n in the weighting factors ($-n$, -1, $+1$ and $+n$ in a 2-bit quantization scheme).
- When disabling the low level products (Table 4 and 5).
- When emphasizing the higher level products (although the influence on the degradation factor was already covered in [20]).

7.4 HARDWARE FOLDING

There are some drawbacks of the designed correlator architecture, one is that it becomes rather large when increasing the input vector size or the number of lags. As shown in Figure 43, after some point the architecture does not fit any more on the (relatively large) *Cyclone IV* FPGA. Another disadvantage is that synthesizing the design at that point is quite a time consuming task (it takes a relatively large circuit more than a day to synthesize on a modern desktop PC).

When looking at Figure 30, it's easy to see that the correlator architecture consists out of somewhat cascaded components (the components are accentuated by Figure 33). This property can be utilize by implementing hardware folding in the architecture. A schematic view of the idea can be viewed in Figure 48.

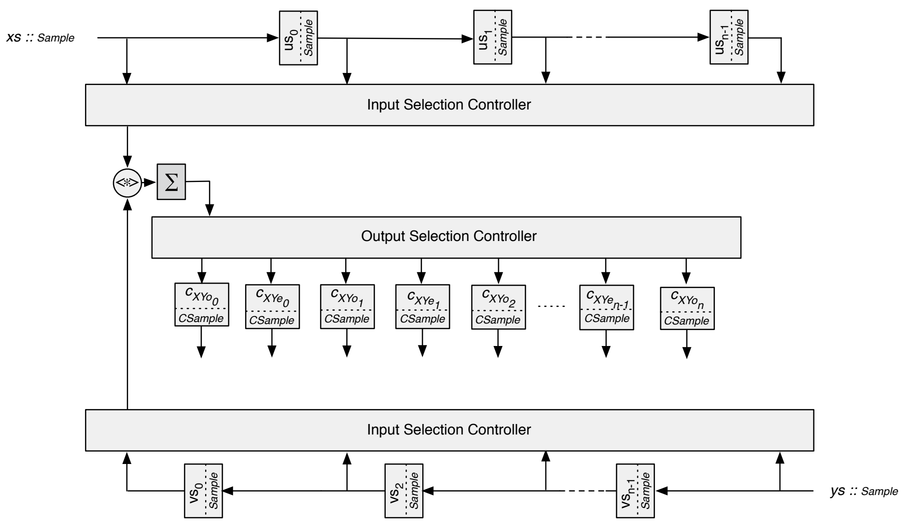


Figure 48: Schematic view of an hardware folded cross-correlator architecture with only one MAC.

As can be seen from the above schematic, folding the correlator structure is not so straightforward as one would first assume. Both inputs of the correlator (*ys* and *xs*) are connected to both ends of the architecture. The input *Samples* will travel trough a certain amount of delays (registers) before reaching the MAC. These delays need to be preserved in the folded architecture, since they are a key element in the correlation operation. To still be able to implement hardware folding, some mechanism is required which every clock-cycle selects the correct two samples from the registerbanks *uss* and *vss* and feeds them to the right MAC (in the above schematic, only one MAC remained after folding) and the answer is put in the correct output register of C_{XY}. This is done by the *Input* and *Output Selection Controllers*.

The main advantage is that the number of LEs used will be cut down, as can be seen by Equation 38 below:

$$\Theta^{\rho}_{Lnc} \approx (2L+1)\cdot(\delta^{uv}_{n} + \sigma_c + \delta^{C_{XY}}_{c}) + \rho \cdot X_n + \gamma \tag{38}$$

Where ρ represents the number of MACs used in the design and γ is the added overhead in LEs due to the *Input* and *Output selection Controllers*. This decrease comes off course with a downside, implementing folding means that the correlation operation now takes several clock-cycles. The maximum frequency the correlation operation can be performed is given by:

$$\Phi^{\rho}_{n} \approx \frac{\Phi_n \cdot \rho}{2L+1} - \lambda \tag{39}$$

Where λ is the latching time delay overhead as result of the folding.

7.5 I/O PINS REDUCTION

In section 5 was described that the bottleneck in fitting the correlator architecture on the FPGA was the number of available pins. It would therefore be wise to implement a circuit in which not every output bit needs it's own pin, but instead sends multiple bits over one pin. One option to realize this can be by using Low-Voltage Differential Signaling (LVDS), which is a high speed and low-power communication system (that is also featured in most of the FPGAs by *Altera*).

