Politecnico di Torino

Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Elettronica
A.a. 2021/2022
Sessione di Laurea Dicembre 2022

Implementation of an Approximated FIR Filter on FPGA for Laser Line Extraction from Pixel Data

Relatore:
Prof. Guido Masera

Candidato:
Lorenzo Poloni

Advisor:
Dott. Arne Kreddig

Al Gatto

Abstract

Approximate computing is an emerging design paradigm that trades in accuracy for resource consumption. Due to the limits of human perception, a certain amount of error is tolerated in most applications that involve the human senses. These include, for instance, image and audio processing.
In this master thesis, approximate computing techniques are employed on an image processing pipeline embedded in a 3D scanner that extracts laser line data from a camera sensor with the goal of computing the shape of the object being scanned. To do this, the position of the laser line must be determined. In the pipeline, the data stream is manipulated by FIR filters that first smooth it to reduce noise levels and then transform the light peak generated by the laser (point of maximum intensity) into a zero by performing a derivative on the smoothed data. The location of the zero is estimated by a special unit. Complex circuits such as arithmetic units (e.g. adders, multipliers and dividers) are good candidates for approximation, as they have a huge impact on the total area and power consumption. Given the complexity of these blocks, their structure is hierarchical and often modular. This, in turn, means that the error can be manipulated at a finer scale by acting on part of the smaller components that make up the circuit. Depending on the kind of approximation one chooses, lower levels (e.g. transistor, logic gates), as well as higher ones (e.g. RTL), can be subject to modifications. The whole design is implemented on FPGA and parameters such as power, area and several types of errors are evaluated. The approximate hardware results are then compared with a software implementation and, subsequently, an exact hardware design. The design space is further explored by acting on parallelism and the filters' architectures. A set of images taken by a 3D scanner's camera are used as test vectors.

Contents

List of Figures

List of Acronyms

FSM finite state machine

SG Savitzky-Golay

ALU arithmetic logic unit

FA full adder

HA half adder

CSEA carry select adder

IC integrated circuit

ER error rate

ED error distance

RED relative error distance

MED mean error distance

NMED normalized mean error distance

MRED mean relative error distance

NED normalized error distance

MSE mean squared error

RMSE root-mean-square error

PDP power-delay product

ADP area-delay product

EDP energy-delay product

LPAA low-power approximate adders

PSNR peak signal-to-noise ratio

KZ Kolmogorov-Zurbenko

MA moving average

LOA lower-part-OR adder

LPL lower-part length

RoBA rounding-based approximate

ETAI error-tolerant adder type I

ETAII error-tolerant adder type II

NT negative threshold

PT positive threshold

LPT low positive threshold

HPT high positive threshold

AM-ER approximate multiplier with configurable error recovery

IPP input pre-processing

WL word-length

VR vertical replacement

HR horizontal replacement

SR square replacement

TR triangle replacement

EXDCr exact restoring divider cell

AXDCr approximate restoring divider cell

AXS approximate subtractor

RCA ripple-carry adder

BKA Brent-Kung adder

KSA Kogge-Stone adder

CSA carry-save adder

CLA carry-lookahead adder

WTA Wallace tree adder

WTM Wallace tree multiplier

BWM Baugh-Wooley multiplier

ALUT adaptive look-up table

ALM adaptive logic module

LAB logic array block

LE logic element

SDC synopsys design constraints

DUT device under test

DF-NS differentiation filter with no smoothing before

DF-2ND second order differentiation filter

DF-S differentiation filter after smoothing

ZC-NA zero-crossing with no average

ZC-A zero-crossing with average

VCD value change dump

CHAPTER 1

Introduction

The purpose of this thesis is to apply approximate computing techniques to a well-established image processing system with the intention of assessing its error tolerance as well as the extent to which resources can be reduced. The assumption is based on the fact that, due to limitations in human perception, a certain amount of inaccuracy can be accepted [1]. A 3D scanner based on triangulation employs a laser and a camera to obtain the 3D model of an object. As shown in Figure 1.1, the laser projects a line on a section of the object and the camera captures this interaction. The system designed in the following chapters has to analyze grayscale images coming from the camera [2]. An example is provided in Figure 1.2. In the pictures the laser line is clearly visible and far brighter than its surroundings. Ideally, everything else except for the laser line region should be dark. The pictures, each with different resolutions, are modeled as a matrix where each element represents a pixel whose value varies from 0 to 255. The matrix is evaluated column by column.

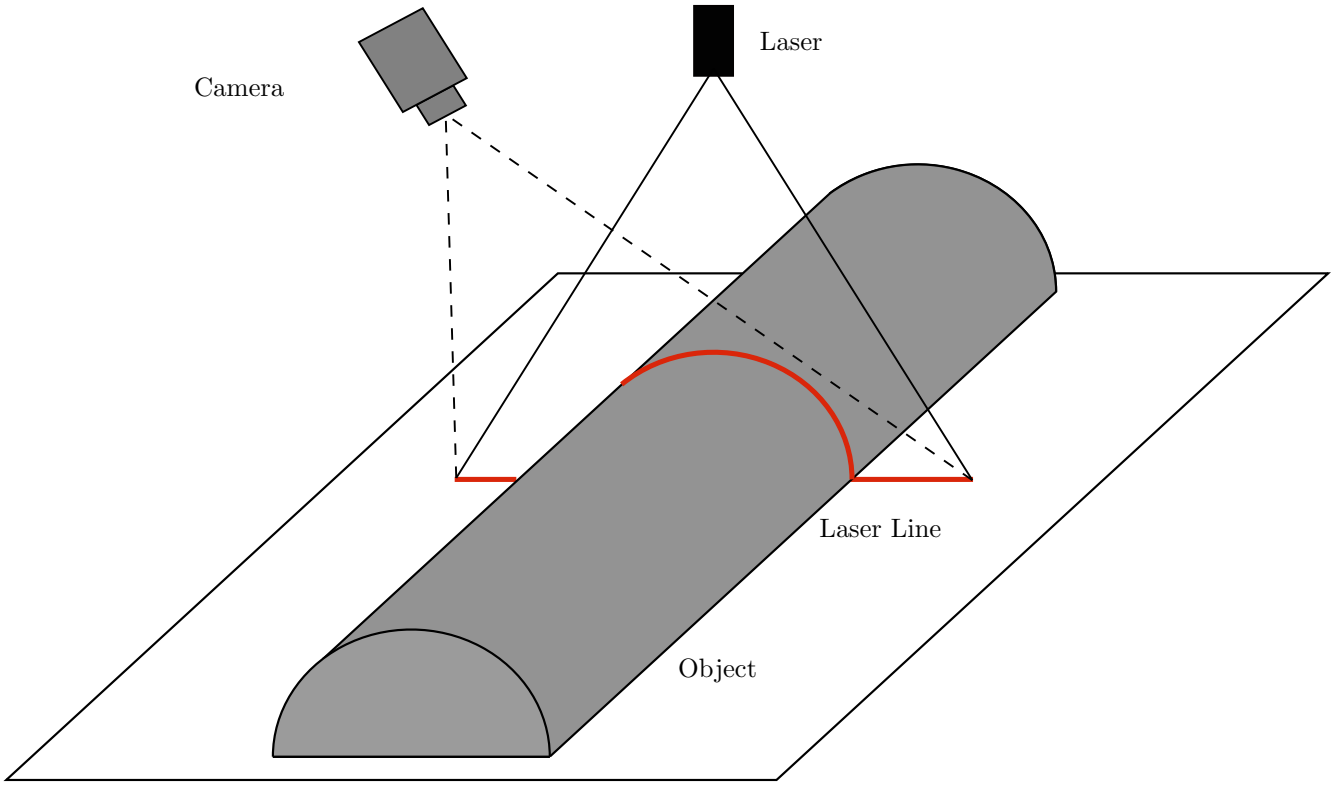


Figure 1.1: Example of a laser line used to scan an object

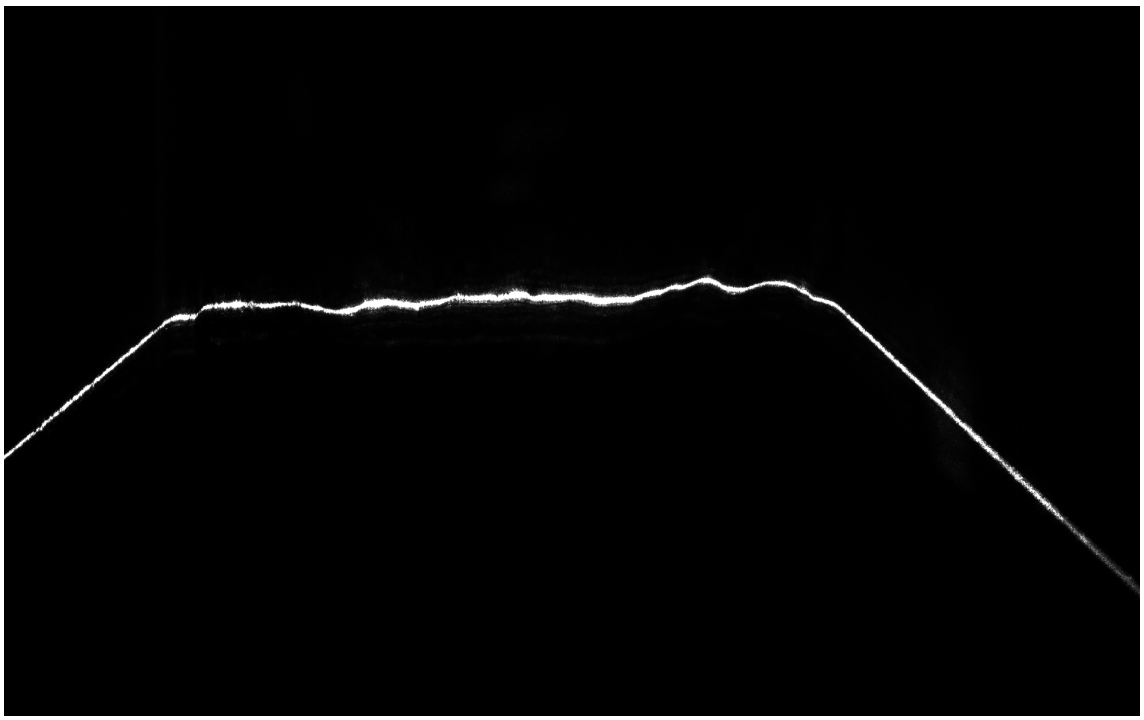


Figure 1.2: Grayscale image of a weld

By shifting the focus to one column where the laser line is present and plotting it on a graph, the reader can notice that it assumes a shape similar to a Gaussian distribution, as shown in Figure 1.3. The purpose of the image processing pipeline is to find the position of the laser line, which means finding the maximum of the Gaussian distribution for each column. To accomplish this task, one needs to define algorithms that are able to identify the position of the laser line. The system that is being designed is thus within the computer vision scope [3].

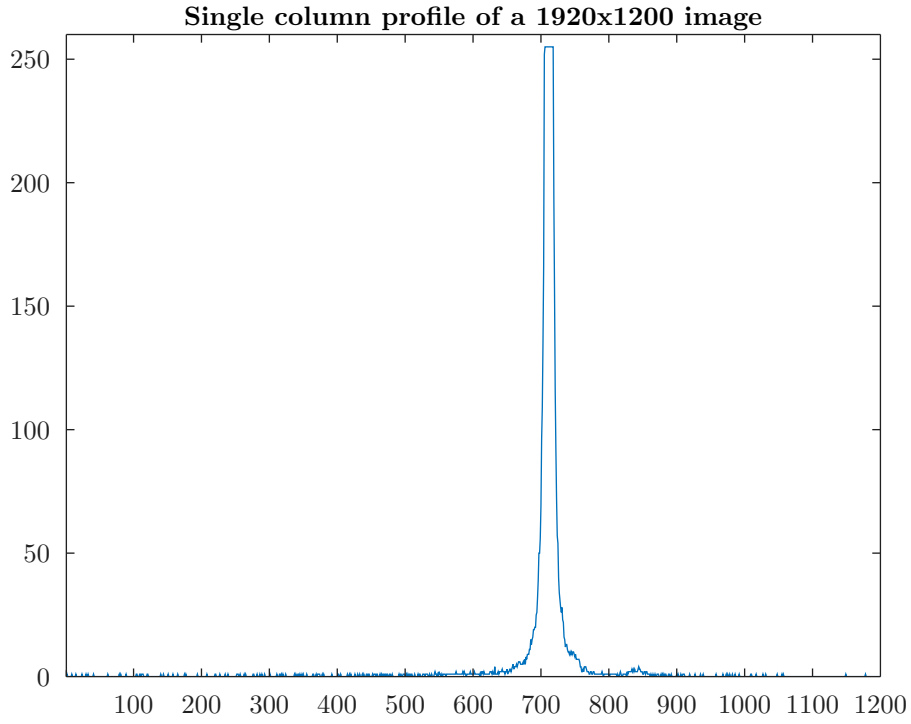


Figure 1.3: The reflection of the laser on the object creates a Gaussian distribution for each column that contains a fragment of the laser line

The algorithm finds the position of the maximum intensity by detecting the zero of a function equal to the derivative of the input Gaussian [4]. As a consequence, two main blocks can be identified in the system:

- Differentiation filter
- Zero-crossing detector

A smoothing filter can also be inserted at the beginning of the chain, depending on the noise levels. It will be shown later that a differentiation filter implemented using SG coefficients is also able to smooth. The extent of this effect depends on the two primary parameters that play a major role in the design: the order (of the fitting polynomial) and the frame length (i.e. the order of the filter). Their importance lies in the fact that they both influence the frequency response of the filter and, thus, the way the input is processed. Both filters are implemented as type I FIR filters. The smoothing filter acts as a prefiltering stage. Its main purpose is to reduce the noise without introducing a distortion in the original signal. As the images received are not ideal, there will always be a certain amount of noise. Most of the time, however, this does not raise any problem. Consequently, depending on the disturbance level, smoothing might or might not be necessary. The effect of this filter on the data, and whether it is essential or not, will be assessed in the coming chapters.
The second block is the differentiation filter. As the name suggests, it applies the differential operator to the input data in order to turn the maximum of the Gaussian into a zero. An example of a typical profile found at the output of this filter is shown in Figure 1.4

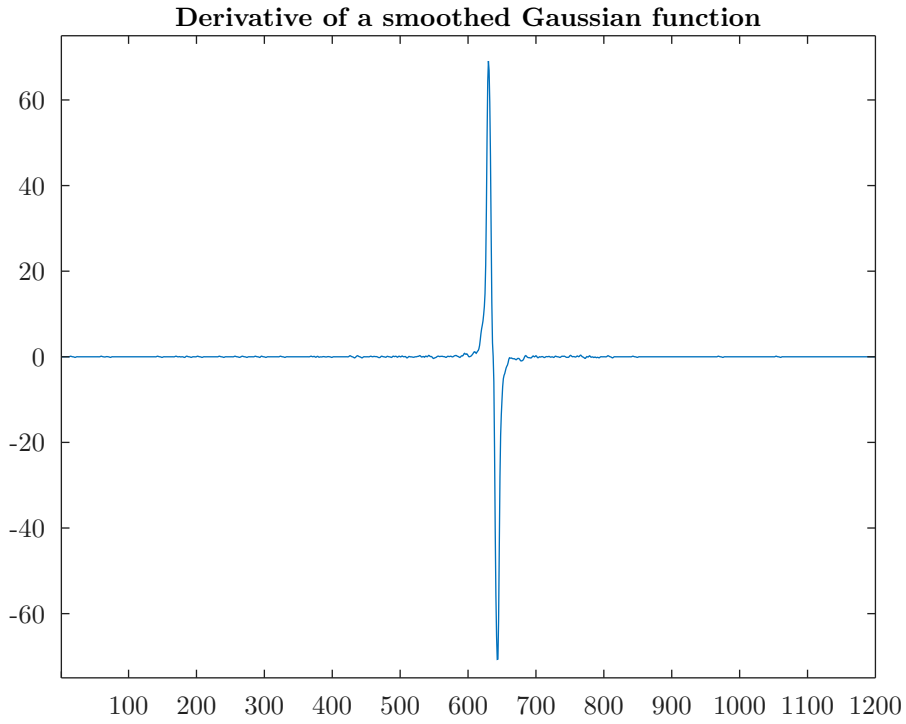


Figure 1.4: Derivative of a smoothed Gaussian distribution

The last block is the most complex one as it implements the core of the algorithm. It features, among other things, a divider, adders, counters, and several comparators. These blocks are driven by a finite state machine (FSM). The zero-crossing detector receives the differentiated signal and tries to find the zero, if any, that corresponds to the laser line position. Ideally, when dealing with a perfect Gaussian, only one zero is expected. In this case the problem greatly simplifies. However, due to noise and other factors, such as the laser line width, multiple zeros can be detected. Finding the right one (i.e. the absolute maximum of the Gaussian) is a complex task. In most cases, the detector will not be able to decide which zero is the actual absolute maximum and will then return more than one position for each column. This issue can be solved with a spline-fitting algorithm. However, as this is a software algorithm, it is not discussed further in this thesis.

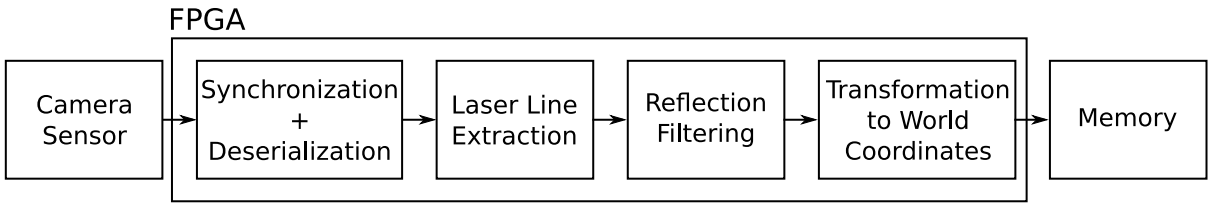


Figure 1.5: 3D scanner processing pipeline

Finally, Figure 1.5 shows the complete pipeline able to process data coming from the camera in order to create a final model of the object scanned. The units described up until this point are contained in the block named "Laser Line Extraction". Reflection filtering and transformation to world coordinates are not part of this project.
Given the presence of a divider, the zero-crossing block might be the slowest and most power-hungry block of the designed system. As the filters contain a lot of registers, the maximum frequency is expected to be higher than that of a divider. However, since these blocks make use of a lot of multipliers, power consumption might add up to those of the zero-crossing circuit. For this reason, efforts to reduce power dissipation are made.

In the following chapters, the full design is examined step by step. First of all, the algorithms are presented. In the initial part, the software implementation of the accurate system is explained. After that, the discussion focuses on the approximate algorithms, chosen from literature, that are applied to the arithmetic units. In the subsequent chapter, the hardware design for the entire system is explored. Exact architectures are introduced and then optimized by exploiting the system's characteristics. The same is done for approximate circuits. Main blocks such as filters and the zero-crossing detection circuit are also optimized. An entire chapter is dedicated to presentation and discussion of the results. Tables and graphs summarize the findings obtained from both simulation and synthesis. Approximate circuits are compared to the exact designs in terms of delay, power, FPGA resources, and error metrics. The last chapter contains a summary of the entire project as well as an insight into what more can be done.

CHAPTER 2

Theoretical framework

In this chapter, some relevant theoretical concepts are introduced. The reader will find most of the concepts covered essential in order to understand the rest of the document. Additional knowledge is disclosed for the sake of completeness.
In the first section, 3D scanners are presented, in particular, their working principle and important properties. As the system employs digital filters, the second section deals with the mathematical model behind SG filters and their main properties. Finally, in the last section, the approximate computing paradigm is introduced.

2.1 3D scanners

3D scanners are devices able to extract certain features, such as shape, from an object, in order to build a 3D model of it. While some limitations on the variety of objects that can be scanned exist, the analysis performed by such instruments does not damage the object. Depending on the application in which the scanner is used, several technologies able to obtain a 3D model have been conceived [5, 6]. The scanner we are going to work with is based on the principle of triangulation [7, 8], which falls within the realm of computer vision [9]. Triangulation is a non-contact active technique as the scanning instrument does not come into contact with the object under test directly, but, rather, blasts a light source (typically a laser) at it, hence active [10]. A camera is placed at a known distance from the laser. Both elements aim at the target. The laser shoots light downwards, in a straight line perpendicular to the nominal plane. This light is then captured by the camera, which is positioned in a way so that it forms a known angle with the laser. The angle, whose vertex is the nominal surface on which the object is placed, acts as a fixed offset. This kind of configuration is called standard geometry [11]. As can be seen from Figure 2.1, the complete set-up looks like a triangle, thus the name triangulation. The camera sensor is normally implemented using CMOS or CCD technologies. Moreover, lenses are present at the output of the light source and at the input of the camera to focus the rays. Applications for this kind of scanner include: position sensing, dynamic measurements as well as thickness and dimensional measurements [12].
The four main geometries employed in a laser line system are: standard, reverse, specular and look-away [11]. Each of them comes with its own advantages and disadvantages that make it suitable for certain applications rather than others. The standard configuration is a general purpose geometry whose main perk is the simplicity of calculations required to determine the shape of the object. This is due to the fact that a change in height of the object along the line produces a variation only in one coordinate of the laser line on the nominal plane, thus yielding a simpler mathematical model. The drawback with this geometry is that it needs an accurate calibration before its use. Another downside is the risk of occlusion. There is a trade-off between the maximum height resolution and the chance of occlusion. Occlusion occurs because the camera is angled in relation to the laser. Since the sensor is not perpendicular to the object, depending on the target, there will be some parts that will not be visible and, thus, occluded. This chance increases by increasing the nominal angle, which also enlarges the height resolution nonetheless.
Figure 2.2 illustrates the relative position between the camera and the laser in different geometries.

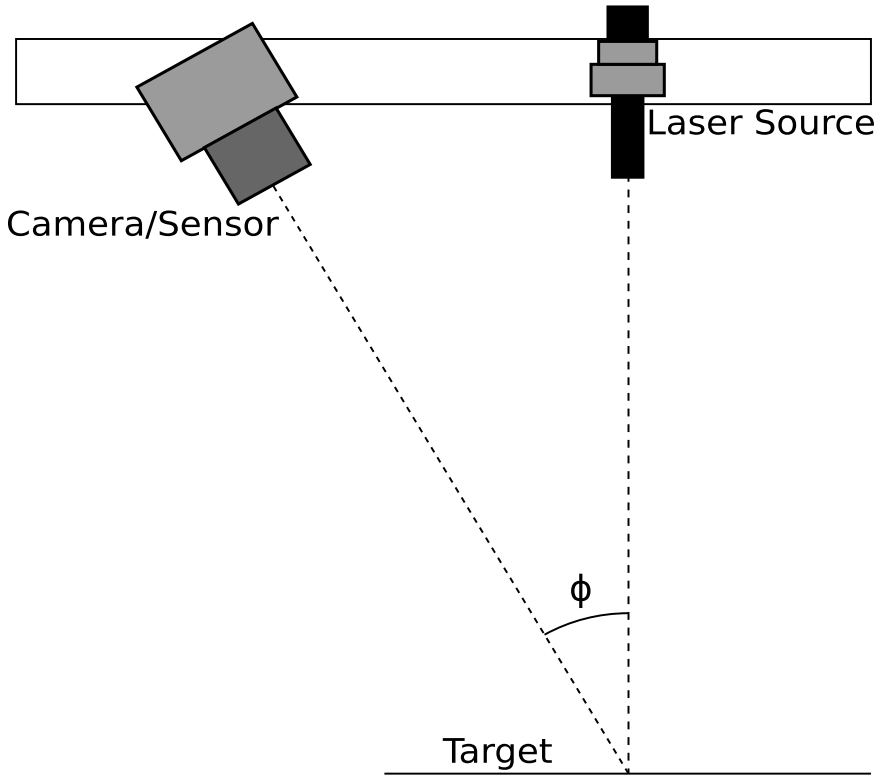


Figure 2.1: 3D Scanner set-up with nominal angle ϕ, standard geometry

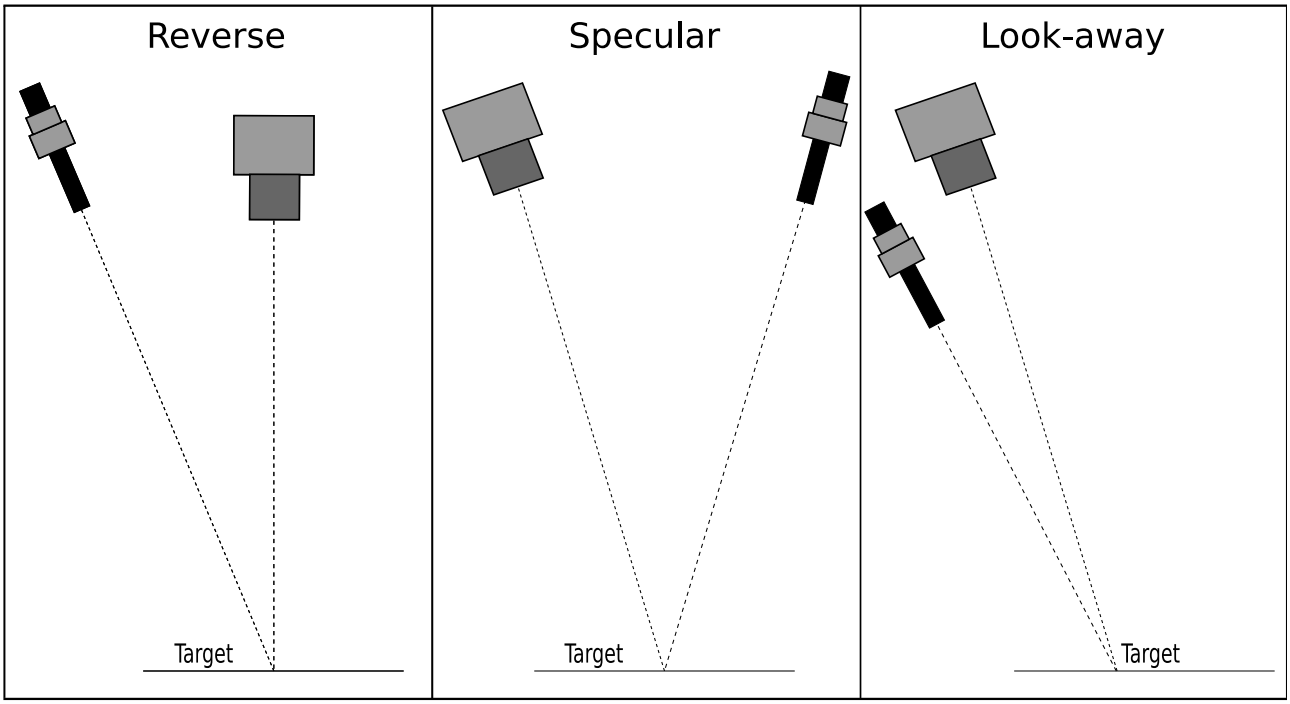


Figure 2.2: The three main geometries besides the standard one

Reverse geometry is obtained by switching the positions of the laser and the camera from standard geometry. This time the camera is perpendicular to the object and no occlusion takes place. Moreover, due to the angled laser illumination, the height resolution is increased. Computation here, however, gets more complex. The increased resolution and computational costs over standard geometry make this configuration suitable for high-accuracy measurements of planar objects.

In specular geometry, neither the camera nor the laser are perpendicular to the target. In fact, they are at similar oblique angles. In this configuration, height resolution is even greater than in the reverse case but this time the camera can sense unwanted specular reflections from the laser that might saturate the detector. Dark objects can benefit from this effect, which makes this geometry particularly suitable for these kind of target. Occlusion is also a major problem.
The last configuration is obtained by placing both the camera and the laser on the same side. This is called look-away geometry. As opposed to the previous geometry, the specular reflection from the laser is dramatically reduced. For this reason, it is primarily used for highly reflective objects. Drawbacks include a lower height resolution and a small chance of occlusion.
In standard geometry, knowing just two variables is enough to estimate the distance between the camera sensor and the item being scanned. When light hits a target that is higher or lower than the stand-off distance (i.e. the nominal distance between the laser and the surface below), the reflection will reach the camera sensor with an angle that can be bigger or smaller than the nominal one. As a consequence, the deflected ray will strike the sensor surface at a point that depends on the height of the object itself [2]. From this distortion it is possible to compute the dimensions of the target. Figure 2.3 shows the working principle of the acquisition system.

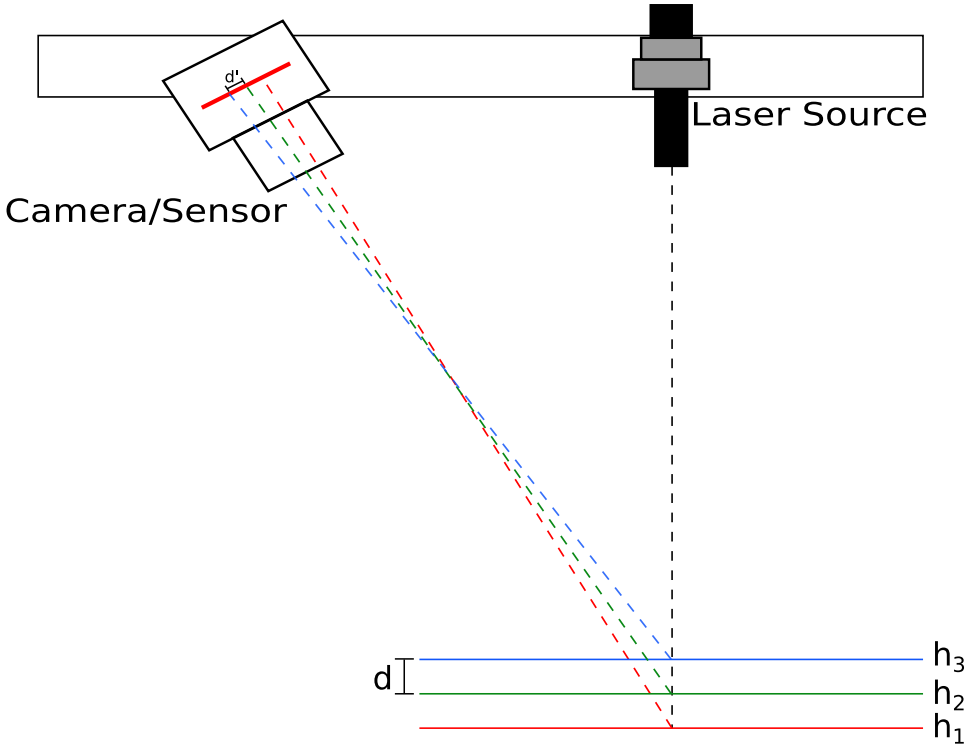


Figure 2.3: Scanning objects at different heights and its effects on the camera sensor. The nominal height is h_2

There is another trade-off to be considered between the maximum measurement range and precision. Given a certain maximum vertical height, we normally want to condition the sensor in order to work at full range. This is done to better exploit its positional sensitivity. Depending on the pixel resolution and the physical size of the chip, the less the vertical variability of the targets, the greater the shift in position on the surface of the sensor. If there is a need to measure objects that come in a variety of sizes, a wider vertical range is important. The risk in this case is that the sensor will not be able to tell apart levels that are not so close to each other. Hence, it experiences a loss in precision [2]. Part of this loss can be compensated using *Subpixeling*.
In image processing and computer vision, subpixeling is part of the so called *Super Resolution Techniques* that aim at increasing an image's quality at a software level using DSP algorithms. Attempts to improve resolution at a hardware level are often much more expensive and ineffective as they involve decreasing the pixel's size, which is limited by technology, or increase the camera sensor's chip area, which increases the overall capacitance [13]. By proceeding instead on an algorithmic level, digital images that would normally be limited to the pixel resolution, can actually reveal details on a finer scale. To obtain this, the most common methods make use of interpolation, fitting or moment-based algorithms [14]. The final result is an image whose resolution is higher than before. Several approaches to improve the image quality by means of subpixeling have been proposed and

are available in literature [13]. The applications range from edge detection [15, 16, 17], change detection [18] and rendering [19] to stereo, satellite and medical imaging [20, 21, 22]. In this thesis, subpixeling is used to compute the position of the zero more accurately. As the derivative of the input data is computed by a digital filter, its values are quantized. This means that when reading the output data, none of the samples will have a value equal to zero but rather the zero is located by detecting two consecutive numbers with opposite signs. The algorithm, that will be explained in detail in Chapter 3, processes these two numbers and performs a division. This operation is carried out on a larger number of bits so that the position of the zero is returned with a higher precision.
In the simplest kind of triangulation the laser scans the object point by point. This technique can be time-consuming. In this thesis, the laser source projects a line as opposed to a single dot. At each acquisition, either the sensor or the object are automatically shifted or rotated in order to achieve a thorough scan. The data structure storing the data from the sensor is chosen depending on the type of object that is being acquired. A $2\frac{1}{2}$-D technique allows us to capture only the surface of a 3D object. Much like a scalar field, considering a classic XYZ three-dimensional space, for each set of coordinates on the XY plane, there exists only one value for the Z component (i.e. a function $f : \mathbb{R}^2 \to \mathbb{R}$). In order to take into account the thickness depth of a target, a 3D approach is needed. In this case more data is collected as there might be multiple points with the same (X,Y) coordinates but with different Z [23]. As a consequence, the data structure is more complex.

2.1.1 Laser line parameters

Three main parameters can be identified:

- Laser line power
- Laser line width
- Line straightness

These properties actually affect the noise present in the final image by introducing measurement errors. It is thus important to keep them under control.
Real laser lines can display some imperfections. Projectors create a beam which is neither homogeneous nor perfectly straight. Moreover, distance from the projector seems to affect this property. These anomalies are mostly caused by manufacturing variations and tolerance effects [11]. Another cause that produces intensity fluctuations is the Speckle effect. Since the laser ray expands while travelling towards the object, the waves follow different paths and reach the target with different phase. The interference of these waves results in a random intensity modulation along the line [4]. It is thus essential to be aware of the power uniformity of the line beam.
Moreover, achieving a narrow line width is fundamental to increase resolution and power density. The laser line width is mainly affected by the focusing of the laser beam through the lens. This happens because the beam is not perfectly circular but rather elliptical. Depending on the orientation of the laser, the line width will be defined by one of the two axes. The choice of the best axis is not straightforward and is determined by several factors such as diffraction caused by lenses.
Finally, to ensure an optimal line straightness, a proper calibration should be performed. It is worth noting that even a calibrated laser line will not be perfectly straight but rather S-shaped [11].

2.2 Savitzky-Golay filters

In the search for a way to pinpoint the position of maximum intensity, Savitzky-Golay (SG) filters represent an extremely solid solution. Therefore, the pipeline described in this thesis features at least one of them. These filters are able to smooth the data and compute the first derivative in one or two steps by choosing the appropriate coefficients. In this section, the properties displayed by SG filters and their perks are discussed.
To understand what a smoothing filter does and the mathematics behind a SG filter, we first need to define what smoothing is. In general, smoothing a data set is the same as shaping an approximating function able to highlight significant patterns in the data, while removing noise. To find this function we need to devise the right algorithm. Given a noisy signal, the algorithm should be able to preserve the information carried by the original signal, while filtering noise out. For this to work, superimposed noise should be out of the bandwidth of our signal, or at least for the most part. Figure 2.4 shows an example of a smoothing operation.

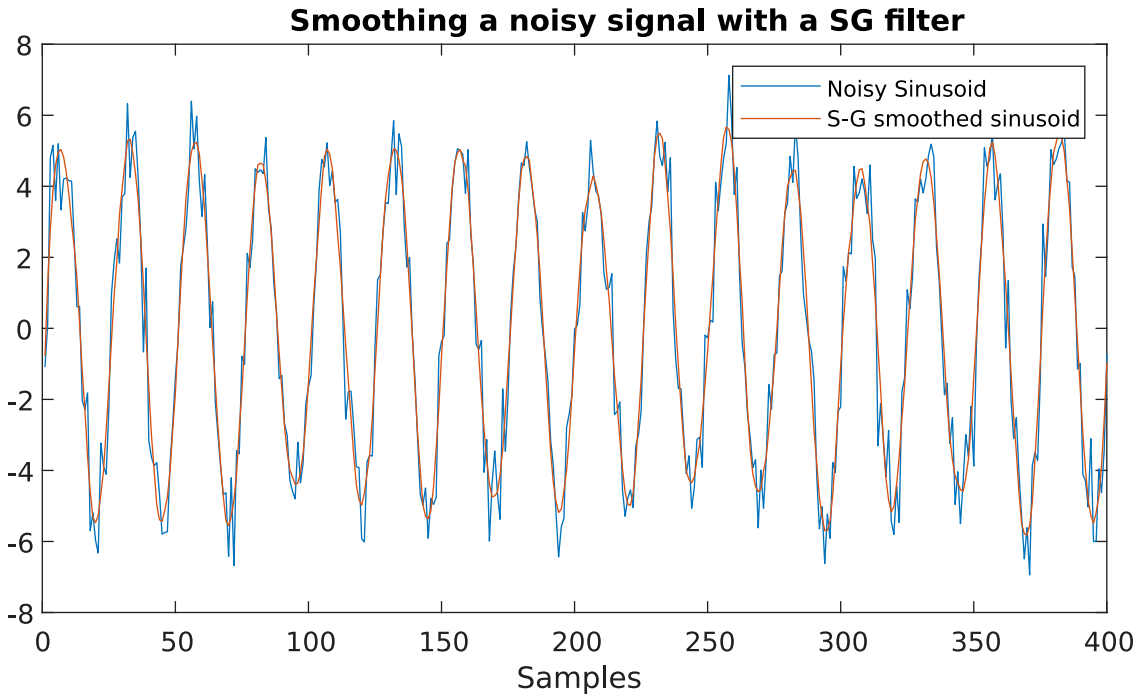


Figure 2.4: Smoothing applied on a signal: the original signal is in blue

Literature shows that several of such algorithms have been conceived over the years. Along with the simpler and common methods such as: moving average (MA), Chebyshev filters and Butterworth filters, one can find: Kalman filters, kernel smoothing, Kolmogorov-Zurbenko (KZ) filters, Laplacian smoothing, spline smoothing, exponential smoothing.
Kalman filters utilize a recursive algorithm to achieve image enhancement. Both the image and the noise are treated as random processes and described by mean and autocorrelation functions. These variables must be time-dependent. The advantage of recursive filters is that less memory is required as they only need the present data and the previous state in order to operate [24, 25]. Bayes filters represent another example of recursive filters and are related to the Kalman model.
Kernel smoothers use a window function such as a Gaussian in order to smooth a set of scattered points locally. Local methods take advantage of the correlation among the nearby pixels as a way to filter the noise. The method does not work if the image is too noisy. For each point, all the data falling inside the window are considered and a weighted average is applied. Normally, the weight is bigger the closer a point is to the centre. Linear and polynomial regressions are more complex alternatives to the simple average [26, 27].
KZ filters are used to smooth the periodogram. They are based on the simple principle of the moving average. However, they are much more robust as the MA is applied several times iteratively much like a cascade of MA filters. The window function produced by KZ filters is known to reduce spectral leakage [28].
Laplacian smoothing is most commonly used for polygonal mesh smoothing. The algorithm traverses every vertex of the mesh with a displacement that depends on the surrounding vertices [29].
Smoothing can also be performed by means of a spline interpolation. Splines are functions defined piecewise by polynomials. As explained in [30], standard polynomial interpolation methods often require the use of high order polynomials that pose major problems and limitations. Moreover, since these functions are analytic, the effect of irregular data in a small region propagates globally. Splines, on the other hand, make use of interpolation in order to obtain a smoothed version of discrete data. In particular, cubic splines are able to produce good results even with a third degree polynomial. In the paper, an iterative smoothing procedure is introduced. It is demonstrated that this method can minimize the least squares errors of the function to be smoothed and its derivatives.
Exponential smoothing deals with time series data much like a MA filter whose window function, instead of remaining constant, decreases exponentially as the samples get older in time. This method looks at past values of a time series and weighs them exponentially to predict future values (forecasting). Trends in the data are better handled by a double exponential filter [31].
Among the techniques listed before, Savitzky-Golay smoothing is the one used in this thesis. In the paper

published by Savitzky and Golay, the authors present an approach to smooth noise-affected data derived from chemical spectrum analyzers [32, 33]. They derived that fitting a polynomial to a set of samples and then determining the value of the resulting function at a single point inside the approximation interval is the same as performing a discrete convolution with a fixed impulse response. The result is a low-pass filter, which nowadays finds many more applications, other than spectroscopy. One notable example is image processing. This kind of filter is very well suited for laser line data management as it offers significant advantages. When it comes to smoothing, for instance, one interesting property is the peak shape preservation [33]. If the signal needs to be differentiated, a SG filter offers the possibility to both smooth and differentiate the input signal in a computational-efficient manner [32].

2.2.1 Mathematical model

In this section, the most important steps to derive an appropriate model describing SG filters are shown. From this analysis, the main properties are inferred and are described in the next paragraph.
The first thing that comes to mind when talking about smoothing is convolution. The results of this operation depend on the chosen convoluting function. In the original paper, Savitzky and Golay applied different, simple functions on the same data points [32]. In its simplest form, convolution can be achieved by a moving average (i.e. a rectangular function). Although this procedure reduces the noise, it also degrades the peak height. This was the case for the other shapes tested as well. It was observed that in general, the peak of the input data was moulded by the convoluting function used. What they found was that while some functions were better than others in trying to maintain the peak height, all of them failed to fully preserve it. To work around this problem, the authors proposed a solution based on the method of the least squares. The only constraint is that the samples must be equally spaced on the x-axis.

Given 2M+1 consecutive samples from the input data x, we want to compute the polynomial $p(n)$ of degree N that best fits these points:

$$p(n) = \sum_{k=0}^{N} a_k n^k, n \in [-M, M]. \tag{2.1}$$

This means, the centre of the samples is at $n = 0$.
The problem now boils down to finding the polynomial coefficients that yield minimum mean-square approximation error for the set of input samples, which translates into:

$$\frac{\partial \varepsilon_N}{\partial a_i} = 0, \tag{2.2}$$

where

$$\varepsilon_N = \sum_{n=-M}^{M} \left(\sum_{k=0}^{N} a_k n^k - x[n] \right)^2 \tag{2.3}$$

and a_i are the coefficients (i = 0, 1, ..., N).
If we wish to identify all the coefficients as a way to obtain the final polynomial, we can work out the set of N+1 equations given by Equation 2.2 in the form:

$$\sum_{k=0}^{N} \left(\sum_{n=-M}^{M} n^{i+k} \right) a_k = \sum_{n=-M}^{M} n^i x[n]. \tag{2.4}$$

These equations are known as *normal equations* [33].
However, solving all the equations in the set is not necessary. The method proposed by Savitzky and Golay required that, for each fixed subset of the input data, only the central point of the approximated function be estimated. After that, the 2M+1 window shifts to the right, adding one trailing point and leaving out the leftmost one. The origin of each new computation is redefined at every iteration. This implies that at the centre, where n = 0, the only coefficient we actually need to compute is $a_0 = p(0)$. Due to this constraint, the authors demonstrated that it is possible to find a set of coefficients that behaves like a convoluting function. The new function was able to smooth the signal while maintaining its peak shape.
They also showed that this procedure is equivalent to the least squares, it is not an approximation. It is

important to note that the new coefficients only depend on the chosen polynomial degree and frame length (i.e. number of input points considered for each iteration), regardless of the input samples. From a computational point of view, the benefits these findings offer are huge, as now it is no longer required to compute a different polynomial for each set of samples.
To understand why the previous statement is true, we can write Equation 2.4 in matrix form. We define $\underline{\underline{A}}$ as a (2M+1)x(N+1) Vandermonde matrix in this way:

$$\underline{\underline{A}} = n^i, \tag{2.5}$$

where -M < n < M and i = 0, 1, ..., N. If $\underline{a}$ is the N+1 vector of the coefficients and $\underline{x}$ is the 2M+1 vector of samples, considering OLS (Ordinary Least Squares) theory, it can be shown that Equation 2.4 is equivalent to:

$$\underline{a} = (\underline{\underline{A}}^T \underline{\underline{A}})^{-1} \underline{\underline{A}}^T * \underline{x}. \tag{2.6}$$

We can define:

$$\underline{\underline{H}} = (\underline{\underline{A}}^T \underline{\underline{A}})^{-1} \underline{\underline{A}}^T \tag{2.7}$$

and write:

$$\underline{a} = \underline{\underline{H}} * \underline{x}. \tag{2.8}$$

Now, if we want to calculate the central point for the sample, we have to find a_0. To do this, we only need the first row of the $\underline{\underline{H}}$ matrix. However, $\underline{\underline{H}}$ does not depend on $\underline{x}$ but only on N and M. Thus, the coefficients do not change once we fix N and M. The computation reduces to:

$$y[0] = a_0 = \sum_{m=-M}^{M} h_{0,m} x_m, \tag{2.9}$$

where $h_{0,m}$ are the elements of the 0th row of the $\underline{\underline{H}}$ matrix. Equation 2.9 is equivalent to:

$$y[n] = \sum_{m=n-M}^{n+M} h[n-m] * x[m] \tag{2.10}$$

when n = 0. Hence proving that least squares smoothing can be seen as "a shift-invariant discrete convolution process" [33]. The result can be used to design a digital filter whose impulse response is given by $h_{0,m}$. This means a number of fixed coefficients normally equal to 2M+1.
Similarly, given the same constraints as before, it can be shown that it is possible to compute the first derivative of the points, too. This method, likewise based on the least squares, extracts the first derivative of the best-fit polynomial. In this case, a_1 (i.e. the second row of the $\underline{\underline{H}}$ matrix is required). Higher order derivatives up to the grade of the polynomial can also be evaluated.
One thing to keep in mind is, that in order to design the filter properly, we must ensure that N $\leq$ 2M. In other words, the polynomial coefficients we want to find must be less or equal than the data samples we process at each iteration. At the limit, when N = 2M, the polynomial fits the 2M+1 data samples perfectly: no smoothing occurs [33].

2.2.2 SG filter's general properties and peculiarities

Aside from peak shape preservation, there are other attributes that characterize Savitzky-Golay filters. Some of them are common to most FIR filters. In the following list, the general properties are summed up:

- Linear phase
- Simple to implement and to simulate
- Intrinsically stable (no loops present)
- Finite precision arithmetic and fixed point arithmetic are supported

To ensure a linear phase, the coefficients must be symmetrical around the centre. Thus, an odd number of taps is required (2M+1) [33]. Linear phase is important to avoid phase distortion that in turn alters the shape of the signal. However, non-linear phase SG filters can also be implemented.
Due to the absence of loops, FIR filters are especially useful when dealing with approximate, imprecise designs,

as the total error can be kept under control. There is no need for a null bias, as numerical errors do not accumulate at each computation. This means that FIR filters can work with fewer bits compared to IIR filters. As a consequence, area and power consumption can be reduced.
The following list highlights some characteristics distinctive of SG filters [33]:

1. If N is an even integer, the filters that result from choosing either N or N+1 are identical. This happens because all coefficients of the impulse response design polynomial whose index is odd are zero.
2. Given a symmetric impulse response, it follows that the frequency response is real. SG filters behave as type I FIR lowpass filters, whose gain, in the passband, is equal to 1.
3. A SG filter where N = 0 and M = 1 is equivalent to a moving average.
4. Frequency response in passband is very flat.
5. The normalized cutoff frequency depends on both N and M. It increases proportionally to N but is inversely proportional to M.
6. SG filters display poor attenuation in stopband regions, around the zeros. If we fix M and increase N, the minimum attenuation in the stopband region increases.