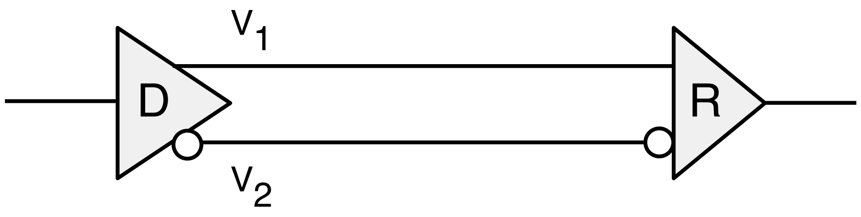


Figure 49: Schematic view an LVDS system.

The binary data at the input of D will be transfered through a buffer to the lines V_1 and V_2. If the input data is a logic 0 than $V_1 = V_2 = V_{cm}$ (a certain common-voltage). When the input data is a logic 1, V_1 becomes larger and V_2 becomes smaller, the input of the receiver R becomes $V_1 - V_2$. The signals are differentially transfered, making the system insensitive of common-noise. The influence of common-noise is that the common-voltage gets shifted, but as these get shifted for V_1 as well for V_2 the differential voltage between them stays the same. Because the system is barely influenced by noise it can perform at low-power, which enables it to perform at high speed.

7.6 FXC

This graduation research concentrated purely on a XFC, but it is also sensible to analyze a FXC. To check how the relations for an FXC hold for a certain input resolution and number of lags, with respect to the necessary LEs, the maximum frequency the correlation operation can be performed, the sensitivity and the power dissipation. And how the FXC holds against the XFC. This should lead to an answer to question when a FXC is an better choice over a XFC or vice versa for certain situations.

7.7 IMPLEMENTING ON ACTUAL FPGA

Everything researched in this thesis is purely theoretical, but to see if all these relations presented in Chapter 5 hold up on an actual FPGA is most likely a wise thing to verify.

7.8 ALGEBRAIC DATA TYPES

In Section 4.4.4 the proceedings to convert a complex correlator architecture to a non-complex one are presented. These are quite small actions, but ideally the top-level entity should pick the right functions for the MAC operation based on the type of the input data (complex or not). This can be realized when implementing an algebraic data type into the architecture on which the complex and non-complex *Sample* are based upon.

7.9 COMPARING DESIGN TO VHDL CORRELATOR

The correlator architecture that was build during this graduation period was written in the relatively new HDL CλaSH. One of the first questions that would arise when CλaSH is presented at the outside world, would be "how does it relates to VHDL?". To give an answer to this question a correlator architecture could be created with VHDL and analyzed and compared to the one written in CλaSH. This will be of course quite some work, but when the results favor for CλaSH, it will be a significant selling point.

7.10 n BY m MULTIPLIER

The Baugh-Wooley multiplier that was created is only able to multiply two vector which both have the same length. But if one input signal could be expressed in lesser bits than the other, it would be more efficient if the multiplier could also operate on two vectors of different length. Note that the analysis on the two complex substitution algorithms which was done in Section 4.3.1 should be reconsidered. Gauss's algorithm is more likely to have a advantage over the standard algorithm, when sign extending one signal isn't necessary anymore.

8 REMARKS ON CλASH

As noted before, CλaSH is an functional HDL created by the CAES research chair at the University of Twente, which was developed only a few years ago in 2009. So this graduation research is still one of the first 'bigger' projects conceived in this relatively new language. First of I like to say that this project was also my introduction to functional programming. During this graduation period I've therefore learned a lot about solving problems in a 'functional' way. Learning to think in new form of abstractions and discovering new ways to represent programs and to think about languages. Which are in my opinion valuable insights which I can benefit from in my future career (even if this only involves solving problems in a imperative languages). I immediately loved the charm of Haskell and its often very clean and elegant solutions to otherwise complex problems. Although my impressions of CλaSH are generally positive, I did had some remarks and suggestions which I have obtained during my experiences working with it. These I would like to present in this Section.

8.1 RECURSION

I started out by learning the basics of Haskell and made some assignment used in the Functional Programming course. I soon realized that for basic problem you almost get forced to use higher-order functions, polymorphism and recursion, which will result in some beautifully compact code, like for instance my first *sum* fuction, which calculates the sum of a list:

```
sum []     = 0
sum (x:xs) = x + sum xs
```

or *reverse*, which reverses a list:

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

But in CλaSH recursion is not available and this is quite a shortcoming, since it's such an important concept of functional programming. The vectors used in CλaSH need to be finite and must have a constant length. Although there are some types of recursion that simple cannot be implemented to hardware, implementing recursion over finite vectors should be realizable, because number of needed hardware elements depends on a known size.

8.2 TYPE SYSTEM

During this thesis a lot of Listings were found which featured type declarations. Like for instance the code of the accumulator:

```
1  (Σ) :: (⟨Bit⟩n, ⟨Bit⟩n) → (⟨Bit⟩n, ⟨Bit⟩n) → (⟨Bit⟩n, ⟨Bit⟩n)
2
3  (Σ) (as, bs) (cs, ds) = (ps, qs)
4    where
5      ps = vtail (as <+> cs)
6      qs = vtail (bs <+> ds)
```

The type declarations (Line 1) are quite easy to understand, the function Σ needs two tuples (each consisting two vectors of type *Bit* and Length *n*) and the result will also be a tuple of the same type. But unfortunately the type declaration in the actual code are a bit polluted from certain type constrains. This is what the actual function looks like:

```
1  (Σ) :: (n ~ Succ (Pred n),
2           IntegerT n,
3           IsPositive n ~ True,
4           Not (IsLT (Compare (Succ n) n)) ~ True) =>
5      ((⟨Bit⟩n, ⟨Bit⟩n) → (⟨Bit⟩n, ⟨Bit⟩n) → (⟨Bit⟩n, ⟨Bit⟩n)
6  (Σ) (as, bs) (cs, ds) = (ps, qs)
7    where
8      ps = vtail (as <+> cs)
9      qs = vtail (bs <+> ds)
```

In this example the type constrains overhead consist of 'only' 4 lines, but I came across much bigger examples (one of themb even expanded over several pages). The constrains here have the following meaning:

Line 1: $n = n + 1 - 1$

Line 2: n is an integer type

Line 3: $n < 0$

Line 5: $!(n + 1 < n)$

As a programmer you don't want to be bothered with such trivial overhead to your functions, certainly not in a language that is as beautifully compact as Haskell. These type constrains are of course necessary (you may not define a list of length -1) but belong within the type declaration of the vector type, not at function level.

8.3 VCOPYN

To keep the listings with code clear and simple, I presented the *vcopyn* function as following:

```
t = vcopyn Dn x
```

Which means that the value x is copied Dn times to create the vector t (resulting in: $<x, x, ..., x>$). But in reality the function would have to be declared as:

```
type N = Vector Dn a

t = vcopyn (undefined :: N) x
```

The function *vcopyn* needs a vector of a certain length. So we first need to declare a new type N, which is a Vector of length Dn of an arbitrary type a (which could basically be anything). The line "undefined :: N" creates a new bottom value of type N (A vector of length Dn with undefined content of type a). The length of this undefined vector will be used as the number of times x will be copied. This comes - from a programmer's point of view - across as unnecessary and impractical.

LIST OF SYMBOLS

$\star$	cross-correlation operator
$\ast$	convolution operator
$\gamma_{xx}[j]$	Discrete autocorrelation function
$\gamma_{xy}[j]$	Discrete cross-correlation function
$\Gamma_{xx}[f]$	Power spectral density function
$\Gamma_{xy}[f]$	Cross power spectral density function
$\mathcal{F}$	Fourier transform
$\mathcal{F}^{-1}$	Inverse Fourier transform
d	Degradation factor
$-V_0$, $+V_0$	Transition levels in quantization scheme
ω	Weighting factors in quantization scheme
A	Multiplicand
B	Multiplier
P	Product ($A \cdot B$)
L	Number of lags used in correlator
n	Input resolution size of correlator
c	Output resolution size of correlator
Θ_{Lnc}	Size of the complex correlator architecture
$\Theta_{Lnc_{\mathbf{R}}}$	Size of the non-complex correlator architecture
δ_n^{us}	Size of one complex input register *us* (which is an element of *uss*)
$\delta_{n_{\mathbf{R}}}^{\text{us}}$	Size of one non-complex input register *us* (which is an element of *uss*)
δ_n^{vs}	Size of one complex input register *vs* (which is an element of *vss*)
$\delta_{n_{\mathbf{R}}}^{\text{vs}}$	Size of one non-complex input register *us* (which is an element of *uss*)
$\delta_c^{\text{C}_{\text{XY}}}$	Size of one complex output register
$\delta_{c_{\mathbf{R}}}^{\text{C}_{\text{XY}}}$	Size of one non-complex output register
X_n	Size of the complex multiplier
χ_n	Size of the multiplier (which is a component of the complex multiplier)