Figures 2.5 to 2.8 show some of the properties previously discussed. They were obtained using MATLAB.
In particular, Figures 2.5 and 2.6 show the impulse responses for two specific SG filters that will be used in the coming chapters. Despite the interpolation, the responses are discrete. The red points represent the $h_{0,m}$ from the $\underline{\underline{H}}$ matrix and thus the filter coefficients. Figure 2.7 shows the frequency response both in magnitude and phase of the same filter. From this image, the flatness of the passband is noticeable, as well as the mediocre stopband attenuation and the linear phase relationship. Finally, Figure 2.8 compares the frequency responses of multiple filters with a fixed frame length M and increasing N. Due to property 1 from the previous list, it is sufficient to show only even values of N. Property 6 can be observed.

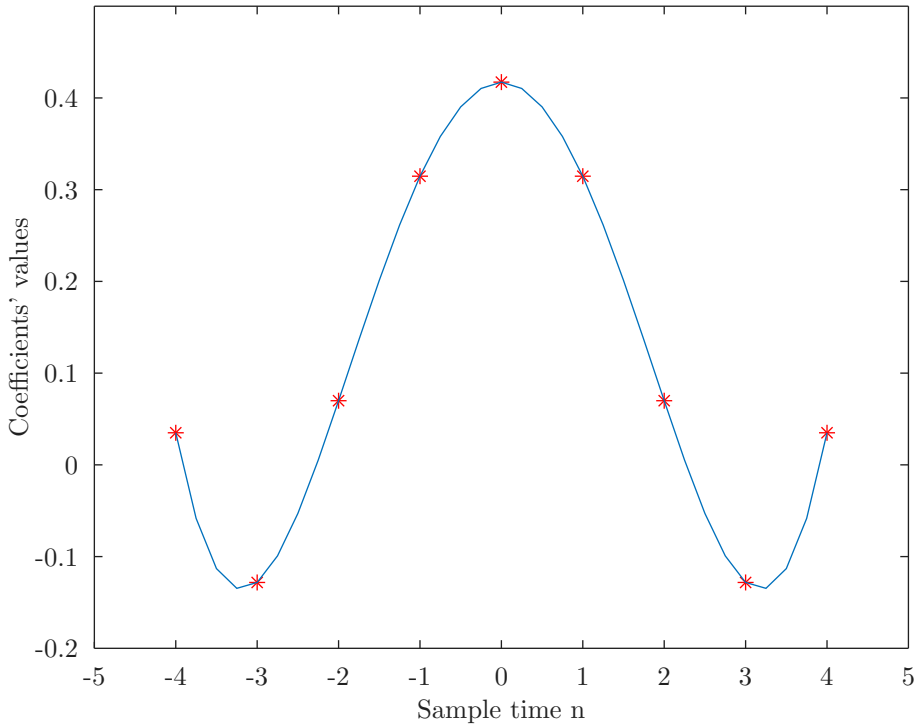


Figure 2.5: Impulse response h of a SG smoothing filter with N = 4 and M = 4

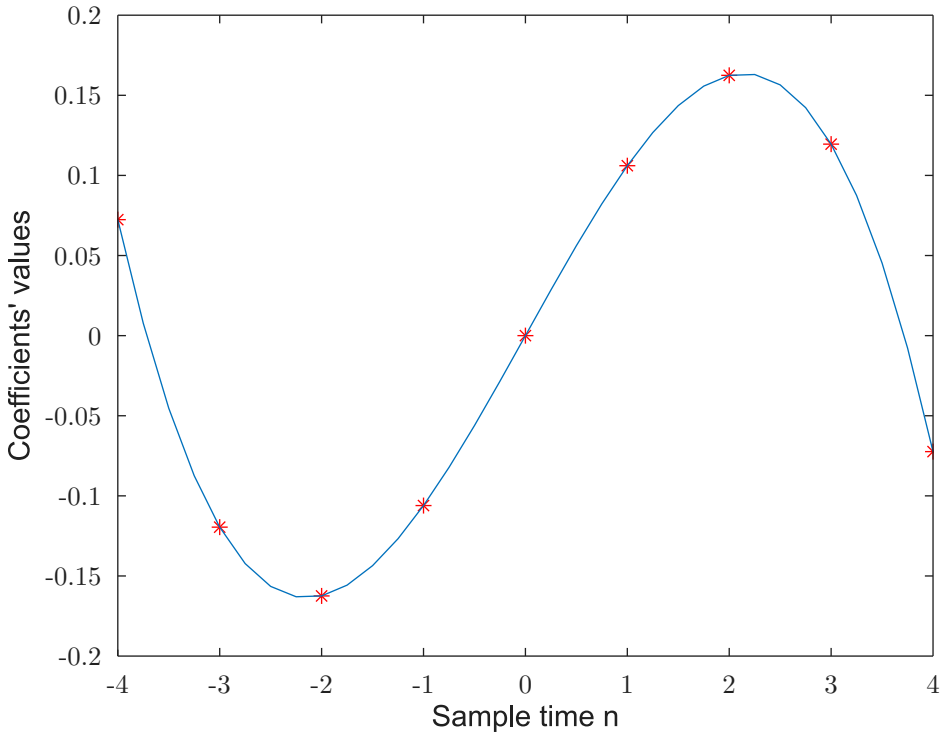


Figure 2.6: Impulse response h of a SG differentiation filter with N = 4 and M = 4

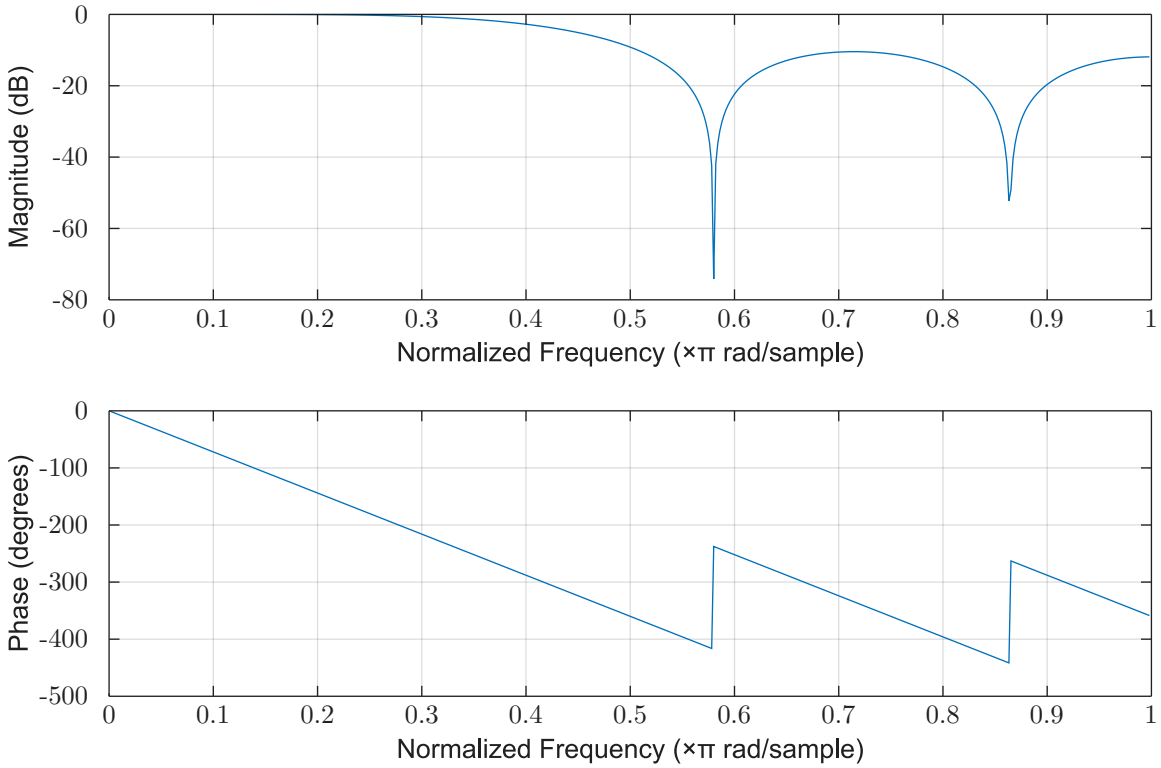


Figure 2.7: Magnitude and phase of a SG filter with N = 4 and M = 4

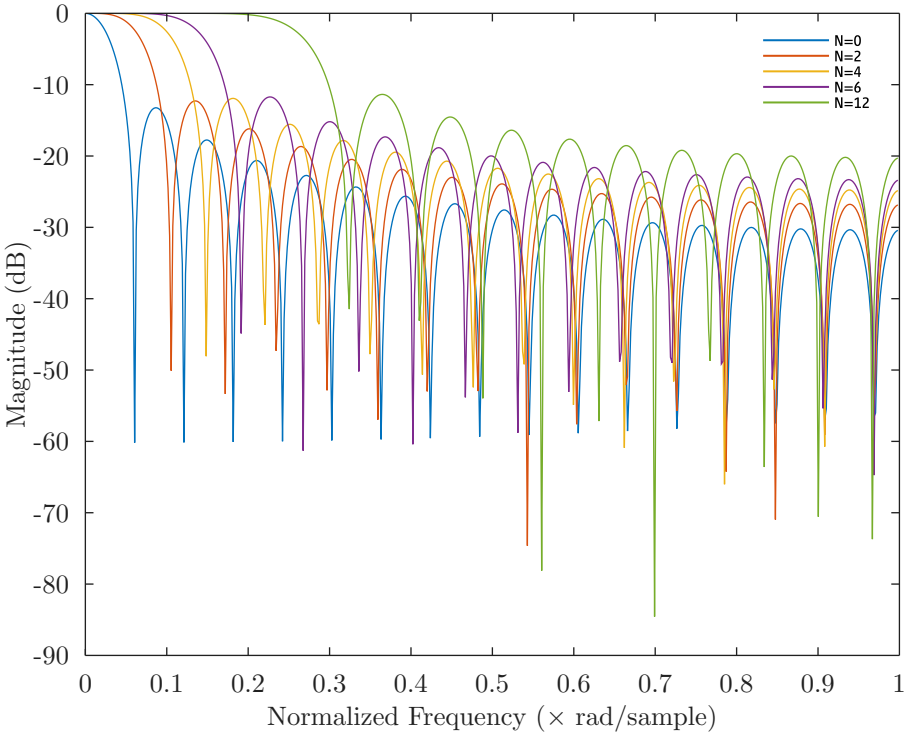


Figure 2.8: Frequency response of a SG filter with fixed M = 16 and increasing N

2.3 Approximate Computing

Approximate computing is a relatively new technique that promises high performance or energy efficient designs in exchange for a degradation of the results' accuracy [1]. In this thesis, deterministic algorithm that produce imprecise results are explored. A deterministic algorithm processes a given input always in the same way, leading to a predictable output. Non-deterministic algorithms also exist for which the same inputs might produce different approximations over time.

Approximate computing is becoming popular due to the increasingly high demand for applications that involve media processing. Moreover, large integrated circuits (ICs) that also work at high frequencies, if not properly optimized, often exceed the power consumption limitations that prevent the circuit from sustaining permanent damage, especially when all units are running simultaneously. That is why, in modern circuits, the number of concurrently working units at some point in time is limited. The area of the IC that cannot work due to power limitations is referred to as *dark silicon* [34]. Nowadays, low-power design techniques are essential to the majority of hardware systems. In particular, arithmetic circuits, and thus arithmetic logic units (ALUs), devoted to media applications, greatly benefit from approximations. Depending on the parallelism, these blocks' complexity can grow exponentially, and so will area. Power is also expected to increase, although it is difficult to estimate by how much as accurate models also depend on topologies/architectures and input data statistics. Approximation methods work well on memories, too [35, 36, 37].

Two types of techniques that produce inexact circuits can be identified [38]. In timing-induced approximation, voltage over-scaling is used to limit power consumption. The difference with ordinary voltage scaling lies in the fact that although the voltage decreases, the frequency is not reduced accordingly. As a consequence, timing violations occur on some of the paths considered critical thus yielding a wrong result. The same effect is achieved by over-clocking. In this thesis, attention is directed to the second approach, which is the functional one. Error is introduced by simplifying the Boolean function of a circuit that in turn reduces its complexity. [34, 39] contain a survey of approximate computing strategies for arithmetic circuits. In general, these techniques can be implemented at any hierarchical level of the circuit, which means from transistor and logic gates levels to entire approximate systems and even in software [40, 41]. In this thesis, focus is placed on applying these strategies to arithmetic circuits. The most common blocks in the system are the adders and multipliers used to put together the two FIR filters. One divider is also present. Subtractors that are part of comparators might

be approximated in specific cases, as explained in Chapter 5. However, in this thesis, they are exact.

2.3.1 Approximate adders

These circuits are normally divided into two parts: the accurate part comprising the MSBs and the inaccurate one that contains the LSBs. This is usually achieved by adjusting some full adder (FA) blocks in the path in order to generate an approximate truth table. Depending on the error one is willing to tolerate, one can either extend or reduce the number of bits affected by error. Several approximation methods have been proposed [39]. The main four ones are listed next:

- Speculative adders
- Segmented adders
- Carry select adders
- Approximate FAs

A speculative adder is faster than its accurate counterpart as it is able to predict the sum bits by looking at the previous k < n LSBs. This means that the carry chain is shorter and, as a consequence, the speed is higher.
In a segmented adder, the computation is sped up by having the adder elements work in parallel, thus scaling down the length of the carry propagation chain. In other words, the adder is segmented into smaller adders each of them working in parallel. Depending on how the carry out of each segment is handled, a design can be more or less accurate.
carry select adder (CSEA) duplicate the number of FAs in order to compute both outcomes decided by the carry in. An approximate CSEA normally employs speculative techniques to predict the value of the carry in. Additionally, it too makes use of segmentation. Another strategy consists in approximating a FA block. Typically, this is done by simplifying the FA logic function, which translates into a simplification at either logic or transistor levels. The approximation only affects the LSBs, while the upper part stays accurate.

2.3.2 Approximate multipliers

Concerning the multipliers, similarly to the adders, four main methodologies can be identified [39]:

- Approximation in generating the partial products
- Approximation in the partial product tree
- Use of approximate adders
- Booth multipliers

The first three techniques are used for unsigned multiplication, while high-radix Booth multipliers have been proven to be fast when working with signed numbers.
In [42], Kulkarni et al. managed to obtain imprecise partial products by approximating a single entry in the Karnaugh Map representing the logic function of a 2x2 multiplier. The computation of the product tree is accurate. To approximate the product tree, one solution proposed in [43] involves the elimination of rows and/or columns of adders that make up the tree. Truncating the LSBs of the partial products is also a possibility. [44] presented a multiplier divided into two sections of which only one is able to perform a multiplication, but on a limited number of bits. The circuit is equipped with a control block that performs an exact multiplication on the LSBs only when the MSBs are 0. Alternatively, only the MSBs yield an accurate result. It can be argued that reducing the critical path in the computation of the sum of the partial products tree is the most effective method to obtain a good trade-off between accuracy and efficiency [1].
As for signed multiplication, a high radix Booth encoder is normally employed. A simple approximation is achieved by truncating the LSBs of the final product. However, this does not offer many advantages compared to an accurate design. The next step would then be to truncate the partial products. This operation introduces a large error that has to be corrected by means of additional circuits.

2.3.3 Approximate dividers

The last arithmetic block discussed in this section is the divider. Since this module is less frequently used than adders and multipliers, only few approximate versions are available in literature. As explained in [39], inexact sequential dividers require a limited number of resources but their latency is significantly greater than other blocks found in the pipeline. Hence, these units constitute a major bottleneck in the overall speed of the circuit. As a consequence, only combinational dividers will be tested and discussed, array dividers in particular. As division is the inverse operation of multiplication, while a multiplier features a multitude of adders, array dividers are made up of subtractors. Most of the times, these building blocks are approximated by logic complexity reduction at transistor and logic levels in order to achieve a decrease in area and power. Each cell of the array contains a 1-bit subtractor and other logic blocks depending on the algorithm. Its behaviour can be summarized by a logic function that will be subjected to approximation. The number and position (weight) of approximate cells in the array depend on the error tolerance of the design. A low power divider based on Vedic mathematics is proposed in [45]. While it is not an approximate divider, it features an algorithm derived from ancient Indian mathematics that promises an important power reduction. In the paper, the algorithm, originally devised in decimal form, is converted into a binary version. As this technique involves complementing the divisor's bits, the circuit has to deal with '0's, '1's and '-1's. For this reason, each bit is encoded by two bits to account for the sign. Once the divisor is complemented, the division can be carried out by using multipliers and adders. A single subtractor is required at the end to decode the result. Simulation has shown that this circuit can reduce power by more than 50% when compared to a conventional divider. This design is featured in [46] where Manikantta et al. present approximate dividers able to achieve even better figures in terms of power and delay at the cost of precision. In particular, they compared circuits based on the well-known restoring and non-restoring algorithms and concluded, among other things, that restoring dividers perform better than their non-restoring counterparts. The claim is also supported in [47]. In [46], it is shown that an exact non-restoring cell contains a 1-bit subtractor and an XOR gate. This gate is replaced by a multiplexer in a restoring cell. Three different approximate configurations in [46] and three more in [47] are obtained by simplifying the logic function associated to either the B_{out} (borrow out) bit or the D (difference) bit. They will be analyzed in the following chapters. Finally, in [39] two other classes of approximate dividers, besides array dividers, are presented: a curve fitting based approach [48] that relies on an antilogarithmic algorithm and a rounding based technique [49] that rounds the divisor. This thesis focuses on restoring array dividers.

2.3.4 Metrics

In this section, the metrics that will be used throughout the thesis are presented. In order to study and evaluate approximate circuits, new figures of merit must be introduced. These metrics are primarily aimed at comparing the inaccurate design with the exact one. The most general ones are defined in [34, 39, 50] and are reported in the equations below.
The first and most important parameter to characterize is the error rate (ER) defined as the probability to get an error at the output of an arithmetic circuit. This rate depends on the input data statistics. In this thesis, uniform distribution is assumed when testing the single arithmetic units. For the whole system a different function is used.

$$ED = |M' - M| \tag{2.11}$$

The error distance (ED) is defined as the positive difference between the approximate result (M') and the accurate result (M). As the name suggests, this parameter represents the distance of the generated inexact output from the exact number, given the same input vectors.

$$\begin{aligned} MED &= \sum_{i=1}^{N} ED_i * P(ED_i) \\ NMED &= \frac{MED}{D} \\ NED &= \frac{ED}{D} \end{aligned} \tag{2.12}$$

From ED, mean error distance (MED), normalized mean error distance (NMED), normalized error distance (NED), relative error distance (RED) and mean relative error distance (MRED), are defined. MED is obtained by taking the average of all the error distances. $P(ED_i)$ is the probability associated to each ED, which is also

the probability of the *ith* input combination. NMED and NED are normalized quantities. D in this formula is the maximum output of the accurate design. Dividing by this number results in a figure of merit insensitive to the word length, for the same circuit. It is useful to compare designs of different sizes [34].

$$\begin{aligned} RED &= \frac{ED}{M} \\ MRED &= \sum_{i=1}^{N} RED_i * P(RED_i) \end{aligned} \tag{2.13}$$

RED is obtained by diving the ED by the accurate result. Given two input combinations generating the same ED, RED will be greater for the combination that produces the smaller exact result. MRED is calculated from RED by evaluating the mean. $P(RED_i)$ is, again, a probability.

$$\begin{aligned} MSE &= \sum_{i=1}^{N} ED_i^2 * P(ED_i) \\ RMSE &= \sqrt{MSE} \end{aligned} \tag{2.14}$$

The last two figures shown in Equation 2.14 are the mean squared error (MSE) and the root-mean-square error (RMSE). MSE is used in statistics to assess the quality of an estimator [51]. In linear regression, this positive quantity tells us how close a regression line is to a set of point [52]. The smaller the MSE is, the closer we are to the line of best fit. In many image processing applications the MSE is exploited to define the peak signal-to-noise ratio (PSNR) used for quality estimation [53]. In approximate computing, MSE, along with RMSE, are employed to estimate the error magnitude [34].
Basic hardware-related metrics include: critical path delay, area, power consumption. These quantities can be put together to obtain compound metrics such as the power-delay product (PDP), the area-delay product (ADP) and the energy-delay product (EDP).
Depending on the application, the designer might need to look at some or all of these variables. For example, one might only be interested in the maximum error of a circuit. In some cases, the chances for this error to occur are negligible, so attention can be shifted to mean or normalized error metrics. Moreover, the designer must be able to compare different approximate circuits based on parallelism, number of inputs and approximation depth.
Along with the previous figures, several methods to analyze inexact circuits have been proposed but, as the field is relatively recent, they apply mostly to approximate adders, are application specific or are technology dependent. A general approach is presented in [38] where Venkatesan et al. came forward with a systematic analysis methodology to compare approximate designs with conventional ones. In the paper, they highlighted the inadequacy of standard verification methods that simply check the equivalence between specification and implementation whereas, in approximate computing the designer is faced with a more complex task. Their approach works for both time-induced approximations, such as voltage over-scaling, and functional ones. Normally, the novel techniques are used in combination with standard verification strategies such as Monte Carlo, exhaustive simulation, SAT solver and BDDs.
In [54], a new model to evaluate the reliability of both approximate and probabilistic adders is presented. The notion of reliability defined in the paper reveals the presence of an error by showing how many bits in the output are incorrect given a certain input. However, it does not consider the weight they have. To complement this metric, ED is used as it takes into account the position of the wrong bits thus displaying the effect of the error on the final result. NED and MED are calculated using sequential probability transition matrices and employed to estimate reliability and accuracy of the designs.
[53] contains a new analytical model that deals with approximate adders. From the analysis, metrics such as the ones defined above, as well as application specific metrics, can be estimated without recurring to simulation (that requires a full model for each circuit). In this way it is easier to compare multiple adders even with different parallelism. As an example, an application specific quantity used in image processing, the PSNR, is related to the generic metric MED.
In [55], a statistical method mainly for low-power approximate adders (LPAA) is introduced. LPAAs, as described by Ayub et al., employ logic simplification at transistor and logic levels so as to reduce capacitances and, subsequently, power consumption. Analytical methods for computing the error probability for multi-bit LPAAs are seldom encountered in literature. The reason behind this stems from the way error probability propagates through each adder stage. Probability of carry-out changes for each new FA and its value cannot be

simply summed to the previous stages' probabilities. Instead, inclusion-exclusion principle is used to detect and remove duplicate terms, inevitably generated, in order to avoid counting them more than once. This process generates a huge number of terms for big adders making its use impractical. For the same reasons, exhaustive simulation is out of the question as well. The proposed model does not rely on inclusion-exclusion principle. Instead, it makes use of a recursive algorithm based on matrix arithmetic proven to be scalable in terms of size and flexible in terms of approximation techniques.
In [56], Mazahir et al. proposed an analytical model able to determine the probability of error and the probability mass function of the error for approximate adders without the need for simulations. They emphasized the limits met by traditional simulation methods that are overcome by their mathematical model.
Additional research on approximate computing is introduced in Chapter 4 where the algorithms chosen for this thesis are presented.

CHAPTER 3

Software design: exact reference models

During the design of a digital circuit, software is omnipresent. This is due to the fact that modern digital ICs are very complex. This concept is referred to as *Computer-aided Engineering* and it applies to all fields of engineering. Software tools take part in speeding up the design process significantly as well as handling optimization, verification and validation, all of which would require a considerable amount of time or be outright impossible when done manually. During the design phase, compilers use heuristic algorithms to achieve logic minimization. From the designer's point of view a circuit modeled through a hardware description language is translated into a schematic at RTL level. Logic synthesis on pen and paper is much harder as its complexity explodes even when only a few logic variables are involved.
The next important step that makes extensive use of software is the verification phase. Through simulation, equivalence between the HDL model and the synthesized circuit is assessed. The results must meet the specifications and constraints set at an earlier stage. Depending on the project, verification can take up more time than any other step. As ASIC chips are extremely expensive to fabricate and cannot be fixed easily [57, 58], a thorough simulation is essential. FPGAs, instead, are programmable and are often used for design emulation or in the event the costs demanded by a custom IC cannot be sustained due to a small scale production of the units involved.
Afterwards, physical design is carried out. If the final product is an ASIC, software is used to perform placing and routing given constraints such as area and delay. Similarly, in an FPGA, software is tasked to execute the place and route operation, so that the programmed circuit behaves like the original model. This chapter focuses on the role software has had on this project and the results that ensued from it.

3.1 Softwares used

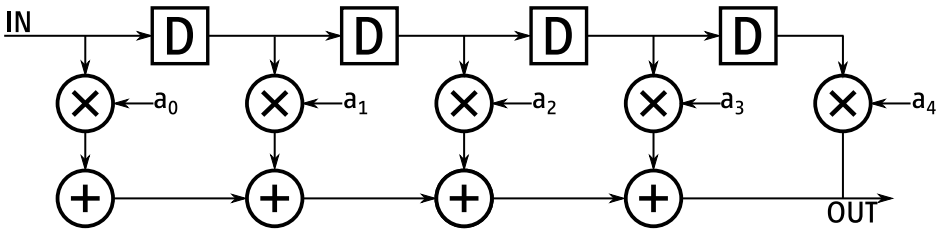
In ASICs, a circuit is synthesized using a technology library supplied by fabrication companies. In this thesis, a Cyclone V FPGA is programmed to replicate the circuit described using VHDL. Intel® Quartus® Prime is used to configure the FPGA and to evaluate information such as delay, area and power. Software versions of the different units are created using MATLAB®. Due to infinite precision, these models act as an archetypal equivalent circuit, be it accurate or approximate. For this reason, they are expected to exhibit better results than the hardware design, which suffers from precision loss owing to a limited number of bits. Outcomes generated by hardware and software models are compared and commented to highlight the impact of this effect as well as that of other sources of noise. To test the behaviour of the circuit, ModelSim is used. A VHDL testbench reads input data from a file and writes the output on a different file. A MATLAB script processes the numbers at the output to evaluate the error metrics previously discussed. It also verifies that the results are correct. As ModelSim simply simulates the VHDL code, a correct behavioural operation of the circuit under test does not imply that it will be synthesizable. So normally, another simulation can be carried out post-synthesis where realistic parameters' estimations can also be extracted. The flexibility of the FPGA renders post-synthesis simulation superfluous as the timing analysis is put into effect by the FPGA synthesis tool and the circuit can be tested directly on the hardware. In this thesis, synthesis is performed by Quartus Prime. All the steps described so far will be explained more in detail in the coming sections. Much of the project is conducted on a Windows 10 system, while time-consuming simulations are carried out on a remote server equipped with a Linux environment that is several degrees faster than an average personal computer. Plots

and charts produced throughout the thesis are generated by MATLAB scripts, while for circuit diagrams and various schemes, *diagrams.net* and *Inkscape* are used.

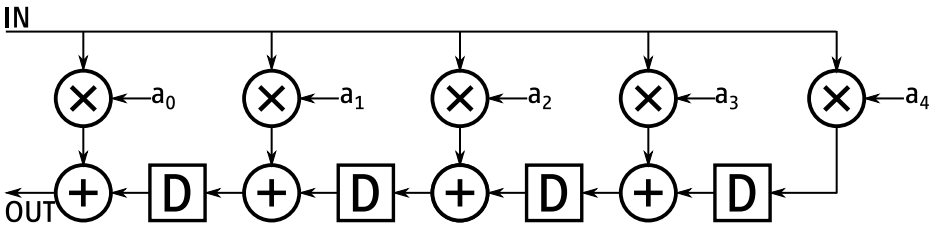
3.2 Filter structure

As stated before, a particular type of digital FIR filters known as SG filters is used for the design. As calculating the coefficients for different orders and frame lengths can be time-consuming, two convenient functions provided by MATLAB are used. The *sgolay* function takes the order of the filter and the frame length as the two main input arguments and generates two matrices. The first one contains the smoothing coefficients, the second one holds coefficients for all derivatives up to the polynomial's order. A filter of frame length $2M + 1$ equal to nine, will generate a $9x9$ smoothing matrix. This means dealing with 81 coefficients instead of just the nine one might expect. This happens because the filter experiences a transient affecting the starting and ending points. In those instants, the mathematical formulation developed in Chapter 2 is no longer applicable. Evaluating the least-squares polynomial only at the central point was enough to obtain a fixed set of coefficients independently of the input data. However, at the boundaries, no central point is present as the filter is either loading the data in the chain or emptying it. As a FIR filter only remembers $N - 1$ past samples (where N is the order of the filter), its transient response will last $N - 1$ cycles [59]. For each of these cycles a different combination of coefficients must be used. Outside of the settling period, instead, only one set of taps is needed as the circuit reaches the so-called steady-state. Then, by looking at the first output matrix, the correct vector to choose from is the whole middle column that contains the steady-state smoothing coefficients. As for the hardware design, transient coefficients are not taken into account as, given the data statistics for this particular application, there is a high chance that both the samples at the beginning and end are very close to zero. Moreover, this helps keeping complexity and thus resources and power consumption low. This does not apply to the software model where the effect at the boundaries is considered.
The second matrix produced by *sgolay* contains, in each column, the coefficients to obtain the derivatives of orders up to the polynomial's degree, including the one of 0th order (smoothing). The column selected for this project is the one containing the first derivative coefficients (i.e. the second one). The first column consisting of the smoothing coefficients can be used for a separate filter if the smoothing provided by the differentiation filter is not enough.
Once the taps are selected, MATLAB's function *conv* is used to obtain the output of the filter by convolving the coefficients with the input vector. The operation is performed only on the central part of the convolution (i.e. steady-state). This is achieved by including the option *'valid'* as an argument. The output vectors related to the starting and final transients are obtained by simple matrix multiplication, they are then combined with the main array. The same result can be observed by using another method called *sgolayfilt*. Other than order and framelength, this function also takes the input vector as an argument and produces the output data by applying a SG FIR smoothing filter that can be either symmetric or non-symmetric. While this solution requires only one line of code, it cannot be implemented for a differentiation filter. This obstacle is overcome by using the generic *filter* function, which requires the vectors of both coefficients and input data. MATLAB also provides the designer with a myriad of interesting functions intended to devise a generic FIR filter, here are some of them: *fir1* (window-based), *fir2* (frequency sampling-based), *firls* (least-squares linear-phase), *firpm* (Parks-McClellan), *fircls* (constrained-least-squares), *rcosdesign* (raised cosine).
When dealing with approximate filters (i.e. filters whose arithmetic blocks are not exact), the aforementioned methods cannot be used. Therefore, a filter that resembles a hardware structure, such as direct or transposed form, is created on MATLAB. The approximated unit is instantiated as a function, while the standard operators are used for blocks that are exact. Different combinations are tested out where either adders or multipliers are replaced by inexact versions, or both of them are.
As noted in [33], designing SG filters with a large order N, results in a badly conditioned problem. This means that the actual number of combinations available, corresponding to different normalized cut-off frequencies, are more limited when compared to other FIR filters. In this work, however, the filters' order does not pose a problem.
A digital filter's components include multipliers, adders and registers. The way they are arranged, though, depends on the structure chosen. This choice is very important as it affects parameters such as delay and stability. For example, by shifting from infinite precision to a limited number of bits, coefficients are inevitably approximated. This error affects the transfer function of the filter as the zeroes and poles move from the desired position. By how much the zeroes and poles move given a certain approximation depends on their relative

positions. If they are close to each other, the displacement will be significant. This is the case for an IIR filter. In this case a cascaded form can improve its stability. FIR filters on the other hand are much more robust as they only have zeroes that are also much more distant from each other. This means that for such circuits, the choice of the right structure is not crucial. Figure 3.1a shows an example of a FIR filter represented in a direct form. The transposed form is shown in Figure 3.1b.
The main advantage of this last structure lies in the fact that its critical path delay does not depend on the number of adder stages present as the registers break the path. This solution can be used when a stringent constraint is imposed on the clock frequency and especially for filters of large order. The problem with this architecture is that, as the registers now appear after the multiplier stage, their parallelism must be doubled, at least, to avoid errors. This bit-width can increase arbitrarily depending on the order of the filter. In this thesis, this effect is expected to be negligible as the filters possess a limited number of coefficients.
One advantage offered by the direct form is the possibility to actually halve the number of multipliers by exploiting coefficients' symmetry. As previously stated, in order to have a linear phase in a FIR filter, its coefficients must exhibit either even or odd symmetry around the centre. This condition is both necessary and sufficient. For instance, if symmetry is even, one can get rid of the second half of multipliers simply by grouping the terms in the equation that share the same coefficients. An example is shown in Figure 3.2. SG smoothing coefficients normally display even symmetry, while for differentiation symmetry is odd. In general, even symmetry is observed for even order derivatives, while odd symmetry for odd order ones. As FIR filters do not contain any loops, they cannot benefit from optimization technique such as folding and loop unrolling. Methods that include pipelining and parallel processing are employed to reduce power consumption. Pipelining can also be used to decrease the delay that affects the direct form.
In the next sections, the two filters used in this project are presented. A small filter size is chosen to speed up the hardware implementation. Higher order filters are simulated only by software. In the following subsections, the filters' parameters and characteristics are briefly described.



(a) FIR filter, direct form



(b) FIR filter, transposed form

Figure 3.1: An example of the two main structures for a FIR filter

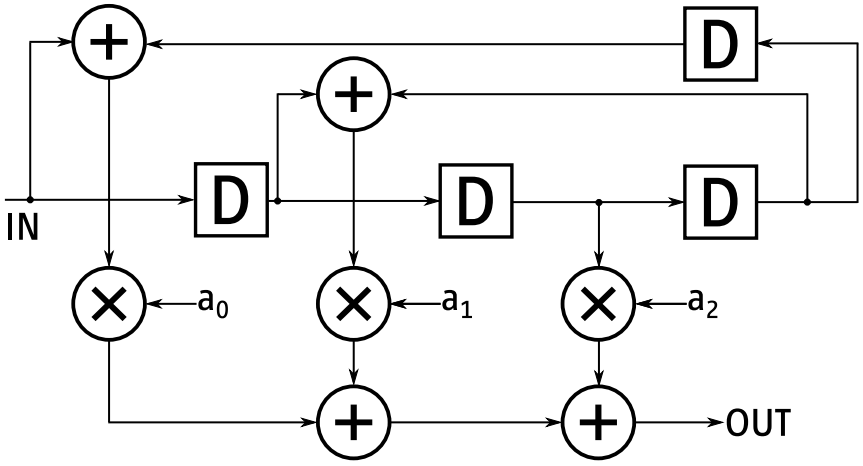


Figure 3.2: Simplified FIR direct form thanks to coefficients' symmetry

3.2.1 Smoothing

Smoothing is an operation that can be carried out by applying a low-pass filter to a data stream. Thus, the first component of the pipeline is a SG smoothing filter that acts as a conditioning circuit in order to reduce high-frequency noise of the input picture. The necessity of such unit depends on the image quality as well as the system's error tolerance. In the thesis, both cases (i.e. computation with or without smoothing) are tested. Similarly to all the other blocks, a software version of the filter is created and simulated before moving on to the hardware implementation. For this task, a MATLAB script that employs some of the functions illustrated before and a Simulink model are used.
A digital filter can be characterized by parameters such as delay, frequency response, cut-off frequency, DC gain and phase relationship. The unilateral z-transform for an FIR filter is given by:

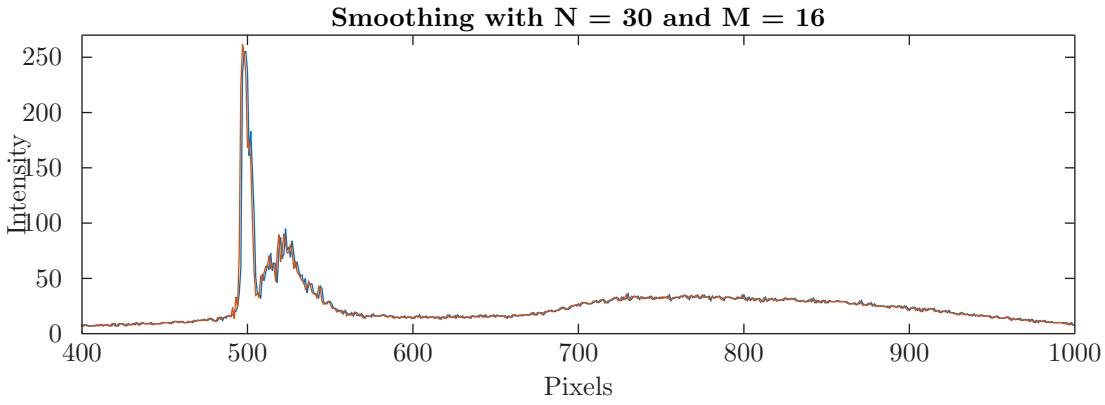
$$H(z) = \sum_{n=0}^{N-1} h(n) z^{-n}. \tag{3.1}$$

By replacing z with $r * e^{jw}$, where $r = 1$ (unit circle), the discrete time fourier transform (DTFT), which represents the frequency response of the filter, is derived:

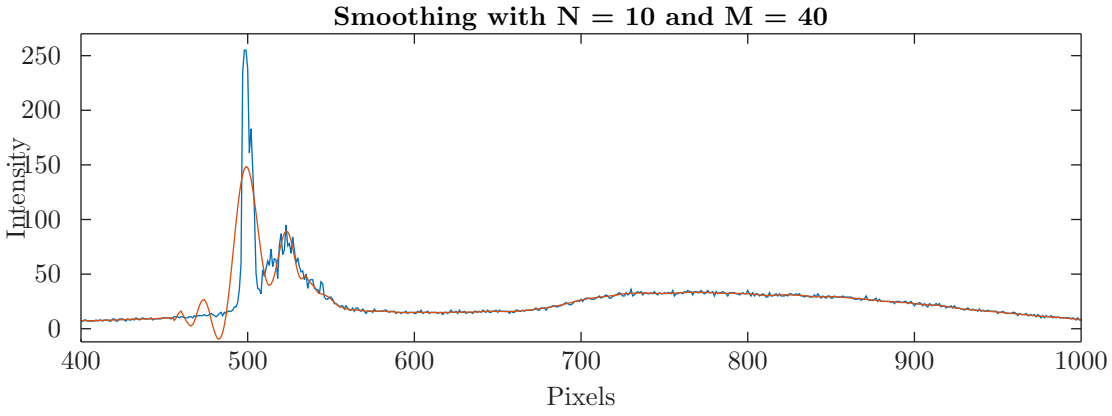
$$H(e^{j\omega}) = \sum_{n=0}^{N-1} h(n) * (e^{j\omega})^{-n}. \tag{3.2}$$

The DC gain is obtained by setting ω to 0. By doing so, it can be seen that this number is equal to the sum of all coefficients. From the theory about SG filters discussed in Chapter 2 and by looking at Figure 2.7, it can be concluded that the DC gain for this kind of filter is 1, regardless of order or frame length. It is possible to vary this number simply by multiplying all the coefficients by the same factor. However, as coefficients do not provide any information on parameters such as delay and cut-off frequency, digital filters' behaviour strongly depends on the chosen sampling frequency (f_s). The MATLAB function *freqz* is used to generate the frequency response for the filter shown in Figure 2.7. Frequencies are normalized so as to decouple the design of digital filters from f_s. Normally, the unit frequency is the Nyquist frequency defined as half f_s. This means that the normalized frequency (f_n) will always assume values between 0 and 1. In MATLAB, f_n is expressed in rad/samples meaning that the Nyquist frequency has a value equal to π instead of 1. In SG filters, the normalized cut-off frequency (f_{cn}) increases as N (order of the fitting polynomial) grows and decreases as M (order of the filter or frame length) rises. To test and compare the smoothing capability of SG filters, three filters with three different f_{cn} are devised. This is mainly done to highlight the effects of the two parameters N and M (order of polynomial and frame length respectively) on the frequency response and also f_{cn}, which affects the extent of the smoothing. The first filter is also the smallest one as it has $N = 4$ and $M = 4$. This is the only unit, among the three, that is further implemented in hardware. Its f_{cn} (defined as the point where the response's magnitude reaches $-3dB$) is equal to $0.41 * \pi\ rad/samples$. The second filter is instead designed using $N = 30$ and $M = 16$. For

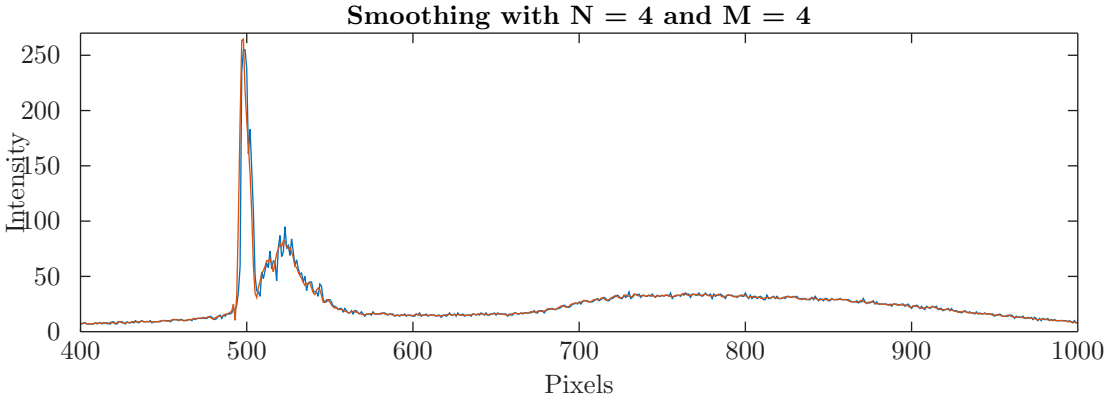
this circuit, the f_{cn} is around $0.8 * \pi$ $rad/samples$. This value is very high because $N \simeq 2M$. As a consequence, the passband is also wide and thus little to no smoothing occurs. On the other hand, for the third filter $N = 10$ and $M = 40$ are chosen. In this case $N < 2M$ meaning that the passband is narrow and the noise is filtered to a greater extent. For this last design, f_{cn} is equal to $0.089 * \pi$ $rad/sample$. Figure 3.3 shows the smoothing effect of the three filters applied on data extracted from an actual image. Figure 3.3a shows the filter with the widest bandwidth. The blue line represents the original signal while the orange line is the smoothed signal. By looking at the picture, it is clear that the two traces are almost indistinguishable. As expected, the smoothing is nearly undetectable. Such high values for N also create a badly conditioned problem in MATLAB. Figure 3.3b shows the filter with the narrowest bandwidth. The signal is heavily smoothed and a considerable amount of its noise and information is lost. The peak also appears to have been attenuated and a transient-like effect appears right before the steep increase in intensity. A middle ground between the first two filters is found in the last example which shows a good amount of smoothing with no side effects. The peak at the output is actually higher than the one from the original signal.



(a) SG filter with $f_{cn} = 0.8 * \pi$ $rad/samples$



(b) SG filter with $f_{cn} = 0.089 * \pi$ $rad/samples$



(c) SG filter with $f_{cn} = 0.41 * \pi$ $rad/samples$

Figure 3.3: Output of three smoothing filters with different bandwidths

The last important parameter of the filter is delay. From the moment an input signal is applied to the instant the output is observed, a certain amount of time needs to pass. In other words, every filter is affected by delay. To better observe the behaviour of a filter, it is desirable to plot input and output signals ideally with no phase shift between them. In the case of FIR filters, the group delay is given by the formula:

$$D = \frac{N-1}{2f_s}, \tag{3.3}$$

where N is the filter's order. The equation depends on f_s. However, as the phase is linear, all frequency components are delayed by the same amount of time. This means that the group delay, defined as the negative derivative of the phase delay, is constant in the passband and can be easily compensated for simply by shifting one signal in time until the two overlap. This also means that if the signal stays within the passband, its group delay (expressed in samples) does not depend on f_s and Equation 3.3 becomes:

$$D = \frac{N-1}{2}. \tag{3.4}$$

This property can be observed using the function *grpdelay* from MATLAB. f_s is still necessary to obtain a group delay expressed in seconds. In our case, the group delay for the three filters, expressed in samples, is roughly equal to M. Note that the N in Equations 3.3 and 3.4 is not the order of the approximating polynomial but the order of the filter, which is equal to $2M+1$ (also known in SG filters as the frame length).

3.2.2 Differentiation

The differentiation filter is placed right after the smoothing one. As the name suggests, the differential operation is applied to the input data. In this thesis, the information coming from the picture has a shape similar to that of a Gaussian. Hence, the typical profile of the function observed at the output of this filter will be that of a function with a maximum, then a zero and finally a minimum. The position of the zero corresponds to the one of the maximum in the unfiltered data (plus the group delay). The search for the zero is tasked to the next unit. When noise is present, the input function might assume a shape which is fairly distant from that of a Gaussian curve. For instance, in Figure 3.3a, two peaks are visible indicating significant noise right next to the laser line. In these cases, more than one zero can be detected at the output, however, only one is correct. This unit is again implemented using a SG filter with a different set of coefficients that are still generated by the function *sgolay*. All the properties and parameters related to FIR filters and discussed in the previous subsection apply here as well. Nevertheless, due to the odd symmetry of the coefficients, filters implementing odd order derivatives produce a null coefficient in the middle of the array. The advantage of such peculiarity is that the filter's chain requires one less multiplier and one less adder. However, this also alters the frequency response by introducing a zero in the origin ($-\infty$). On top of that, zeroes with opposite signs acting like poles make the first derivative filter look like a bandpass filter. Figure 3.4 shows an example of a typical impulse response. It can be seen that every frequency component is attenuated to some extent as the magnitude peak lies completely below $0dB$. The f_{cn} can be computed on both sides of the main lobe by reaching the $-3dB$ mark relative to the point of minimum attenuation.