τ_n^+	Size of the adder (which is a component of the complex multiplier)
τ_n^-	Size of the subtracter (which is a component of the complex multiplier)
σ_c	Size of the complex accumulator
$\sigma_{c_\mathbf{R}}$	Size of the non-complex accumulator
P_{Lnc}	Number of I/O pins necessary for the complex correlator architectrue
$P_{Lnc_\mathbf{R}}$	Number of I/O pins necessary for the non-complex correlator architectrue
r	Represents the required number of non-data pins (power-supply, ground, clocks)
Φ_n	The maximum clock rate possible at which the complex correlator operation can be perfomed
$\Phi_{n_\mathbf{R}}$	The maximum clock rate possible at which the non-complex correlator operation can be perfomed
$f_{\max}$	The maximum achievable frequency of the FPGA

A

APPENDIX A: CASE STUDY SFDR RESULTS

In Section 5.3 the steps for the SFDR casestudy were described. In this Appendix the results of this casestudy will be presented. Beginning with an example of a graph of the spectral density function for a correlator with an input resolution of 32 bits and 64 lags. This can be seen below in Figure 50.

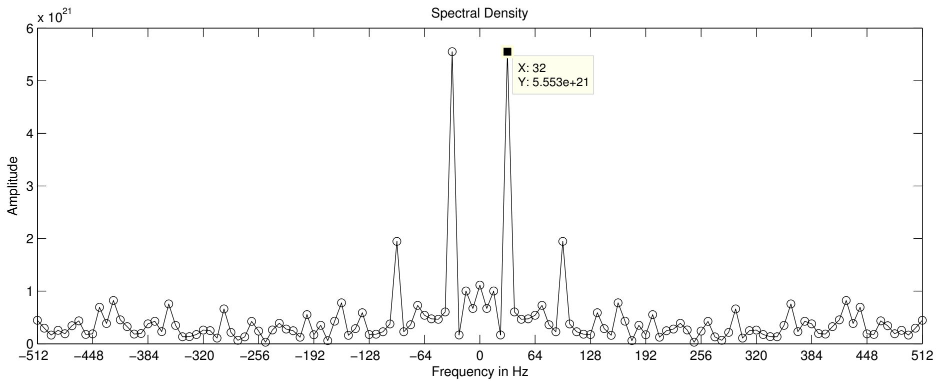


Figure 50: Example spectral density graph of a correlator with input resolution of 32-bits and 64 lags.

The fundamental peak at 32 Hz can be clearly seen, also the spur at 96 Hz, resulting in a SFDR of: $10 \cdot^{10} \log(\frac{fnd}{spur}) = 4.55$ dB. Also the effect of the number of lags is clearly visible. 64 lags means for the correlator architecture that it has $2 \cdot lags + 1 = 129$ MACs and thus also 129 output registers. Since the FFT operation doubles this amount, we can see 158 vertices (dots) in the graph, which means that increasing the number of lags, the accuracy will improve. So if the number of lags is as small as 2, the only information that can be fetched from the spectral density function is, that a 'certain peak' is found (Figure 51).

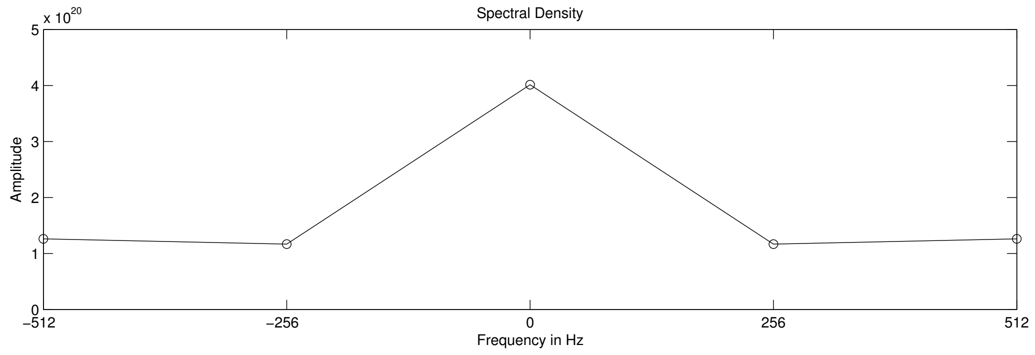


Figure 51: Spectral density graph of a correlator with input resolution of 32-bits and 2 lags.