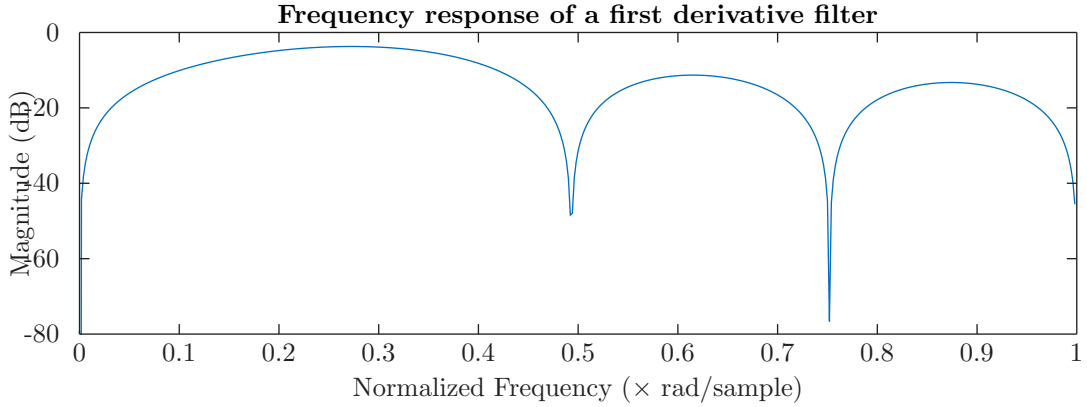
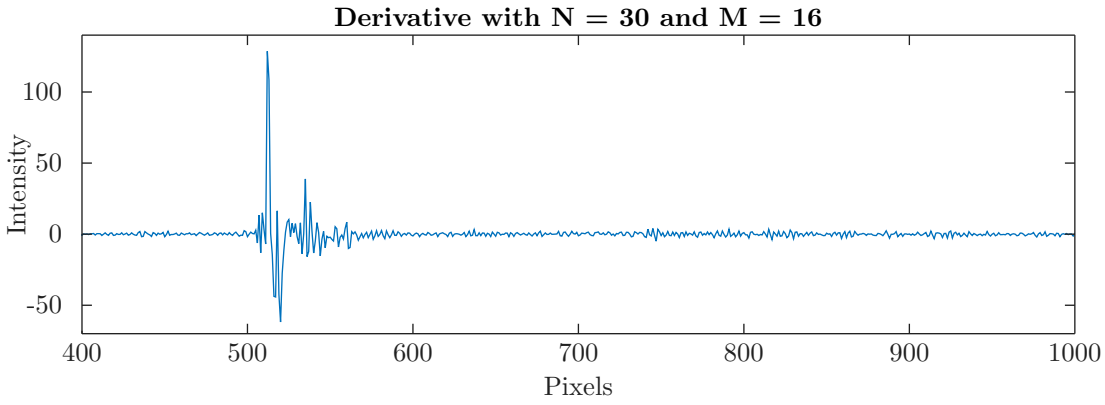


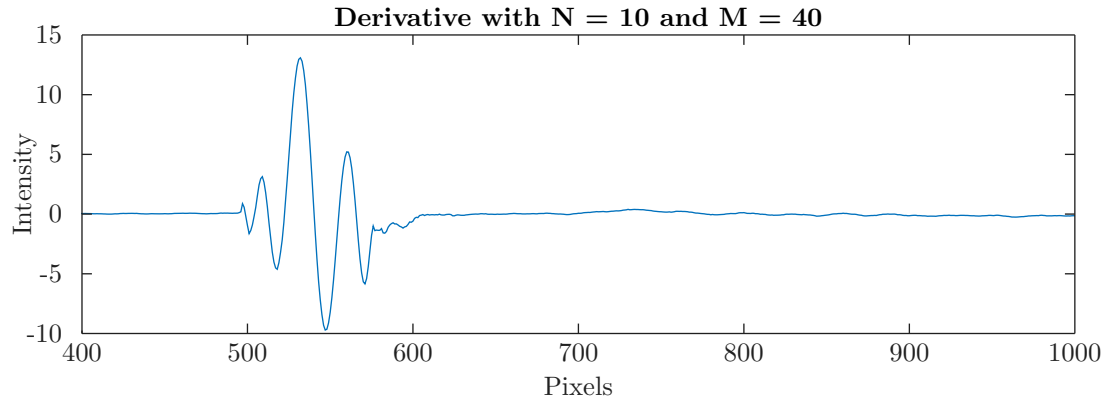
Figure 3.4: Typical first derivative frequency response

In Figure 3.5, the signals coming out of the same three differentiation filters seen before are shown. The input signal is also equal to the one previously employed and it is sent in directly with no smoothing in front. By

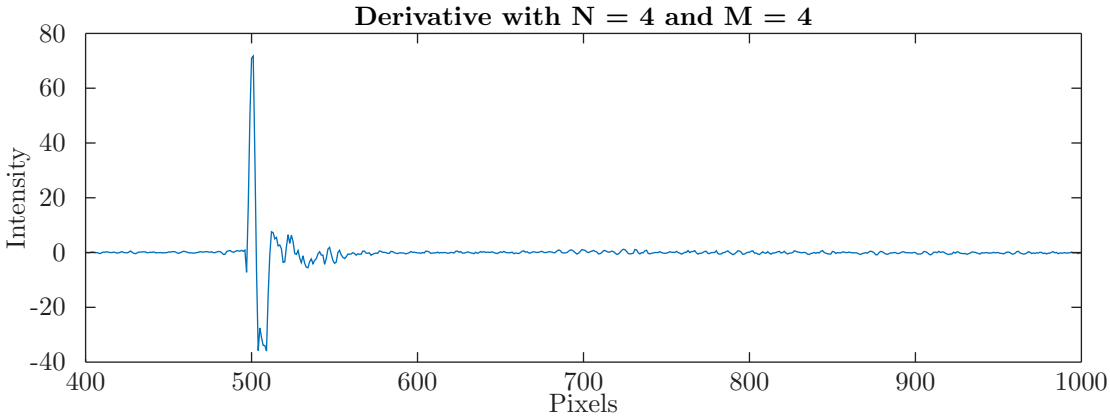
doing so, it is possible to observe that this kind of filter is also able to smooth the data to some extent, while, at the same time, performing the derivative. The signal in Figure 3.5a appears very noisy. It is the output of the filter having $N = 30$ and $M = 16$. An absolute maximum and an absolute minimum can be identified. Ideally, only one zero should lie between them but in this case there are three. Besides them, due to noise presence, many more zeroes exist along the signal. However, it will be shown later that these anomalies can be effectively identified and discarded during the zero-crossing computation. As for the three zeroes in the middle, identifying the correct one is much more intricate. Figure 3.5b shows instead the filter obtained by setting $N = 10$ and $M = 40$. As the bandwidth is narrow, noise is almost non-existent. It appears as if only one zero is present between the two absolute extreme values. The problem with this solution, however, is that the maximum and the minimum are notably attenuated and thus they are not easily distinguishable from the other relative peaks and valleys. This means that a further computation might reveal the presence of more than one zero. The last picture, Figure 3.5c, shows a more balanced solution where some noise is filtered while the main extreme values are still way above the rest. With this configuration, it is expected that only one zero will be detected.



(a) Differentiation filter with very wide bandwidth



(b) Differentiation filter with very narrow bandwidth



(c) Differentiation filter with reasonable bandwidth

Figure 3.5: Output of three differentiation filters with different bandwidths

If instead differentiation is applied right after smoothing, more combinations are possible. However, using one or both filters with extreme bandwidths is never a good idea. Combining two $N = 4, M = 4$ filters seems like the most sensible solution. Figure 3.6 shows this result, while Figure 3.7 shows the Simulink model consisting of smoothing and differentiation filters connected in series used to simulate their behaviour.

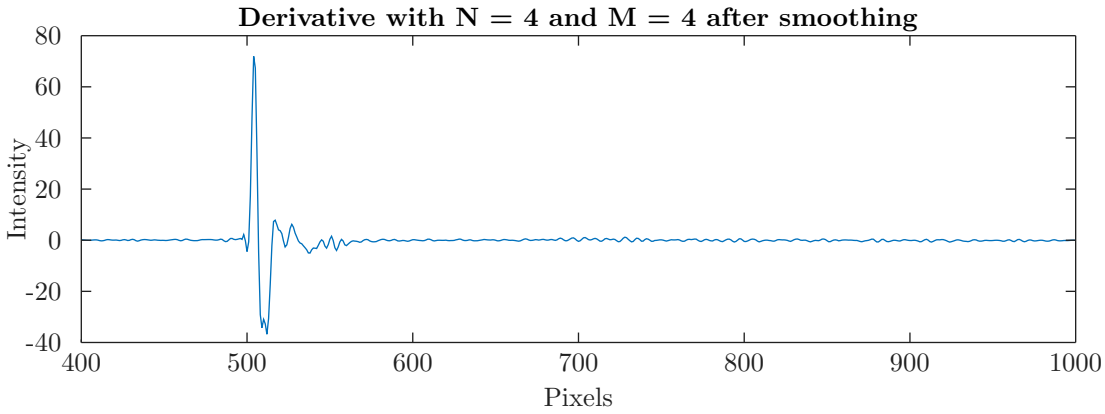


Figure 3.6: Output after a smoothing-differentiation filter chain

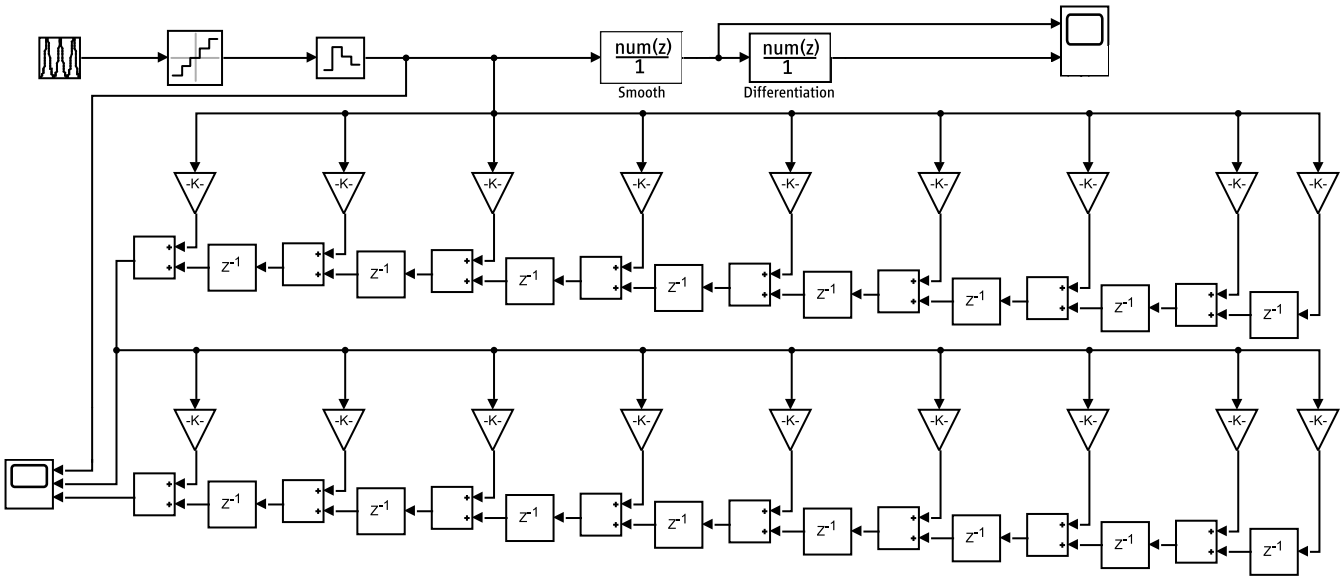


Figure 3.7: Schematic of the two filters in Simulink

One last interesting properties about SG filters is explained in [32]. There exists one particular configuration that allows for a great circuit simplification. When $N = 2$, the coefficients for the first derivative follow a straight line as shown in Figure 3.8. As expected, symmetry is odd and the tap in the centre is 0. Moreover, each of the coefficients is a multiple of the smallest one that is different from zero. By exploiting this feature, it results that only one multiplication is needed for each cycle, whereas a multi-operand adder can be used for addition. At the time of this paper's publication, computers were not nearly as powerful as they are in the present. Hence, this solution reduced computation time significantly. Today, this optimization would not make a difference when dealing with most software applications but it can be extremely useful in hardware as multipliers are the most complex blocks in a filter. In Chapter 5, this implementation is discussed from a hardware point of view. A considerable reduction in resources and thus area is expected for this kind of filter. Figure 3.9 shows a scheme describing this principle. The only multiplier present is the block on the right side with gain K, where K is the value of the smallest positive coefficient. The rest of them is derived from this one. The other amplifiers are still multipliers in software, however, they can be turned into simple shift operations when switching to hardware.

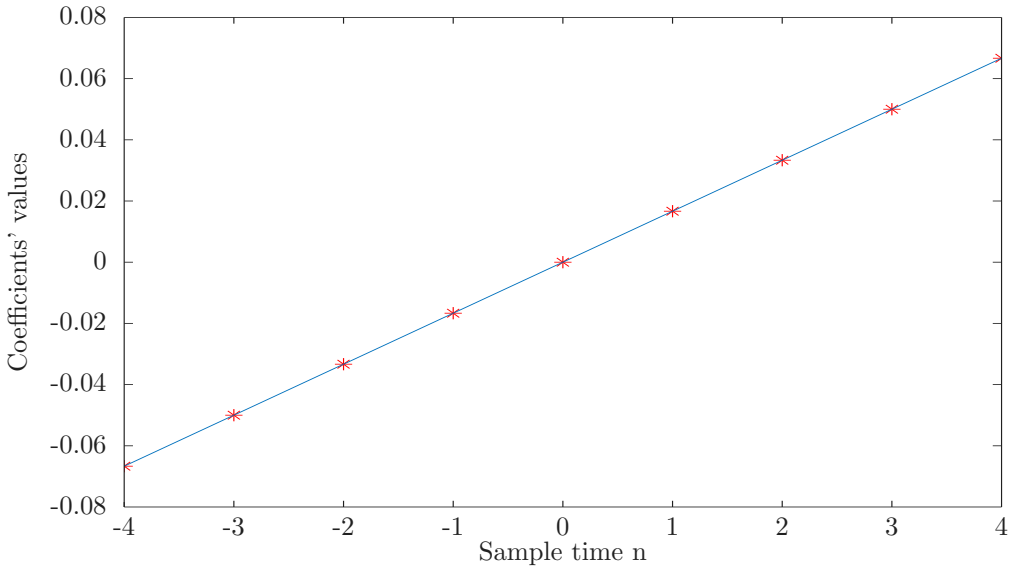


Figure 3.8: Impulse response h of a S-G differentiation filter with N = 2 and M = 4

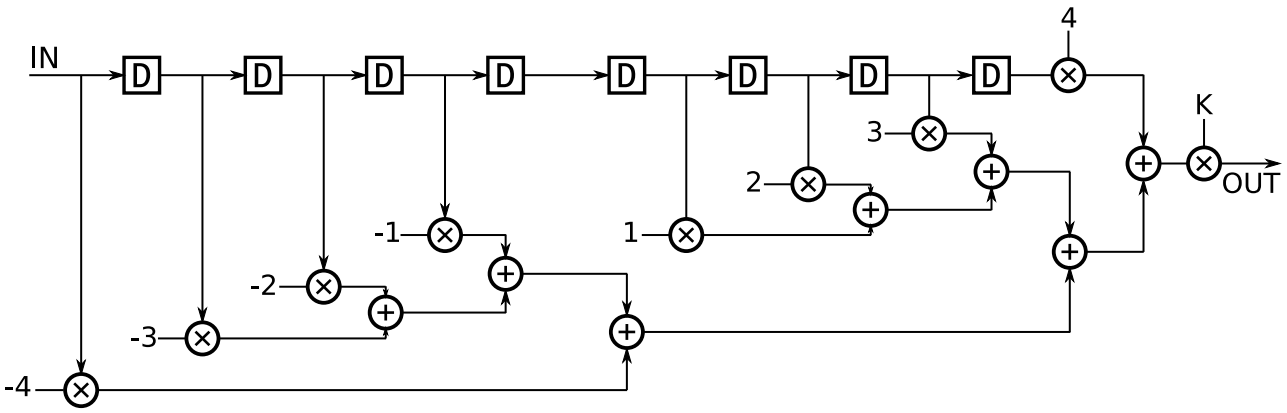


Figure 3.9: Schematic of a SG filter with $N = 2$ and $M = 4$ implementing first derivative

3.3 Zero-crossing algorithm

The zero-crossing detector is the most complex block in the pipeline. Given an image coming from a camera pointed at an object, columns are extracted and elaborated one after the other. For each column, only one peak corresponding to the laser line is expected. When this signal is differentiated, the function at the output will cross the x-axis ideally once. The detector's main purpose is to find the position (in pixels) of the points that reach the '0' mark on the y-axis. The mapping of a pixel position to real world coordinates is not discussed in this thesis.

As the signal is discrete, chances are that a point corresponding to the maximum intensity is not equal to precisely zero after differentiation, unless the image is highly overexposed. However, this is not supposed to happen. Thus, the first obstacle to overcome is the need of some kind of interpolation able to extract the right position of the zero even when no point is equal to exactly '0'. On top of that, high levels of noise can create more than one zero in the output signal. The second obstacle, hence, arises from the necessity to be able to discern legitimate zeroes from invalid ones. When noise levels are comparable to signal levels (e.g. noise peak is almost as high as the positive peak that stems from the derivative of the Gaussian), more than one zero can be found. At this stage, there is no way of knowing which value is the correct one. However, in the majority of cases the algorithm is able to find the valid zero while leaving out the others. The algorithm needs to find the zero between the absolute maximum and the absolute minimum. Ideally, only one maximum and one minimum

should exist, while the rest of the signal is flat. In reality, there are other relative extreme values. As the height of the absolute maximum is not known, theoretically each one of them should be inspected. Luckily, one can rule some maxima out by placing a threshold below which, anything is considered noise. This also sets a limit on the minimum intensity the laser line has to possess in order to be detected by the circuit. A negative threshold (NT) is placed for the minimum. As the original signal is a Gaussian, the derivative will always have a shape for which the maximum comes first, then the 0 and finally the minimum as shown in Figure 3.10. This means that when the computation starts, only the positive threshold (PT) is checked because a maximum is expected at some point, not a minimum. The first sample above the threshold signals the presence of a peak nearby, which is probably not noise. In reality, the algorithm is not waiting for the peak but rather for the zero. When two consecutive samples having two different signs are detected, it results that the zero lies between the two numbers. To compute the correct position, a simple model based on basic geometry is proposed. As the two points hiding the zero are normally close to each other, the curve connecting them can be approximated to a straight line. If instead they are far apart, it is possibly due to a function varies quickly. Rapid monotonic variations usually indicate that noise is not present in that particular section and thus, the straight line approximation is still acceptable. Figure 3.10 shows an example of an ideal curve when no noise is present.

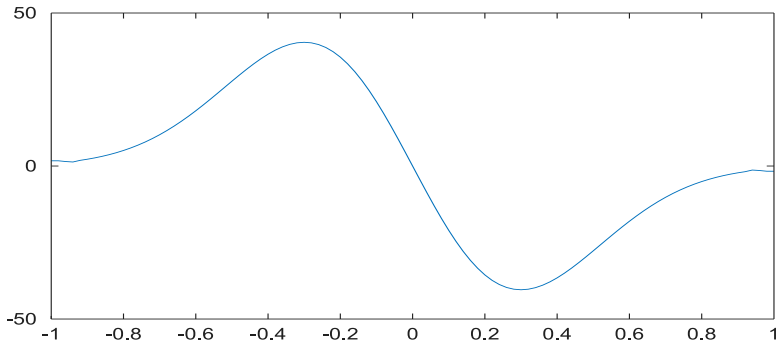


Figure 3.10: Output of a first derivative filter when an ideal Gaussian is used as input

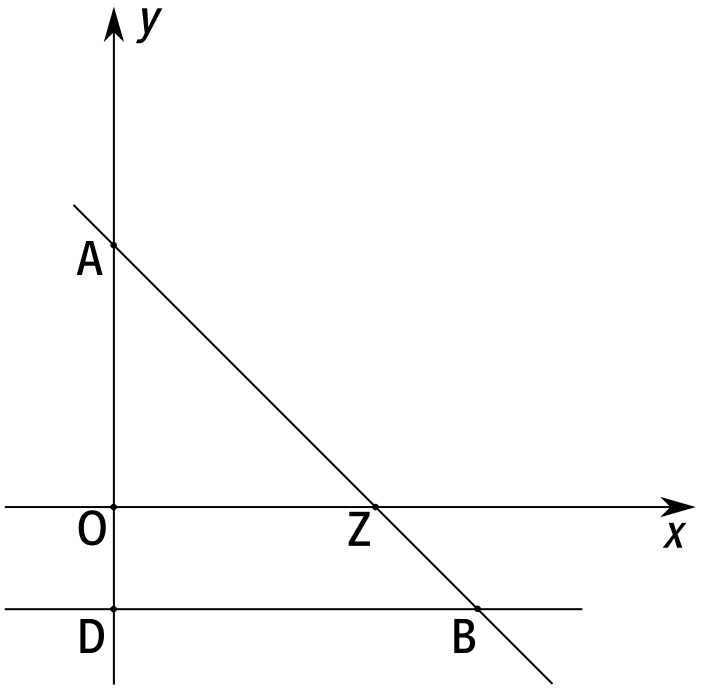


Figure 3.11: Simple scheme used to derive the equation needed to locate the zero

Figure 3.11 shows a representation of the principle regarding the calculation of the position of the zero. Point A represents the positive value, while point B the negative. Between them, the zero must be found. The two points are connected through a straight line that intercepts the x-axis where the zero (point Z) lies. O represents the origin, while D is the projection of segment $\overline{ZB}$ on the y-axis. Due to the intercept theorem, it can be seen that triangle $\triangle ZOA$ and triangle $\triangle BDA$ are similar as they have the same angles. This entails that the following relationship between the two holds:

$$\frac{\overline{AO}}{\overline{OZ}} = \frac{\overline{AD}}{\overline{DB}}. \tag{3.5}$$

As the position is computed as an offset distance from sample A, the unknown quantity to obtain is the segment $\overline{OZ}$. By working on Equation 3.5, we get:

$$\overline{OZ} = \overline{AO} * \frac{\overline{DB}}{\overline{AD}}. \tag{3.6}$$

Now, $\overline{AO}$ is equal to the value of the sample in A, while $\overline{AD}$ is the difference between the sample in A and the one in B. Given that B is always negative, $\overline{AD}$ is always positive. $\overline{DB}$ instead, is the distance between two consecutive samples. As the signal is time-discrete, this length of this segment is set to 1. Thus, the final equation boils down to:

$$\overline{OZ} = \frac{\overline{AO}}{\overline{AD}}. \tag{3.7}$$

Given that O is equal to 0, 3.7 becomes:

$$\overline{OZ} = \frac{A_y}{A_y - B_y},\ A_y > B_y. \tag{3.8}$$

For this operation one subtractor and one divider are required. The position must be calculated as accurately as possible, therefore the zero is expressed in software in a double format. In hardware, to increase precision, the bit-width of the divider is expanded by means of subpixeling. Another correction in the position involves the delay of the filter, which is constant, and also depends on whether smoothing was performed earlier on or not. If both filters with $N = 4$ and $M = 4$ are used, the total delay is roughly 8 samples. It should be noted that the zero is considered only if one sample is positive and the next one is negative as that is the only case where the zero can originate from the signal rather than from the noise. Zeroes resulting from an increasing section of the function are not counted. Right after the zero a minimum is expected. Thus, when the NT is crossed, the program stores the value of the last zero and then keeps processing other samples waiting for a possible new value that crosses the PT. A situation in which the NT is crossed before the positive one is not expected, if it occurs, the zero between the relative extreme values is not considered. It is important to note that the two threshold are used to achieve hysteresis. Once the PT is activated, the algorithm will find all the zeroes in between until the negative limit is crossed. Multiple zeroes between the two extreme values can arise when the original signal is saturated to maximum intensity for a certain number of consecutive samples (i.e. the laser line is wide). In this case all of the singularities are considered and an average is computed to get the middle position. All the zeroes encountered before the positive edge or after the negative one, are not counted. The thresholds are chosen empirically and might depend on the application. They cannot be too big or else even a laser line intense enough to be visible is ignored in the computation but they also cannot be too small or the noise will produce false positives. The hardware version allows the user to set and modify this value from the outside. To make the algorithm more robust, a second positive threshold is used, which is much higher than the previous one. This works because normally the laser line's intensity is well above the noise around it. In these cases, it is nearly impossible for the noise to go beyond this limit and thus only one zero is returned. Once the high positive threshold (HPT) is passed and the conversion is completed, another conversion is possible only if the same threshold is exceeded: the low positive threshold (LPT) does not work anymore.
Even with two thresholds, at least two cases exist, in which more than one zero can be detected. When the signal goes beyond the HPT, there is a tiny chance the noise also goes beyond that. If that happens, the signal and noise are indistinguishable and thus both zeroes are counted. The other situation takes place when the signal is not intense enough and does not go beyond the HPT. In that circumstance, the noise that is able to go beyond the LPT can also be responsible for the generation of falsely identified zeroes. Both cases can only happen if the noise is located far from the laser line as the NT must be passed before starting a new computation. The misidentified zeroes far from the laser line are filtered at a later time by a spline fitting algorithm realized in software. However, this computation is not discussed here.

Figures 3.12 to 3.15 show the working principle of an algorithm that employs two thresholds. The code is applied to columns coming from real images. In the first example, the effect of the HPT is demonstrated (Figure 3.12). The straight lines correspond to the three thresholds, the zero-level is also highlighted. The red dots represent the crossing of the LPT, the black one is for the HPT, the x represents a zero of interest, but not necessarily valid and the asterisk indicates when the NT is exceeded. From the image, it can be seen that the signal remains within the boundaries marked by the LPT and the NT until the LPT is passed. Right after that, the HPT is also crossed. Due to the discrete nature of the signal, the first points to go beyond the thresholds can be far from the actual lines. After the first peak, the zero position is estimated and once the NT is crossed, the algorithm stops searching for zeroes. Normally, a new computation would start once the LPT is crossed. In this case, however, a new zero can be detected only when the HPT is passed. Therefore, the two following zeroes are ignored. If instead an invalid zero lies before the actual signal peak, the noise will be included in the computation and the algorithm will return multiple locations. This possibility is shown in Figure 3.13.

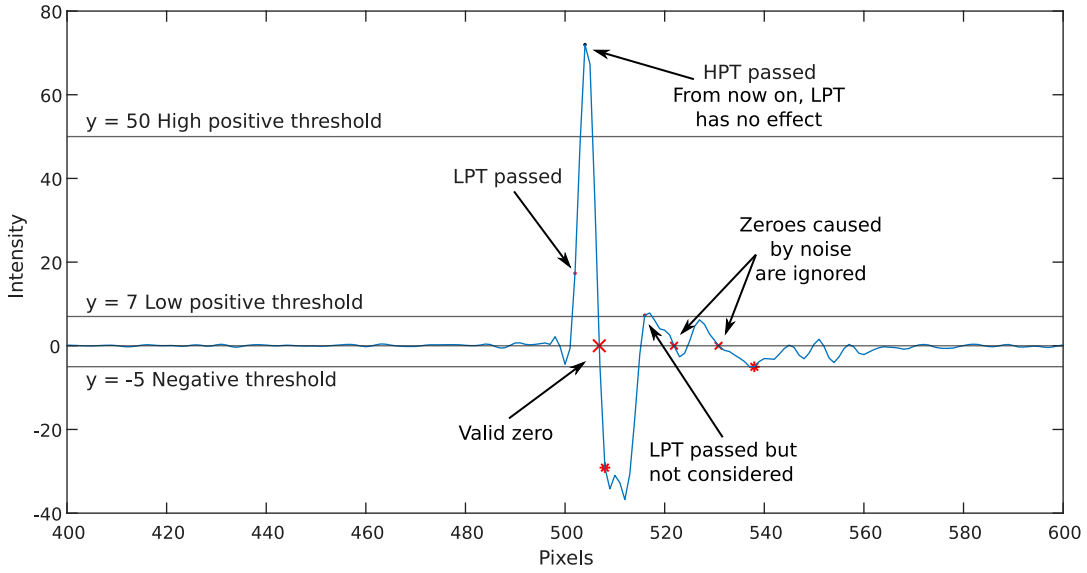


Figure 3.12: The second threshold effectively filters the noise

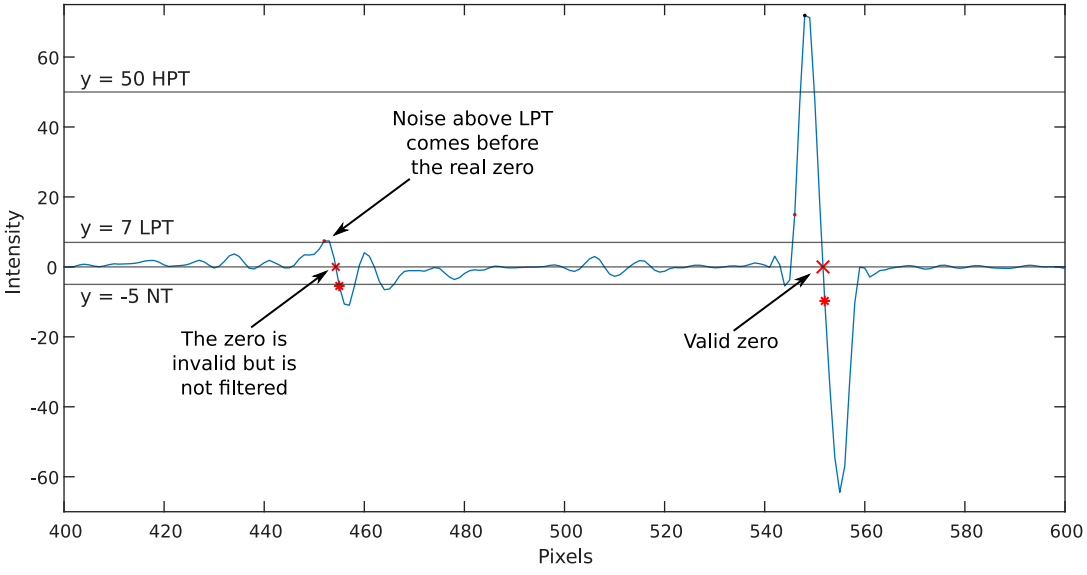


Figure 3.13: The second threshold is exceeded after the noise, which is not filtered

Figure 3.14 shows the effects of a wide laser line on the derivative of the signal. Three neighbouring zeroes

are visible. Since no threshold is crossed in between those points, it is safe to assume that all three are valid. While picking the first zero and discarding the other two might be a feasible strategy, calculating an average value seems to be more accurate. For this reason, the algorithm takes into consideration all zeroes until the NT is passed. Following that, the arithmetic mean is computed. The new value roughly falls in the middle of the laser line. Finally, Figure 3.15 illustrates why a LPT is sometimes necessary. In the column, the laser line is still faint and the peak does not reach a significant level. As a consequence, it is not able to exceed the HPT and the coming zero would be rejected were it not for the LPT. In this case, the actual laser line and any noise able to surpass the threshold are indistinguishable. However, it appears that noise is directly related to the intensity of the laser line. Thus, dim laser lines normally have no noise at all, as can be seen from the flatness of the region around the perturbation caused by the peak.

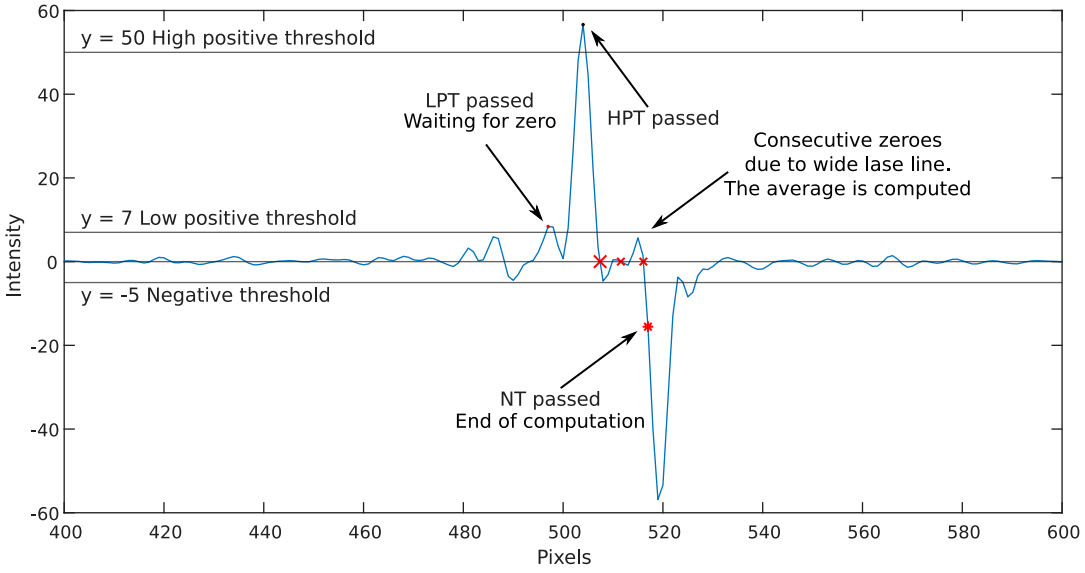


Figure 3.14: Situation in which a computation of the average is required

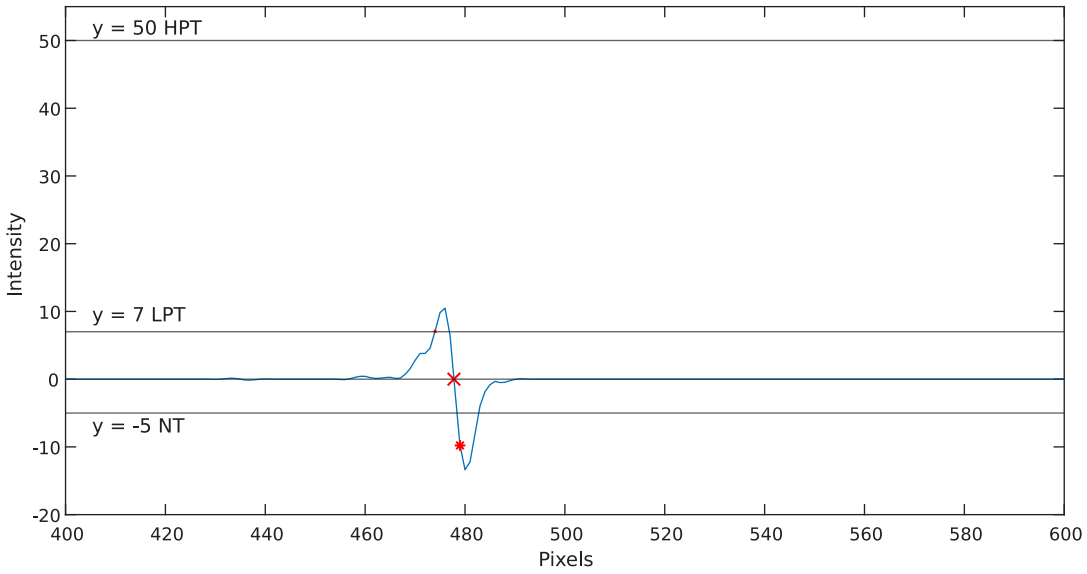


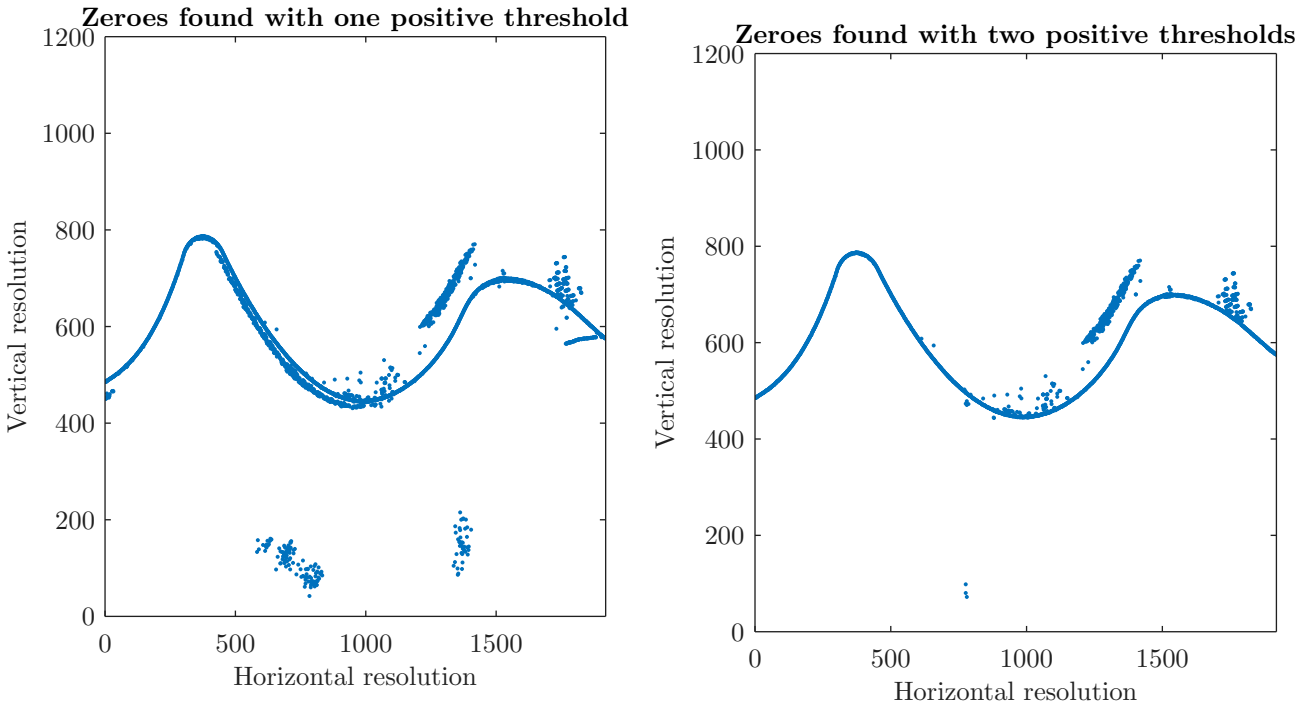
Figure 3.15: Faint laser lines do not go beyond the HPT

The software algorithm is visibly simpler than the hardware implementation that also involves a divider, counters, comparators, and everything is handled by an FSM. In Chapter 5 the circuit will be discussed. An

algorithm for the zero-crossing detector is shown in Algorithm 1. This code is applied to every column in the image after the initial processing carried out by the filters. The images used to the test the system all have a horizontal resolution of 1920 pixels, the vertical resolution varies. Algorithm 1 is the most complete and robust. Three threshold values can be distinguished: two positive and one negative. Flags are used to store information about past samples. This information is essential to understand what to do next. In the hardware version, flags are replaced by states managed by the FSM. When the LPT is exceeded, conversion flag is set only if the HPT was not already surpassed. This flag validates the calculation of the zero position, meaning that the coming zero is probably caused by the signal and not just noise. If a sample goes beyond the second threshold as well, the HPT flag is set. This variable is not cleared until the next column is processed effectively blocking more noisy zeroes from being detected. As the second threshold can hold a very high value, a zero preceded by samples above this number can be validated with higher confidence. Since noise hardly ever gets that high, one can assume that the zero is valid and no other occurrences are detected unless they go beyond this second threshold. This model can be used for a generic scanner that deals with different objects with varied profiles and intensities. A flow chart describing the algorithm responsible for simulating the complete pipeline (i.e. filters and zero-crossing detection) is shown in Figure 3.21. The software model of the whole system is implemented using MATLAB. Figure 3.16b shows an example of this code with a very noisy image (Figure 3.17). By comparing it with the version with only one threshold in Figure 3.16a, a problem with this algorithm is visible. The HPT is able to filter all the noise only after it has been passed and the appropriate flag has been set. That is why only the side below the laser line is clean while the upper side is not affected. This problem can be fixed by running the program twice and processing each column first from top and then from bottom. Alternatively, the zeroes that go beyond the HPT can be selected once the computation is over. However, this is not always necessary. Multiple zeroes can be found and tolerated as long as the laser line is also very well visible and continuous. The zeros generated by the noise can be detected and ruled out at a later stage, as discussed previously. Thus, a simpler solution, using only one positive threshold and one negative is acceptable. A circuit can largely benefit from switching to a simpler solution, especially in terms of resources and complexity.

Algorithm 1 Zero-crossing algorithm

```
for index from 0 to vertical image resolution do
    if intensity[index] > low positive threshold then
        if intensity[index] > high positive threshold then
            set high positive threshold flag
            set conversion flag
        else if high positive threshold flag is NOT set then
            set conversion flag
        end if
    else if intensity[index − 1] < negative threshold AND conversion flag is set then
        set commit flag
    end if
    if conversion flag is set AND sample(n) < 0 AND sample(n − 1) > 0 then
        zero position = zero position + sample(n−1) / (sample(n−1)−sample(n)) + delay correction
        increment zero counter
    end if
    if commit flag is set then
        compute average zero position
        store average zero position
        clear conversion flag
        clear commit flag
        clear zero position
    end if
end for
```



(a) Zeroes found by a detector using only one positive threshold

(b) Zeroes found by a detector using two positive thresholds

Figure 3.16: Comparison of two zero-crossing detection solutions: each point plotted is a zero

Algorithm 2 shows how the pseudo-code simplifies when one threshold is removed. In particular, the number of branches is reduced as one of the flags completely disappears. If instead the object scanned by the system is known, and it always produces a laser line that is very bright and present through each column, as in Figure 3.17, noise can completely be filtered simply by choosing a single PT that is high enough. The same noisy images in Figure 3.16 appear in Figure 3.18 spotless even though only one threshold is used. This happens because all the noise has an intensity well below the threshold. This method cannot be applied for an image such as the one in Figure 3.19 as the laser line at the edges is faint and, in those columns, it be treated as noise and ignored.

Algorithm 2 Zero-crossing algorithm with one positive threshold

for index from 0 to vertical image resolution **do**
  if $intensity[index] > positive\ threshold$ **then**
    set conversion flag
  else if $intensity[index-1] < negative\ threshold\ AND\ conversion\ flag\ is\ set$ **then**
    set commit flag
  end if
  if $conversion\ flag\ is\ set\ AND\ sample(n) < 0\ AND\ sample(n-1) > 0$ **then**
    $zero\ position = zero\ position + \frac{sample(n-1)}{sample(n-1)-sample(n)} + delay\ correction$
    increment zero counter
  end if
  if $commit\ flag\ is\ set$ **then**
    compute average zero position
    store average zero position
    clear conversion flag
    clear commit flag
    clear zero position
  end if
end for

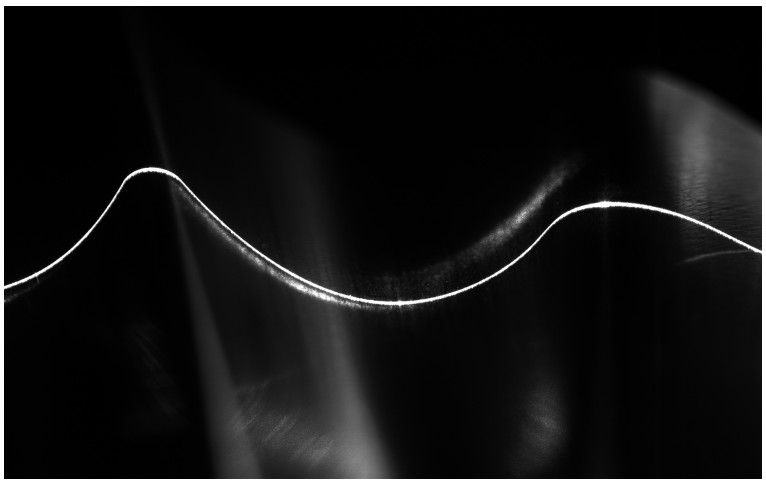


Figure 3.17: Noisy image with a continuous laser line

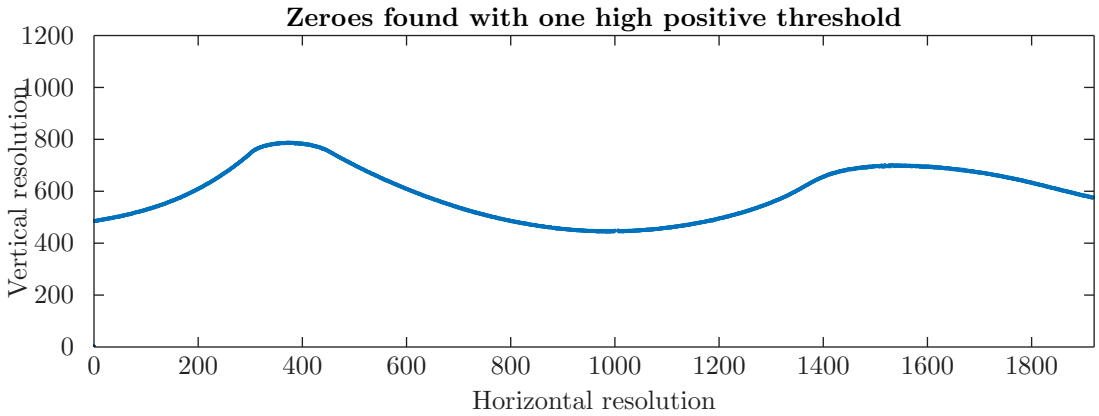


Figure 3.18: Result after using one very high positive threshold

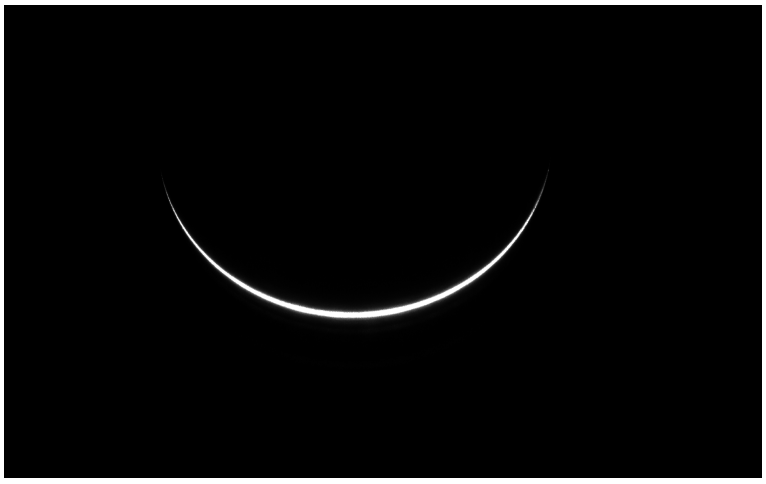


Figure 3.19: In a sphere, the laser line is not visible in each column

Another possibility that greatly simplifies the algorithm consists in removing the calculation of the average. By doing this, the NT also becomes superfluous as now every zero found is stored, as long as the NT is passed. This simplification is shown in Algorithm 3. Figure 3.20 shows an example where the average is needed. As a lot of consecutive samples are constant at maximum intensity, it is reasonable to choose a position for the maximum that falls right in the middle of the plateau. This is achieved by computing the average. Without it, the zero would

be found at the beginning of the peak. A circuit that does not use this feature might still produce acceptable results depending on the width of the laser line. For objects that do not produce a wide laser line due to low exposure time, this simplification can be implemented.