In Figure 52 the impact of increasing the number of lags for different input vector sizes can be seen.

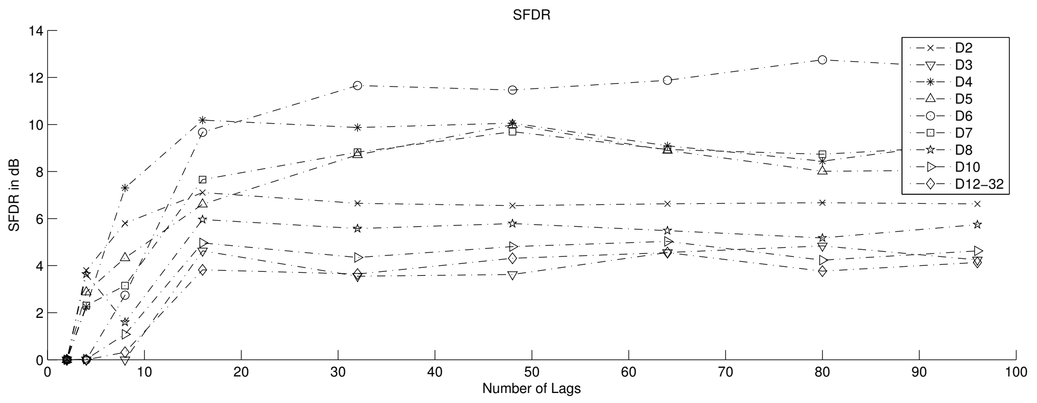


Figure 52: SFDR Results for different input vectors and lags.

When looking at the above graph, we can observe different things. First, up to 16 lags, a great improvement in the SFDR can be seen, but after that the lines become quite straight. So, increasing the number of lags, does make the data more reliable - in the sense of estimating the place in the frequency band where peaks occur. Another observation is that the input resolution of *D*4 up to *D*7 seem to have the biggest SFDR. Where the SFDR of larger resolutions (*D*1 till *D*32) won't be larger than around 4dB. Which is quite surprising, since it contradicts with *Equation A.5* that Mark Oude Alink presented in [37]:

$$\mathrm{SFDR}_{\mathrm{ADC}} \approx 8.07b + 3.29[\mathrm{dB}] \tag{40}$$

Where b represent the number of bits used. This equation states that the SFDR for an input resolution of 32 bits should be around 281.69 dB. Which indicates that its highly likely that there exist (at this point unclear) errors in the simulation. Unfortunately due limitation in available time, I'm unable to correct the simulation and run it again (running the simulation alone, would take several days). So with some pain in my heart I'm forced to push this section to the future works (Chapter 7).

BIBLIOGRAPHY

[1] S.S. Haykin. *Digital communications*, volume 5. Wiley, 1988.

[2] V. Valenta, R. Marsalek, G. Baudoin, M. Villegas, M. Suarez, and F. Robert. Survey on spectrum utilization in europe: Measurements, analyses and observations. In *Cognitive Radio Oriented Wireless Networks & Communications (CROWNCOM), 2010 Proceedings of the Fifth International Conference on*, pages 1–5. IEEE, 2010.

[3] M. Sherman, A.N. Mody, R. Martinez, C. Rodriguez, and R. Reddy. IEEE standards supporting cognitive radio and networks, dynamic spectrum access, and coexistence. *Communications Magazine, IEEE*, 46(7):72–79, 2008.

[4] M.S. Oude Alink. A crosscorrelation cmos spectrum analyzer with improved sfdr for cognitive radio. 2010.

[5] J. Mitola III. *Cognitive Radio*. PhD thesis, Royal Institute of Technology (KTH), Sweden, May 2000.

[6] J. Mitola III. *Software radio architecture*. Wiley Online Library, 2000.

[7] J. Mitola III and G.Q. Maguire Jr. Cognitive radio: making software radios more personal. *Personal Communications, IEEE*, 6(4):13–18, 1999.

[8] J. Mitola III. *Cognitive radio: Model-based competence for software radios*. PhD thesis, KTH, 1999.

[9] J. Mitola III. Cognitive radio for flexible mobile multimedia communications. In *Mobile Multimedia Communications, 1999.(MoMuC'99) 1999 IEEE International Workshop on*, pages 3–10. Ieee, 1999.

[10] I.F. Akyildiz, W.Y. Lee, M.C. Vuran, and S. Mohanty. Next generation/dynamic spectrum access/cognitive radio wireless networks: a survey. *Computer Networks*, 50(13):2127–2159, 2006.