Algorithm 3 Zero-crossing algorithm with no average calculation

for index from 0 to vertical image resolution **do**
  if $intensity[index] > low\ positive\ threshold$ **then**
    if $intensity[index] > high\ positive\ threshold$ **then**
      set high positive threshold flag
      set conversion flag
    else if high positive threshold flag is NOT set **then**
      set conversion flag
    end if
  end if
  if $conversion\ flag\ is\ set\ AND\ sample(n) < 0\ AND\ sample(n-1) > 0$ **then**
    $zero\ position = \frac{sample(n-1)}{sample(n-1)-sample(n)} + delay\ correction$
    store zero position
    clear conversion flag
  end if
end for

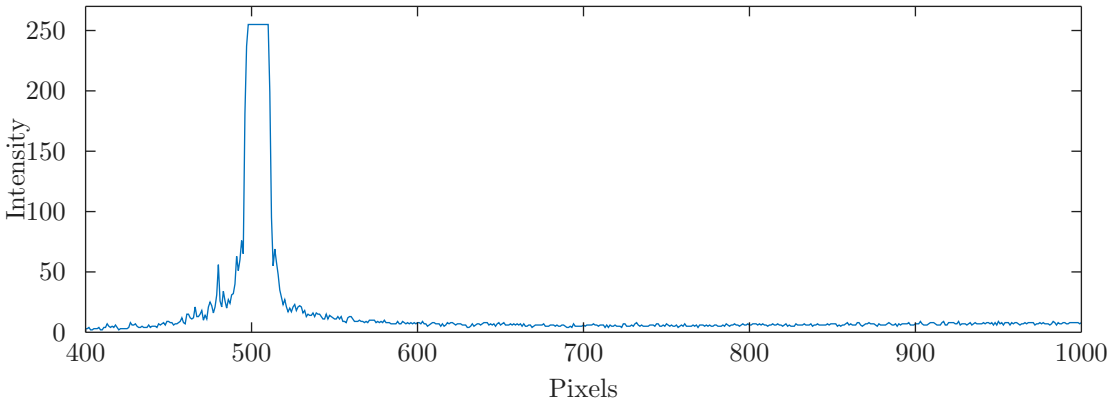


Figure 3.20: Column coming from an image highlighting a thick laser line

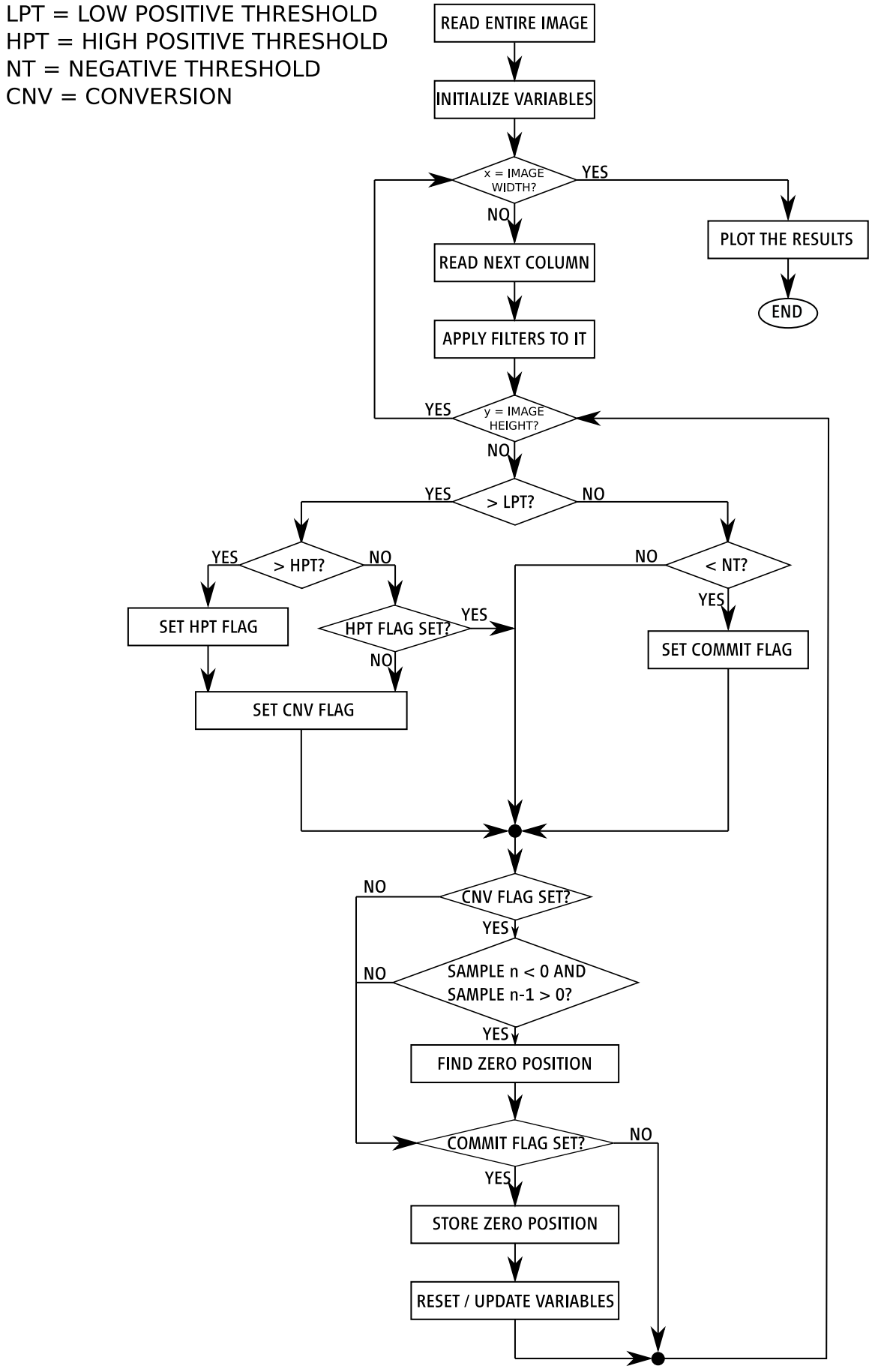


Figure 3.21: Flow chart

CHAPTER 4

Approximate computing algorithms

In this chapter, the algorithms used to produce inexact software versions of arithmetic circuits are discussed. They include adders, multipliers, and dividers. A general overview is presented here, while the actual hardware implementation chosen for this project is clarified in Chapter 5.
The code for these units is created and simulated on MATLAB. Normally, simulation regarding the filters and the zero-crossing detector can be carried out on MATLAB with infinite precision using the *double* data type. However, as approximate algorithms are intended for hardware applications, some of their software models must deal with binary numbers. Thus, functions that convert from binary to decimal and vice versa are included in the modules. To have a standard word-length (WL) and to make simulation faster, a fixed-point format with a 16-bit WL is used. Of those, 7 bits are reserved for the fractional part and the remaining 9 are used for the integer part. As the pipeline deals with signed numbers, the MSB stores the information about the sign. The smoothing filter handles integer data ranging from 0 to 255. However, two of its coefficients are negative. Therefore, 9 bits are required for the integer part. This range can be reduced at the output of the differentiation filter as one less bit is needed, which can be gifted to the fractional part.
When it comes to coefficients, a 7-bit fractional part can produce some issues. Due to the unitary gain of the filters, the taps' value is always less than '1'. Cramming these numbers into a 7-bit slot can by itself cause some significant loss of information when compared to the original coefficients, even in the event an exact filter is used. While the system still works, the limited precision constitutes the first kind of approximation. It will be explained later that the fractional part is expanded when subpixeling is applied. The built-in functions *fi* and *sfi* provided by MATLAB can convert a decimal number into its fixed-point representation with a desired WL. Along with them, custom functions that deal with fixed-point conversion are used. Bitwise operators such as OR, AND, and arithmetic shift can also be found.

4.1 Lower-part-OR adder

A possible technique to design an inexact adder involves the approximation of the FAs that manage the LSBs. [60] and [61] belong to this category as well as the lower-part-OR adder (LOA), which is introduced in [43]. As the name suggests, this circuit splits the computation of the sum in two. The n least significant bits (lower part) are approximated by replacing FAs with OR gates as shown in Figure 4.1. Therefore, the approximation is carried out at gate level, rather than at transistor level. A bitwise OR operation between the two operands is applied in the lower part. The m MSBs (upper part) are computed by an exact adder with no constraints on the choice of architecture. As $m + n$ must be equal to the total bit-width of the input numbers, their sum is constant. By changing one of them, different approximation depths can be achieved. Given a fixed WL, the precision is expected to decrease as n rises, whereas power and area are supposed to drop. To mitigate the error, an additional two-input AND gate is added to generate the carry-in bit for the accurate part. Its output assumes the value of '1' in the event the most significant approximate bits of the two operands are '1'. Without it, a systematic error arises when dealing with small numbers. This happens when the magnitude of the operands is comparable with the size of the lower part (n bits) since it affects the MSBs of the result. For input data that display a full dynamic range, this problem is diminished as the MSBs overshadow the error introduced by the carry-in bit of the upper part, which has a negligible weight.

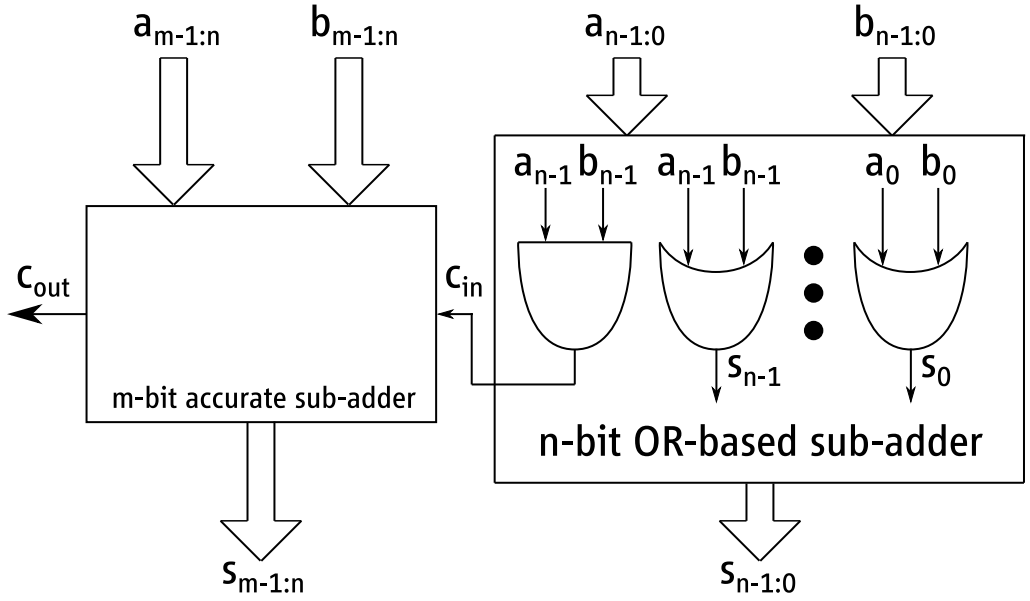


Figure 4.1: Implementation of a LOA

In [43], Mahdiani et al. computed the error probability for the circuit. Equation 4.1 shows that the ER does not depend on the WL but only on n, also called lower-part length (LPL). This implies that a large LPL is always responsible for a high error rate. However, as the error affects the LSBs, its magnitude does not constitute a problem in many applications.

$$ER_{LOA} = 1 - \left(\frac{3}{4}\right)^{LPL}. \tag{4.1}$$

Two noteworthy features belonging to this approximate adder are its maximum error range and its bias. Due to its nearly symmetric output error range, as shown in Equation 4.2, the LOA is suitable for both signed and unsigned addition. A consequence of this symmetry is reflected by the fact that the circuit is almost unbiased as error with opposite signs compensate each other. Mahdiani et al. estimated an average error of -0.25. This number depends neither on the WL nor on the LPL.

$$\begin{aligned} MAX_{LOA} &= 2^{LPL-1} - 1, \\ MIN_{LOA} &= -2^{LPL-1}. \end{aligned} \tag{4.2}$$

As for the MSE, it grows exponentially with LPL. Equation 4.3 highlights this relationship:

$$MSE_{LOA} = 2^{2*LPL-4}. \tag{4.3}$$

Finally, in [39], it is shown that the delay for an n-bit LOA is roughly equal to:

$$D_{LOA} = O(log(n - LPL)). \tag{4.4}$$

Also in [39], Jiang et al. compared this design to other approximate adders. They found that, in relation to other units, the LOA features low MRED and average error, whereas it displays one of the highest ER. As loops are not present in FIR filters, the average error finds no use in this thesis. The LOA also performed very well in terms of PDP and ADP.

4.2 Error-tolerant adder type II

The error-tolerant adder type II (ETAII) is an approximate adder introduced by Zhu et al. in [62]. As it is composed of smaller adders working in parallel, it belongs to the class of segmented adders along with [63, 64, 65, 66]. The ETAII represents the evolution of a previous version called error-tolerant adder type I (ETAI). Similarly to the LOA, this kind of adder divides the calculation into two parts. An upper part carrying the MSBs, which is accurate, and a lower part, which is subjected to approximation. Again, aside

from a fixed WL, no constraints are imposed of the bit size of the inexact section. To understand how this circuit works, we can start from the middle, i.e. from the point where the two partitions meet. On the left, an accurate sum is carried out starting from the LSB and proceeding leftwards, as one would normally do. For this operation, any ordinary adder can do the trick. As for the right side, the computation moves from the MSB towards the right. These two parts work independently, thus producing a faster circuit. Moreover, delay is further reduced by relinquishing carry propagation in the inexact portion. To mitigate the error that emerges from this approximation, Zhu et al. came up with a special strategy. Starting from the left, each pair of bits belonging to the two operands is checked. If both A_i and B_i are equal to '1', their sum bit S_i is set to '1' and so are all the other sum bits to the right. Any other input bit combination generates a correct sum bit, much like a half adder with no carry-out bit. Figure 4.2 shows an example of this technique.

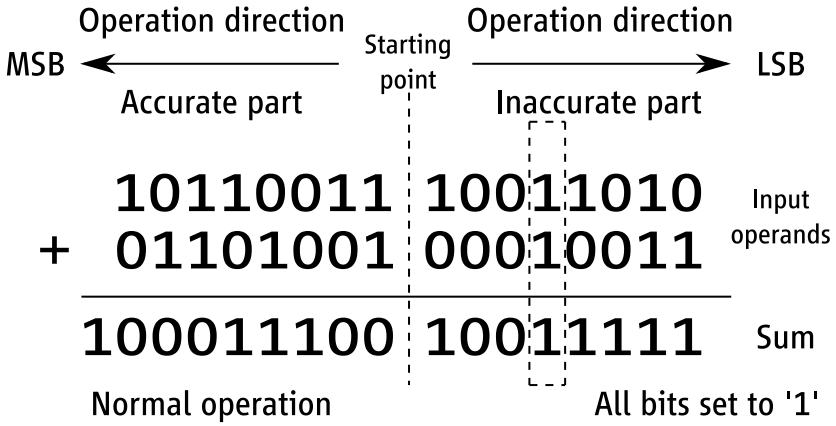


Figure 4.2: Example of addition performed by an ETAI

As stated in [62], the ETAI generates a large error when it comes to summing two small numbers that are contained entirely in the approximate part. The ETAII is proposed as a solution to this problem. Instead of completely discarding the carry propagation, this second adder improves precision by splitting the path into smaller units designated for carry propagation. This approximation works on the assumption that the worst case (i.e. the carry is generated at the LSB) rarely happens, thus a split-up chain is able to restrict the maximum error rate to a limited number of cases.
This new circuit is realized by diving the n-bit width of the operands into m blocks (with $m > 1$). As a result, each block manages n/m bits, which are handled by two different units: the carry generator and the sum generator. Figure 4.3 shows a scheme of the ETAII:

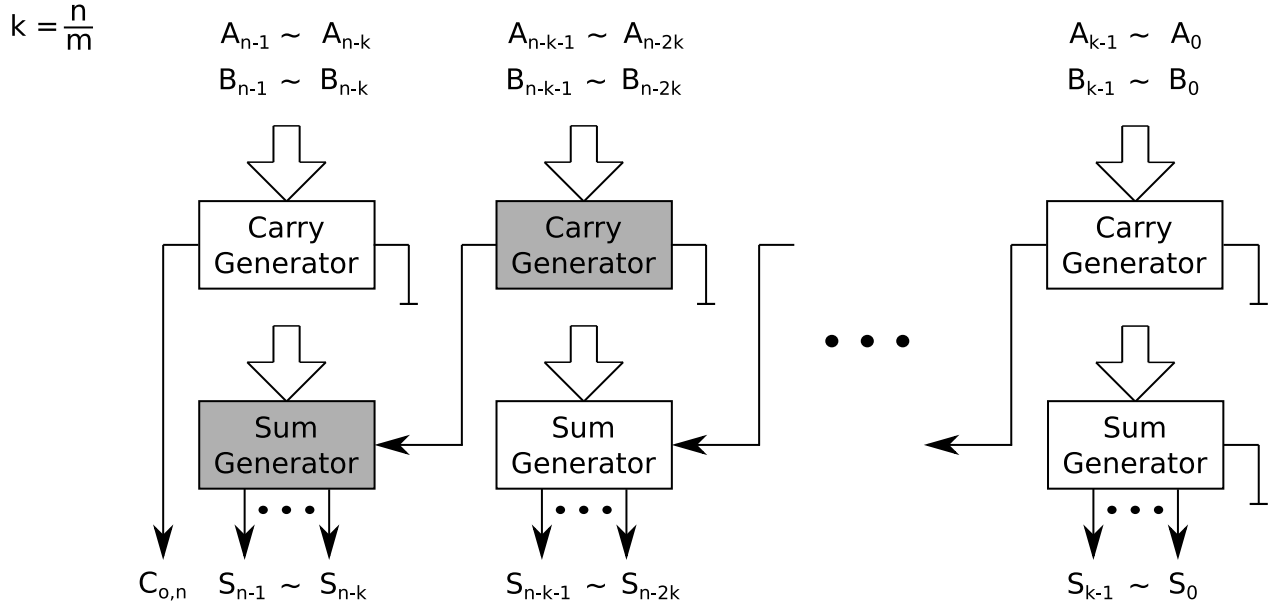


Figure 4.3: Schematic of an ETAII

The carry generator is responsible for the generation of the carry. As the chain is split, this circuit does not receive any information from other blocks, so the carry is propagated up until the next sum generator. This unit computes the output sum from both the input operands and the knowledge provided by the previous carry generator. By using this design, the longest path is $\frac{2n}{m}$ bits long, whereas the delay is reduced by $m/2$ times compared to a conventional adder. Breaking up the circuit also reduces the number of glitches caused by carry propagation, which negatively affect the overall power consumption. It goes without saying that the bigger m is, the shorter the carry path is. This produces a design that is faster but less accurate. On the other hand, small values of m result in a more robust circuit that approaches its exact equivalents. From the data gathered by Zhu et al., it seems that a size of 4 bits for each block represents a good compromise. This width is chosen for this project.
Lastly, the data also show that, while the ETAII dramatically improve accuracy when handling small number, the opposite happens for large input operands. Solutions to this problem include extending the carry propagation chain only for the MSBs. However, this kind of adder will not be discussed here.

4.3 Rounding-based approximate multiplier

The rounding-based approximate (RoBA) multiplier employs rounding techniques with the aim of expressing its operands as powers of 2. This method is presented by Zendegani et al. in [67]. Multiplication by numbers expressed as 2^n greatly simplifies the whole computation as now the operation can be implemented merely by left-shifting one of the operands. As a consequence, this design is area efficient and thus, power saving. The idea behind this approximate multiplier can be expressed by Equation 4.5.

$$A * B = (A_r - A) * (B_r - B) + A_r * B + B_r * A - A_r * B_r, \tag{4.5}$$

where A_r and B_r are the rounded versions of the two operands A and B. Equation 4.5 produces exact results. Given that $(A_r - A) * (B_r - B)$ is difficult to compute and its weight is not predominant, an approximation can be achieved simply by leaving this term out. In this way, Equation 4.6 is obtained:

$$A * B \approx A_r * B + B_r * A - A_r * B_r. \tag{4.6}$$

Now the multiplier must perform three multiplications. Even so, since it deals with numbers rounded to the closest 2^n, they can be plainly implemented as shift operations. Thus, the scheme for the final model, shown in Figure 4.4, make use of three barrel shifters, one adder and one subtractor.

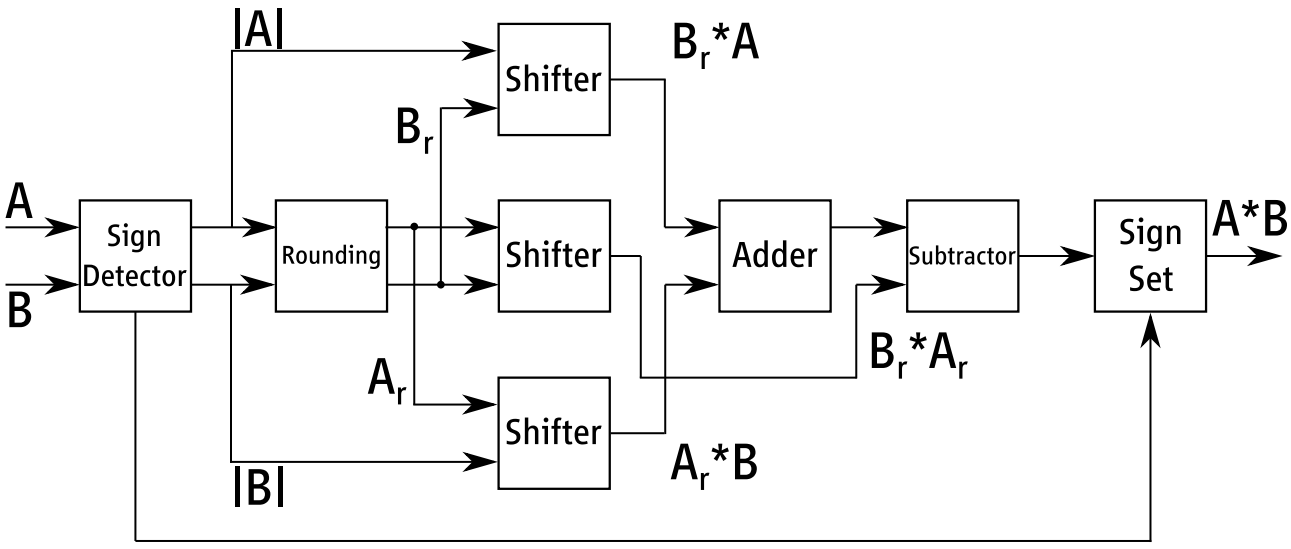


Figure 4.4: Schematic of a RoBA multiplier

The rounding block, as the name suggests, is responsible for approximating the input operands. The algorithm presented in [67] is suited for unsigned numbers as they are simpler to approximate. That is why negative numbers are first converted into positive. The "sign detector" keeps track of the signs at play and applies the correct one to the result at the output through the "sign set" block. Also in [67], the case where an operand finds itself exactly between two consecutive powers of two is examined. For instance, when either A or B are equal to $3 * 2^{n-2}$, where n is a positive integer greater than '1', the number can be either rounded up towards 2^n or downwards 2^{n-1}. In this event, the authors suggest that rounding to the larger power of two should be favored as it results in lower area occupation. The only exception occurs when the number to be rounded is '3' ($n = 2$). In this circumstance, the operand is scaled down to '2'. Considering that the choice has no effect on accuracy, the up-rounding approach is used also in this thesis, although the system is implemented on a different kind of hardware so the advantages might not be noticeable. The general rounding paradigm for the i-th bit of an n-bit operand A is shown in Equation 4.7:

$$A_r[i] = (\overline{A[i]} * A[i-1] * A[i-2] + A[i] * \overline{A[i-1]}) * \prod_{i=i+1}^{n-1} \overline{A[i]}, \tag{4.7}$$

where $*$ represents an AND operation and $+$ is an OR operation. From this equation it can be seen that $A_r[i]$ is one in two cases: when $A[i]$ is 1 and both the bits on its left (MSBs) and the one bit on its right are 0 or when $A[i]$ and all the bits on the left are 0, while the two bits on its right are 1. Bits at the boundaries of A undergo a slightly different rounding procedure as shown in Equations 4.8:

$$\begin{aligned} A_r[n-1] &= \overline{A[n-1]} * A[n-2] * A[n-3] + A[n-1] * \overline{A[n-2]}, \\ A_r[2] &= A[2] * \overline{A[1]} * \prod_{i=3}^{n-1} \overline{A[i]}, \\ A_r[1] &= A[1] * \prod_{i=2}^{n-1} \overline{A[i]}, \\ A_r[0] &= A[0] * \prod_{i=1}^{n-1} \overline{A[i]}. \end{aligned} \tag{4.8}$$

In order to be able to exploit the advantage of the 2^n format, input data must be positive. As shown in Figure 4.4, when dealing with signed numbers, a conversion must be carried out first. The appropriate sign of the output must be computed and a correction applied. To further approximate the multiplier, the two's complement conversion, normally implemented as $\overline{Y} + 1$, can be realized simply as a one's complement, or $\overline{Y}$. These extra blocks are not needed when unsigned numbers are handled.
Finally, due to the use of rounded numbers, only a limited amount of situations can occur at the input of the subtractor. Consequently, its logic function can be simplified as will be discussed in Chapter 5.
The output error for this multiplier can be expressed by the formula:

$$error(A, B) = \frac{(A_r - A) * (B_r - B)}{AB}. \tag{4.9}$$

By substituting the condition for the worst case error, it follows that the maximum error for both signed and unsigned multiplication is roughly equal to 11.1%. The maximum error occurs when both operands fall in the middle of the interval between two powers of two. In this case, the ED is at its peak. For an n-bit unsigned multiplier, the number of these intervals is equal to $n - 1$. Hence, the input combinations that can give rise to the maximum error are $(n-1)^2$. The maximum error rate can be computed by dividing this number by all the possible 2n-bit outputs and its inversely proportional to the WL. This rate increases for signed multiplication for which the possible combinations that produce the worst case error are $(2(n-2))^2$. On the other hand, when one of the input operands is equal to a power of two, the result generated by the circuit is exact. For an unsigned multiplier, there are $2(n+1) * 2^n - (n+1)^2$ possible combinations that yield a correct outcome. For a signed implementation this number almost doubles: $n * 2^{n+2} - 4n^2$.

4.4 Approximate multiplier with configurable error recovery

The approximate multiplier with configurable error recovery (AM-ER) is obtained by approximating the adders required to compute sums in the product tree. Similar approximation techniques are explored in [68, 69, 70]. The

AM-ER is introduced in [71]. As the name suggests, the design is equipped with an error recovery strategy to contain the overall error. The AM-ER works by cutting the critical path that results from the carry propagation chain, thus enabling parallel computation of the product tree. Since this feature generates a high error rate, a correction mechanism is devised. An error signal is generated concurrently with each sum bit, hence this generation does not create additional delay. As a result, Liu et al. claim that this new circuit has a shorter critical path delay than that of a normal FA.
The approximation technique conceived for this circuit is straightforward. The first conditioning to the data is effected by the input pre-processing (IPP). This unit works on the principle that, in a sum operation, bits with the same weight can be switched without affecting the final result. In other words, the i-th bit of operand A and the i-th bit of operand B are interchangeable. The goal is to separate as many '1's from the '0's as possible by confining them in one of the two operands. The algorithm works as follows: given the input numbers A and B, for each pair of bits a swap occurs whenever $A_i = 0$ and $B_i = 1$. As for the three remaining combinations, nothing is done. In this way, A will contain mostly '1's, while B contains '1's only when both A_i and B_i are equal to '1'. The IPP is thus characterized by the logic functions shown in Equation 4.10. The pre-processed operands are distinguished by a dot on top of them.

$$\begin{aligned} \dot{A}_i &= A_i \; OR \; B_i, \\ \dot{B}_i &= A_i \; AND \; B_i. \end{aligned} \tag{4.10}$$

It is easy to notice that these equations also represent the propagate and generate signals in a parallel prefix adder such as the carry-lookahead.
Once the pre-processing stage is completed, the circuit is able to compute the sum in parallel. As $\dot{B}$ is equivalent to the generate bit, a value of 1 for $\dot{B}_i$ means that a carry was produced. This bit is sent to the next stage where it could further propagate in the event $\dot{A}_{i+1} = 1$ and $\dot{B}_{i+1} = 0$. In this case, the sum bit S_{i+1} would be '0' and the carry bit would move on to the $i + 2$ stage. However, as the adder is approximated, S_{i+1} ends up being '1', the carry does not go past that point and an error bit is set to signal this occurrence. Each digit has an error bit that turns to '1' whenever a carry is blocked. The unit that deals with the error signals is described later. Table 4.1 shows the truth table for this kind of circuit.

$\dot{B}_{i+1}\dot{B}_i$	$\dot{A}_{i+1}$	S_{i+1}	E_{i+1}
00	$\dot{A}_{i+1}$	$\dot{A}_{i+1}$	0
01	$\dot{A}_{i+1}$	1	$\dot{A}_{i+1}$
10	1	0	0
11	1	1	0

Table 4.1: Truth table for the approximate adder

The truth table can be summed up by Equation 4.11, where sum and error bits are expressed as a function of the input operands, before entering the IPP unit:

$$\begin{aligned} S_{i+1} &= (A_{i+1} \; XOR \; B_{i+1}) \; OR \; (A_i \; AND \; B_i), \\ E_{i+1} &= (A_{i+1} \; XOR \; B_{i+1}) \; AND \; (A_i \; AND \; B_i). \end{aligned} \tag{4.11}$$

It is worth noting that the error E is always non-negative and that the result, when incorrect, is always smaller than the accurate one. Hence, to compensate for the error, a second adder can be used. It is possible to show that:

$$S = S' + E, \tag{4.12}$$

where S' is the approximate sum and S is the correct result.
The approximate adder is tasked with summing the partial products in the product tree. For each pair of numbers in the tree, a string of error bits is generated. Instead of fixing the error at the output of every adder, the compensation is carried out on the final result. For this to be possible, all the errors along the tree must be first accumulated and then corrected. By using an exact adder for each accumulation stage, and a final adder to implement Equation 4.12, one can obtain the correct result at the output with no approximation. Due to the data statistics of the error vectors, however, Liu et al. propose an approximation technique for the pairs in the tree [71]. As the error bit is '1' only when the carry propagates, it is safe to assume that the generic array E will have more '0's than '1's. Therefore, the probability that two different arrays exhibit a '1' in the same

position is low. Thus, the compensating adder can be implemented by means of OR gates much like a LOA. OR gates produce a correct result in 3 out of 4 cases. The wrong case is the least likely to occur.
Depending on the depth of the tree, an arbitrary number of error vectors are present. The overall error is obtained by performing a bitwise OR on the whole set. Equation 4.13 shows the operation on the generic i-th bit for an n-bit vector:

$$E_i = E1_i \; OR \; E2_i \; OR \; ... \; OR \; En_i. \tag{4.13}$$

The design can be further simplified by performing the OR operation and subsequently the error correction only on a limited number of bits. As the MSBs have a bigger impact on the overall accuracy, it is sufficient to handle a subset of elements from the vectors. The number of bits chosen depends on the target precision. However, as the '1's in the error vectors are sparse, even a conservative choice can produce good results. Figure 4.5 shows an example of a 8x8 multiplier for which only the 4 MSBs are selected for error correction. Out of all the seven error vectors generated, only the ones that contain bits with the same weight as the four most significant ones are considered. These are: A3, A4, A6 and A7. The remaining elements are used only for the final sum.

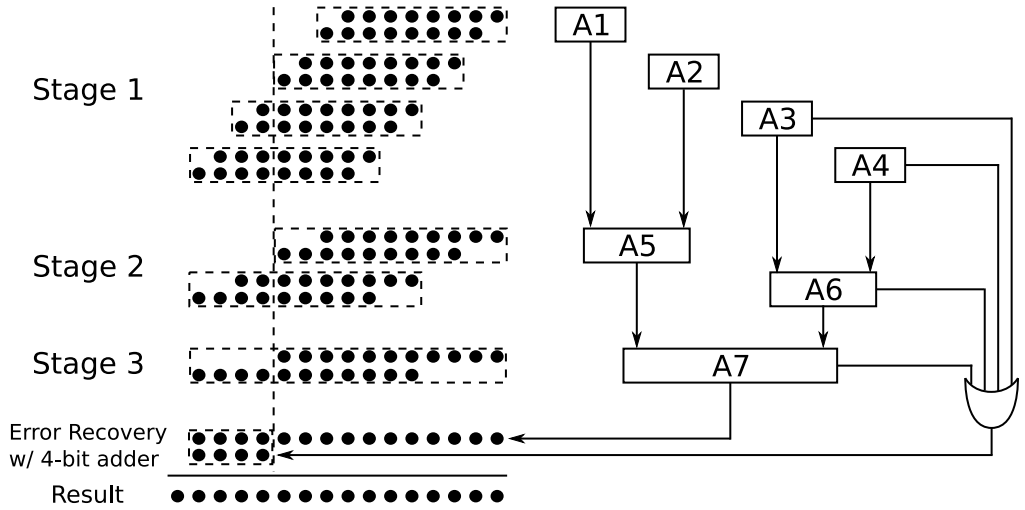


Figure 4.5: Approximate adder dealing with the product tree of an 8x8 multiplier, the 4-bit adder is exact

The results presented in [71] show that only few MSBs are needed in order for NMED and MRED to drop dramatically. The decline in these figures actually slows down after a certain point and almost flattens, thus indicating that including more bits than necessary might give no advantage at all. The approximate adder cell obtained from Equation 4.11 has four gates, whereas an exact FA has five. By defining as τ_g an arbitrary gate delay, Liu et al. estimated a delay of $2\tau_g$ for the inexact cell compared to $3\tau_g$ of the FA. As for the multiplier, the AM-ER is weighed against the well-known Wallace and Dadda trees. Equation 4.14 represents the delay calculated for an n-bit AM-ER whose product tree grows logarithmically with n. The formula also takes into account the error accumulation.

$$D_{AM-ER} = (2 * \lceil log_2(n) \rceil + 1) * \tau_g. \tag{4.14}$$

When compared to the accurate multipliers, the delay gap increases with the WL. It is estimated that for a 2^k-bit multiplier, the delay related to the AM-ER increases as $2k+1$, whereas the Wallace and Dadda trees show a trend close to $5k$. Power is also greatly reduced, with better results for higher WLs. In certain circumstances, a 16-bit AM-ER can experience a decrease in power consumption greater than 60%.

4.5 Restoring divider

Division is the inverse of multiplication. Although the basic algorithms and architectures able to perform division look similar to the ones related to multiplication, this operation is a bit more complicated. As previously stated, the system's pipeline contains only one divider. However, given its complexity, it represents the perfect candidate for approximation.
Integer division makes use of the following quantities:

- A dividend z expressed on 2*k bits.

- A divisor d of k bits.
- A quotient q of k bits.
- A remainder s of k bits.

The following relation holds: $z = d * q + s$. Starting from z, a sequential restoring unsigned divider computes q iteratively. For each cycle, a partial reminder s_i is obtain by subtraction and shifting exactly like pen and paper division. As s_i must be non-negative, the correct value for q_i is the one for which $s \geq 0$. Moreover, the relation $|s| < |d|$ must be true. Research has shown that approximate array dividers using a restoring algorithm perform better in terms of power and area than their non-restoring counterparts, at the expense of accuracy [46, 47]. This is partly due to the fact that the non-restoring divider requires an additional correction stage for the remainder. For this reason, a restoring divider is selected for this project. Using this approach, the subtraction is carried out and the sign of s_i is tested at every iteration. Only if it is not negative can s and q be updated. In this thesis a high-radix approach is preferred. An array divider is able to implement division in just one cycle. This fully combinational solution has the advantage of being flexible: approximation can be achieved on different levels. The design is also modular. All cells are the same and, due to granularity, different approximation strategies can be implemented where only some of the cells are simplified. Moreover, the regularity of such dividers makes it easy to introduce pipeline registers [72]. Still, by looking at the array-like circuit in Figure 4.6, it can be seen that the division algorithm itself is always sequential. This means that unlike multiplication, that can be carried out in parallel like a multi-operand addition, here the previous stage, and the related error, affect the following one. Each row produces a quotient bit, which is actually reused. It results that the accumulated error must be kept under control.

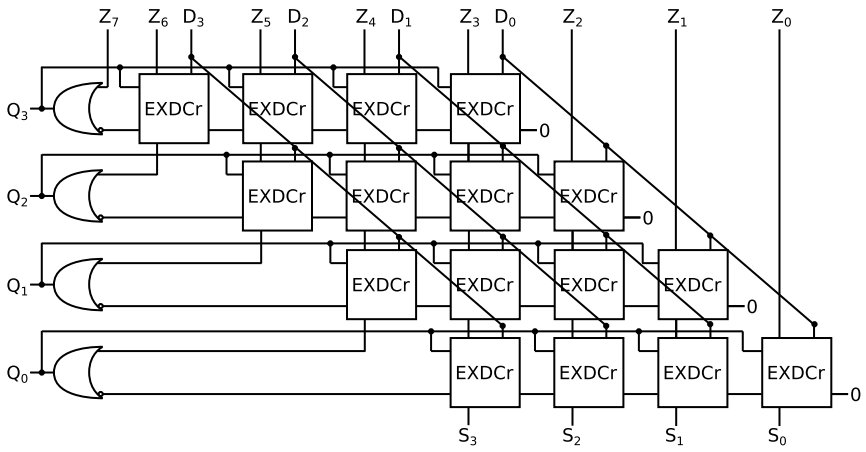


Figure 4.6: Schematic of an 8x4 restoring array divider (unsigned)

Approximation is carried out on the building blocks of the array. A restoring divider's cell is made of a 1-bit subtractor and a multiplexer. Its inputs are: the dividend z, the divisor d, the borrow-in b_{in}, and the quotient q, whereas its outputs are the remainder s and the borrow-out b_{out}. Figure 4.7 shows an exact restoring cell.

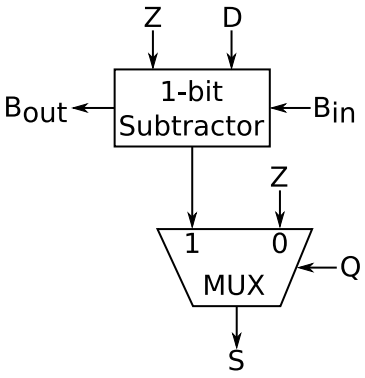


Figure 4.7: Example of an exact restoring cell

Approximation occurs on two levels: first, the primary cells are simplified by acting on their logic functions and later they are used to replace the exact cells in the array with different possible arrangements. Four substitution strategies can be identified: vertical replacement (VR), horizontal replacement (HR), square replacement (SR), and triangle replacement (TR). Different methods affect q's and s's accuracy in different ways. In VR, the exact cells in the least significant columns are replaced by approximate equivalents. The number of columns that undergoes this process depends on the specifications: higher approximation depth means lower power and area but larger error. HR, as the name suggests, aims at replacing an entire row. In particular, the one responsible for computing the LSB of q. This line also produces the final remainder. That is why this approach is feasible only when the value of s is not important. Moving from the bottom row up, multiple lines can be replaced depending on the final accuracy. SR is obtained by combining both VR and HR. The last technique, TR, is similar to SR but the area covered by the approximate cells is triangular. Figure 4.8 shows an example of these methods. An alternative to cell replacement is cell truncation, where the exact cell is removed and not replaced.

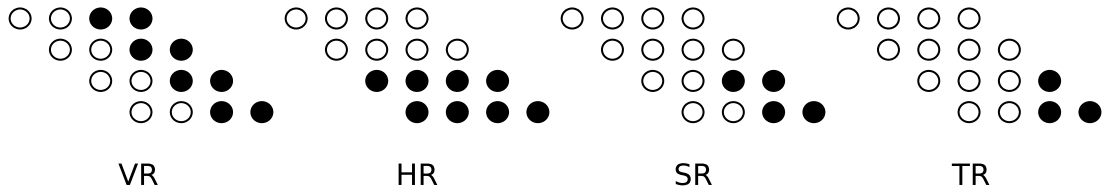


Figure 4.8: The four replacement strategies for an array divider

Four approximate subtractor cells are created by means of logic reduction. A 1-bit subtractor consists of three inputs: X, Y and B_{in}, and two outputs D (difference) and B_{out}. The logic functions for an exact cell are shown in Equation 4.15:

$$\begin{aligned} D &= X \oplus Y \oplus B_{in}, \\ B_{out} &= B_{in}(\overline{X \oplus Y}) + \overline{X}Y. \end{aligned} \tag{4.15}$$

From 4.15, the logic functions for an exact restoring divider cell (EXDCr) can be obtained:

$$\begin{aligned} S &= Q(X \oplus Y \oplus B_{in}) + \overline{Q}X, \\ B_{out} &= B_{in}(\overline{X \oplus Y}) + \overline{X}Y. \end{aligned} \tag{4.16}$$

The first approximate subtractor (AXS), named AXS1, is defined by the following functions:

$$\begin{aligned} D &= \overline{B_{in}(X + Y) + XY}, \\ B_{out} &= \overline{X}. \end{aligned} \tag{4.17}$$

In [47], it is stated that the error on B_{out} is more important than the one on D. That is why, for AXS2 and AXS3, only the equation for D is approximated:

$$D = X \oplus Y + B_{in}, \text{ (AXS2)}, \tag{4.18}$$

$$D = B_{out}, \text{ (AXS3)}. \tag{4.19}$$

Finally the last unit, AXS4, is defined by:

$$\begin{aligned} D &= B_{in}(X + \overline{Y}) + X\overline{Y}, \\ B_{out} &= B_{in}. \end{aligned} \tag{4.20}$$

Four approximate array dividers are created using both EXDCrs and approximate restoring divider cells (AXDCrs) with different replacement methods and approximation depths. Based on the findings in [47], the

TR offers the best accuracy in terms of MED and NED. When it comes to power consumption, VR and HR are able to save the most among the proposed replacement schemes. Cell truncation naturally dissipates the least amount of power as entire blocks are eliminated and no substitutes are introduced. However, this solution negatively affects the final accuracy. When considering the trade-off between error and power consumption, the TR approach seems to be the best choice. When different WLs are tested, results show that, in general, the NED decreases, while power consumption grows moderately. Out of the four approximated cells, AXS1 and AXS2, combined with TR, are expected to offer the most advantageous compromise between power and accuracy. Due to the specific application related to the zero-crossing detector, the divider in this thesis deals with unsigned operands and its result is a fixed point number between 0 and 1.

CHAPTER 5

Hardware design

After a first discussion on the system and the approximate algorithms, this chapter outlines the hardware implementation in detail. First, the exact circuit is analyzed starting from the top-level entities (i.e. filters, and zero-crossing) and moving downwards to the basic blocks. Once this step is completed, approximate arithmetic circuits take the place of the accurate units one by one. In a filter, three versions are implemented and tested: one that makes use of approximate adders, one featuring approximate multipliers only and one that uses both. An exact design is created alongside the approximated ones for the sake of comparison. In Chapter 6, benefits and shortcomings related to approximate circuits are highlighted. Several figures of merits are considered. These include the metrics introduced in Chapter 2 and also post-synthesis information such as resource utilization, maximum frequency, and power consumption. Final results such as filtering and zero-detection are compared with the software version as well. For this project, the Intel®Cyclone®V FPGA is used. In particular, the design is synthesized on an Altera DE1 SoC board mounting a Cyclone V 5CSEMA5F31C6 FPGA. This model features 32070 adaptive logic modules (ALMs), 85000 logic elements (LEs), 3207 logic array blocks (LABs), 128300 registers, 6 PLLs, 4450 kbits of embedded memory, and 457 total pins. The board mounts a dual-core ARM Cortex-A9 processor with a maximum clock frequency of 925 MHz. Four 50MHz clocks are fed directly to the FPGA to synchronize the user logic. The FPGA's core voltage is 1.1V, while the GPIOs can operate on different levels up to a maximum of 3.3V. Parameters such as ALM and LAB are required to estimate the resource consumption. Their purpose and what they represent will be explained in Chapter 6.

5.1 Exact design

Three main blocks are described using VHDL. These are: the smoothing filter, the differentiation filter and the zero crossing detector. Each of them comes in multiple versions characterized by one or more changes in relation to the others. The circuits presented next are already optimized and display the best possible performance out of all the previous variants. Optimizations are achieved by exploiting the known data statistics or by rearranging the equations in the filter's case. Moreover, dealing with constant coefficients greatly simplifies the multipliers. Pipelining is used to improve speed and power consumption at the expense of area. Finally, the Intel Quartus compiler might apply additional improvements such as logic minimization and register retiming. The optimization mode for the compiler is set on "balanced". This means that delay, area and power figures are given equal priority during synthesis. In this project, the main arithmetic units are implemented structurally. In other words, the VHDL arithmetic operators (such as $+$ and $*$) defined in the numeric_std library are not employed. Instead specific architectures are chosen and described on a logic gate level. Although efforts are being made in literature to create libraries for approximate circuits [73, 74], approximate designs cannot make use of these operators. Comparators, multiplexers and encoders, instead, are implemented behaviourally using combinational VHDL statements. For this reason, the internal realization of these units is up to the synthesizer and is not visible from the Quartus RTL Viewer.

5.2 Filters

In this section the hardware implementation of SG filters is introduced. These units use registers, multipliers, adders as well as subtractors as main components. A direct form is preferred, as explained in Chapter 3.

Such a structure is easier to implement and optimize. Moreover, the advantages of the transposed form are not significant when handling low order filters such as the ones discussed here. The long combinational path observed in the direct architecture can be fixed by means of pipelining.
Both filters use a degree four polynomial and a framelength of nine to perform the least-squares procedure. This means, resource-wise that eight registers, eight adders and nine multipliers are needed for each filter. Nevertheless, coefficients' symmetry can be exploited to obtain the first optimization. In this way, the number of multipliers can be halved. The number of registers does not change. A standard FIR filter equation for the design under test can be expressed as:

$$y[n] = a_0x[n]+a_1x[n-1]+a_2x[n-2]+a_3x[n-3]+a_4x[n-4]+a_5x[n-5]+a_6x[n-6]+a_7x[n-7]+a_8x[n-8]. \quad (5.1)$$

Following simplification, two different equations can be derived depending on the properties of the coefficients, as explained in the next sections. But first, two exact adder architectures are presented that are found in both filters.

5.2.1 Brent-Kung adder

The Brent-Kung adder (BKA) is a type of parallel prefix adder [75]. All parallel prefix structures are based on the carry-lookahead adder (CLA) and work on the assumption that defining an associative operator enables parallel computation of the operands. In a CLA, the generate and propagate signals are obtained from the input operands using the following expressions:

$$\begin{aligned} g_i &= a_i \ AND \ b_i \\ p_i &= a_i \ XOR \ b_i. \end{aligned} \quad (5.2)$$

When both the i-th bits of two numbers are equal to 1, a carry is produced. The generate bits are therefore computed via an AND gate. The propagate signal, as the name suggests, determines whether the carry is propagated to higher order bits or not. In the case where the i-th bits are opposite in value, the carry generated in a previous stage can propagate. As an example, a 4-bit CLA is able to compute each carry bit by employing the following logic functions:

$$\begin{aligned} c_1 &= g_0 + p_0 * c_{in}, \\ c_2 &= g_1 + g_0 * p_1 + c_{in} * p_0 * p_1, \\ c_3 &= g_2 + g_1 * p_2 + g_0 * p_1 * p_2 + c_{in} * p_0 * p_1 * p_2, \\ c_4 &= g_3 + g_2 * p_3 + g_1 * p_2 * p_3 + g_0 * p_1 * p_2 * p_3 + c_{in} * p_0 * p_1 * p_2 * p_3, \end{aligned} \quad (5.3)$$

where + denotes an OR operation and $*$ represents an AND operation. From the expressions, it can be seen that each signal can be computed starting from elements that are readily available from the beginning. Carry $i+1$ needs not wait for carry i as in a ripple-carry adder (RCA). However, as the equation is recursive, its complexity rapidly escalates after a certain word-length. This phenomenon can be explained by considering that for each new bit, all the previous carry bits are recalculated and combined with the new g and p signals. This is necessary to avoid waiting for the least significant carry bits to propagate. For this reason, typical CLA implementations never exceed a 4-bit width. To obtain larger circuits with a reasonable complexity, the CLA blocks can be connected in a ripple-carry fashion. This type of circuit is used to model the carry generation block in one of the approximate adders, as explained later on.
Parallel prefix adders take advantage of a tree-like structure able to compute all carry bits in one go. These values are necessary to calculate the final sum. Given the two input operands, a pre-calculation of all generate and propagate signals is performed. Hence, g_i and p_i are produced starting from Equation 5.2. Similarly to a CLA, the g and p signals must be combined recursively in order to obtain each carry bit. To group these logic values, an associative operator "$\circ$" is defined. This operator handles two pairs of generate and propagate signals (e.g. (g_x,p_x) and (g_y,p_y)) and combines them to create a new pair of bits called group generate and group propagate determined by Equation 5.4:

$$(G_{x,y}, P_{x,y}) = (g_x, p_x) \circ (g_y, p_y) = (g_x + p_x * g_y, p_x * p_y). \quad (5.4)$$

Since only associative operators (OR, AND) are used, it follows that $\circ$ is also associative. Equations 5.3 reveal that the higher the weight of the carry, the more complex its expression is. $G_{x,y}, P_{x,y}$ can be combined recursively with successive sets to form new pairs with extended range. In general:

$$(G_{x,j}, P_{x,j}) = (G_{x,y}, P_{x,y}) \circ (G_{i,j}, P_{i,j}). \quad (5.5)$$

Owing to this strategy, the computation can proceed in parallel at a faster pace. Figure 5.1 shows an example of a parallel prefix adder, in particular a Kogge-Stone adder (KSA) [76]. Each black dot is tasked with performing the associative operation described in Equation 5.5. For instance, $P_{1,0}$ is obtained by computing $p_1 * p_0$. At the same time, $P_{3,2}$ is estimated from p_3 and p_2. Finally, $P_{3,0} = p_3 * p_2 * p_1 * p_0$ is computed by the GP block for $P_{3,2}$ and $P_{1,0}$. The same is true for G, although the expression is more complex. By making use of recursion, the range can be extended until all carry bits are available. It can be shown that the height of the tree, and thus the delay, grows logarithmically with the number of input bits.

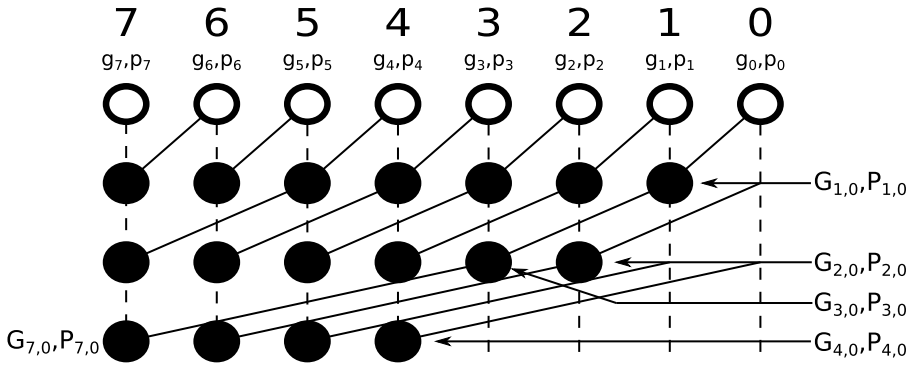


Figure 5.1: 8-bit Kogge-Stone adder

Depending on how the operator is applied, several designs can be created. Besides the KSA and the BKA, circuits possessing a tree structure include the Ladner-Fischer adder [77] and the Han-Carlson adder [78]. A summary of these units can be found at [79]. In general the BKA uses the least amount of area and requires less power to function. Comparative studies [80, 81] demonstrated that for short word-lengths the differences among tree adders are not so pronounced. The BKA is the accurate adder of choice for this project, as it requires the least GP blocks. Figure 5.2 shows an example of an 8-bit BKA. Unlike the KSA, this design requires more stages, yet it allocates less resources. For this unit, the computation is split. In the upper part, the minimum number of blocks necessary to calculate the most significant carry bit $(G_{7,0}, P_{7,0})$ is instantiated. This operation also generates $(G_{1,0}, P_{1,0})$, $(G_{3,0}, P_{3,0})$. The remaining pairs are evaluated in the last two stages.

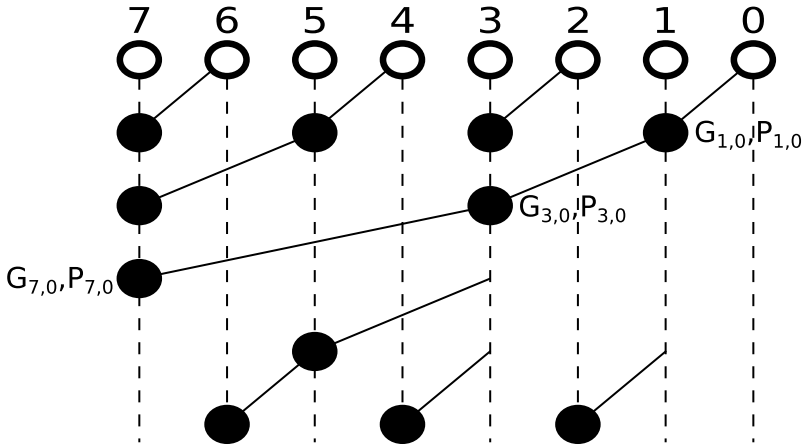


Figure 5.2: 8-bit Brent-Kung adder

Besides the greater tree depth, another disadvantage the BKA has to face when compared to a KSA is the larger fan-out. While in the KSA each GP block has a maximum fan-out of two, the number doubles when the BKA is considered. However, as explained in Section 6.2, the synthesis results show that the BKA is actually faster and definitely less complex than the KSA.
In the smoothing filter, four 2-operand BKAs are needed to combine pairs of samples that share the same coefficient prior to multiplication. In this way, the nine taps are reduced to five. The sample in the middle is fed directly to the multiplier. Given the input range, the BKA takes two 8-bit operands and returns a 9-bit result. As the addition is unsigned, the overflow condition can be detected simply by checking the carry-out

bit, which also plays the role of the MSB. Although the isolated coefficient (b_4) appears to be the largest, it is not responsible for the widest output range. In fact, the sample fed to the multiplier does not pass through any adder and therefore it can still be expressed on 8 bits rather than 9. The adder is optimized by eliminating the carry-in signal and by instantiating only the GP blocks that are actually needed for the sum. The schematic of the adder is shown in Figure 5.2.

5.2.2 Wallace tree multi-operand adder

The second kind of adder found inside the filters is the carry-save adder (CSA), which is known to be particularly suitable for multi-operand addition [82]. In this case, the sum of four numbers must be computed. Unlike a RCA, which features a chain of FAs, the blocks in this circuit work in parallel. This means that the bottleneck caused by the carry propagation is now deferred until the last step, resulting in a much faster adder. In a CSA, the distinction between the input operands' ports and the carry-in one is ignored as the attention is shifted onto the weight of these bits instead. Each FA takes three input signals with equivalent weight and produces a sum bit that shares the same weight. A carry bit is also generated whose weight is positioned right above the sum bit. Therefore, it can be said that the FAs in a CSA work as 3-2 compressors. Figure 5.3 shows examples of two different additions carried out by a CSA. On the left is a sum of five 8-bit operands, whereas on the right only four of them take part in the computation. The dot notation is used to highlight the allocation of the FAs according to the weight of each bit whose value is irrelevant [83]. Each of the bits belonging to operands, carry and sum signals is represented by a dot of a certain colour. All the dots on the same column have the same weight. Blocks created by grouping three dots inside the black rectangle symbolize FAs, while boxes containing only two dots symbolize half adders half adders (HAs). Both of these adders must process dots with the same weight and produce a sum bit and a carry bit also in dot notation. As can be seen from the figure, the carry bit's weight is always one position ahead of the sum one. As a consequence, any adder at column 2^0 produces a carry bit that has a weight of 2^1. Hence, no carry out is found at the LSB, unless a carry-in is present. Several levels of FAs are needed to process the five operands. However, due to the compression ratio of the FA/HA, the circuit always returns two numbers rather than the final sum. By refusing to acknowledge the carry propagation, the CSA is bound to produce a result that is said to be in carry-store form. The output of a CSA is made of two numeric strings that contain the sum and carry bits. To obtain the final result, these strings must be added together by means of a standard two-operand adder. The main advantage of such implementation is that the carry propagation is processed only at the end. For this reason, the delay of the whole adder is determined by the delay of the circuit tasked with the final addition, which can have any architecture. As the LSBs do not produce a carry, the final adder operates on fewer bits than the ones that constitute the output. While not strictly necessary, extra HAs can be strategically added to further reduce the size of the last adder.

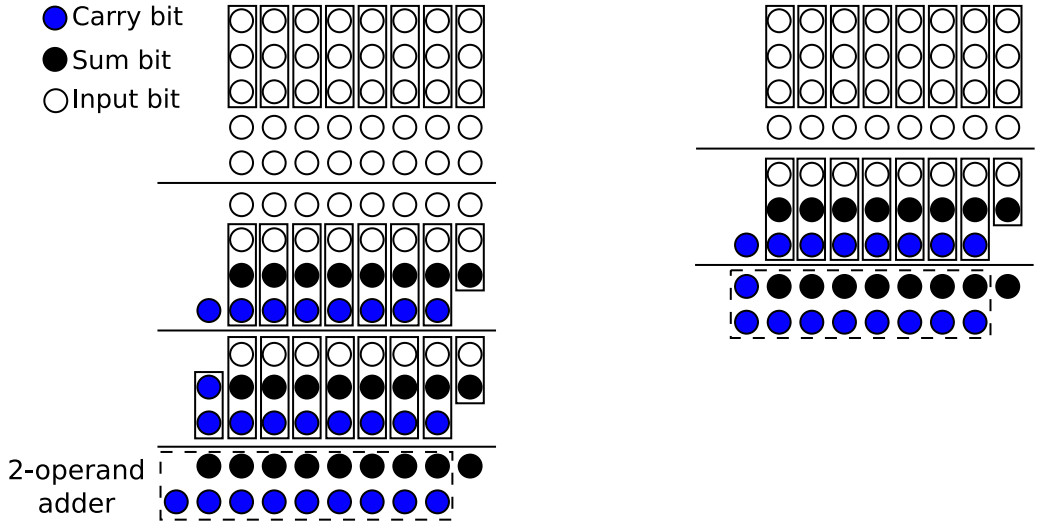


Figure 5.3: On the left: Wallace tree adder dealing with five 8-bit operands. On the right the same adder working with four 8-bit operand

The arrangement of the FAs in the tree is not random. The rules are provided by the Wallace tree algorithm. With this strategy, each level is divided into groups of three rows. Depending on the amount of dots, as many

FAs (3 dots) or HAs (2 dots) as possible are allocated. For instance, in a situation in which more than five operands are summed, at least two rows of FAs would be instantiated in the first stage. The single dots are simply moved to the next level where they maintain the same position. This process is iterated until two operands are left.
Given n operands on k bits, the number of levels required to reduce the tree to just two numbers is given by the following recursive equation:

$$h(n) = 1 + h\left(\left\lceil \frac{2n}{3} \right\rceil\right). \tag{5.6}$$

Equation 5.6 shows that the height of the tree is logarithmic. Moreover, the speed of the whole sum boils down to the choice of the final adder.

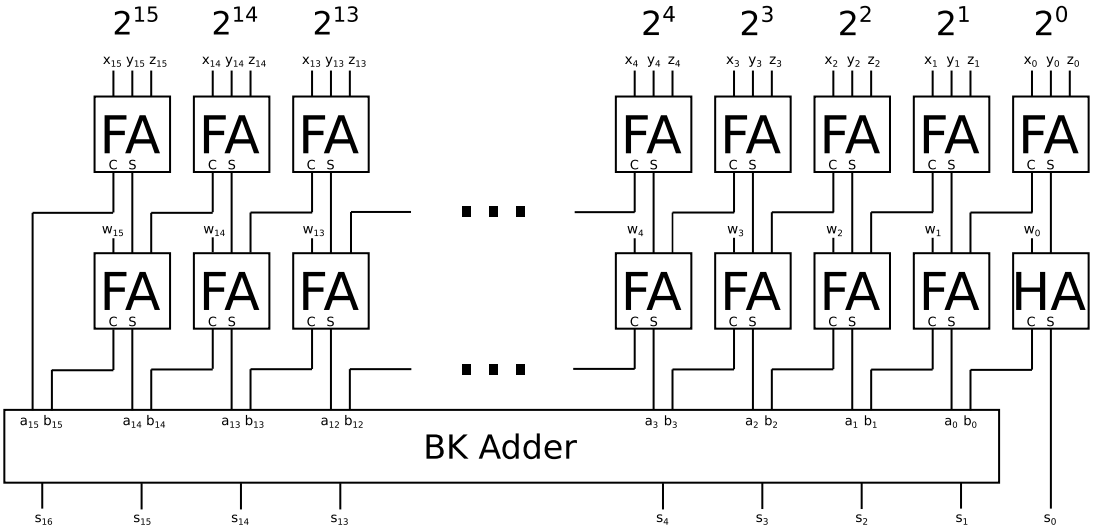


Figure 5.4: Implementation of the Wallace tree adder in the smoothing filter

In the smoothing filter, a Wallace tree adder (WTA) is used to sum four 16-bit operands. Figure 5.4 shows the tree of FAs and the final BKA. The circuit adds together the 16-bit operands, coming out of the multipliers that make use of the coefficients to scale the input samples. An optimized 15-bit BKA is employed as the final adder. The output is provided on 17 bits. An aggressive optimization is achieved by knowing the input dynamics, i.e. the output range of the multiplier. As some operands require less than 16 bits to be represented, their MSBs are always zero. Therefore, some FAs can be spared or turned into HAs. However, this circuit works only for coefficients smaller than a certain quantity and only if the input data is fed in a specific order.
As combinational multipliers need to process a large number of partial products, a multi-operand adder such as the WTA is often the preferred solution. In this project, all the exact multi-operand adders and unsigned multipliers are based on the Wallace tree. The Dadda tree algorithm is a valid alternative [84]. Unlike the Wallace multiplier, this circuit instantiates the minimum quantity of FAs/HAs to reduce the number of rows to a value related to a specific parameter proposed for the algorithm. The advantages of one strategy over the other in terms of area, delay or power are minor. Thus, the Wallace algorithm was preferred owing to its easier implementation.

5.2.3 Smoothing filter

A smoothing filter exhibits an even symmetry. For this reason, the pairs of same-value coefficients can be grouped except for the one in the middle that occurs only once. This amounts to a total of five multipliers and eight adders. Figure 5.5 represents a direct mapping of Equation 5.7. The image shows the smoothing filter's basic architecture that makes use of the optimization just analyzed. A register, which is not displayed here, is added at the input to decouple the circuit from the rest of the system, and to protect it from spurious switching activity that would negatively affect power consumption.

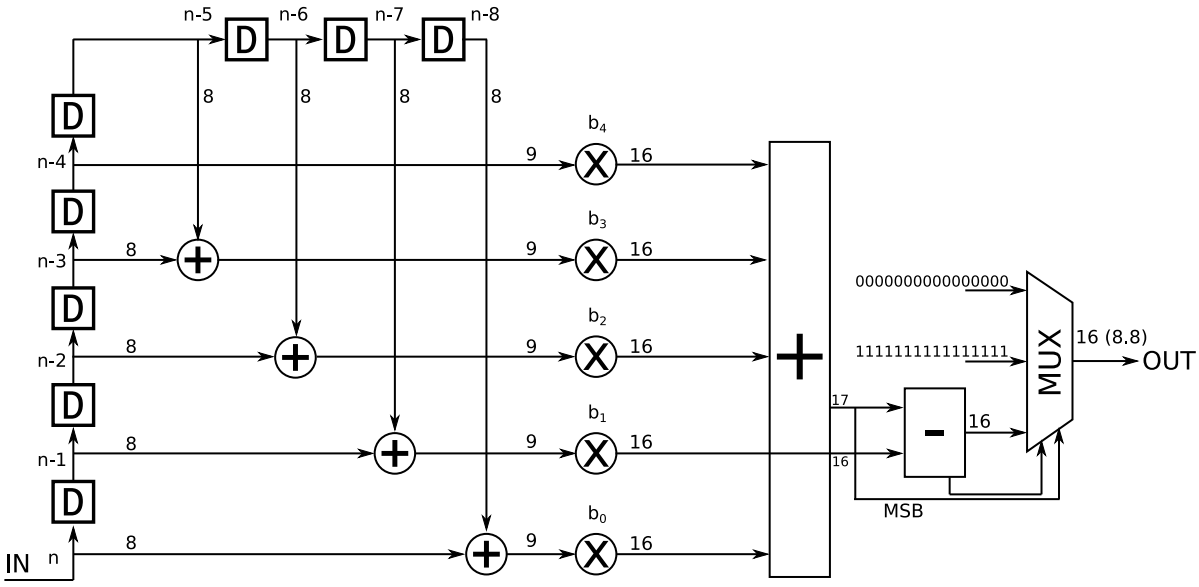


Figure 5.5: Actual smoothing filter implementation

$$y[n] = b_0(x[n]+x[n-8])+b_1(x[n-1]+x[n-7])+b_2(x[n-2]+x[n-6])+b_3(x[n-3]+x[n-5])+b_4x[n-4]. \quad (5.7)$$

A smoothing filter returns a less noisy version of the input signal, which ranges from 0 to 255. Thus, this circuit is fully unsigned. The input data is defined on 8 bits. Four adders are responsible for the grouping of the samples that share the same coefficients. These first blocks manipulate integer numbers. As the result can get as high as 510, the output is extended to nine bits. After this step, five unsigned multipliers are used to scale down the samples based on the coefficients. Since their magnitude is always less than 1, the output is smaller than the input. Nonetheless, the product is extended to 16 bits as it is converted to a fixed point number. Of these 16 bits, eight are assigned to the integer part, while the remaining eight account for the fractional part. Four coefficients are positive, while one is negative. A multi-operand adder simultaneously sums the four operands multiplied by the positive coefficients. This calculation is at risk of producing an overflow, which happens when the output goes beyond 255. The multiplication by the fifth and last coefficient is, again, unsigned. However, the result is rendered negative by performing a subtraction instead of an addition. Its value is subtracted from the signal coming out of the multi-operand adder. This operation might result in a negative output. The effect observed here is expected but unwanted. Normally, this number is small and does not influence the outcome of the subsequent stages. However, to keep the output unsigned, any negative number is forced to 0. This action does not introduce any distortion on relevant information, such as the peak, which is far from being negative. Clipping is applied to saturate any detected overflow at 256 (actually 255.999...), which avoids adding an extra bit. This operation is enforced by a multiplexer that inspects the MSB coming from the adder and the carry-out from the subtractor to identify any out-of-range exception. While a negative result can be corrected at any step, a peak that exceeds 255 must be scaled only at the end, so as not to affect the final height and shape of the laser line. This is why this task is carried out as late as possible and not right outside the adder. The output is on 16 bits: the minimum number of bits to represent the integer part are eight, the other half is used by the fractional one.
Pipeline registers are inserted so that all the paths produce similar delays. To perform this optimization, all the blocks in the filters have been synthesized separately, and data has been collected. The information concerning the speed of the single units, as well as resources, are commented in Chapter 6. All registers are implemented as generic entities. They are driven by a clock signal and are updated on the rising edge. Additionally, they feature a synchronous active-high reset. No enable signal is present.

5.2.3.1 Wallace tree multiplier

The last arithmetic block discussed is the multiplier. The architecture chosen for the smoothing filter that supports unsigned multiplication is the Wallace tree multiplier (WTM) [85]. This kind of circuit is able to

calculate the result in one cycle as it is fully combinational. It first computes the partial products and then sums them together by means of a WTA and a final BKA. In this way, the total sum is achieved in much less time than that required by several two-operand adders. The WTA intended for the multiplier follows the same working principles of the one previously introduced. The only difference is that for a multiplier, operands are not aligned. Instead, they are shifted left by a number of positions that depends on the weight of the operand bit considered. Moreover, the number of partial products is much bigger. Figure 5.6 shows the calculation of nine partial products based on a Wallace tree approach. As explained earlier, groups of three operands are formed and as many FAs/HAs as possible are instantiated. As the 1-bit adders, represented by black rectangles, act as compressors, the number of operands reduces when going down a level. Each block produces a sum dot at the same position of the $i-th$ bits from the operands and a carry dot whose weight is equal to $i+1$. Once the number of operands reaches two, a BKA computes the final result.

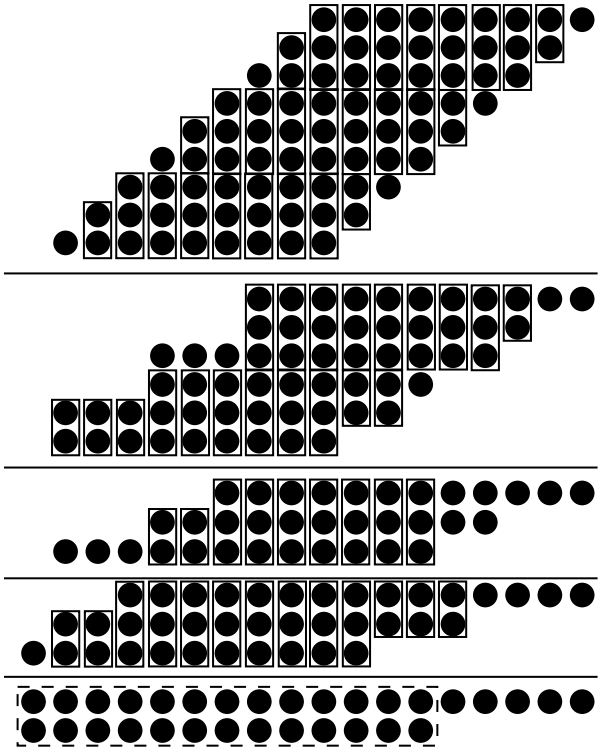


Figure 5.6: Dot notation for a 9x9 Wallace tree multiplier

In a smoothing filter, the WTM collects the BKA's 9-bit output and multiplies it by the respective fractional coefficient adapted on nine bits. Hardware designed for integer circuits is perfectly compatible with fixed-point numbers and it can be reused. However, the correct position of the result's decimal point must be determined to extract the correct value. In the proposed data processing pipeline, both integer and fractional parts double in size, so the decimal point moves leftwards. However, a full 18-bit output dynamics is not necessary. Since the image data is always multiplied by a number, which is less than 1, the output range never doubles due to the fact that the result is smaller than the input. Moreover, by knowing the value of each coefficient, accurate estimations can be performed to obtain information on the maximum number the multiplier can produce. In this case, to increase the fractional part precision, the coefficient is shifted left by ten positions ($*2^{10}$). To better explain this, suppose the numbers 156 and 0.0724 must be multiplied. The first operand comes from the subtractor and could be expressed as a 16-bit fixed point number with a 9-bit integer part and a 7-bit fractional part in the form: "010011100.0000000". Applying the same format to the coefficient leads to "000000000.0001001". It is important to note that few bits contribute to the fractional part precision and a lot of digits are not significant overall. As the filter's multipliers always process an integer number and a fractional one, the extra zeroes can be eliminated by shifting the coefficient left. By getting rid of unnecessary zeroes, the multiplication can be carried out on a lower number of bits, namely nine. The two operands are now perfectly aligned. In this way, while the integer number is unchanged, the coefficient is expressed on nine bits which are all fractional. The modified operands look like this: "010011100" and "001001010". This process moves the decimal point of the output to a new position, but once it is located, the result can be obtained. Out of the $2*9$ output bits, only

16 are selected.
In reality, the multipliers' actual implementation is much simpler in terms of resource consumption and delay than the concept shown in Figure 5.6. As one of the input operands is fixed, the circuits can be greatly simplified. To understand how, it is sufficient to know that the specific bit sequence of each coefficient is coded inside the hardware. In other words, since the value is known, whenever an input sample is multiplied by a zero from the coefficient, that partial product is automatically removed. The result is a circuit that multiplies any input by a constant value. Again, this implementation is extremely specific, so a different entity for each coefficient must be devised. The complexity of a multiplier ultimately depends on the number of zeroes its coefficient contains. The more there are, the simpler is the implementation. For example, the binary representation of the first coefficient b_0 is equal to "000010010". As the number has two '1's, only two partial products are effectively produced and properly weighed. In this case, the complexity and delay match those of an adder. Thus, a simple BKA is used directly on the samples. A multi-operand adder is actually needed only for the last two coefficients b_3 and b_4, which generate three and five partial product respectively. In this worst case, nevertheless, the final circuit is simpler than the one shown in Figure 5.6. This simplification is highlighted in Figure 5.7. The remaining four partial products are omitted as they are all zero. The points discussed so far imply that these multipliers take only one input operand. As some of these circuits are as simple as adders, it might be convenient to develop approximated versions based on inexact adders rather than multipliers.

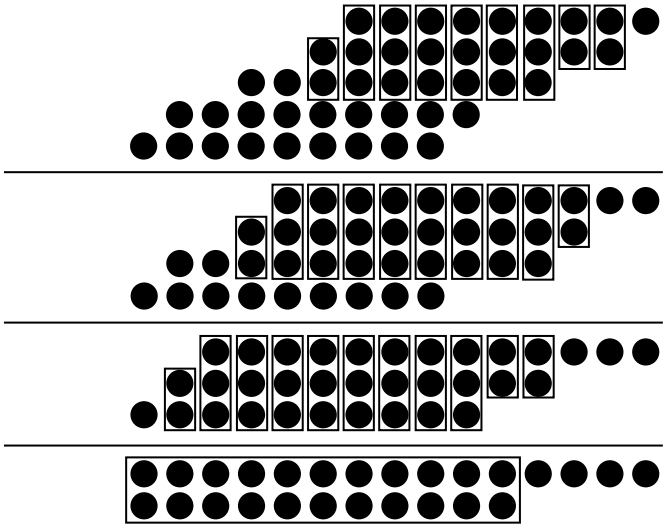


Figure 5.7: Dot notation for an optimized 9x9 Wallace tree multiplier

5.2.4 Differentiation filter

The differentiation filter normally requires only eight multipliers and seven adders. This is due to the fact that the middle coefficient g_4 is always 0. As a result, the simplified filter features only four multipliers. However, due to the odd symmetry displayed by the differentiation filter, the grouping of the terms in Equation 5.1 (excluding g_4) results in a subtraction rather than an addition. This is visible in the following equation:

$$y[n] = g_0(x[n-8] - x[n]) + g_1(x[n-1] - x[n-7]) + g_2(x[n-2] - x[n-6]) + g_3(x[n-3] - x[n-5]). \quad (5.8)$$

As a consequence, the differentiation filter has three adders and four subtractors. A subtractor is a slightly more complex version of an adder given the two's complement operation required at the beginning. However, by acknowledging the nature of the data at hand, the circuit can, again, be made simpler. It is known that the input samples are always non-negative, so the sign bit is 0 for the first operand and 1 for the second (after inversion). Moreover, the carry-in signal is fixed to '1' so it does not appear in the input list. Instead, the logic functions defined internally are adjusted accordingly. Needless to say, the differentiation filter deals with signed numbers. Figure 5.8 shows the entire circuit. It can be seen that the multiplexer featured in the smoothing filter is no longer present. This is because the differentiation filter produces a narrower output range. No overflow can occur and negative numbers are accepted.
The Baugh-Wooley multiplier (BWM) is chosen to manage signed numbers. As in the WTM case, the coefficients are shifted left to increase precision. In addition, the output range is expanded to produce a fixed-point

number. However, as the output is smaller than the input, it does not double. Hence, the decimal point's position is determined and a subset of bits is selected from the full $2n$ result. The use of subtractors allows all the coefficients to be positive, which simplifies the implementation.
The final addition is, once again, carried out by a four-operand WTA. However, as the Wallace tree technique is successful only with unsigned operands, a sign extension must be performed in order to obtain sensible results. This is why the internal parallelism of this circuit is wider than necessary. All two-operands adders and subtractors in the filter are based on the BKA. Carry-in and carry-out signals are included only if needed. Different BKAs in various sizes are implemented depending on their purpose. The total number of existing variants may increase the project's size but it ensures the highest level of optimization. Multiple filter versions also exist. A differentiation filter might be instantiated by itself or it can come after a smoothing filter. Finally, the second order filter discussed earlier, whose coefficients lay on a straight line and are thus proportional, is also implemented, as explained in Subsection 5.2.5.

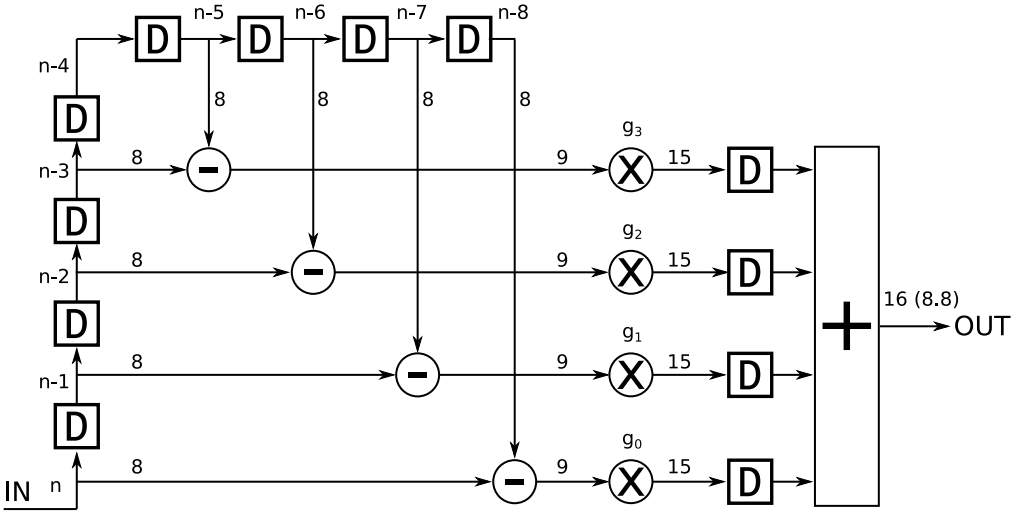


Figure 5.8: Actual differentiation filter implementation

For a lone differentiation filter, the input data ranges from 0 to 255. However, the output of the subtractors is a signed number going from -255 to 255. For this reason, the result must be extended to nine bits. Nonetheless, since the sign bits of the input operands are known, the actual BKA implementation defines its input ports on just eight bits and then extends the output to nine. To realize the two's complement of the second operand, the carry-in is always set to '1': logic functions that depend on this signal are simplified. A BWM is placed right after each subtractor. The reason behind its choice is explained in the following subsection.

5.2.4.1 Baugh-Wooley multiplier

The exact multiplier of the differentiation filter is represented by the BWM [86]. This multiplier was chosen, because, unlike the WTM, it can deal with signed numbers, while instantiating roughly the same amount of resources of an unsigned unit. Its operation can be understood by taking a look at the expressions that describe signed multiplication. Equation 5.9 shows that when dealing with signed numbers, the weight related to the MSBs of a pair of operands, i.e. the sign bit, is negative.

$$A * B = \left(-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i \right) * \left(-b_{n-1}2^{n-1} + \sum_{j=0}^{n-2} b_j 2^j \right). \tag{5.9}$$

Expanding the equation results in:

$$A * B = a_{n-1}b_{n-1}2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j} - 2^{n-1}\sum_{i=0}^{n-2} a_i b_{n-1} 2^i - 2^{n-1}\sum_{j=0}^{n-2} a_{n-1} b_j 2^j. \tag{5.10}$$

Equation 5.10 demonstrates that signed multiplication relies on subtraction. In particular, the last two terms must be subtracted from the rest in order to compute the result. The subtraction is turned into an addition by

performing the two's complement on the negative terms. From the expression, it can be seen that both numbers are $n-1$ bits wide, and they carry the same weight starting from $n-1$ to $2n-3$.

position	$2n-1$	$2n-2$	$2n-3$	...	n	$n-1$	$n-2$	...	0
X	0	0	x_{n-2}	...	x_1	x_0	0	...	0
Y	0	0	y_{n-2}	...	y_1	y_0	0	...	0

These numbers are first extended to $2n-1$ bits and then inverted. The extension is the result of a zero-padding both on the left and on the right side.

position	$2n-1$	$2n-2$	$2n-3$	...	n	$n-1$	$n-2$	...	0
$-X$	1	1	$\overline{x_{n-2}}$	...	$\overline{x_1}$	$\overline{x_0}$	1	...	1+1
$-Y$	1	1	$\overline{y_{n-2}}$	...	$\overline{y_1}$	$\overline{y_0}$	1	...	1+1

After negation, all the padding zeros have been turned into ones. A '1' is added at the LSB of both terms in order to accomplish the two's complement.

position	$2n-1$	$2n-2$	$2n-3$	...	n	$n-1$	$n-2$	...	0
$-X$	1	1	x_{n-2}	...	x_1	x_0+1	0	...	0
$-Y$	1	1	y_{n-2}	...	y_1	y_0+1	0	...	0

This operation turns all lower padding bits back to zero. The carry propagates until it reaches the first actual bits of the complemented number at position $n-1$. Although these bits are not known, both operands end up with a '+1' in that same position. Thus, the carry is further propagated to position n. Figure 5.9 shows the partial products obtained from the terms of Equation 5.10 for an 8x8 BWM. The terms that need to be subtracted are the ones with a line over them. The '1' with a weight equal to n is visible. The second '1' is, again, a result of the carry stemming from the MSBs of the numbers which are '1's. It ends up in position $2n-1$. These expedients create a signed multiplier that works with no sign extension, greatly reducing the total number of required resources.

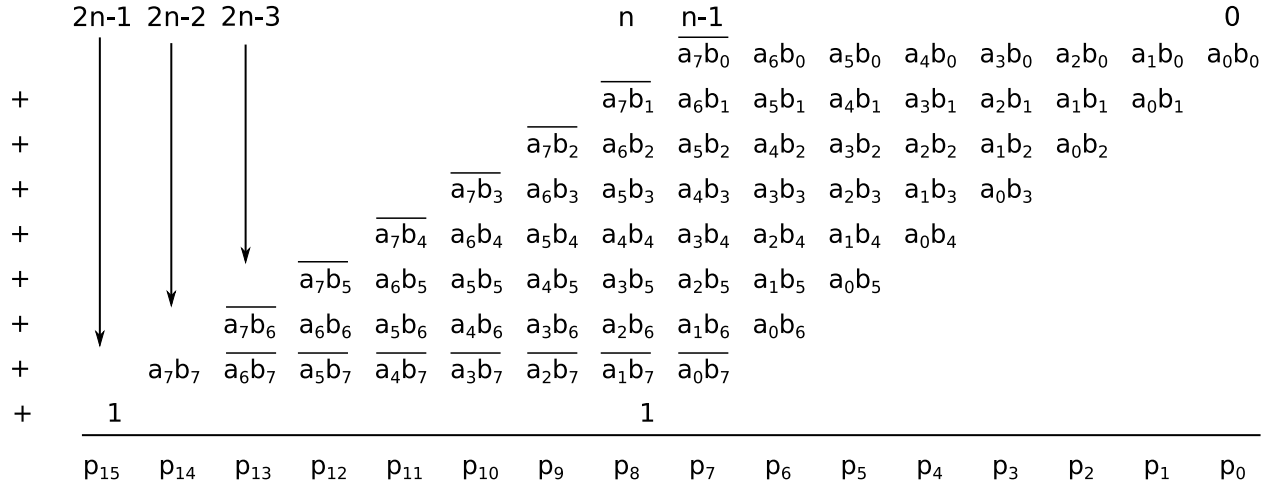


Figure 5.9: The working principle of a Baugh-Wooley multipler

Figure 5.10 shows the example of a 4x4 multiplier. Two kinds of basic cells can be distinguished. The white cell, which will be referred to as a regular cell, is made of a FA and an AND gate. In the gray one, identified as inverted cell, the AND is replaced by a NAND. The NAND gate is necessary to negate the numbers that need to be subtracted. An additional row of FAs in a ripple carry configuration is required to add the ones at the end. As the BWM is a kind of array multiplier, its structure is regular and consistent. A more regular design makes for a more orderly layout. Parameters such as power, delay and area are affected by it. The same cannot be said for the WTM.

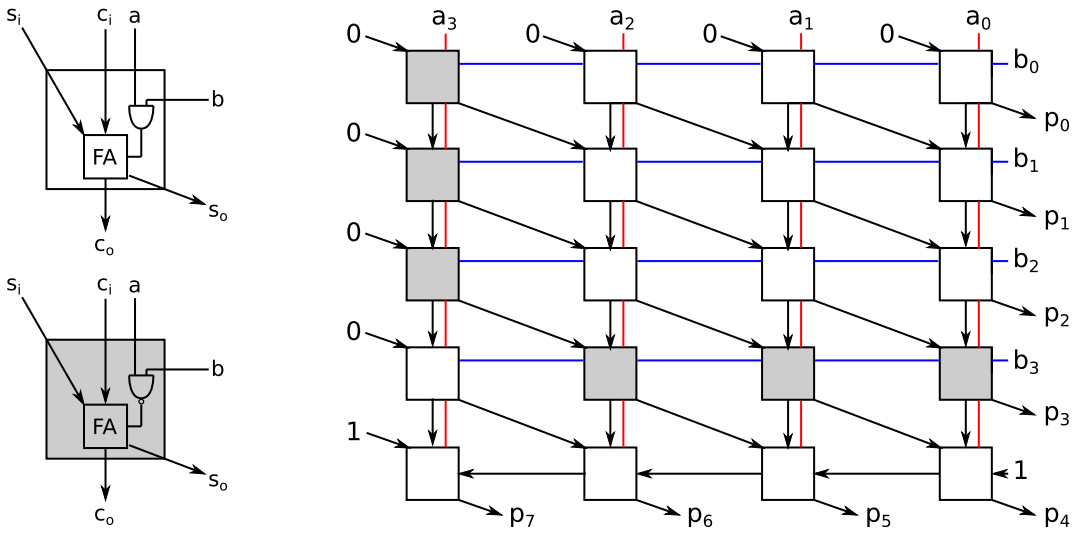


Figure 5.10: Example of a standard 4x4 Baugh-Wooley multiplier

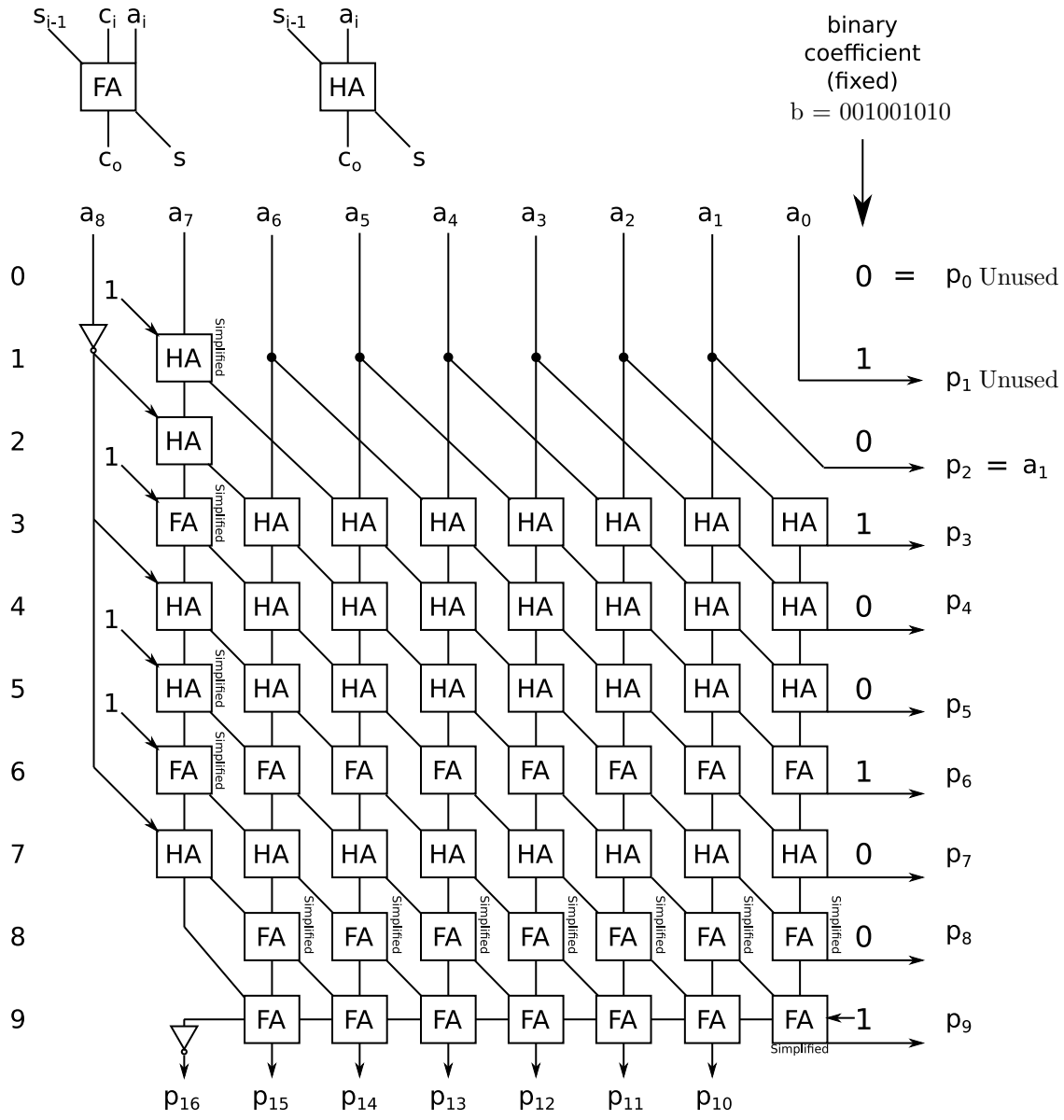


Figure 5.11: Implementation of a Baugh-Wooley multiplier for one coefficient

During the optimization process, however, this regularity might be lost in an attempt to reduce area. Similarly to the WTMs used for the smoothing filter, each of the BWMs features only one input. As the coefficients are known, four different multiplier circuits are created, one for each of them. Figure 5.11 shows an example of the multiplication by g_1 which only contains three '1's.
Similar to the smoothing filter, these multipliers take a 9-bit input coming from a subtractor and multiply it by a 9-bit extended precision coefficient. While the coefficient is always greater than zero, the result can be either positive or negative. In Figure 5.11, the row index is indicated on the left, the bits on the right represent the coefficient's binary form starting from the LSB. In a generic multiplier, the input on the right would represent the operand b. However, in this application, the coefficient is fixed. Starting from the top, nothing is instantiated until a '1' from b is encountered, as regular cells have no valid inputs. The inverted cells, on the other hand, contain a NAND gate. Whenever a '0' is input, the NAND gate returns '1'. Thus, every b_i equal to '0', generates a fixed '1' in the most significant column (a_8). When the first '1' is reached, regular cells can be still left out. This is due to the fact that, at this point, there are two inputs: one coming from operand a and one from b. However, as b_i is fixed to '1', the AND gate inside simply propagates a_i. As a consequence, the bits from a slide to the right by one position for each new '0', until the second '1' is found. No logic is introduced. Nonetheless, the leftmost regular cell is instantiated in the form of an HA that adds a_7 to the '1' generated by the previous NAND gate. This adder is simplified as one of the inputs is constant. When b_i is '1', the NAND gate in the inverted cell works as an inverter. The column next to the most significant one receives at least two inputs from the upper row and thus needs adders. The second '1' from b requires HAs to be instantiated. Column 7, which already used one, upgrades it to a simplified FA. As HAs and FAs produce two outputs, all the rows below the second '1' from b necessitate them. A HA is placed when b is '0' and a FA when it is '1'. The second to last row is made of inverted cells. As b_8 is always '0', the NAND output is fixed to '1'. FAs are instantiated to deal with three inputs. However, because of the constant input, they are simplified. Similar considerations can be made about the last row and the generation of p_{16}. In is important to note that the AND and NAND gates are never needed and the regular and inverted cells either reduce to 1-bit adders or are simply omitted. The output number is obtained by determining the decimal point position. In this case bits from p_{16} down to p_2 are considered (15 bits in total). Resources instrumental in obtaining bits beyond the MSB are not instantiated, hence the big X on some cells in Figure 5.11. By knowing the output data statistics it was determined that the maximum integer value coming out of the multipliers can be expressed on 7 bits. The remaining 8 bits are fractional. While a 16-bit output would add a bit to the fractional part, this gain in precision would be been lost after the WTA, as the integer part expands.
A 4-operand WTA is placed after the multipliers to compute the final sum. While its inputs are on 15 bits, internally the sign is extended to 16 bits which is also the width of the output. The BKA inside the WTA works on 15 bits and needs no carry-in.
In the next subsection, two more versions of the differentiation filter are presented. The first variant employs a second degree polynomial to fit the data. While the number of coefficients is unchanged, the number of resources needed is different. The second variant possesses the same parameters as the standard differentiation filter. However, while it instantiates the same amount of units, their word-length is bigger.

5.2.5 Differentiation filter: variants

A SG filter of order two and framelength nine is also tested. As explained before, this filter's coefficients lie on a straight line and are thus proportional. The following relations apply:

$$\begin{aligned} g_3 &= \frac{g_2}{2} = \frac{g_1}{3} = \frac{g_0}{4}, \\ g_3 &= -g_5 = -\frac{g_6}{2} = -\frac{g_7}{3} = -\frac{g_8}{4}. \end{aligned} \tag{5.11}$$

By replacing these identities in the FIR filter equation (Equation 5.1), the result is:

$$y[n] = (4*(x[n]-x[n-8])+3*(x[n-1]-x[n-7])+2*(x[n-2]-x[n-6])+(x[n-3]-x[n-5]))*g_3. \tag{5.12}$$

This means that the number of coefficients can be reduced to just one (g_3). The others are obtained by multiplying the input data by a constant (implemented by shifting). Hence, a single multiplier is required. It is placed at the end of the chain, before the output. By comparing the two versions discussed so far, it can be seen that the subtractors at the input are unchanged. Following subtraction, in place of the multiplier, a WTA is used to sum the four operands extracted from Equation 5.12, which become five once the multiplication by 3 is implemented as $p = 2a + a$, where $a = x[n-1] - x[n-7]$ to allow for the use of shifting.

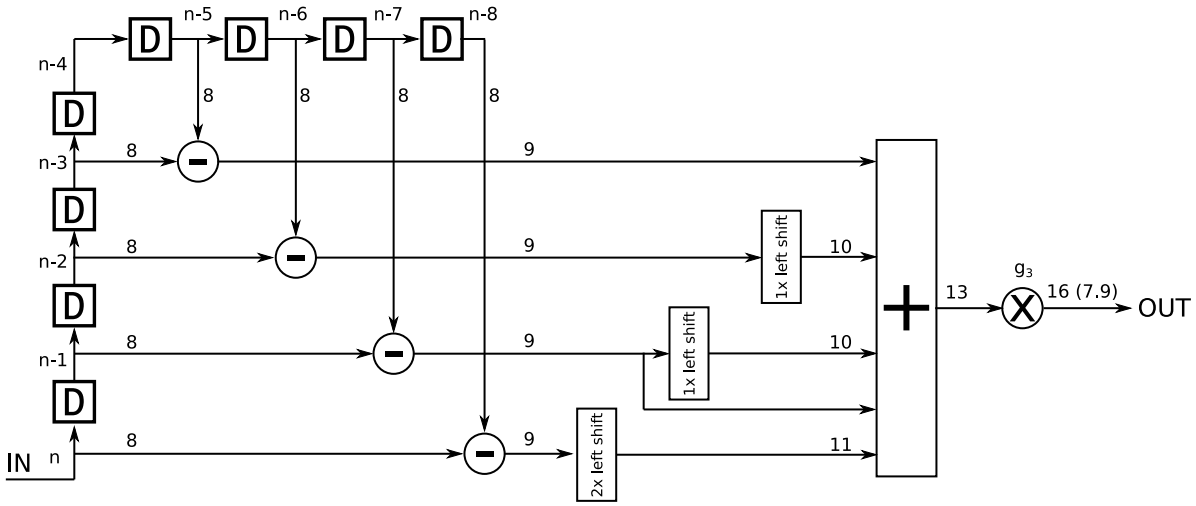


Figure 5.12: Second order differentiation filter implementation

As shown in Figure 5.12, the numbers at the output of the subtractor are defined on nine bits. Some of them are extended by shifting left. The longest word produced is 11 bits long after doing 2 left shifts. That's why a word-length of 11 was picked for the WTA. No simplifications based on the input word-length of the different operands are allowed here, unlike in the smoothing filter. That is because a signed sum requires the sign bits to be extended to 13, which is also the width of the output. The 13-bit number produced by the WTA is sent to the single BWM, which performs the final operation. This circuit is optimized for for coefficient g_3, which is the only one left in the design. Out of the 26 bits at the output only 16 bits are selected. As the output range for a filter of this order is narrower, seven bits are used for the integer part and nine for the fractional part, making this filter slightly more precise. Moreover, the entire filter operates on integer numbers. The fixed-point result is generated by the multiplier as the final output.
The last differentiation filter is instantiated after the smoothing procedure. The equation and the schematic for this filter do not change. However, unlike the first derivative filter, which has 8-bit inputs, this version receives the unsigned fixed-point data expressed on 16 bits. This means that the Brent-Kung subtractors also work on 16 bits and produce 17-bit outputs. Similarly, multipliers feature an extended input range to match the subtractor output. However, the bits selected as output reduce to 15 bits as in the first version. From this point on, the circuit is the same: a WTA adds all the operands to produce the signed 16-bit output. The coefficients fed to the BWMs are not re-computed on 17 bits to improve precision. Their 9-bit versions from earlier are simply extended to 17 bits by means of zero-padding. This is because the precision obtained by this operation would be lost at a later stage once the output word is shrunk.