[11] S.S. Haykin, D.J. Thomson, and J.H. Reed. Spectrum sensing for cognitive radio. *Proceedings of the IEEE*, 97(5):849–877, 2009.

[12] M.S. Oude Alink, E.A.M. Klumperink, M.C.M Soer, A.B.J Kokkeler, and B. Nauta. A cmos-compatible spectrum analyzer for cognitive radio exploiting crosscorrelation to improve linearity and noise performance. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 59(3):479–492, 2011.

[13] J.A. Westphal. Some astronomical applications of cross-correlation techniques. *The Astrophysical Journal*, 142:1661, 1965.

[14] J.G. Proakis, C.M. Rader, F. Ling, C.L. Nikias, M. Moonen, and I.K. Proudler. *Algorithms for statistical signal processing*. Prentice Hall, 2002.

[15] J. Bunton. Ska correlator advances. *Experimental Astronomy*, 17:251–259, 2004. 10.1007/s10686-005-5661-5.

[16] M.S. Oude Alink, E.A.M. Klumperink, M.C.M Soer, A.B.J Kokkeler, and B. Nauta. A 50Mhz-To-1.5Ghz Cross-Correlation CMOS Spectrum Analyzer for Cognitive Radio with 89dB SFDR in 1Mhz RBW. *New Frontiers in Dynamic Spectrum, 2010 IEEE Symposium on*, pages 1–6, January 2010.

[17] A. Bos. High Speed 2-Bit Correlator Chip for Radio Astronomy. *Instrumentation and Measurement, IEEE Transactions on*, 40:591–595, June 1991.

[18] FK Bowers and R. Klingler. Quantization noise of correlation spectrometers. *Astronomy and Astrophysics Supplement Series*, 15:373, 1974.

[19] B.F.C. Cooper. Correlators with two-bit quantization. *Australian Journal of Physics*, 23:521, 1970.

[20] B.G. Clark. Internal mem. *National Radio Astronomy Observatory, Charlottesville, Virginia, U.S.A.*, 1966.

[21] V. Standard. Language reference manual. *IEEE Std*, pages 1076–1987, 1988.

[22] Accellera's Extensions to Verilog. Systemverilog 3.1a language reference manual. *IEEE Std*, 2004.

[23] S.T. Taft. *Ada 2005 reference manual: language and standard libraries: international standard ISO/IEC 8652/1995 (E) with technical corrigendum 1 and amendment 1*, volume 4348. Springer Verlag, 2006.

[24] D.M. Ritchie, S.C. Johnson, M.E. Lesk, and B.W. Kernighan. The c programming language. *Bell Sys. Tech. J*, 57:1991–2019, 1978.

[25] B. Fuchs. Verilog hdl vs. vhdl: For the first time user. 1995.

[26] IEEE Design Automation Standards Committee et al. Std 1076–2008, ieee standard vhdl language reference manual. *IEEE, New York, NY, USA*, 2008.

[27] C.P.R. Baaij. CλasH: From Haskell To Hardware. Master's thesis, University of Twente, December 2009.

[28] C.P.R. Baaij, M. Kooijman, J. Kuper, M.E.T. Gerards, and E. Molenkamp. Tool demonstration: Clash-from haskell to hardware. 2009.

[29] C.P.R. Baaij, R. Wester, A. Niedermeier, A. Boeijink, M. Gerards, J. Kuper, et al. Functional hardware design in cλash. 2010.

[30] J Kuper. Functional specifications of hardware architectures. november 2012.

[31] S.P. Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, et al. Haskell 98: a non-strict, purely functional language, 1999. *URL: http://www. haskell. org/definition*.

[32] G.H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[33] B. Parhami. *Computer arithmetic*. Oxford university press, 2000.

[34] C.R. Baugh and B.A. Wooley. A two's complement parallel array multiplication algorithm. *IEEE Transactions on Computers*, 22(12):1045–1047, 1973.

[35] Donald E Knuth and TheArtof ComputerProgramming. Volume 2: Seminumerical algorithms. *The Art of Computer Programming*, page 519, 1997.

[36] Altera Corporation. Cyclone iv device handbook: Logic elements and logic array blocks in cyclone iv devices. pages 1–8, november 2009.

[37] M.S. Oude Alink. *RF spectrum sensing in CMOS exploiting crosscorrelation*. PhD thesis, University of Twente, The Netherlands, 2013.