5.3 Zero-crossing circuit

In Chapter 6, two versions of the zero-crossing circuit are tested. In this stage, the data coming from the derivative filter are scrutinized to find the position of the zero. The computation starts when two consecutive samples of opposite signs are observed. To increase robustness, thresholds are defined in order to filter part of the noise. Comparators detect a threshold crossing, while a special circuit checks for a sign change. The main calculation makes use of a divider to linearly interpolate the zero position between the two samples. An FSM takes decisions based on status signals and generates the appropriate control signals. The first version features two thresholds: one positive and one negative. The second implementation operates on the positive one only. The main purpose of a negative threshold (NT) is to compute the average position of a zero when multiple occurrences that are close to each other are detected. Removing the NT results in inaccurate zero coordinates in those cases in which the laser line is exceedingly wide. It will be shown later that, while the filters alone are able to reach a maximum frequency of several hundreds of MHz, the speed of this unit is limited by the array divider. Even with three stages of fine-grain pipelining, the circuit synthesized on a Cyclone V FPGA is not able to reach 100 Mhz. Some improvements are obtained post approximation. The zero-crossing system is also pipelined. However, not much can be done on the the maximum frequency without altering the timing of the FSM.

Figure 5.13 shows the circuit tasked with locating the derivative zeroes using two thresholds and a divider which is not internally pipelined.

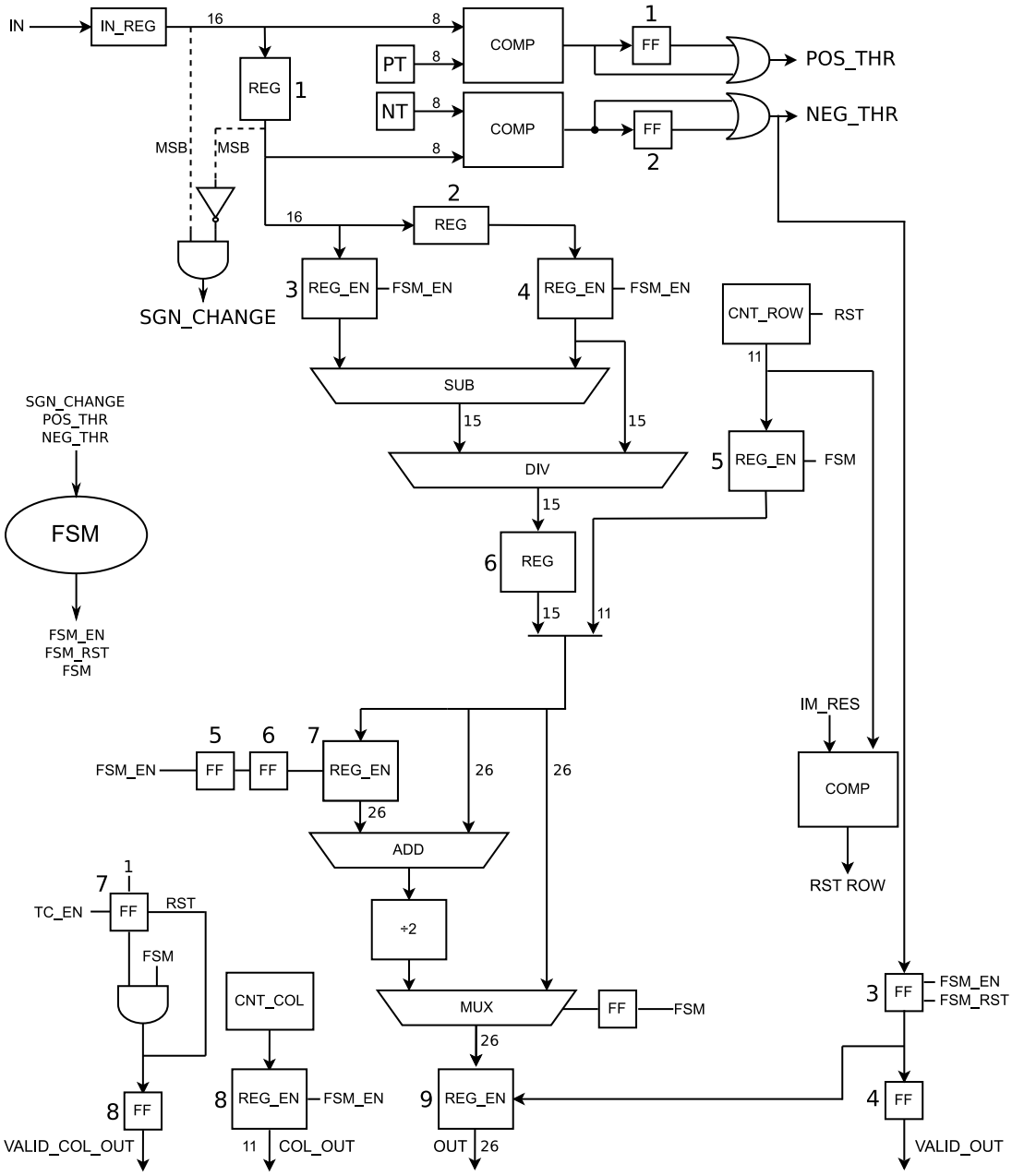


Figure 5.13: Zero-crossing circuit featuring two thresholds and average computation

The input register is needed to separate the circuit from the rest of the system pipeline. This is essential to avoid spurious commutations caused by noise that could impact power consumption. The FSM provides an active-high reset state that sets every sequential element to zero. Right after the input register, the 16-bit input signal is fed to a comparator that checks whether the positive threshold (PT) is passed. This value is specified as an input port so that it can be set up by the user based on the particular application. The bit at the comparator output is then inspected for a high value by the FSM, which then moves from an idle state, in which nothing is done, to a situation in which a zero is expected. In this second waiting condition, no new control signals are asserted. Instead, the control unit starts polling the path which detects a sign change by comparing the sign bits of two consecutive samples. The inversion is valid in one direction only (positive to negative). For the testing to be possible, one of the two successive samples is stored in register 1 for one cycle. To generate a match, a positive value should be present at the output of this register, whereas the newly sampled number should be negative. In the idle state, before PT is reached, a sign change generates no action from the FSM, as the zero in between is considered invalid. Once the sign inversion is detected, the FSM waits for the next rising

clock edge to further move the two samples with opposite signs down to register 2. A new state is reached that enables registers 3 and 4 via an FSM control signal. At their inputs are the two samples that will be used to compute the zero position. The FSM maintains this state only for one cycle as the registers need to sample once. It is important to ensure that only meaningful data is processed. That is why some sequential units are equipped with an enable signal, while others need to inspect all the data passing through to avoid missing precious information. The decoupling between the first part of the circuit and the complex arithmetic blocks placed next prevents unnecessary commutations that can propagate in depth at each clock cycle and increase power usage. As the enable is synchronous, the input is sampled at the following cycle. At this point, the data moves to the subtractor first and the divider next to put Equation 3.8 into effect. These operations can be carried out through different states depending on the status of the NT. A specific stage is devised to wait for this event. However, the amount of cycles required to observe the NT is not known in advance. Thus, part of the datapath keeps processing the data regardless of the state sequence. Only a reset signal can disrupt the flow at this point. The FSM waits for the status signal coming from the comparator notifying the presence of a sample that goes below the threshold. For timing reasons, this unit examines the input numbers that have been delayed by register 1. A shift of one cycle between the two comparators is crucial to correctly handle the particular cases in which the NT and the PT are crossed by two consecutive samples. This is also where FF1 and FF2 prove useful: the negative sample responsible for the sign change might already have a value that is below the NT. When this happens, the FSM is in a state that is not sensitive to the signal coming from the comparator. Thus, to avoid losing this bit of information, its output is saved for future use. It is then read by the following state specifically created to wait for the NT. A detailed description of all the states and their roles is given in the following subsection. Some particular cases related to the input samples are also illustrated.
Once the division is over, register 6 samples the output. This register's purpose is simply to reduce the combinational path of the circuit. Although given the bottleneck caused by the divider, it might not offer any advantages. The enable signal is not present since the registers on top prevent the output from changing unless a valid result is produced. As the dividend is always smaller than the divisor, the divider's output lies between 0 and 1. For this reason, the integer number indicating the row at which the zero was found, is simply attached to the fractional value. The row number is generated by a counter that resets M times, once every N clock cycles, where M and N are the horizontal and vertical resolution respectively. These numbers depend on the resolution of the image and can thus be specified by the user. A comparator sends a terminal count signal which passes through the FSM and forces it into a reset state. However, for timing reasons, it is also fed directly to some registers by ORing it to the external reset so that they do not have to wait for the state to change. To avoid unnecessarily erasing other memory elements, this particular reset signal is employed only by the counter, and registers 1 and 5. After register 6, the final path is crossed. Register 7 manages the average calculation and is enabled in the same state as registers 3 and 4. In reality, the enable signal reaches the register two clock cycles later due to FF5 and FF6. This is done to maintain a correct timing. Register 7 is designed to sample and retain the first zero for a number of samples spanning from the PT to the NT. This means that if multiple zeroes are detected, they will enter registers 3 and 4 but will not be stored in register 7. Because of this difference, the first zero is computed in one state, while all the other zeroes are obtained by means of a different state. As the enabling state is reached only once for each PT crossing, the register does not sample more than once per series of zeroes. While waiting for the NT to be crossed, the particular case in which another zero is found can arise. In this situation, the FSM transitions into the state that enables only registers 3 and 4. This is done for every zero between the first one and the NT detection. Each new zero passes through the divider and is discarded unless the NT is intersected. When that happens, the last zero in the sequence has been found. The unit then adds the first and the last zeroes and divides by 2 (right shift). Thus, the circuit does not actually compute an accurate average. Instead, it selects the two extremes and divides by two. This small approximation is necessary given that the number of zeroes between PT and NT is not known beforehand. The simplification avoids a potentially large number of operations on 26 bits and the need for a second, even bigger divider that would further reduce the maximum frequency. Depending on the number of zeroes found, a multiplexer guided by the FSM has to choose between the average and the result that comes directly from the divider. Once the NT is passed, the circuit goes back to *IDLE* state.
Registers 8 and 9 deal with the outputs. The row number combined with the fractional part represents one signal, while the column is sent out as a separate number. The column counter specifies the column on which the zeroes might lie. This circuit defines the horizontal resolution and is never reset as once the final number is reached, the whole image has been processed and the computation is over. As multiple zeroes for the same column are allowed, this number is output separately and specified only once to reduce the parallelism. The

moment in which a valid output is produced by the circuit is unknown from the outside. Therefore, two signals are used to validate the two outputs. These bits are evaluated by the testbench in order to know when to write a result on the output file. Register 8 features an enable signal to avoid unnecessary sampling (e.g. when a column has no zeroes there is no need to generate its number). It is activated by the FSM once the first zero is detected. As for register 9, the enable avoids sampling results in the middle that must be discarded in the case of an approximate average calculation. The enable and valid out signals are activated by the NT, as seen from the diagram. NT works like the *commit flag* in the software version: it marks the end of computation. They are, again, delayed by flip-flops to respect the timing. The numbers processed by this unit come from the differentiation filter and are thus defined on 16 bits.

5.3.1 FSM

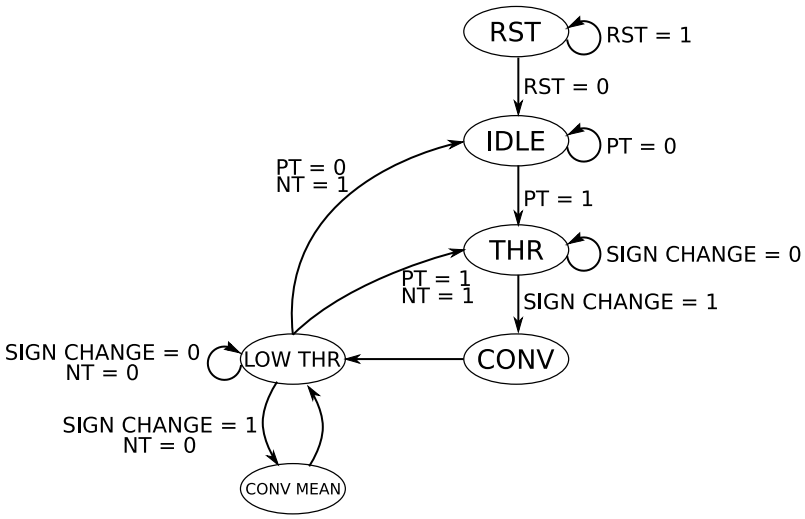


Figure 5.14: Zero-crossing circuit featuring two thresholds and average computation

The FSM used for the circuit with two thresholds is shown in Figure 5.14. This control unit features six states. Some of them work simply like software flags, while others actually drive the datapath via control signals. The reset state is activated either when a reset signal is sent from the outside or when the elaboration of a column is complete. In the first case, all memory elements are set to 0 and the processing computation stops as long as the reset signal is high. In the second case, the elaboration does not go further but the registers might retain their last value. As the reset bit comes from the outside, it can be activated at any moment during the computation. Thus, every state is sensitive to the reset and is connected directly to the reset state. The jump takes place at the first occurring rising clock edge. Synchronizers are necessary to avoid metastability due to the asynchronous signal. Once the signal is deactivated the machine moves to the *IDLE* state and waits for the PT to be crossed. When the PT is met *THR* is accessed. This state is equivalent to the *conversion flag* in Algorithm 2. No commands are issued to the datapath. Instead, the FSM becomes sensitive to the status bit that is active when the signs of two consecutive inputs reveal the presence of a zero in between them. When a sign change is detected, the *CONV* state is reached and registers 3 and 4 are enabled. The two input operands are sampled in the following cycle. At this point, it is certain that at least one zero exists for a certain column. Therefore, register 8, responsible for storing the contents of the column counter, is also enabled. This happens only once per column, provided that a zero is present. FF5 also gets this control signal at the input. After *CONV*, the FSM moves inevitably to *LOW THR*, which waits for the NT to be reached. Depending on the variability of the input data, several situations can occur. Consequently, this state has many different redirections. Normally, if a column contains only one zero, the FSM waits in the *LOW THR* state for the NT. Afterwards, the machine goes back to *IDLE*. When noise is present and the filtered signal varies fast enough, something unusual might happen. The NT comparator processes the number stored in register 1. At this point, a new sample is present at the output of the input register. Thus, in the same cycle, the PT comparator is already evaluating possible PT crossings. If this occurs right after the NT is passed, the FSM moves directly to *THR* instead of idling. Figure 5.15 shows a simple example of a discrete signal that produces this situation. As this case might occur as early as the *CONV* state, flip-flops 1 and 2 save the state of the comparators. In *LOW THR*, the FSM starts

searching for NT. Both the FF bits coming from the previous cycle and the new ones are inspected. With this strategy, all the status checks are lumped together in one state. This avoids having to deal with multiple states defined by the same conditions.

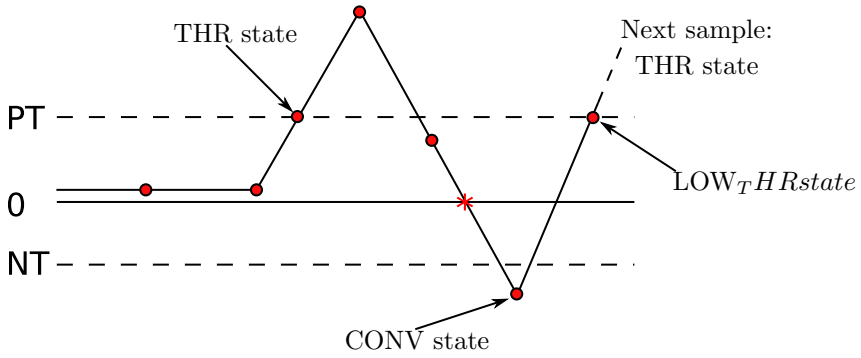


Figure 5.15: Zero-crossing circuit featuring two thresholds and average computation

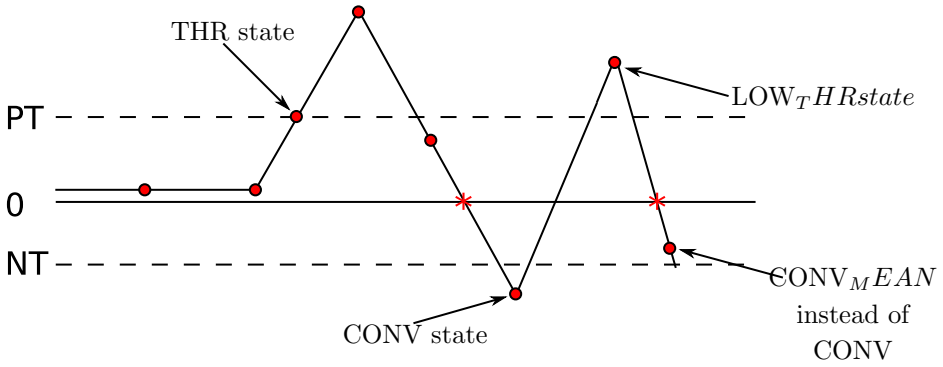


Figure 5.16: Zero-crossing circuit featuring two thresholds and average computation

Figure 5.16 shows a rare case that is processed incorrectly when using the approach presented. As the NT is passed, the second zero should be separated from the first one. Instead, an average is calculated and only one number is observed at the output. In other words, the state *CONV MEAN* is reached instead of *CONV*. The computational error is not significant as the two zeros are really close to each other. *CONV MEAN* is a state that can be accessed only from *LOW THR*. The FSM jumps to this state in the event a new zero is detected but the NT was not yet passed. All the zeroes that occur after the first one are computed by it. *CONV MEAN* is similar to *CONV* as it enables registers 3 and 4 but not register 7. This last memory element stores the first zero of a series that is then summed to the last one to compute the average. This is why two different states are devised. All the zeroes between the first and the last are discarded. The FSM jumps between *LOW THR* and *CONV MEAN* several times until the NT is observed. Moreover, in *CONV MEAN*, a signal coming from the FSM acts as the multiplexer control signal and chooses the path containing the adder. The signal is sampled by a flip-flop before the multiplexer. This is done to retain the correct multiplexer status even if the FSM reverts back to *LOW THR*. The flipflop is reset once the NT is passed. The signal out of the NT comparator is used directly as the valid out after being delayed the right amount by flip-flops. A special circuit featuring FF7 and FF8 is used for the validation signal of the column as it only has to be active one clock cycle for each column (provided it contains a zero). In order to simplify the zero-crossing circuit, the mean computation section can be left out. In this case, the NT is not necessary anymore and both the datapath and the control unit simplify greatly. Figure 5.17 shows the new system. It can be seen that the comparator for NT is not needed anymore. Moreover, most of the flip-flops used before disappear. The same applies to the register, the BKA and the multiplexer instantiated specifically for the average. The pipeline register (indicated as register 6 in Figure 5.13) is also no longer necessary as there is no combinational path between that and the output register. Since registers 3 and 4 already act as "enable barriers", the subsequent memory element does not make use of an enable signal. The same FSM control bit that enables registers 3 and 4 also validates the output number after a delay of 3 flipflops or more (if additional pipelining is required). The FSM reduces to the one in Figure 5.18.

As can be expected, the *LOW THR* and *CONV MEAN* states disappear. The status signals include the reset, the PT comparator, the sign change and the terminal count. The control signal that activates registers 3 and 4 and produces the valid output signal is the only present.

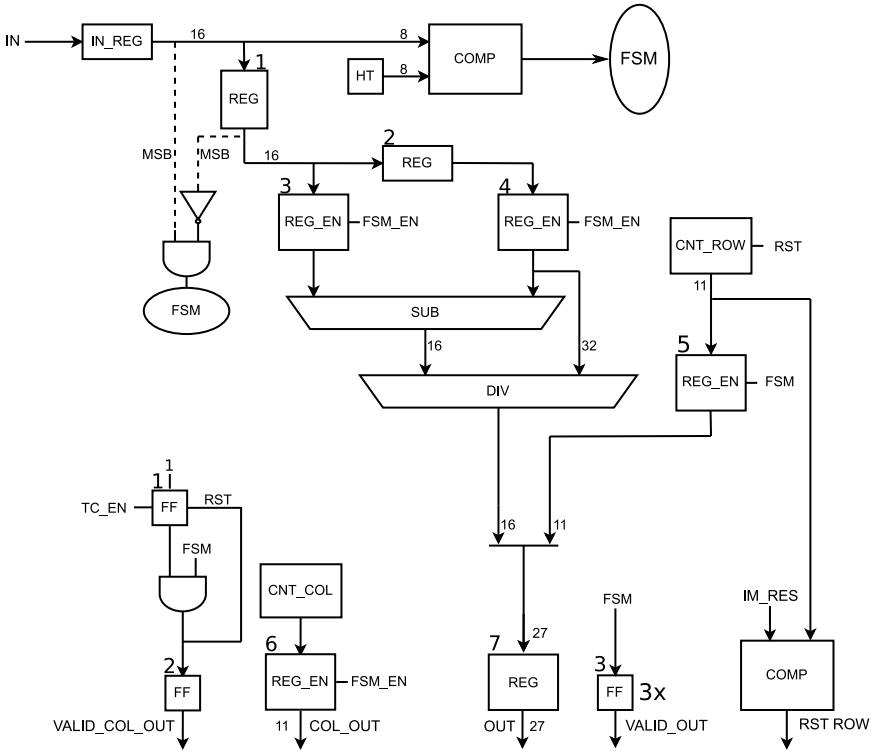


Figure 5.17: Zero-crossing circuit featuring two thresholds and average computation

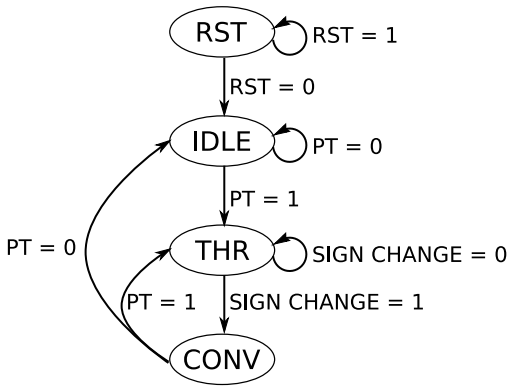


Figure 5.18: Zero-crossing circuit featuring two thresholds and average computation

5.3.2 Exact divider

An unsigned divider is used to compute the zero position. As previously shown in Figure 4.6, an array divider's dividend z should be double the size of the divisor d. In this case, both input numbers are 15 bits wide, so the dividend must be represented on 30. It follows that the upper half of the dividend is always null and is, thus, wasted. Moreover, as the output is always less than 1, an integer division would always return a quotient q equal to 0 and a remainder s equal to z. To prevent this, the 15-bit input acting as the dividend is shifted left by 15 positions so that the lower part is always 0. In this way, the quotient contains the full final result, which is also a completely fractional number defined on 15 bits. Starting from the eight decimal bits coming out of

the differentiation filter, the number at the output of the divider is expanded to 15 bits. Hence, the overall precision on the position of the zero is dramatically improved, in fact, it doubles. In this case, subpixeling is achieved by exploiting the redundant bits required by the divider. This procedure is similar to the one used for the multipliers in the filters, where coefficients are scaled up to produce a more accurate result. Because the lower part of z has all bits set to 0, the cells belonging to the first column from the right, which also have no b_{in}, can be greatly reduced. By looking at Equation 4.16, when two of the inputs disappear, the remainder is obtained by a simple AND gate. Additional improvements are achieved by considering that the remainder signal, produced by the last row, is not used. As a consequence, the logic functions of these cells can also be simplified and only b_{out} is computed to get q_0. The divider features three levels of pipeline registers as shown in Figure 5.20. They are necessary to reduce the delay of the circuit, which due to its limited speed, determines the maximum frequency of the whole system. The final version, however, contains five pipeline levels, each with three rows, for a total of 15 bits. It is important to note that the divider in Figure 5.20 is not the one implemented, although they have the same architecture. In the divider, the partial remainders propagate from top to bottom, while the quotient is evaluated from right to left. However, given the presence of a feedback loop in each row, the quotient computation cannot be pipelined. A series of registers is instantiated right after the q output bits as a means to delay them by the appropriate amount of cycles to obtain a meaningful result. In Figure 5.19 the contents of the blocks on each row are revealed. Every q_i is fed back to the corresponding unit where it is used by all cells to compute the partial reminder. The cells implement Equation 4.16. The last block is simpler as no remainder is produced.

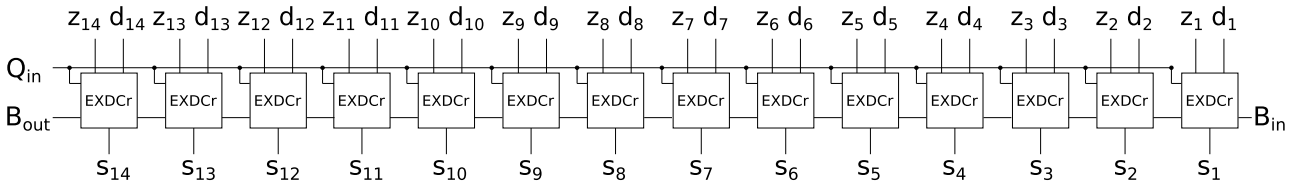


Figure 5.19: Single row of the 15 constituting the array divider

The 15-bit subtractor preceding the divider is implemented using a BKA. The considerations made for the filter apply to it as well. To keep the total error under control and avoid nullifying the effects of subpixeling, this circuit is not approximated. The result of the subtraction is always a positive number. Thus, the sign bits of the two operands are left out. This is why the circuit works on 15 bits rather than 16.
The final adder that deals with the average computation is also a BKA and it can be approximated. The section of the circuit responsible for the calculation of the average is not present in the zero-crossing version with no NT. As shown in Figure 5.13, the output of the divider and of the row register are combined to form a 26-bit number. Hence, all the components after that point, including the BKA, share the same parallelism. Both counters are defined on 11 bits, setting a limit on the maximum resolution supported. Their implementation is behavioural. This applies also to the multiplexer.

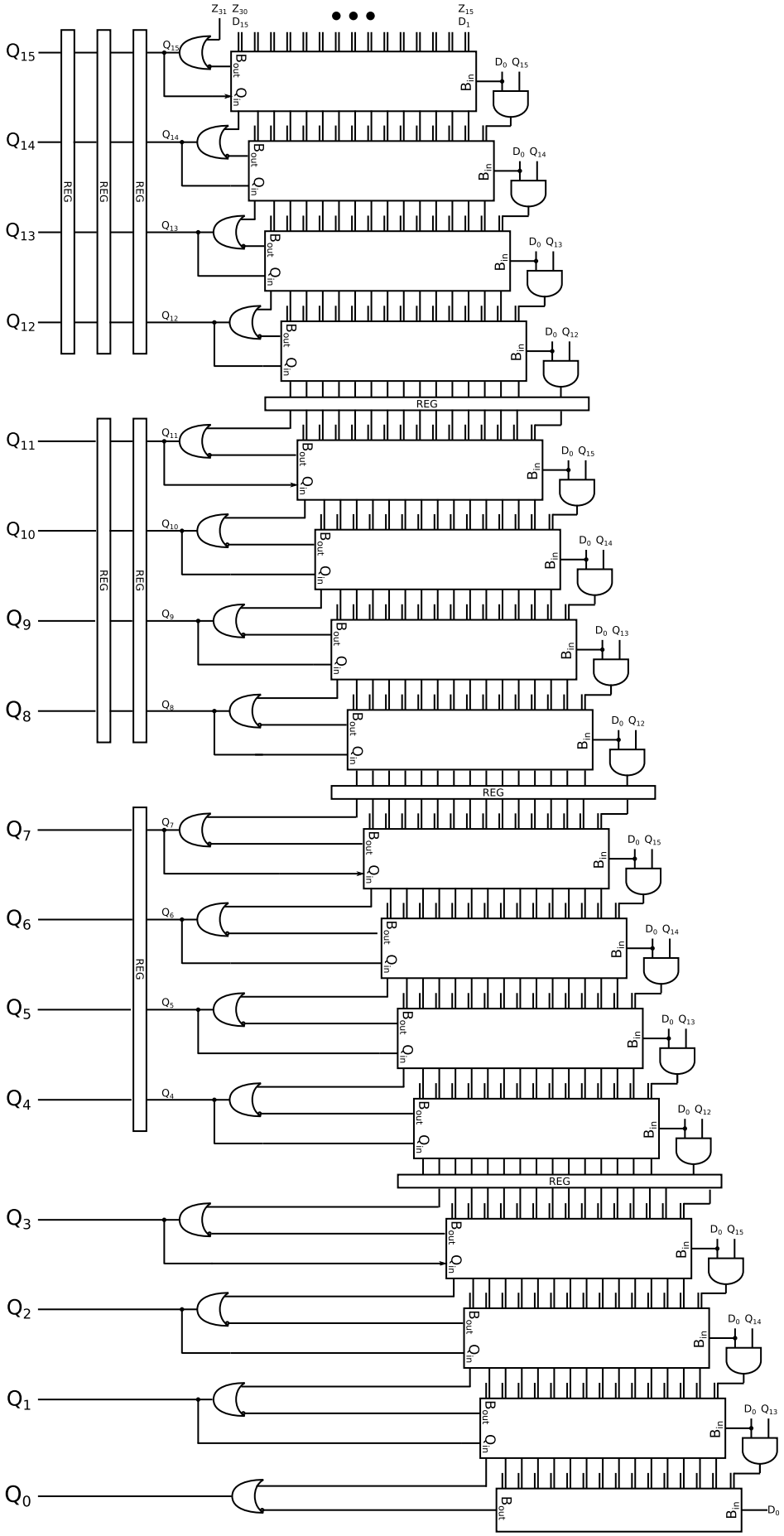


Figure 5.20: 32x16 pipelined array divider

5.4 Approximate arithmetic circuits

The hardware implementation of the five approximate circuits is discussed in this section. Each units is simulated individually and the most important metrics are estimated. This evaluation is performed on the software version of these circuits, implemented in MATLAB, when there is no difference with the hardware counterpart. In other cases, the VHDL model is tested. Either way, the metrics are computed on MATLAB. As the purpose of this project is to assess the impact of approximate circuits as components of a bigger system, the single units are not tested extensively. A more comprehensive analysis can be found in the papers cited in this thesis.

5.4.1 LOA

As explained in Chapter 4, this first adder shown in Figure 4.1, is divided into two parts. The approximated section is simply implemented using OR gates. A final AND gate is inserted to generate the carry-in for the accurate part, which makes use of a BKA. The circuit is employed either as an adder or a subtractor, depending on its purpose, and it can handle signed and unsigned numbers. When the input operands are defined on an even number of bits, the accurate and approximate parts are split equally. In case of an odd number, the leftover bit is assigned to the accurate part to avoid degrading precision excessively. A problem arises when adapting the LOA to work as a subtractor. As in the approximate section no carry propagation takes place, performing the two's complement by inverting the input bits and then adding '1' as the carry-in is not possible. This entails that such an operation must be carried out at an earlier time by instantiating a specialized circuit that adds a '1' to the inverted number. While this adding unit is much simpler than a standard adder, it partially cancels the benefits of approximate computing. In particular, the delay increases significantly and the only way to fix it is by inserting a pipeline register. Alternatively, one might just skip the +1 operation and simply perform the one's complement. It will be shown that this procedure dramatically raises the ER. However, the maximum ED is not affected so the overall effect on the filter does not compromise the validity of the results. It is important to note that the expressions defined for some of the metrics concerning the LOA, such as Equations 4.1 and 4.3, are not valid for this last version. This happens because virtually all input combinations can generate an error. Normally, given two bits at position i (one for each operand), an error is produced by the pair only in case both bits are equal to '1'.

5.4.1.1 9-bit LOA adder and subtractor

A 9-bit LOA adder [43] is found in the smoothing filter. As explained before, the unit deals with image samples expressed as non-negative integer numbers, i.e. all positive numbers including zero. In this circuit, four bits are reserved for the accurate part. The remaining 4 bits are approximated. A 4-bit BKA returns the MSBs of the result, as well as the carry-out representing the ninth bit. Given the small word-length, a thorough simulation for this circuit was performed and all the input combinations were tested. In this case, 2^{16} possible combinations exist. Due to the odd symmetry of the coefficients, the adder is replaced by a subtractor in the differentiation filter. The hardware implementation and the bit-widths of the two parts is the same. However, the second input is simply inverted. This introduces an additional error, which does not affect the quality of the filter's output signal significantly and can thus be tolerated. A graphical side-by-side comparison with the exact filter output will be presented in Chapter 6.
The simulation of the 9-bit LOA subtractor is performed by testing all combinations once. As the input samples are defined on eight bits, this number amounts to 65 536 (2^{16}). The output range goes from -255 to 255, nine bits are required to account for the sign. In the first simulation, the correct input is fed to the subtractor. In other words, the two's complement of the second operand is performed correctly. The 4-bit approximate part is examined separately. When considering two operands, 256 possible combinations exist. The ER estimated in [43] is expressed by Equation 4.1 and it depends on the amount of bits assigned to the approximate portion. For a 4-bit lower part, this number is expected to be 0.6836, or roughly 68%. The simulation result obtained by generating every possible combination matches exactly this theoretical value, as shown in Table 5.3. The maximum ED was found to be 8, which is also the weight of the approximate MSB. By analyzing the simulation data, it was also possible to extract the probabilities associated to each ED starting from 0 (correct result). In Table 5.1 the figures related to the EDs are shown. The second column contains the number of output combinations that exhibit the specific ED indicated in the previous one. As expected, the output cases add up to 2^{16}. The third column shows the number of input combinations that generate a specific ED by taking into account only the lower four bits. This is why they are only 256 in total. To obtain the full number of

combinations shown in the second column, it is sufficient to multiply each entry by 256. The fourth column is put together by dividing either of the two previous columns by the total cases, e.g. 2^{16} for column two and 256 for column three. This probability is used to compute the MED. The last column includes the mathematical model to predict the probabilities even without any simulation.

ED	Output cases	Input combinations	Probability	Probability estimation
0	20736	81	0.3164	0.75^4
1	7168	28	0.1094	$0.75^3 * 0.25 + 0.25^4$
2	7680	30	0.1172	$0.75^3 * 0.25 + 0.75 * 0.25^3$
3	3072	12	0.0469	$0.75^2 * 0.25^2 + 0.75 * 0.25^3$
4	9216	36	0.1406	$0.75^3 * 0.25 + 0.75^2 * 0.25^2$
5	3072	12	0.0469	$0.75^2 * 0.25^2 + 0.75 * 0.25^3$
6	4608	18	0.0703	$0.75^2 * 0.25^2 + 0.75^2 * 0.25^2$
7	3072	12	0.0469	$0.75 * 0.25^3 + 0.75^2 * 0.25^2$
8	6912	27	0.1055	$0.75^3 * 0.25$

Table 5.1: Error distance probabilities for standard LOA adders and subtractors.

From the table it can be seen that a large portion of combinations yields a correct result ($ED = 0$). The reason is that, for each pair of bits with equal weights coming from the two operands, only one combination out of four generates an error. In other words, the sample space: $A_iB_i = \{(0,0);(0,1);(1,0);(1,1)\}$ exists for any pair i of bits belonging to operands A and B. The condition for which no error is detected happens anytime all the four approximate bits do not display the erroneous case. This means that none of the four pairs exhibits the (1,1) combination. This takes place with a 3/4 chance for each couple. For instance, the numbers:

operand 1 = xxxx1001,
operand 2 = xxxx0100,

will produce no error, as the (1,1) combination is not present. The joint probability is obtained by multiplying this estimate by itself four times (i.e. 0.75^4). On the contrary, the operands:

operand 1 = xxxx1001,
operand 2 = xxxx0101,

will yield an incorrect result, due to the (1,1) that affects the LSBs. This case, however, is less frequent and its overall probability is given by $0.75^3 * 0.25$, i.e. three correct pairs (0.75^3) times one incorrect pair (0.25). As the erroneous combination occurs at position $2^0 = 1$, the ED for the two operands will be one. Similarly, an ED of eight is observed when the pair of approximate bits that weighs the most (2^3), is equal to (1,1). The probability of this event is 1/4 multiplied by the probability that each of the remaining three pairs produce an exact result (0.75^3).
By this logic, the following operands should generate the maximum ED of the circuit:

operand 1 = xxxx1111,
operand 2 = xxxx1111.

In this case, all pairs produce a wrong result. By summing the weights of each incorrect pair, the ED would be equal to 15. However, this does not happen. All the combinations that would normally generate an ED greater than eight have their EDs reduced to a number between one and seven. This is possible because of the AND gate that serves as a carry-in for the BKA and it achieves error compensation. Thus, the second terms that appear in Table 5.1 under the "Probability estimation" column take into account the additional combinations whose EDs have been mitigated by the AND gate.
By looking at the previous example, an ED equal to 1 is obtained not only when the pair at position 2^0 is equal to (1,1) but also when:

operand 1 = xxxx1111,
operand 2 = xxxx1111.

This additional combination has a $0.25 * 4$ probability, as shown in Table 5.1. In general, to predict the final ED when error compensation is exploited two cases can be distinguished. They depend on the value of the most significant bits in the approximate part, which in this example have a weight equal to 2^3. This pair can either produce the right result by showing one of the three correct combinations or generate the wrong result by ORing two '1's. In the first case, the ED can be calculated simply by accumulating the weight of the pairs of bits, from the operands, that are both equal to 1. For example an ED of 7 can be seen as $2^2 + 2^1 + 2^0$, which

means the three least significant pairs are all equal to 1. Figure 5.21 shows some examples.

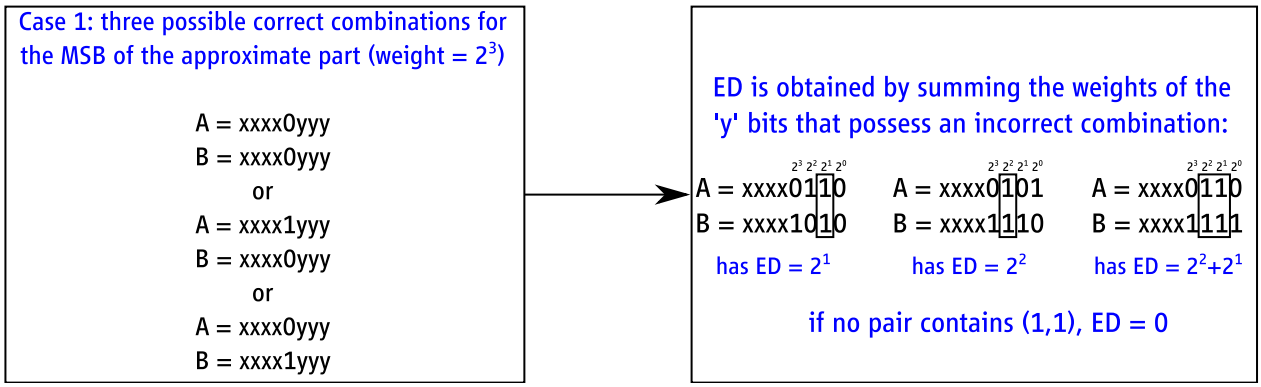


Figure 5.21: Error estimation in LOA, case 1

If the erroneous combination appears solely on the MSBs of the approximate part, the number will have an ED of eight. As explained earlier, if additional pairs contain an error, the ED is greater than eight. However, the error compensation reduces the overall ED. Thus, in this case, a subtraction can be performed rather than an addition. To obtain an ED of two, for instance, the operation to be performed is $2^3 - 2^2 - 2^1 = 2$. This principle is shown in Figure 5.22.

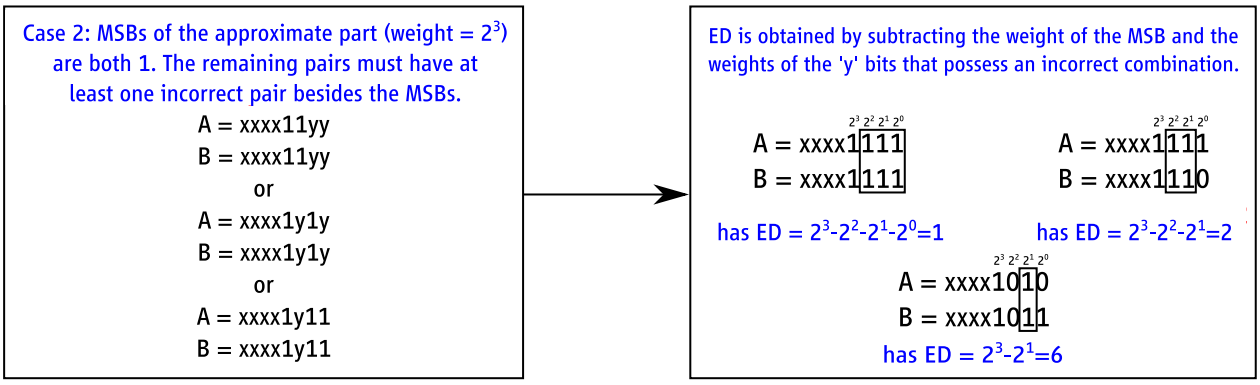


Figure 5.22: Error estimation in LOA, case 2

In case of an ED of two, three additional combinations are present. By simply computing $0.75^3 * 0.25$, the number of possible arrangements is equal to 27. However, the simulation shows that 30 combinations have an ED equal to two. The effect of the error compensation mechanism explains the initial discrepancy between the probability estimated from the empirical data and the one obtained without considering these extra combinations. Simply multiplying the probability of each pair is not enough in this case as it does not consider the full range of possibilities. The three additional combinations occur when

operand 1 = xxxx111y,
operand 2 = xxxx111z,

where yz can be "00", "01" or "10" but never "11". This explains the presence of three extra arrangements associated with this particular ED. Depending on which pair of bits is not fixed to "11", several additional combinations can add up to the total. They can be obtained by applying $0.75^x * 2^{2x}$, where x is the number of pairs of approximate bits that cannot be equal to "11". $2x$ takes into account the fact that two operands are present. The result is proportional to the size of the approximate part.
The data presented in Table 5.1 also apply to the approximate adder, whereas everything changes when the one's complement version of LOA subtractor is tested. Table 5.2 shows the results.

ED	Output cases	Input combinations	Probability	Probability estimation
0	256	1	0.0039	0.25^4
1	21504	84	0.3281	$0.75^4 + 0.75 * 0.25^3$
2	7680	30	0.1172	$0.75^3 * 0.25 + 0.75 * 0.25^3$
3	9216	36	0.1406	$0.75^3 * 0.25 + 0.75^2 * 0.25^2$
4	3072	12	0.0469	$0.75^2 * 0.25^2 + 0.75 * 0.25^3$
5	9216	36	0.1406	$0.75^3 * 0.25 + 0.75^2 * 0.25^2$
6	4608	18	0.0703	$0.75^2 * 0.25^2 + 0.75^2 * 0.25^2$
7	9216	36	0.1406	$0.75^2 * 0.25^2 + 0.75^3 * 0.25$
8	768	3	0.0117	$0.75 * 0.25^3$

Table 5.2: Error distance probabilities for one's complement LOA subtractor

As expected, the most occurring ED is one. All the combinations that previously resulted in a correct number, now suffer from the error of the omitted +1 addition. Only 256 combinations out of 65536 provide a correct result, which translates into a single configuration out of 256 when focusing on the approximate bits. This configuration is:

operand 1 = xxxx1111,
operand 2 = xxxx1111,

and it is again a result of the compensating AND gate. In general, the two situations explained before can still be applied to estimate each ED probability. However, to obtain the new ED a 1 must be added to the final value when calculating the sum. In case of subtraction, the 1 must be subtracted. For example, an ED of 3 can normally be seen as $2^1 + 2^0$, meaning that a pair of bits in those positions should exhibit the incorrect combination. In this new situation, this ED is observed when the erroneous combination solely affects 2^1, as the +1 originates from the partial two's complement. In the second case, when the approximate MSBs are both 1, the ED, previously determined by $2^3 - 2^2 - 2^0$, is simply estimated by $2^3 - 2^2$. Table 5.3 shows the main metrics for the LOA. The results are based on the simulation of an 8-bit circuit. Two sutractors, one that performs the two's complement and the other the one's complement, are tested. Along with them, an unsigned adder is also simulated.

Metric	Adder	Two's complement subtractor	One's complement subtractor
ER	0.6836	0.6836	0.9961
MED	2.875	2.875	3.375
Max ED	8	8	8
Max NED	0.0157	0.0314	0.0314
MSE	16	16	16.5
RMSE	4	4	4.062
NMED	0.0056	0.0113	0.0132
Mean error	0.25	0.25	-0.75
Max RED	0.5	8	7
Min RED	0	-3.5	-8
MRED	0.0149	0.0248	0.0126

Table 5.3: Important metrics comparison between different LOA adder and subtractor designs

The figures from the table demonstrate that the metrics for the adder and the subtractor match in most cases. The main differences stem from the fact that the adder works with unsigned numbers, unlike the subtractor. Quantities like ER and MSE show values equal to the ones expected. On the contrary, the one's complement subtractor displays a huge ER and a higher MED, due to its ED distribution. It is interesting to note that the mean error was expected to be -0.25. However, the simulation produced a different number (i.e. 0.25) for both the adder and the subtractor. This might be due to the data processed by these units: the adders and subtractors have 8-bit positive inputs, while outputs are on nine bits. Although only a particular kind of LOA used in this project is analyzed, the considerations and the results discussed can be extended to any circuit of any word-length, with the proper adjustments. All the other version featured in the project are listed in Table 5.4

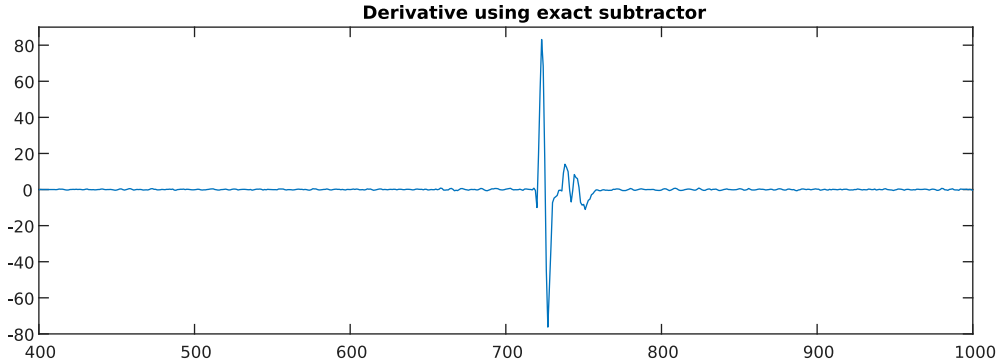
Type	Input bits	Output bits	Data type	Accurate bits	Approx. bits	Used in
Adder	8	9	Unsigned int	4+carry-out	4	Smoothing
Adder	15	16	Unsigned FP	8+carry-out	7	Smoothing
Subtractor	16	16	Unsigned FP	8	8	Smoothing
Subtractor	8	9	Signed int	4+carry-out	4	Differentiation
Adder	15	15	Signed FP	8	7	Differentiation
Subtractor	8	9	Signed int	4+carry-out	4	2nd order derivative
Adder	11	11	Signed int	6	5	2nd order derivative
Subtractor	16	17	Signed FP	8+carry-out	8	Diff. after smooth
Adder	15	15	Signed FP	8	7	Diff. after smooth
Adder	26	26	Unsigned FP	14	12	Zero-crossing

Table 5.4: List of LOA units instantiated in the system

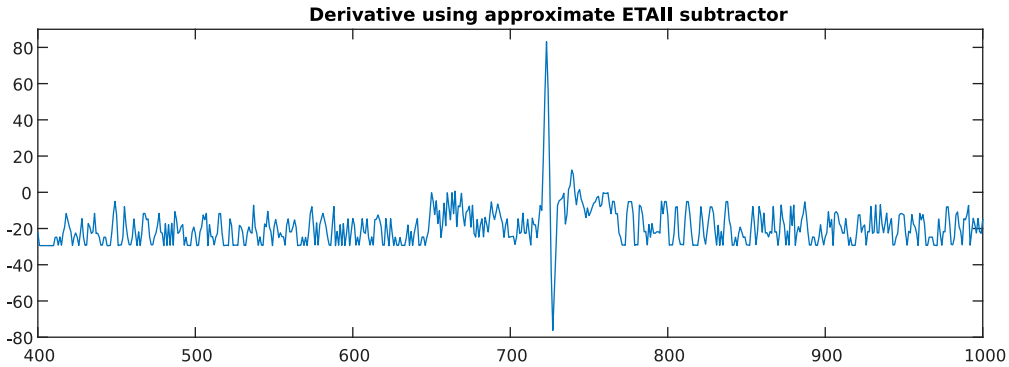
5.4.2 ETAII

The ETAII adder splits the carry propagation chain that produces a faster implementation while introducing an error [62]. Also for this circuit, several hardware versions exist that operate with different word-length and either signed or unsigned numbers. Similarly to the LOA, the ETAII subtractor faces some hurdles when implementing the two's complement operation. As the carry chain is sliced, setting the carry-in signal to '1' following bit inversion would only affect the first carry generation block, responsible for the LSBs. This sub-unit is then plugged into the neighbouring sum generation block, where it stops advancing. Consequently, unless a carry bit is generated in the subsequent carry generation block, the rest of the circuit is not updated to the correct result. This problem affect also adders. ETAII circuits are made of two main blocks shown in Figure 4.3: the sum generator which produces the sum bits, and the carry generator, which produces one carry-out bit connected to the following sum generator. Different variants of the carry generation block exist, as explained later. Since the blocks' size is limited to three or four bits in most cases, the choice of implementation does not play a role as the advantages on area and delay among different architectures are negligible at this scale. For this reason, following the design introduced in [62], the carry generator is based on the CLA, while the sum generator features a RCA.
The ETAII subtractor designed for the single differentiation filter works on nine bits. While both the exact implementation and the LOA version are defined on eight bits plus a carry-out, this subtractor extends the input operands to nine bits and then computes the result. This is done to better split the circuit and avoid blocks that are either too big or too small in size. Moreover, with this strategy, the last carry generator, which computes the final carry-out (tenth bit), can be left out. The nine bits are split and sent to three sum generators that manage three bits each. The carry generators, as stated before, are only two. The second operand is always inverted before entering the subtractor. The +1 that realizes the two's complement is input as the carry-in. A version similar to the LOA, performing the one's complement, is also tested. Since the input operands are unsigned, sign extension bits are known and fixed. The first number is padded with a zero, the second one is inverted to 1 before entering the subtractor. As a result, the block that processes this pair of bits could be potentially optimized. However, this is not done as a way to preserve modularity at code level and improve readability. The sum generators define a carry-in port that is connected to the previous carry generator. This last circuit normally works on a section of the operands and requires no other input. A first variant is created to account for the carry-in bit that the first carry generator (LSBs) needs in a subtractor. This signal is not listed as an input port, instead, the logic function inside are adjusted to take into account the fixed '1'. On the other hand, the subtractor that implements the one's complement does not need this version. For convenience, these circuits will be called plus-one-subtractor and plus-zero-subtractor respectively. The ED distribution for the plus-one-subtractor is shown in Table 5.5. Again, all the 2^{16} input combinations are tested and the input range undergoes no change. Roughly 92% of the results appear to be correct. However, the few that are left over, exhibit a huge ED. For the filters to produce a reliable output signal, the maximum ED for an approximate circuit must be limited. The maximum amount admissible grows with the word-length. By itself, the ER does not say much about the final outcome, as the error can be negligible. In this case, although the ER is pretty low, the ED is unacceptable even if very few cases produce it. The effects on the final waveform can be catastrophic, as shown in Figure 5.23. In the graph, a differentiation filter mounting a 9-bit approximate ETAII subtractor processes a column. All the other components are exact. Pairs of inputs such as $(0, 0)$ produce the biggest ED. That is

why the zero axis is exceedingly shifted downwards. While the zero can still be extracted from Figure 5.23b, it is important to avoid the further degradation introduced when other approximate circuits are added to the filter. For this reason, ETAII circuits displaying this behaviour were corrected, as explained later in this section.



(a) Output of an exact filter



(b) Effect of the approximate 9-bit subtractor on the differentiation filter's output

Figure 5.23: The same column processed by an exact differentiation filter (a) and an approximate filter using the ETAII subtractor (b)

Table 5.6 summarizes the EDs for the plus-zero-subtractor. The outcome is similar to the plus-one-version. The metrics of the two designs are compared in Table 5.7. A 9-bit adder using the same architecture is also tested. This circuit is used in the smoothing filter and deals with unsigned numbers, no two's complement is necessary.

ED	Output cases	Input combinations	Probability
0	60928	81	0.3164
64	4608	28	0.1094

Table 5.5: Error distance probabilities

The ED of 64 is observed for input combinations that generate a carry bit in the first block (three LSBs), originating either from the operands or because of the carry-in, that needs to propagate all the way to the MSBs. Due to the approximation, this transmission is hampered by the discontinuities in the chain. A simple example that highlights this issue is demonstrated by the pair of operands $(0, 0)$. A subtractor whose carry-in bit is set to '1', will propagate this carry no further than the next block. Before computing the subtraction, the bits for b are inverted:

$$\text{a} = \text{“000000000”},$$
$$\text{b} = \text{“111111111”}.$$

Normally, the carry-in bit set to one would generate a carry that is propagated from the LSB to the MSB. However, in this case, the carry-in stops at the following sum block unless a new carry is generated in the adjacent carry generator. Figure 5.24 shows this problem.

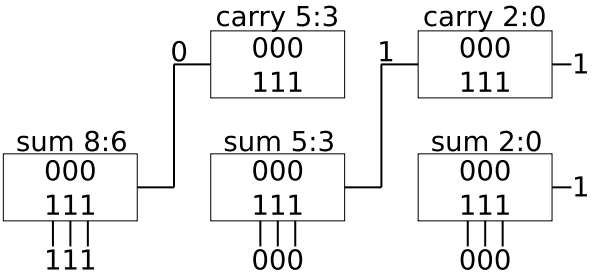


Figure 5.24: Example of a result with maximum ED

The propagation of the carry-in allows sum generators to produce correct results from bit 5 down to bit 0. However, the leftmost carry generator works independently and does not interact with the previous stages. Given the three input bits, the resulting carry-out is, of course, 0. As a consequence, due to the incomplete inversion of the MSBs, the final result is -64.

ED	Output cases	Input combinations	Probability
1	61952	81	0.3164
65	3584	28	0.1094

Table 5.6: Error distance probabilities

The plus-zero-subtractor never produces a correct result. Due to the missing addition at the beginning, the EDs are shifted by 1. The number of output cases that fall in the maximum ED tier is lower than before. The same distribution applies to the adder as well. However, as it does not perform the two's complement, its EDs are 0 and 64 rather than 1 and 65.

Metric	Adder	plus-one-subtractor	plus-zero-subtractor
ER	0.0547	0.0703	1
MED	3.5	4.5	4.5
Max ED	64	64	65
Max NED	0.2510	0.2510	0.2549
MSE	224	288	232
RMSE	15	16.9706	15.2315
NMED	0.0137	0.0176	0.0176
Mean error	-3.5	-4.5	-4.5
Max RED	1	64	65
Min RED	0	-1.1228	-1.1404
MRED	0.0171	0.43	0.43

Table 5.7: Error distance probabilities

As expected, out of the three circuits, the adder exhibits the best results. However, as its maximum ED is 64, it also needs a correction. This kind of expedient is not always needed: it appears that larger circuits are not affected by this error as much. Despite having a 100% ER, when equipped in a filter, the plus-zero-subtractor produces a better output result compared to the plus-one counterpart. Leaving the carry-in bit out introduces a negligible error, as observed also with the LOA. Nevertheless, both results are unacceptable. To overcome this obstacle, the solution adopted in [62] is considered. To reduce the error, Zhu et al. connected the carry generators processing the MSBs by feeding the carry-out of one block to the carry-in of the next one. In other words, n-bit CLAs are combined externally in a ripple-carry fashion as shown in Figure 5.25. In this way, the advantages of the CLA over the RCA are still preserved. The second variant of the carry generator, created specifically for this purpose, features and additional input port to account for the carry-in.

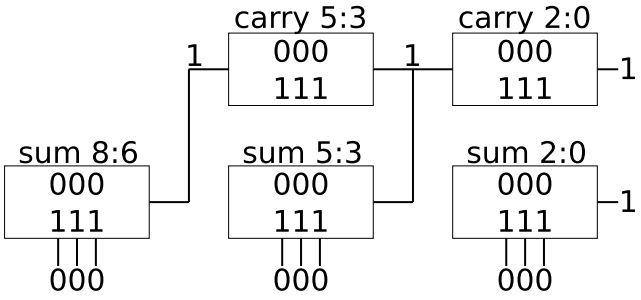


Figure 5.25: Exact ETAII circuit after correction

In 9-bit adder and subtractor, only two carry generators exist and by connecting them, the chain is restored, resulting in an exact entity. Despite not being approximate, this circuit is employed in the filters as its performance is on par with other inexact adders, without the burden of the error. Units with larger word-lengths tolerate the error better given its much less predominant weight, which exclusively affects the fractional bits. As a consequence, large circuits do not make use of this connection. Only in some cases is the connection required for the MSBs, but it is never extended to all the carry generators. Thus, most of the ETAII components found in the project are approximate. On the other hand, the adder shows better results overall, such as lower ER and mean error, as it does not manage the two's complement. The maximum ED, however, is still 64. Similarly to the subtractor, an ED of 64 occurs when the carry generated at the LSBs does not propagate until the end. In the adder's case the three MSBs are all equal to 0 instead of 1.
When compared to a LOA of the same size, the ETAII shows a negative average error. Its sign can accessory to the choice of the thresholds in the zero-crossing circuit. A negative error will shift the noise downwards so that it does not trigger the PT. Due to the larger ED, quantities such as the MED also increase. It is interesting to note the difference between the RED for an adder versus the one for a subtractor. As the former works with unsigned numbers, the errors start arising when the final result affects the MSBs, i.e. when it is greater than 63. Thus, the maximum RED value is observed in those cases which produce an output equal to 64. As for the subtractor, due to bit inversion, the MSBs are affected especially when dealing with small numbers. A RED of 64 is detected when $a - b = 1$ (exact result), unless a is a power of 2. As for the minimum RED, the unsigned adder does not accept negative numbers.
In Table 5.8 lists all the versions of the ETAII-based arithmetic circuits found in the system. The "exact" and "appr." keywords in the first column indicate whether the carry generators are all connected together or not. Some of the approximate circuits might also require joining some blocks together, but only in part. As explained earlier, only the smallest units need to be exact. Normally, the same number of carry generators and sum generators is instantiated. In case the carry-out is not needed, the leftmost carry generator is omitted.

Type	Input bits	Output bits	Data type	Carry blocks	Sum blocks	Bits per block	Used in
Exact adder	8	9	Unsigned int	2	3	3	Smoothing
Appr. Adder	15	16	Unsigned FP	3	4	4	Smoothing
Appr. Sub.	16	16	Unsigned FP	4	4	4	Smoothing
Exact Sub.	8	9	Signed int	2	3	3	Differentiation
Appr. Adder	15	15	Signed FP	4	5	3	Differentiation
Exact Sub.	8	9	Signed int	2	3	3	2nd order derivative
Exact Adder	11	11	Signed int	2	3	4	2nd order derivative
Appr. Sub.	16	17	Signed FP	4	4	4	Diff. after smooth
Appr. Adder	15	15	Signed FP	4	5	3	Diff. after smooth

Table 5.8: List of ETAII units instantiated in the system

5.4.3 RoBA multiplier

The RoBA multiplier turns hardware multiplication into a series of shift operations as well as additions and subtractions. Given the extent to which the exact multipliers are simplified, this circuit, too, needs extensive optimizations. The key point to keep in mind is that multipliers in a filter are characterized by a constant input, i.e. the coefficient. Following this consideration, the circuit was simplified. Therefore, only the samples are processed, while the coefficient is hard-wired inside the multiplier at the cost of creating a unique component for

each coefficient. The full circuit presented in Figure 4.4 is reduced to the one in Figure 5.26. In this new scheme, a signed sample is first inverted by means of a series of XOR gates performing a bit-wise operation. One of the two inputs is connected to the sign bit of the operand, which causes each gate to invert its second input signal in the case of a negative number. Thus, all the sample's bits but the most significant one are complemented. If the operand is positive, it is not affected by the bit inversion. As internally the circuit is unsigned, the sign bit is not included in the computation considering it would always be equal to zero. Similarly to the subtractors previously discussed, the one's complement is preferred over the accurate conversion. That is why no extra adder compensates for the '1' after the XOR operation. While this generates a further layer of approximation, the error does not impact the result significantly. This is shown in Table 5.9. Due to the excellent performance of the exact circuits, this multiplier struggles to reach figures that are as good. For this reason, additional approximations, such as the one just introduced, are beneficial to the downsizing of the multiplier. After the possible inversion, the operand goes through the rounding block, which implements Equation 4.8 to produce a number in the form of the closest power of two. To decrease the error, this component's output has one extra bit relative to the input. This extension is essential to the subset of numbers close to the upper limit of the total range that must be rounded up to obtain the approximated value. For instance, an 8-bit number such as 250 is much closer to 256 rather than 128. By allocating the extra bit, the maximum error is greatly reduced. The rounding result is then sent to an encoder that produces an integer number from '0' to n, where n is the sample's word-length. This output drives a barrel shifter that displaces the bits to the left. The number of shift positions supported by this unit depends on the word-length. In the case of fixed-point numbers, the fractional part has a negative weight. In other words, a right shift might is necessary. However, to keep the design simple, any number below '1' is considered '0' and no right shift is performed. The coefficient does not undergo any of these operations, as its rounded version is known ahead of time. As a consequence, the number of shifting positions is fixed. The optimizations applied to the adder and subtractor produce two types of approximate circuits. A RoBA multiplier implements Equation 4.6. However, the scheme in Figure 5.26 does not perform any subtraction. This is a direct consequence of the fixed coefficients: the subtractor can be completely left out, while still obtaining the correct result. Using this strategy, instead of summing A_rB and AB_r first, $A_rB - A_rB_r$ is carried out. A_rB_r is a number whose bits are all '0' except for one. The position of the '1' depends on the value of the operands. Before subtraction, when performing the two's complement for such a number, all the zeroes after the single '1' are inverted. Depending on the value of the coefficient, two rounding schemes exist: rounding up or rounding down. These cases give rise to two slightly different implementations.

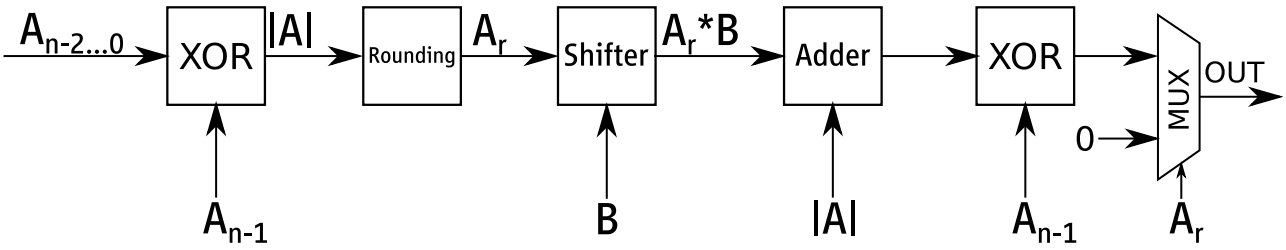


Figure 5.26: Simplified version of the RoBA multiplier

As explained before, in a multiplier, coefficients are aligned with the integer samples to increase precision. Despite being fractional, once they are shifted, they can be treated as integers. In the implemented RoBA multipliers, operand A represents the sample, while operand B is the constant coefficient. With the rounding up strategy, a coefficient whose binary representation is equivalent to 122, will be approximated to 128, i.e. it is rounded up. In binary, this becomes:

$$B = 001111010, \text{ and} \tag{5.13}$$

$$B_r = 010000000. \tag{5.14}$$

By knowing B and B_r, A_rB_r is obtained by shifting A_r by B_r rather than the other way around. In this way, the barrel shifter is not needed. Even though operand A is not known, the products A_rB and A_rB_r will have a similar magnitude. In other words, their most significant ones can either be in the same position or exhibit, at most, a 1-bit difference in weight. In the case of up-rounding, A_rB_r is always greater than A_rB. In this

example:

$$A_rB = 00001xxxx,$$
$$A_rB_r = 000100000 \text{ (original), and}$$
$$A_rB_r = 111100000 \text{ (two's complement).}$$

Two characteristics can be noticed. First: the result of the subtraction is always negative and second: the lower part of the result is equal to the lower part of A_rB, due to the fact that A_rB_r solely contains zeroes. Moreover, there is no need to compute A_rB. This number is generated when the coefficient B is shifted by an arbitrary amount dictated by A. However, this operation just introduces trailing zeroes, which can be safely left out. The same applies to A_rB_r. The subtraction result is simply obtained by taking B and padding it with ones. At this point, the barrel shifter moves the number in the left direction by a quantity that depends on A. Therefore, the amount of padding ones necessary and, in turn, the bit-width of this number are defined by the maximum shift allowed and are thus constant. The number of trailing zeroes that results from this procedure is not known beforehand and are therefore kept. This is not the case for AB_r. As the number of shifts is fixed, the trailing bits are simply dropped. The operand coming from the barrel shifter is split into two parts. Its lower part would face the hypothetical trailing zeroes of AB_r and is thus transferred directly to the output. Its upper part is sent to the adder. In this way, the addition can be performed on a much smaller circuit whose bit-width is determined by A, after the XOR inversion. The subtraction result, properly shifted, and AB_r, which is simply A, are summed together. For this operation, a BKA is used. This unit could not be approximated as it operates on the MSBs of the result. A visual representation of the steps followed by the optimized RoBA multiplier is depicted in Figure 5.27. In the example, $A = 14$ and $B = 122$

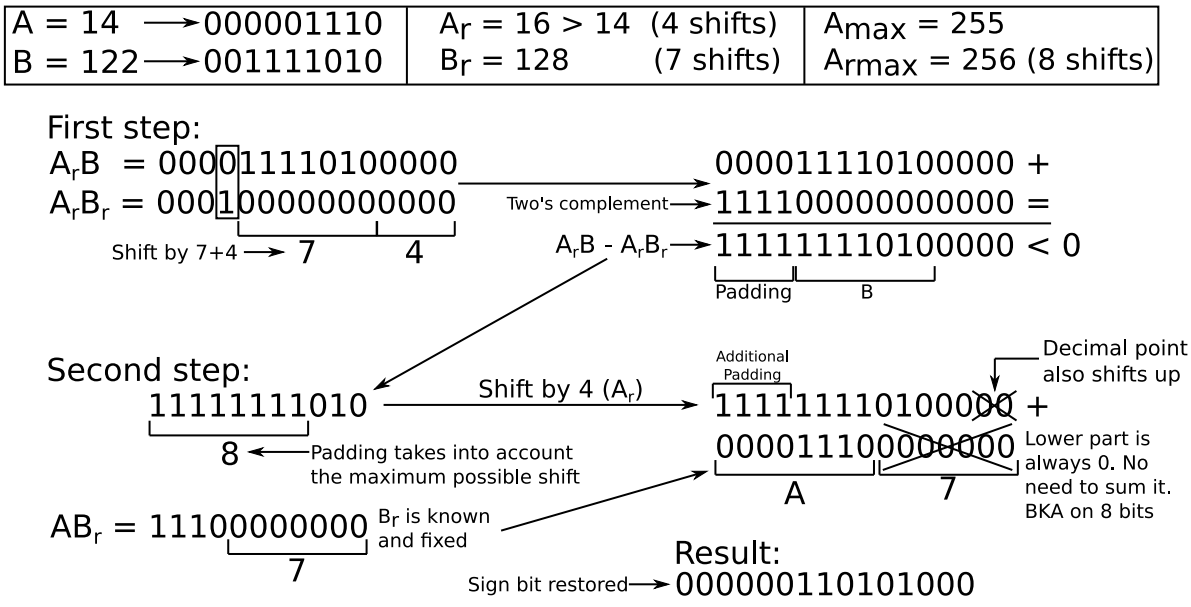


Figure 5.27: RoBA optimization concept

After the addition, the second XOR block converts the number in the same way and on the same conditions as before. As coefficients are always positive, the sign of the result is the same as A. Also in this case, one's complement is preferred. However, since the output number is extended to include a fractional part, the weight of the missing '1' is much smaller than before. A multiplexer at the end chooses between the result arriving from the chain and '0'. Given that the multiplication has turned into an addition, the multiplexer nullifies the output in the case of a sample equal to zero. A different strategy would be to design a barrel shifter able to completely shift the operand to the left, leaving only zeroes. However, given the complexity of a barrel shifter, a multiplexer might produce a faster design.

In the event the coefficient is rounded downwards, A_rB_r and A_rB have the same size. Thus, to obtain the result it is sufficient to take B and replace its most significant one with a zero. This operation does not require any resources. The number is again padded, this time with zeroes, depending on the maximum shift. It is interesting to note that due to the left shifts, the vast majority of the binary outputs have at least two trailing zeroes. This means that almost all the results from the RoBA are even and divisible by four. This is true when the two's complement is carried out correctly or, alternatively, when unsigned numbers are processed.

As explained before, there are four types of multipliers. The ones in the single differentiation filter work on 9-bit integer numbers coming from a subtractor. These operands can be negative and, thus, sign inversion is applied when needed. After this step, the samples are reduced to eight bits and processed. The multipliers in the smoothing filter work on 9-bit unsigned numbers. This means that the circuit is even simpler as there is no sign inversion and no errors arise from the simplified two's complement. Finally, the differentiation filter attached to the smoothing filter receives fractional numbers that are 17 bits long. Internally, the parallelism is 16 bits. As stated previously, instead of allowing both left and right shifts, fractional numbers are set to zero. The differentiation filters produce a result on 15 bits, while the smoothing filters generate 16 bits. The second order differentiation filter features a distinctive architecture that makes use of only one multiplier. The input operands are on 13 bits, while the output is resized to 16 bits. The sizes of the rounding block, the encoder, the barrel shifter, and the BKA are directly proportional to the input bit-width. From the description presented so far, it is clear that the RoBA multiplier is generally much more complex than an exact implementation made solely of a series of FAs. Despite the optimizations, it is expected that this circuit will perform more poorly than the accurate ones. As some exact multipliers are as simple as an adder, a simple replacement with an approximate adder might be the best solution.
Unlike the approximate adders, simulation for this unit was performed on the hardware implementation. This is due to the way software models are devised. As the adders approximations have an effect on the single bits, their software versions are implemented using bitwise operators provided by MATLAB as a way to simulate logic gates. Thus, these models behave exactly like the real circuits. The RoBA instead, modifies the whole operand and thus, a software multiplication can be carried out in decimal format. The software variant would return better results due to the higher precision. The 9-bit versions are tested and the usual metrics are collected. In the following tables the results for each multiplier in both differentiation filter and smoothing filter are listed. The former has four coefficients, whose integer representation is reported in the top row of Table 5.9. To compute the fixed-point result, it is sufficient to multiply the samples by these integer numbers and then again by 2^{-10} to shift right. In the case of the smoothing filter 2^{-9} is the constant, while for the second order filter it is 2^{-15}.
In a differentiation filter, the multipliers define only one input which ranges from -255 to 255. Thus, the number of total combinations to test reduces to just 2^9. The results refer to signed multipliers that perform a two's complement in both conversion. Thus, it is expected that the final version of the multiplier will produce a bigger error. Each multiplier is identified by a number, which is the result of a direct conversion of the fixed-point coefficient from binary to integer. The difference between a two's complement version and a less accurate one is not significant in terms of metrics. Yet, Table 5.9 also includes column *RoBA 166 approx.* presenting data from one of the one's complement versions next to its counterpart, *RoBA 166*. To better highlight the differences, the coefficient with the worst error performance was chosen. For the sake of comparison, a generic RoBA multiplier was tested along side them. This multiplier takes two signed input numbers and tests all 2^{18} combinations. Its operands are treated as integers.

Metric	RoBA 74	RoBA 122	RoBA 109	RoBA 166	RoBA 166 approx.	Integer
ER	0.9706	0.9706	0.9746	0.9706	0.9843	0.9309
MED	0.2087	0.1252	0.3966	0.7932	0.8001	455.06
Max ED	0.6250	0.3750	1.1875	2.375	2.4961	4096
Max NED	0.0339	0.0123	0.0437	0.0575	0.0604	0.0625
MSE	0.0746	0.0268	0.2692	1.0767	1.0864	608955
RMSE	0.2731	0.1638	0.5188	1.0377	1.0423	780.36
NMED	0.0113	0.0041	0.0146	0.0192	0.0194	0.0069
Mean error	0	0	0	0	0.079	0
Max RED	0.045	0.0164	0.0581	0.0736	0.0763	0.1111
Min RED	-0.045	-0.0164	-0.0581	-0.0736	-1	-0.1111
MRED	0	0	0	0	-0.0064	0

Table 5.9: Error distance probabilities of RoBA multipliers for a differentiation filter

The results show that the signed RoBA multiplier has a high ER and a reasonable MED. The maximum ED is observed for operands located exactly in the middle of two consecutive powers of two. The distance between the number in the middle and its closest approximation, i.e. the maximum ED, increases proportionally with

the word-length. This happens because the range from a bit to the next grows larger as it moves toward bigger weights. In this case, the highest number for an operand is 255. The distance between 128 and 256 is clearly wider than the distance between 64 and 128. Thus the maximum ED happens for A equal to either -192 or 192, while in the integer multiplier, -192 and 192 can be rearranged to generate four different combinations. The maximum ED also depends on the coefficient, which is fixed. With equal range, the multipliers whose coefficients are closer to their respective power of two will produce a lower error than the ones further away. In light of these considerations, the worst case for the standard differentiation filter is observed in the multiplier designed for the coefficient 166. The metrics obtained from the integer version are several orders of magnitude above the rest, as the result is not scaled by 2^{-10}. The mean error of 0 is due to the fact that the range is symmetrical and the negative numbers produce the same results as their positive counterpart, but with opposite sign. The same can be said for the MRED. The one's complement version instead does not follow this trend due to a discrepancy between the negative results and the positive ones, which do not suffer from this error.
Table 5.10 shows data related to the smoothing filter, which has five coefficients. While the numbers are again on 9-bits, the range now goes from 0 to 510. Despite the absence of the two's complement, the maximum ED is sometimes higher than before because the distance between 512 and 256 is larger than the distance between 256 and 128. The number in the middle, in this case, is 384 and it generates the biggest error. As for the coefficients, the farthest from its approximation is 214. An integer version is generated as well. The number of testing combinations is the same as before.

Metric	RoBA 18	RoBA 36	RoBA 66	RoBA 161	RoBA 214	Integer
ER	0.9804	0.9804	0.9804	0.9824	0.9648	0.9613
MED	0.167	0.334	0.167	2.7552	1.75	1820.4
Max ED	0.5	1	0.5	8.25	5.25	16384
Max NED	0.0279	0.0279	0.0076	0.0514	0.0493	0.0627
MSE	0.0477	0.1909	0.0477	12.9903	5.2511	9740190
RMSE	0.2184	0.4369	0.2184	3.6042	2.2915	3120.9
NMED	0.0093	0.0093	0.0025	0.0172	0.0164	0.0070
Mean error	0.0019	0.0039	0.0019	0.0318	-0.0401	-0.2442
Max RED	0.037	0.037	0.0101	0.0683	0.0654	0.1111
MRED	0.0189	0.0189	0.0051	0.0348	0.0332	0.0287

Table 5.10: Error distance probabilities for a smoothing filter

It can be seen that the maximum ED for the coefficient 161 is much higher than the one for coefficient 166 in the previous filter. Although the coefficients are similar, the input range is doubled. Moreover, the scaling factor is halved (2^{-9}). Although 214 is farther from 256 than 161 is from 128, this last multiplier receives its operand directly from the input of the filter rather than from the subtractor, as its coefficient is the middle one. The error is thus lower because of the reduced dynamics. The RoBA 161 also features the highest MED. As negative numbers are no longer present the mean error and the MRED are not 0 anymore. The minimum RED is fixed to 0 for all designs.
Table 5.11 summarizes the main features of the RoBA multiplier by filter.

Filter	Number of multipliers	Data type	Input bits	BKA output bits	Rounding bits	Encoder bits
Differentiation	4	Signed	8	9	9	9
Smoothing	5	Unsigned	9	10	10	10
Smoothing (unpaired)	1	Unsigned	8	9	9	9
Differentiation (2nd ord.)	1	Signed	12	12	12	12
Differentiation (after smooth)	4	Signed	16	13/14	9	9

Table 5.11: List of RoBA units instantiated in the system

5.4.4 Approximate multiplier with partial error recovery

The AM-ER is the second inexact multiplier implemented. It approximates the partial product computation by simplifying its adders [71]. The carry propagation is removed and instead, an error signal is generated to keep track of the lost information that would be provided by the carry. While theoretically the sum and error signals can be added together to obtain the correct result, only some of the most significant error bits are actually

used resulting in an approximate circuit. The final sum and part of the error signal are added together by a BKA. Its size depends on the preferred bit-width of the error signal. The wider it is, the more accurate is the result. As no carry propagation occurs, each pair of bits is independent from the neighbouring ones. Thus, in theory, the algorithm can be applied starting from any direction. In the implementation, two operands are processed at a time, and for each pair of equal-weight bits, a sum bit and an error bit are generated. After the first iteration, $n/2$ words made of single sum bits, where n is the operands' parallelism, are obtained. For the subsequent levels of the partial product tree, each couple of sum words undergoes the same procedure to create a new sum word on a lower branch. The same applies to the error signals, which are required at a later step. This process goes on until one sum word remains. On the contrary, none of the error arrays are processed until the very end, where they are added to the final approximate sum. In the error array, the i-th bit turns to '1' only in one specific case, i.e. when $A_{i-1} = B_{i-1} = 1$ and $A_i \neq B_i$. Therefore, it is expected that few '1's are present while the rest is mostly '0's. Because of this, in the paper [71], a further approximation is presented: instead of summing the error bits, a simple OR operation is performed.
Also in this thesis, a different circuit for each multiplier is created in order to optimize the design to the fullest. Unlike the RoBA multiplier, this solution is expected to produce better results in terms of delay than the exact version. This is because it simplifies the existing adders instead of adding more logic. Although a final BKA is present, its width is much smaller than the original size. The number and types of multiplier created is the same as in the RoBA. As before, due to the fact that one input operand is fixed, the multipliers have one input port only. Moreover, to further optimize them, the partial products equal to zero are not considered. Finally, the error array is not computed for all pairs, but only for the required MSBs. Figure 5.28 shows a spreadsheet in which the multiplication algorithm is applied to a 9-bit operand (495), assuming that the second one is fixed and equal to 161.

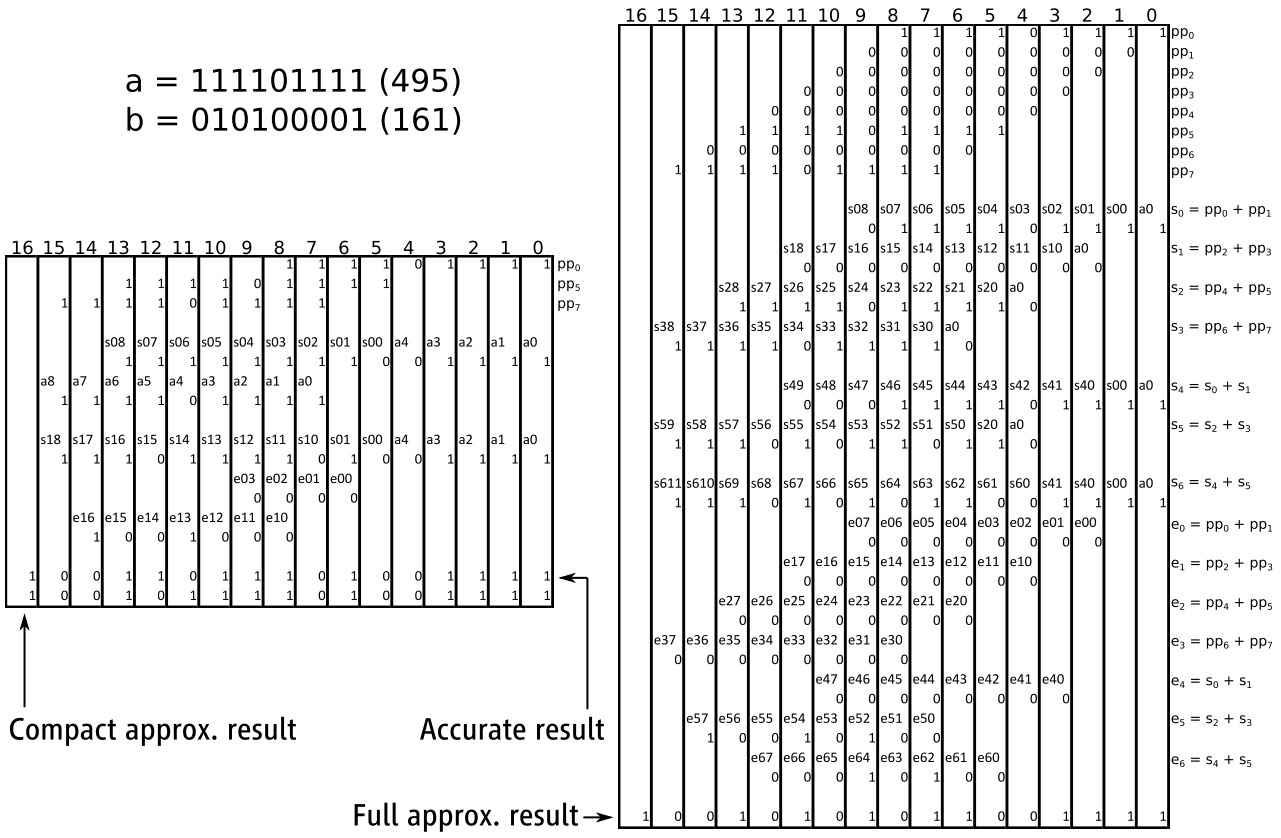


Figure 5.28: An example of the AM-ER algorithm

The multiplication is unsigned. On the right side, the full multiplication is performed as if both operands were unknown. Only eight partial products are present as the last one is zero. This is done to have an even

number of partial products. The final result is obtained by summing the s_6 along with all the error signals below. Exact addition is carried out only on the four MSBs whereas the remaining bits employs OR operators. The same is done for the computation on the left, which represents the simplified version. Given that the binary representation of "161" contains three '1's, only three properly shifted partial products are generated. From the figure it is possible to see that the full algorithm proceeds like a standard multiplication except for the fact that sum bits are not always correct. Due to the operands alignment, the sum words begin when the first pair of bits from the right is encountered. The error arrays are also arranged by weight and start at the second pair of bits from the right. This is possible because, due to the error bits' logic function, the first pair would always yield '0'. As each coefficient is unique, the length and weight of these arrays may vary depending on its binary pattern. The optimization on the left provides two advantages. The first and most obvious is that the final circuit will be much simpler as it requires fewer logic. The second one is that, in general, the accuracy increases. In the figure, the seven error strings on the right reduce to just two after the optimization. As a consequence, the chances that two or more '1's are in the same position reduce. This makes the OR operation less error-prone. In the standard version, columns 8 and 9 might contain multiple '1's in the worst case. If all these bits are simply ORed and the resulting carry is ignored, a much bigger error is generated. Moreover, it is also evident that the total number of '1's in the error array is much lower in the optimized version. As each '1' represents an error, the less of them, the better. The information that was previously stored in the error arrays is now stored in the sum. This means that even by completely neglecting error compensation, the optimized design would generate a significantly more accurate result. The numbers are compared in Figure 5.29.

Exact			1	0	0	1	1	0	1	1	1	0	1	0	0	1	1	1	1	79695
Compact			1	0	0	1	0	1	1	1	1	0	1	0	0	1	1	1	1	77647
Full			1	0	0	1	0	1	0	1	0	1	1	0	0	1	1	1	1	76495

Figure 5.29: AM-ER multiplication results for different strategies

It can be seen that the full version has a bigger ED. As explained earlier, the two main sources of errors are: the size of the final adder that compensates for the accumulated inaccuracies and the method chosen to sum the error arrays together (usually OR gates). However, as each optimized multiplier configuration is unique, the results can vary significantly among different coefficients. For this reason, it is difficult to estimate the metrics for each circuit analytically. For instance, even if the BKA width was fixed for all circuits, the number of error arrays, whose weight is high enough to reach the MSBs, is not always the same. In some cases, only one of them manages to pull through, while the other arrays are not even computed. In other cases, it is more than one. In the first situation, no OR operation needs to be carried out leading to a more accurate design. The more error arrays overlap, the higher the chances are to observe multiple '1's in the same position. There are also cases in which employing an actual adder to sum the error arrays, rather than the OR operation, results in a loss in accuracy. This happens when only part of the error is compensated: the adder introduces carry bits from a certain position and ignores all the previous bits. The carry might propagate but it modifies only the MSBs. This can cause a greater ED than it would with the OR solution. Finally, the fixed-point number at the result is not rounded, the LSBs are simply truncated. This further approximation affects also the RoBA multiplier.
The AM-ER implements both signed and unsigned multiplication. As expected, a signed operation requires sign extension to work. Unsigned multiplication is simpler as the single sum and error arrays are usually shorter. Similarly to the previous designs, the multipliers found in the differentiation filter take 9-bit input samples and produce a 15-bit result, while in the smoothing filter the output is on 16 bits wide. If the differentiation happens after the smoothing, the inputs are represented on 17 bits. The second order differentiation filter instead uses a multiplier that accepts 13-bit inputs and generates a 16-bit output. In all cases, a 5-bit BKA is selected for the final sum. The number of error bits included in this addition depends on the coefficient. When multiple arrays are present, an OR operation is first applied to get a single array. The part of the result that is not compensated for is obtained directly from the final sum word, which might contain errors. The sum and error bits are computed by means of simple logic functions highlighted in Equations 4.11.

Metric	AM-ER 74	AM-ER 122	AM-ER 109	AM-ER 166	AM-ER 546
ER	0.8102	0.9393	0.9569	0.9022	0.9863
MED	0.2613	0.9638	0.9242	0.7162	1.0003
Max ED	1.2051	5.7520	6.4395	2.6895	4.6567
Max NED	0.0654	0.1893	0.2372	0.0651	0.1096
MSE	0.1384	2.0720	2.5233	0.8723	2.1331
RMSE	0.3720	1.4395	1.5885	0.9340	1.4605
NMED	0.0142	0.0317	0.0340	0.0173	0.0235
Mean error	-0.2613	-0.9638	-0.9242	-0.7162	-1.0003
Max RED	0.1741	0.5344	0.5145	0.2929	0.1171
Min RED	-1.0841	-2.7158	-1.1692	-2.1711	-1.1090
MRED	-0.0166	-0.0464	-0.0488	-0.0281	-0.0472

Table 5.12: Error distance probabilities for a multiplier used in a differentiation filter

Table 5.12 shows a wide metrics variability among different designs for the differentiation filter. One important thing to notice is that despite the relatively high maximum ED, the MED is rather low. This means that very few configurations are able to generate a big EDs. The metrics obtained from the smoothing filter are listed in Table 5.13

Metric	AM-ER 18	AM-ER 36	AM-ER 66	AM-ER 161	AM-ER 214
ER	0.4384	0.4384	0.4521	0.7847	0.9336
MED	0.1174	0.2348	0.4658	1.2593	2.3633
Max ED	0.6250	1.25	2.5	5.627	10.8281
Max NED	0.0349	0.0349	0.0380	0.0351	0.1016
MSE	0.0460	0.1840	0.7123	3.8955	9.4574
RMSE	0.2144	0.4289	0.8440	1.9737	3.0753
NMED	0.0065	0.0065	0.0071	0.0079	0.0222
Mean error	-0.1174	-0.2348	-0.4658	-1.2593	-2.3633
Max RED	0.1891	0.1891	0.0554	0.2763	0.4716
MRED	0.0191	0.0191	0.0127	0.0239	0.0609

Table 5.13: Error distance probabilities for a multiplier used in a smoothing filter

AM-ER 18, 36 and 66 are the simplest AM-ER circuits. This is a consequence of the small number of '1's in them, i.e. 2, which generates only two partial products. Thus the error occurrences are much rarer giving rise to a very low ER. AM-ERs 18 and 36 share many of the results because the two coefficients are similar: 36 is obtained by shifting 18 left. AM-ER 66 is the only multiplier that uses no error correction as there is no error bits in the most significant positions. On the other hand, AM-ER 214 with its five '1's performs the worst in terms of all the metrics. As explained in [71], the result is always equal to or smaller than the exact one. This is the reason why the mean error is negative and its absolute value is equal to the MED.
The numbers that cause the maximum ED vary depending on the implementation set by the coefficient. A large error is expected when a lot of error bits are '1', or when they have the same weight. Multiple '1's create an error proportional to their weight. Simulation is carried out in the same way as it was for the RoBA. As expected, the worst-case ED is observed for those number with a lot of '1's and a few '0's in between. These operands, when partial products are computed, produce the condition that generates an error multiple times as they are shifted. The relative position of the pairs of numbers due to shifting, depend on the coefficient binary pattern. Table 5.14 shows some examples of number generating maximum ED. The "# (number) of error arrays" column refers to the arrays that are actually computed. The "Coefficient" columns highlights the number of '1's present in each coefficient. In case multiple numbers generate the same maximum ED, only one of them is reported.

Multiplier	Type	Coefficient	# of partial products	# of error strings	Worst case operand	Used in
AM-ER 74	Signed	001001010	3	2	011110111	Diff. filt.
AM-ER 122	Signed	001111010	5	4	101110111	Diff. filt.
AM-ER 109	Signed	001101101	5	3	101111110	Diff. filt.
AM-ER 166	Signed	010100110	4	2	011011111	Diff. filt.
AM-ER 18	Unsigned	000010010	2	1	101111011	Smooth. filt.
AM-ER 36	Unsigned	000100100	2	1	101111011	Smooth. filt.
AM-ER 66	Unsigned	001000010	2	0	111011010	Smooth. filt.
AM-ER 161	Unsigned	010100001	3	1	110111011	Smooth. filt.
AM-ER 214	Unsigned	011010110	5	2	010111011	Smooth. filt.
AM-ER 546	Signed	0001000100010	3	1	1011001110111	Diff. filt. 2nd

Table 5.14: List of AM-ER units instantiated in the system

5.4.5 Approximate divider and approximate divider cells

In an array divider, approximation is achieved by working on the single cells. As explained earlier, four different cells introduced in [46, 47] are tested. The main approximation strategy implemented is the triangle replacement (TR). However, both horizontal replacement (HR) and vertical replacement (VR) are found. The divider deals with two 15-bit signals. They originate from the differentiation filter but by the time they reach the divider they turn into unsigned numbers and the sign bit is no longer needed. This happens because the divider works on the segment between two samples, which is always positive. The dividend z is always less than or equal to the divisor d. With this in mind, some simplifications are possible. As z is supposed to be on 30 bits, its lower half is set to '0' to avoid having a quotient q that is null for all inputs. For this reason, the rightmost column contains cells where only one input out of three, i.e. d, varies. As shown in Figure 5.20, this column is reduced to an AND gate in the exact divider. Given its simplicity, it was replaced by approximate cells by performing a VR. Depending on the logic function implemented by the inexact cells tested, the AND gate is either simplified or left untouched.
A second optimization regards the last row responsible for both the final remainder s and q_0. As s is not needed, this row was simplified in the exact divider to an extent that allows for a correct computation of q_0 only. To reduce the critical path, the last row is completely removed in the approximate divider and d_0 is directly assigned to q_0. Thus, the total number of rows reduces to fourteen. The fourteenth row now becomes the last and is simplified since s_{13} is no longer needed, whereas q_1 is still computed. Similarly to the rightmost column, the last row is also approximated. The logic functions related to the inexact cells are reduced so that only b_{out} (needed for q) is computed, while s is left out. Thus, instead of a pure TR, also VR and HR are partially implemented. The results of this approximation are shown in Figure 5.30. The exact cells are black-filled, whereas the approximate cells are red. Shades of black and red represent simplified versions of a cell. The blue/light blue squares symbolize those cells that handle solely b_{out}. The picture highlights the replacement strategies, hence, it shows no interconnections. A smaller divider is shown in Figure 4.6. Both exact and approximate circuits possess the same structure, only the cells change.
The red square can represent one of the four approximation schemes. The first inexact cell, AXS1, is obtained from Equation 4.17. When z and b_{in} are '0', both outputs, i.e. s and b_{out}, are fixed at '1'. This happens at the least significant column (column 0). Compared to the exact design, the AND gate disappears, which means that also the adjacent cells (gray and pink) that receive the '1's as input are also simplified. This is done by replacing the corresponding input signal in the logic functions with a constant '1'. The remaining cells are exact. As for the last row, only b_{out} is computed. The blue cells are exact, whereas the light blue ones are simplified. In the case of AXS1, the input is simply inverted.
The second cell, whose function was introduced in Equation 4.18, is called AXS2. This cell only approximates the partial remainder s while keeping b_{out} accurate. For this reason, the rightmost column's cell is approximated to an AND gate as was the case in the exact divider. Unlike the AXS, the second column cannot be simplified since d_0 is found at the input, as opposed to a fixed value. As the last row only computes b_{out}, its implementation is equivalent to that of the exact divider. The same thing is true for AXS3, which is described by Equation 4.19.

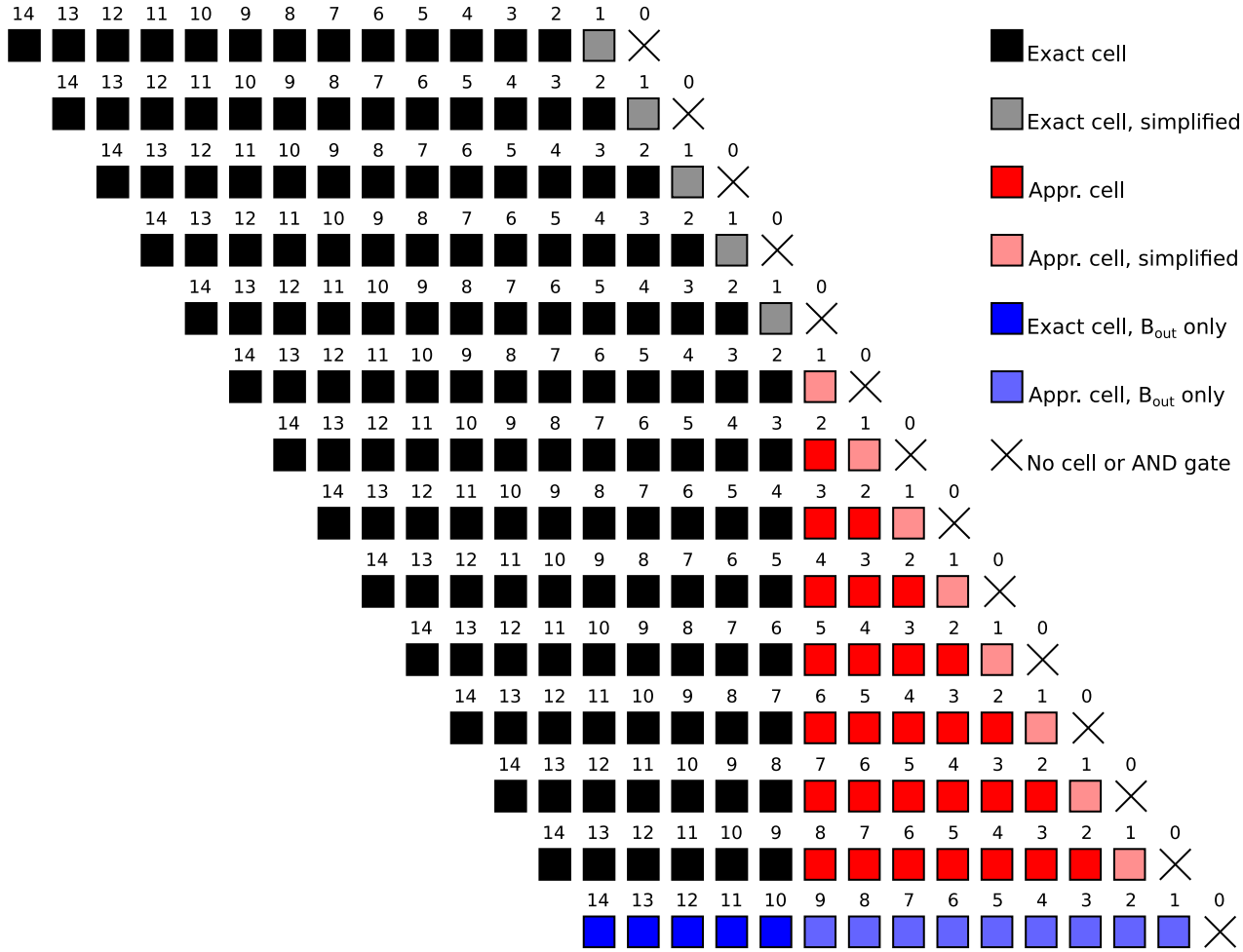


Figure 5.30: Divider's hybrid replacement strategy

The last cell, AXS4, is presented in Equation 4.20. In this case, $b_{out} = b_{in}$. This means that for a single row, the first b_{in} is transmitted all the way to the output without any change. Figure 4.6 shows that the initial b_{in} is always '0'. This produces a q that is constantly equal to '1', after negation. Moreover, the cells belonging to the first column receive two inputs at '0'. Due to the logic function of this divider cell, when two inputs are '0', so are the outputs. This causes a chain reaction where the '0's are used by the nearby cells and propagate. As a consequence, without a correction, the approximated cells would be equivalent to a wire and disappear. While this is a possibility, the resulting error is not admissible. To work around this problem, the rightmost cells estimate b_{out} as $not(z_i)$ AND d_0, which is suggested in [46]. In this way, a cell with two inputs at '0' produces no partial remainder, while b_{out} is equal to d_0. The second column from the left receives a constant zero and thus its cells can be slightly simplified. As for the last row, approximate cells are not instantiated as d_0, i.e. the first b_{in}, simply propagates until the first exact cell. Each architecture is tested using 10^7 random pairs of input numbers, for which $d > z$ always applies. It is important to stress that some simplifications work only if this condition is true. Metrics are collected and the results are shown in Table 5.15. Starting from the bottom-right block, the depth of approximation was increased in a triangular pattern until the ED reached figures deemed unacceptable ($> 10^{-3}$). The limit is reached by the structure in Figure 5.30, which is the one chosen. The numbers in the table refer to this circuit.
As the result is always a number between 0 and 1, the absolute values of some metrics are considerably smaller when compared to the arithmetic circuits already seen. Given that the ED magnitude is comparable to the limited precision provided by 15 fractional bits, any ED below 2^{-15} is set to '0' and the final result is considered exact. This also affects the overall ER and the MED. Although the maximum ED is a relatively big number, the MED is not. In the case of the divider an ED less than 10^{-3} is considered reasonable. The input combinations that generate a greater error distance are observed to occur approximately 0.01% of the time. This generally happens when both operands are the same, case for which an exact result equal to '1' would be produced.

Metric	AXS1	AXS2	AXS3	AXS4
ER	0.4542	0.2551	0.2563	0.4471
MED	3.0007e-05	1.1489e-05	1.1609e-05	2.3321e-05
Max ED	0.0527	0.0078	0.0019	0.0395
MSE	1.2014e-08	01.4986e-09	5.5015e-10	7.5108e-09
RMSE	1.0961e-04	3.8711e-05	7.2082e-05	8.6665e-05
Mean error	-2.9199e-05	-1.0054e-05	-1.1187e-05	-2.2929e-05
MRED	1.0192e-04	7.0460e-05	7.2082e-05	8.2695e-05

Table 5.15: Error distance probabilities for the various approximate divider architectures

As the maximum output value for this kind of divider is '1', the NED and NMED are not included in the metrics estimation, since they are always equal to the ED and the MED respectively. As the critical path propagates from top to bottom, a TR does not improve speed as much as a HR. To get better results without degrading accuracy too much, it is possible to keep applying HR to the lower rows, while preserving TR. This, combined with pipelining can reduce the critical path considerably. A TR reduces the number of resources but does not do much for delay. To notice some improvements at least half of the cells should be replaced by the inexact ones. It is also important to note that this type of approximation works best at transistor level. While, the approximate cells are clearly simpler, the benefits they provide at a logic gate level are not always so noticeable.
Table 5.15 shows that the ERs are relatively low. As expected, the most accurate designs are the ones that only approximate s, while leaving the function for b_{out} unchanged. These are the AXS2 and the AXS3. Moreover, the functions for s in both cells preserve the AND gate at column 0 instead of removing it, further reducing the error. In terms of accuracy, the AXS1 and the AXS4 perform the worst. Although the approximation is more aggressive for the AXS4, its results are better than the AXS1. As explained in Chapter 6, the improvements seen by the divider in terms of delay and power are inversely proportional to their accuracy. It will be shown in Chapter 6 that both the AXS2 and AXS3 provide a performance which is similar to the exact implementation. Improvements after synthesis can be noticed when employing the AXS1 and AXS4. However, the reduction in delay is not significant. The AXS4 was chosen as the final divider on account of the fact that it represents a good compromise between accuracy and performance.

CHAPTER 6

Results and Evaluation

This chapter is centred on the simulation of the entire pipeline. The results are presented and discussed. After the simulation, the single arithmetic circuits are synthesized and the most important parameters such as area, power and delay are collected. In the meantime, both the exact and approximate units are put together to build the main blocks of the pipeline. These circuits are also simulated and synthesized. The analysis of the filters and the zero-crossing detector is carried out in terms of error metrics and post-synthesis parameters. A comparison between accurate and approximate circuits and, when possible, with their software versions is performed.
As explained in the previous chapter, the whole system is synthesized on a Cyclone V 5CSEMA5F31C6 FPGA. The Quartus software provides several tools to aid the design process. The first step is called "analysis and synthesis". For this part, the VHDL code is compiled and syntax errors or discrepancies among the signals are detected. The design is also mapped on the specific technology and state machines are modelled. The compiler employs logic minimization techniques and removes redundant logic. For this project, the optimization strategy is balanced. This means that power, delay and area are given the same importance. At this point, the physical placement has not been performed, nor have the interconnections been laid down yet. After synthesis, the fitter executes the physical design by assigning the logic functions to the appropriate lookup tables and connects them. It checks for timing violations and iterates the placing and routing procedure until the constraints are met or until Quartus detects that the constraints cannot be satisfied.
Circuits in the project are divided into several categories depending on the function, e.g. multiplication, addition, filters, etc. Power analysis for each group is performed using the same simulation time, input vectors and frequency, which is dictated by the slowest circuit. This number was obtained during a previous synthesis aimed at finding the maximum frequency. In this case the period selected is 1 ns, which is impossible to reach. The accuracy of this analysis depends on the information at hand. The most rigorous way requires the creation of a value change dump (VCD) file for the software. This file contains the signal activities of each node in the circuit. The signal activities are registered by means of a gate level simulation. Despite being much faster than the previous approach, post-mapping simulation does not take into account the delays caused by wires and is thus less accurate than gate level simulation, which is available only after fitting. It is not performed on the VHDL code but rather on a VHO file holding the netlist generated post-fitting. Given the size of this project, simulation time for a gate level approach ranged from a couple of minutes to an hour depending on the size of the design. Thus, all the units' switching activities are collected at gate level. A longer simulation time generates a more accurate and bigger VCD file. For complex circuits such as filters, a simulation time of 2 ms was enough to get a high accuracy estimation. However, for smaller arithmetic circuits, the accuracy was medium, as stated in the power report. The figures shown in the coming tables are computed as a worst-case scenario rather than a typical situation. Toggling rate for unspecified internal signals is obtained through vectorless estimation, which statistically estimates the signal activity of a node by looking at connected nodes. For unspecified I/O signals, a default toggle rate of 12.5% is set.
As said earlier, the analysis estimates the maximum power based on the following operating conditions: voltage: 1.1 V, junction temperature range: 0-85 °C. This parameters are also used during timing analysis to ensure that the circuit works at the maximum frequency computed under every condition. Two types of worst cases exist in the timing analyzer: slow 85 °C and slow 0 °C. Both have the slowest silicon, due to process variations, and the lowest supply voltage. The only difference is the operating temperature. While one would expect the colder

circuit to be faster, this is not always the case. Normally, performance worsens when temperature increases due to a reduced mobility, which has a direct effect on the saturation current of a MOSFET. The threshold voltage also controls this parameter in the opposite direction, i.e. the higher the temperature, the lower the threshold voltage, the higher the saturation current. The delay as a function of temperature depends on the prevailing effect of the two. If the effect of the threshold voltage is greater than the one due to mobility, the speed decreases with temperature.

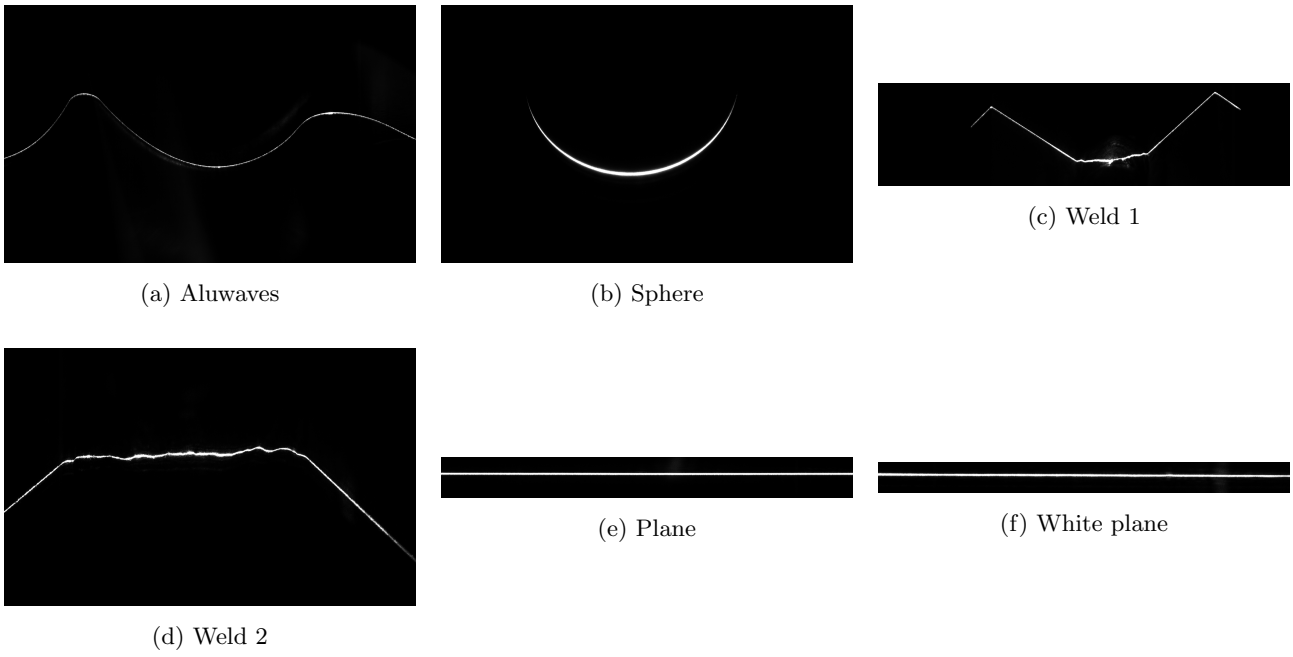
6.1 Pre-synthesis simulation

Each unit and its components were tested by means of a functional simulation aimed at verifying the equivalence between the VHDL model and the behaviour observed in the results. Whenever possible, all the available input combinations were used. This is the case with small arithmetic circuits, whose error metrics are discussed in Chapter 5. In Section 6.2, delay, power, and area are reported for all units, while error metrics estimations and comparisons of any kind are reserved to the complex blocks, e.g. filters. The verification process makes use of a simple VHDL testbench that reads input vectors from a text file, applies them to the device under test (DUT) and writes the results on an output file. The input samples are randomly generated for the arithmetic circuits. For the filters instead the real grayscale images are converted into binary pixel by pixel. The zero-crossing circuit receives the data from the filters. Depending on the DUT, a clock signal might be present or not. During functional simulation the actual clock frequency is not important. The appropriate period is necessary only during power analysis. The output files contain results in binary form. They are converted into decimal by MATLAB scripts tailored for each design. The code performs the same operation done by the DUT separately and then compares the two values to ensure they are identical. Error metrics and plots are also performed by MATLAB.
While a software implementation is more accurate than the same exact circuit, the error measured is simply the difference in precision between the two. The simulations are aimed at proving the correctness of the results. For the sake of comparison, the accurate circuits are synthesized as well, and parameters such as FPGA resource consumption (area), delay, and power are estimated. It is expected that the approximate design show lower area, delay and power consumption compared to the exact designs and that these figures decrease proportionally to the error that affects the circuits.

6.1.1 Filters

In this section, the results of the four designed filters are compared. These are: the smoothing filter, the differentiation filter with no smoothing before (DF-NS), the second order differentiation filter (DF-2ND), and the differentiation filter after smoothing (DF-S). For each of them seven versions exist that arise from some of the possible combinations of approximate arithmetic units. Thus, next to the exact design, filters containing only one class of approximate circuits are tested, i.e. adders or multipliers. Based on the simulation outcome, two additional filters are created for which all arithmetic components are inexact. As two approximate adders (LOA and ETAII) and two approximate multipliers (RoBA and AM-ER) have been studied, two final filters are formed by combining them. The chosen couples are: LOA-RoBA and ETAII-AM-ER. By switching the circuits, two additional filters, i.e. LOA-AM-ER and ETAII-RoBA, can be crafted. However, due to time restrictions, they are not tested here. The exact filters make use of BKAs, BWMs, and WTMs.
For the simulation, MATLAB generates the input files by reading grayscale images of the laser line as integers and then converting them into binary. The images with higher resolutions contain $> 2 * 10^6$ rows and a size of 20 MB. The output files are roughly double the size because of the increased bit-width. The total occupation of all the output files generated by the simulation is around 10 GB. The simulation was automatized by means of a TCL script. The script contains a loop that modifies the specific lines of the testbench to test the 32 images available. The MATLAB code also reads each file in the folder and converts the results to decimal in order to perform the comparison with the exact value. The error metrics are saved in a separate file.
Approximate filters are created by replacing the adders and multipliers inside with their inexact counterparts. Although the final circuits replace both, simulation was first performed by replacing either the adders or the multipliers while keeping the other exact.
Five types of laser line images from five different artifacts were used for testing, that is: a sphere's profile, aluwaves, a weld and two types of flat planes. Aluwaves refer to a milled aluminum part that has a curvature. As for the weld object, it was created by fusing two L profiles together in the middle. Some examples are shown

in Figure 6.1. The two L profiles are visible in Figure 6.1c. Figure 6.1d represents the same object as Figure 6.1c but flipped and observed at a closer distance. For every object, images with different exposure times were taken. The total number of tested images is 32, and all of them are listed in Appendix A.



(a) Aluwaves

(b) Sphere

(c) Weld 1

(d) Weld 2

(e) Plane

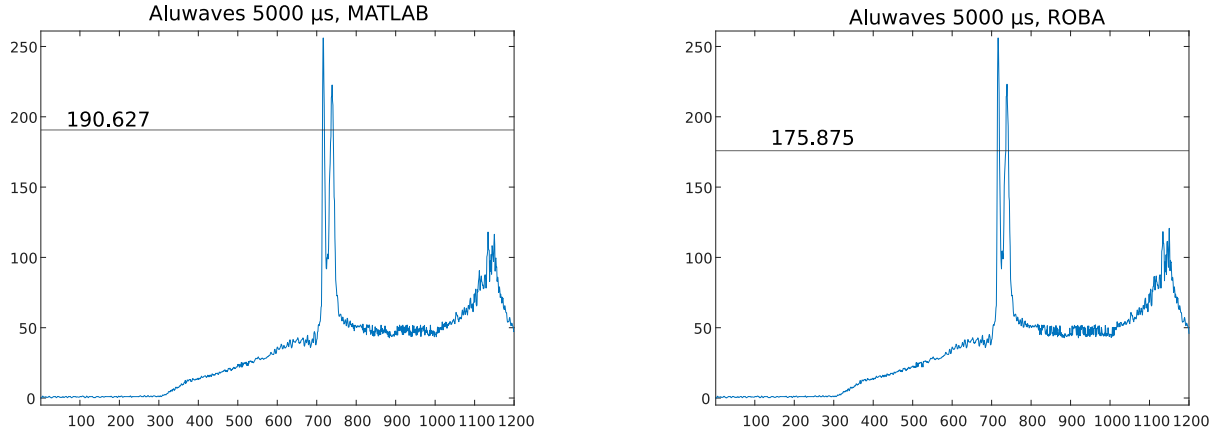
(f) White plane

Figure 6.1: Some of the object captured by the camera and analyzed

In the following tables, the names of the scanned objects are complemented with the exposure time, which appears in ascending order. For filters, four metrics are estimated. They are: error distance (ED), error rate (ER), mean error distance (MED), and mean error. The total number of simulations performed to explore the design space is 24. To avoid showing 24 tables, each made of 32 rows, only the error metrics of some objects are reported at the beginning. At the end of the tables, the maximum and average metrics for all the 32 pictures are shown. The information supplied by these two sets of numbers summarizes the error performance of the algorithm and provides enough information to assess the accuracy of an approximate design. The approximate results are compared with a software version of the filters implemented using the MATLAB *filter* function. A comparison with the exact hardware version has also been performed. However, the differences with the software implementation are only visible after several decimal places, while the metrics reported here are rounded at five significant digits. An exception is seen for the ER. Due to the significant contrasts, the ERs for both versions appear in the tables. The ER is greater for the software version due to the fact that the approximate circuit is compared to a number with a much higher precision. On the contrary, the exact filters implemented in hardware have equal bit-widths and thus provide a more realistic ER.
All the graphs and the tables shown in the following Subsections (6.1.1.1 to 6.1.1.6) have been generated by a MATLAB script that analyzes all the 32 images and detects, for each of them, the column with the largest ED. This column is plotted for both exact and approximate versions. The two points affected by the largest error are highlighted. This is based on the assumption that the columns with the biggest ED will show noticeable differences when the approximate design is placed side by side with the exact one. While this might not always be the worst case visually, it is assumed that it is very close to it. For mildly approximated algorithms, even the graph produced in worst case conditions are hard to tell apart. With a higher exposure time, the laser line is thicker but the noise also increases. A wide laser line can create multiple zeroes, however, most of the noise can be filtered by the zero-crossing detection by setting a high enough PT. The extra noise caused by an approximate circuit is not a problem as long as the region around the zero is not distorted. It is thus important that the position of the zero does not change.
While the software version of the differentiation filters was created by means of a built-in function, the smoothing filter was simulated by a custom code that limited the output between 0 and 255. This function implements a direct form FIR filter but it saturates the output samples in both directions at each cycle. The results for the DF-S are obtained by applying the output samples coming from a smoothing filter of the same type. For example, a LOA DF-S reads data from a LOA smoothing filter.

6.1.1.1 RoBA filter

Based on the data presented in the following sections, the RoBA multiplier is the most accurate of all the circuits discussed here.
The first analyzed circuit is the smoothing filter. While the maximum ED for the smoothing filter is high compared to other filters, the other metrics show remarkable results. Figure 6.2 shows an example on a particularly noisy image. The large ED between the approximate design on the right and the correct one on the left is due to the relative position of the points involved. Despite being far from the correct number, the approximate point still falls in the range admitted by the filter. Thus, the final results are not altered and the smoothed image is not affected. In fact, the differences between the two graphs are imperceptible. This demonstrates that the maximum ED is the least reliable of the metrics due to its erratic behaviour also when exposure time increases. While the other figures are directly proportional to this time, the maximum ED displays a weaker correlation. This is because it is a local parameter, which is not able to take into account the whole system. The metrics estimation is shown in Table 6.1. It can be seen that the error increases when exposure time increases. This is not surprising considering that the longer the exposure time, the higher the noise levels. This effect is visible in Figure 6.2. In the image, the noise level raises above zero from a certain point on. Due to the small oscillations, the noise is responsible for large EDs compared to a signal whose samples are constant and very close to zero. The data derived from the images is summarized in the last two rows of the following tables by computing the average and the maximum for the four metrics. As said before, the relatively high ED has no impact on the shape of the smoothed signals.



(a) MATLAB implementation

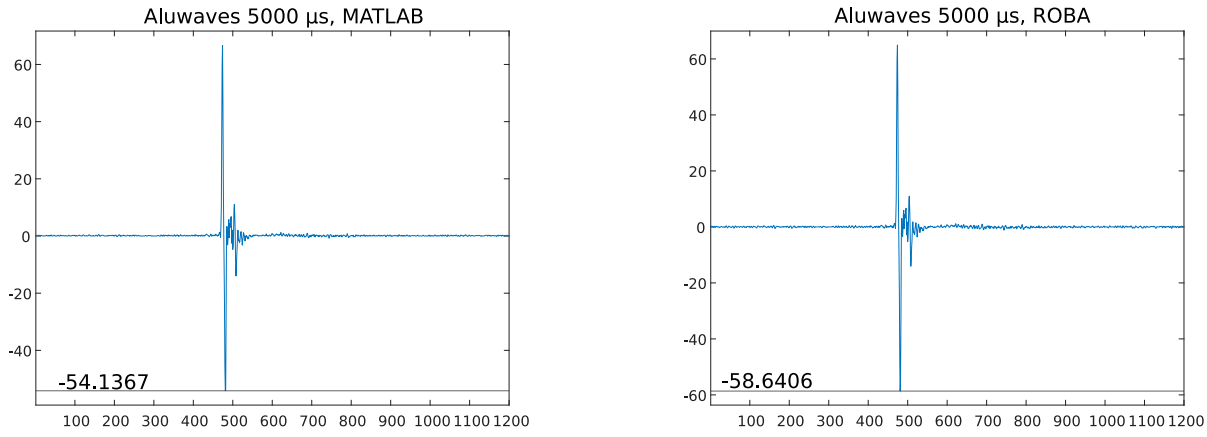
(b) Approximate circuit

Figure 6.2: Results of one column provided by the exact smoothing filter from MATLAB vs. the RoBA approximate circuit implemented using VHDL

The RoBA differentiation filter comes in three variants. The first one examined is the DF-NS. Similarly to the previous circuit, this filter is incredibly accurate. Moreover, the maximum ED measured here is much more reasonable than before. Two example images obtained from the simulation are plotted in Figure 6.3. The difference between the software and hardware versions is unnoticeable. Compared to the RoBA smoothing filter, the average metrics are slightly worse. Moreover, in the approximate implementation, the minimum occurs at a lower position than what is observed in the exact circuit. This phenomenon is witnessed during the analysis of other approximate differentiation filters. The problem is not visible in the smoothing filters due to saturation. Table 6.2 contains some of the results with increasing exposure time. The average and maximum metrics of all the images simulated can be found in Table 6.3.

Image	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	12.099	0.1899	0.0507	0.0068	-0.0014
aluwaves 500 μs	13.543	0.3924	0.2092	0.0330	-0.0032
aluwaves 1000 μs	13.842	0.6179	0.3430	0.0607	-0.0071
aluwaves 2000 μs	14.078	0.7656	0.5027	0.1065	-0.0120
aluwaves 5000 μs	14.752	0.9918	0.8216	0.2297	-0.0281
plane 10 μs	4.7830	0.1020	0.0332	0.0668	-0.0010
plane 20 μs	12.178	0.1241	0.0779	0.0755	-0.0008
plane 40 μs	13.791	0.1749	0.0980	0.1307	-0.0429
plane 80 μs	14.122	0.2774	0.1236	0.1406	-0.0182
plane 160 μs	13.795	0.4838	0.1765	0.1368	-0.0174
plane 320 μs	13.771	0.8481	0.4306	0.1719	-0.0502
plane 640 μs	13.481	0.9906	0.8609	0.2597	-0.0423
sphere 100 μs	13.344	0.1369	0.0157	0.0081	-0.0006
sphere 200 μs	13.777	0.1683	0.0217	0.0151	-0.0011
sphere 500 μs	14.214	0.2371	0.0347	0.0179	-0.0000
sphere 1000 μs	14.132	0.2860	0.0544	0.0206	0.0001
sphere 2000 μs	14.190	0.3989	0.0950	0.0266	0.0011
sphere 5000 μs	14.868	0.6141	0.2193	0.0425	-0.0008
weld 10 μs	13.406	0.1230	0.0298	0.0122	-0.0020
weld 100 μs	13.501	0.2207	0.0565	0.0240	-0.0038
weld 500 μs	13.932	0.4943	0.2679	0.0605	-0.0104
weld 1000 μs 1	14.355	0.6180	0.4210	0.0982	-0.0005
weld 1000 μs 2	14.554	0.5420	0.1568	0.0423	-0.0061
weld 2000 μs	14.583	0.6805	0.5212	0.1443	-0.0091
weld 5000 μs	14.693	0.7394	0.6343	0.2620	-0.0299
white plane 20 μs	2.3400	0.2392	0.0972	0.0181	-0.0014
white plane 50 μs	10.835	0.3542	0.1467	0.0806	-0.0072
white plane 100 μs	13.737	0.4905	0.2082	0.1868	-0.0338
white plane 200 μs	13.971	0.6754	0.3019	0.2387	-0.0335
white plane 500 μs	13.776	0.9527	0.6307	0.2954	-0.0468
white plane 1000 μs	13.842	0.9912	0.8854	0.3777	-0.0336
white plane 2000 μs	14.102	0.9926	0.9511	0.5178	-0.0516
Average	13.137	0.4973	0.2972	0.1211	-0.0155
Max	14.868	0.9926	0.9511	0.5178	-0.0516

Table 6.1: Error evaluation of the RoBA smoothing filter



(a) MATLAB implementation

(b) Approximate circuit

Figure 6.3: Results of one column provided by the exact differentiation (no smoothing) filter from MATLAB vs. the RoBA approximate circuit implemented using VHDL

Image	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	3.3547	0.2136	0.1308	0.0221	0.0205
aluwaves 500 μs	3.9159	0.4207	0.2782	0.0504	0.0453
aluwaves 1000 μs	4.2640	0.6557	0.4410	0.0793	0.0726
aluwaves 2000 μs	4.4494	0.8000	0.5656	0.1041	0.0968
aluwaves 5000 μs	4.5039	0.9938	0.7475	0.1428	0.1317
plane 10 μs	1.8579	0.1274	0.1080	0.0306	0.0255
plane 20 μs	3.7995	0.1524	0.1293	0.0534	0.0265
plane 40 μs	3.9538	0.2074	0.1678	0.0699	0.0266
plane 80 μs	3.8536	0.3215	0.2430	0.0888	0.0519
plane 160 μs	3.7042	0.5468	0.3946	0.1158	0.0739
plane 320 μs	3.7797	0.8919	0.6652	0.1706	0.1130
plane 640 μs	3.9365	0.9967	0.8196	0.2154	0.1459
sphere 100 μs	3.2101	0.1698	0.0929	0.0146	0.0130
sphere 200 μs	4.0123	0.2067	0.1152	0.0201	0.0163
sphere 500 μs	4.0032	0.2878	0.1664	0.0299	0.0243
sphere 1000 μs	4.1611	0.3390	0.2038	0.0363	0.0306
sphere 2000 μs	4.0405	0.4609	0.2894	0.0496	0.0449
sphere 5000 μs	4.0487	0.6658	0.4363	0.0749	0.0705
weld 10 μs	3.6682	0.1468	0.0933	0.0184	0.0154
weld 100 μs	4.0466	0.2544	0.1641	0.0331	0.0270
weld 500 μs	4.3660	0.5199	0.3487	0.0707	0.0592
weld 1000 μs 1	4.3336	0.6401	0.4541	0.0924	0.0792
weld 1000 μs 2	4.4616	0.6078	0.3997	0.0744	0.0647
weld 2000 μs	4.4660	0.6913	0.5224	0.1085	0.0933
weld 5000 μs	4.2486	0.7571	0.6143	0.1287	0.1111
white plane 20 μs	0.9422	0.2946	0.2085	0.0380	0.0361
white plane 50 μs	3.1034	0.4152	0.3000	0.0705	0.0532
white plane 100 μs	4.4694	0.5637	0.4156	0.1260	0.0750
white plane 200 μs	3.9877	0.7525	0.5816	0.1853	0.1089
white plane 500 μs	4.1188	0.9831	0.8257	0.2447	0.1527
white plane 1000 μs	4.1876	0.9991	0.9014	0.2658	0.1725
white plane 2000 μs	4.7374	0.9998	0.9269	0.2765	0.1834

Table 6.2: Error evaluation of the RoBA differentiation filter (no smoothing)

	ED	ER (SW)	ER (HW)	MED	Mean err
Average	3.8746	0.5338	0.3985	0.0969	0.0706
Max	4.7374	0.9998	0.9269	0.2765	0.1834

Table 6.3: Error evaluation of the RoBA differentiation filter (no smoothing)

The RoBA DF-2ND is the most accurate design yet. This is mainly due to the fact that this filter features only one multiplier at the output while the arithmetic circuits before exclusively deal with integer numbers. All DF-2ND filters have a stronger smoothing capability compared to the standard one of order four. This is due to a reduced cut-off frequency. As a result, the attenuation makes its output range narrower and the shape of the output signal more rounded. Table 6.4 shows the error metrics for this filter.

Image	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	0.5512	0.2119	0.1115	0.0021	0.0014
aluwaves 500 μs	0.5512	0.4182	0.2304	0.0048	0.0028
aluwaves 1000 μs	0.5512	0.6514	0.3635	0.0070	0.0042
aluwaves 2000 μs	0.5512	0.7946	0.4630	0.0087	0.0049
aluwaves 5000 μs	0.5512	0.9907	0.6074	0.0111	0.0061
plane 10 μs	0.2803	0.1271	0.1059	0.0042	0.0012
plane 20 μs	0.5512	0.1521	0.1250	0.0103	0.0010
plane 40 μs	0.5512	0.2068	0.1594	0.0147	0.0017
plane 80 μs	0.5512	0.3199	0.2272	0.0189	-0.0000
plane 160 μs	0.5512	0.5428	0.3576	0.0232	0.0004
plane 320 μs	0.5512	0.8871	0.5793	0.0271	0.0039
plane 640 μs	0.5512	0.9900	0.7283	0.0274	0.0074
sphere 100 μs	0.5428	0.1684	0.0869	0.0019	0.0012
sphere 200 μs	0.5512	0.2048	0.1063	0.0028	0.0014
sphere 500 μs	0.5512	0.2845	0.1496	0.0038	0.0019
sphere 1000 μs	0.5512	0.3343	0.1789	0.0044	0.0023
sphere 2000 μs	0.5512	0.4535	0.2458	0.0057	0.0031
sphere 5000 μs	0.5512	0.6567	0.3587	0.0070	0.0043
weld 10 μs	0.5501	0.1457	0.0856	0.0024	0.0010
weld 100 μs	0.5512	0.2521	0.1447	0.0042	0.0017
weld 500 μs	0.5512	0.5154	0.2913	0.0079	0.0031
weld 1000 μs 1	0.5512	0.6357	0.3714	0.0094	0.0036
weld 1000 μs 2	0.5512	0.6021	0.3350	0.0076	0.0041
weld 2000 μs	0.5512	0.6891	0.4259	0.0108	0.0038
weld 5000 μs	0.5512	0.7487	0.5125	0.0122	0.0045
white plane 20 μs	0.1418	0.2931	0.1975	0.0038	0.0021
white plane 50 μs	0.5293	0.4127	0.2816	0.0115	0.0029
white plane 100 μs	0.5512	0.5599	0.3804	0.0234	0.0039
white plane 200 μs	0.5512	0.7483	0.5199	0.0277	0.0042
white plane 500 μs	0.5512	0.9799	0.7241	0.0331	0.0058
white plane 1000 μs	0.5512	0.9874	0.8058	0.0346	0.0057
white plane 2000 μs	0.5512	0.9713	0.8523	0.0401	0.0059
Average	0.5289	0.5293	0.3473	0.0129	0.0032
Max	0.5512	0.9907	0.8523	0.0401	0.0074

Table 6.4: Error evaluation of the RoBA differentiation 2nd order filter (no smoothing)

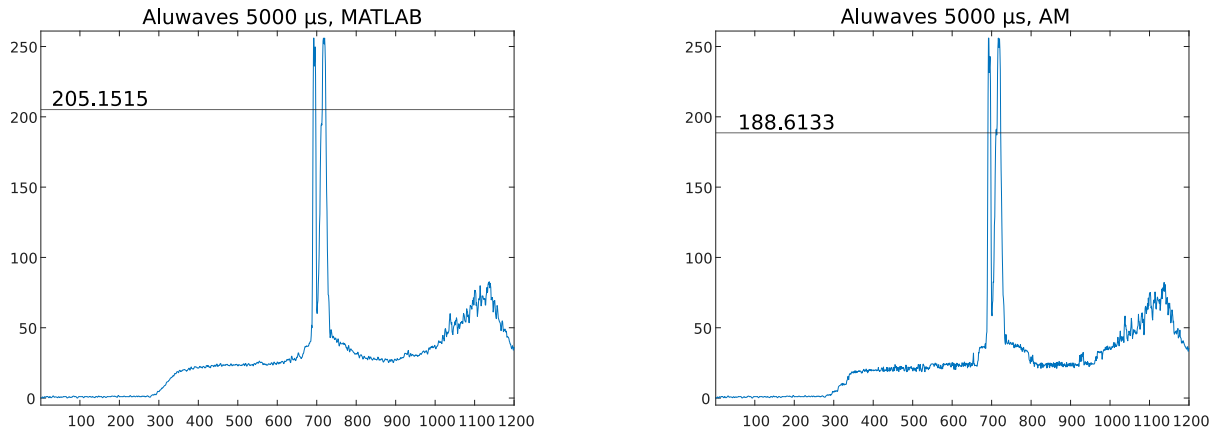
Finally the RoBA differentiation filter placed after the smoothing procedure shows similar results to the standalone version. Although the input data is already approximated by the smoothing stage, this filter still manages to provide very accurate results. Table 6.5 shows the data related to the filter.

Image	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	5.3811	0.2520	0.2388	0.0222	-0.0000
aluwaves 500 μs	6.0186	0.4654	0.4472	0.0500	-0.0002
aluwaves 1000 μs	6.7184	0.7128	0.6870	0.0777	0.0000
aluwaves 2000 μs	6.8080	0.8414	0.8179	0.0991	0.0002
aluwaves 5000 μs	6.3526	0.9938	0.9785	0.1364	-0.0011
plane 10 μs	2.3741	0.1830	0.1727	0.0264	-0.0004
plane 20 μs	4.6017	0.2067	0.1959	0.0544	-0.0032
plane 40 μs	5.9345	0.2685	0.2548	0.0936	0.0044
plane 80 μs	5.4181	0.3924	0.3731	0.1171	0.0137
plane 160 μs	5.6794	0.6394	0.6105	0.1450	0.0038
plane 320 μs	5.3191	0.9223	0.9040	0.1864	-0.0004
plane 640 μs	5.3834	0.9978	0.9864	0.2164	0.0010
sphere 100 μs	4.9032	0.2524	0.2294	0.0168	-0.0001
sphere 200 μs	6.4843	0.2982	0.2722	0.0236	-0.0001
sphere 500 μs	5.6060	0.3864	0.3568	0.0329	0.0000
sphere 1000 μs	5.1590	0.4304	0.4016	0.0387	-0.0005
sphere 2000 μs	5.2445	0.5448	0.5158	0.0524	-0.0005
sphere 5000 μs	5.9875	0.7307	0.7020	0.0781	-0.0005
weld 10 μs	5.6814	0.1967	0.1825	0.0185	0.0003
weld 100 μs	6.2576	0.3052	0.2882	0.0347	0.0006
weld 500 μs	6.7762	0.5537	0.5359	0.0746	0.0008
weld 1000 μs 1	6.0870	0.6622	0.6463	0.0939	0.0004
weld 1000 μs 2	5.9871	0.6921	0.6616	0.0760	-0.0003
weld 2000 μs	5.8171	0.7103	0.6955	0.1072	-0.0009
weld 5000 μs	7.1008	0.7867	0.7683	0.1339	-0.0026
white plane 20 μs	0.9678	0.4109	0.3809	0.0272	-0.0002
white plane 50 μs	4.9540	0.5266	0.4966	0.0747	-0.0010
white plane 100 μs	6.7027	0.6771	0.6453	0.1499	0.0005
white plane 200 μs	5.8154	0.8340	0.8068	0.2057	0.0006
white plane 500 μs	5.4603	0.9928	0.9796	0.2473	-0.0023
white plane 1000 μs	5.2221	0.9999	0.9883	0.2578	-0.0017
white plane 2000 μs	6.0189	1.0000	0.9883	0.2800	-0.0008
Average	5.5694	0.5896	0.5690	0.1015	0.0003
Max	7.1008	1.0000	0.9883	0.2800	0.0137

Table 6.5: Error evaluation of the RoBA differentiation filter (after smoothing)

6.1.1.2 AM-ER filter

The second set of filters is implemented by replacing the accurate multipliers with AM-ERs. The adders are still exact. Starting from the smoothing filter, this multiplier produces very good results. What makes it even better than the RoBA is its simplicity. A large ED is again observed in the smoothing filter. However, this does not constitute a problem as the approximate image maintains its similarities with the exact version. The results for the smoothing filter are shown in Table 6.6. The average and the maximum values are shown instead in Table 6.7. Figure 6.4 displays the output of the approximate smoothing filter next to the exact version. As opposed to the RoBA, some differences between the two images are visible. The effects of the approximations, however, are minimal.



(a) MATLAB implementation

(b) Approximate circuit

Figure 6.4: Results of one column provided by the exact smoothing filter from MATLAB vs. the AM-ER approximate circuit implemented using VHDL

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	12.702	0.1899	0.0116	0.0160	-0.0159
aluwaves 500 μs	16.250	0.3924	0.1105	0.1058	-0.1055
aluwaves 1000 μs	17.169	0.6179	0.2029	0.2467	-0.2461
aluwaves 2000 μs	15.929	0.7656	0.3207	0.4617	-0.4605
aluwaves 5000 μs	16.538	0.9918	0.5743	0.9842	-0.9810
plane 10 μs	10.078	0.1019	0.0494	0.0957	-0.0954
plane 20 μs	16.983	0.1240	0.0590	0.1353	-0.1342
plane 40 μs	16.227	0.1749	0.0673	0.1725	-0.1705
plane 80 μs	16.258	0.2774	0.0802	0.1863	-0.1848
plane 160 μs	16.248	0.4838	0.1053	0.2250	-0.2235
plane 320 μs	16.426	0.8481	0.1948	0.3199	-0.3180
plane 640 μs	16.100	0.9907	0.5147	0.6463	-0.6427
sphere 100 μs	15.774	0.1369	0.0102	0.0202	-0.0201
sphere 200 μs	15.822	0.1683	0.0137	0.0285	-0.0282
sphere 500 μs	15.910	0.2371	0.0216	0.0408	-0.0404
sphere 1000 μs	16.183	0.2860	0.0312	0.0558	-0.0554
sphere 2000 μs	16.857	0.3989	0.0490	0.0829	-0.0824
sphere 5000 μs	16.343	0.6141	0.1008	0.1516	-0.1509
weld 10 μs	15.427	0.1230	0.0161	0.0273	-0.0271
weld 100 μs	15.530	0.2207	0.0288	0.0484	-0.0481
weld 500 μs	16.200	0.4943	0.0964	0.1306	-0.1298
weld 1000 μs 1	16.546	0.6180	0.2483	0.2756	-0.2745
weld 1000 μs 2	15.859	0.5420	0.0603	0.0850	-0.0843
weld 2000 μs	15.814	0.6805	0.3930	0.5379	-0.5365
weld 5000 μs	17.322	0.7395	0.5219	0.9413	-0.9372
white plane 20 μs	9.6655	0.2392	0.0626	0.0796	-0.0795
white plane 50 μs	15.418	0.3542	0.1001	0.2032	-0.2022
white plane 100 μs	16.667	0.4906	0.1342	0.3037	-0.3015
white plane 200 μs	15.984	0.6754	0.1866	0.3800	-0.3769
white plane 500 μs	15.562	0.9526	0.3393	0.5823	-0.5777
white plane 1000 μs	15.796	0.9912	0.5980	0.9261	-0.9199
white plane 2000 μs	19.544	0.9927	0.8537	1.5013	-1.4951

Table 6.6: Error evaluation of the AM-ER smoothing filter

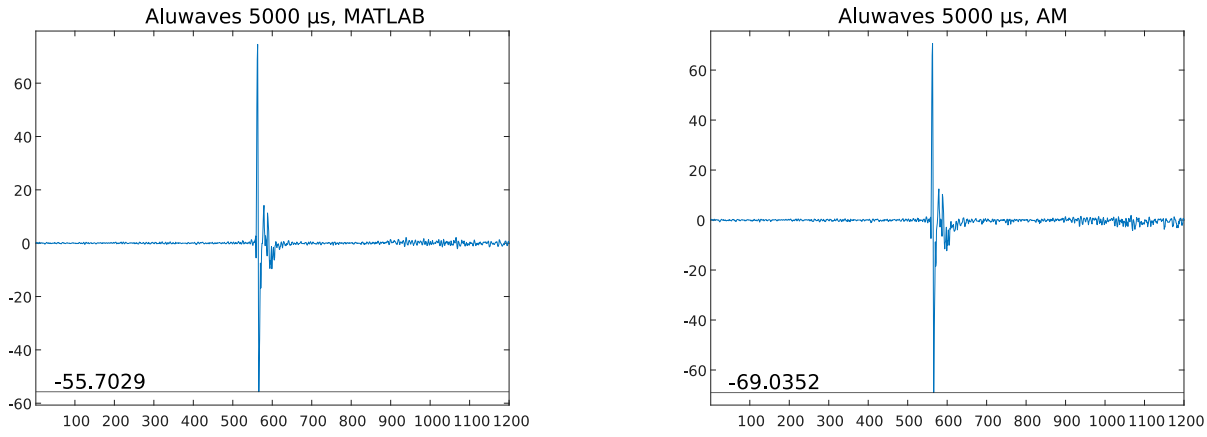
	ED	ER (SW)	ER (HW)	MED	Mean err
Average	15.785	0.4973	0.1924	0.3124	-0.3108
Max	19.544	0.9927	0.8537	1.5013	-1.4951

Table 6.7: Error evaluation of the AM-ER smoothing filter

Despite the large ED, the AM-ER DF-NS is very accurate. As seen from Figure 6.5, the main error lies in the distance between the two minima in the plots. This gap is so large that it constitutes the maximum ED not only for the object in the figure but for every other image examined. This problem is once again not visible in the smoothing filter. Table 6.8 shows the summarized metrics for this filter.

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	10.452	0.2136	0.1067	0.0371	-0.0371
aluwaves 500 μs	13.312	0.4207	0.2273	0.0787	-0.0787
aluwaves 1000 μs	13.410	0.6557	0.3642	0.1247	-0.1247
aluwaves 2000 μs	13.571	0.8000	0.4764	0.1860	-0.1860
aluwaves 5000 μs	13.332	0.9938	0.6437	0.3088	-0.3088
plane 10 μs	8.6000	0.1274	0.0980	0.1452	-0.1452
plane 20 μs	10.910	0.1524	0.1142	0.1940	-0.1940
plane 40 μs	12.688	0.2074	0.1448	0.2379	-0.2379
plane 80 μs	12.694	0.3215	0.2073	0.2790	-0.2790
plane 160 μs	12.343	0.5468	0.3333	0.3264	-0.3264
plane 320 μs	12.107	0.8919	0.5553	0.4023	-0.4023
plane 640 μs	11.285	0.9967	0.6926	0.5131	-0.5131
sphere 100 μs	12.228	0.1698	0.0739	0.0293	-0.0293
sphere 200 μs	12.940	0.2067	0.0918	0.0385	-0.0385
sphere 500 μs	13.466	0.2877	0.1338	0.0518	-0.0518
sphere 1000 μs	11.991	0.3389	0.1651	0.0620	-0.0620
sphere 2000 μs	11.936	0.4608	0.2374	0.0808	-0.0808
sphere 5000 μs	11.658	0.6657	0.3640	0.1174	-0.1172
weld 10 μs	11.400	0.1468	0.0767	0.0492	-0.0492
weld 100 μs	12.632	0.2544	0.1353	0.0752	-0.0751
weld 500 μs	13.591	0.5199	0.2878	0.1395	-0.1395
weld 1000 μs 1	14.150	0.6401	0.3781	0.1852	-0.1852
weld 1000 μs 2	13.933	0.6078	0.3330	0.1227	-0.1227
weld 2000 μs	12.067	0.6913	0.4431	0.2451	-0.2451
weld 5000 μs	13.053	0.7571	0.5379	0.3721	-0.3721
white plane 20 μs	5.0644	0.2946	0.1762	0.1604	-0.1604
white plane 50 μs	9.9360	0.4152	0.2570	0.2677	-0.2677
white plane 100 μs	13.836	0.5637	0.3490	0.3798	-0.3798
white plane 200 μs	13.950	0.7525	0.4855	0.4991	-0.4991
white plane 500 μs	11.729	0.9831	0.7006	0.6597	-0.6597
white plane 1000 μs	11.876	0.9991	0.7887	0.8096	-0.8096
white plane 2000 μs	14.721	0.9998	0.8653	0.9912	-0.9912
Average	12.214	0.5338	0.3389	0.2553	-0.2553
Max	14.721	0.9998	0.8653	0.9912	-0.9912

Table 6.8: Error evaluation of the AM-ER differentiation filter (no smoothing)



(a) MATLAB implementation

(b) Approximate circuit

Figure 6.5: Results of one column provided by the exact differentiation (no smoothing) filter from MATLAB vs. the AM-ER approximate circuit implemented using VHDL

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	4.3082	0.2119	0.0455	0.0094	-0.0094
aluwaves 500 μs	4.6467	0.4182	0.0966	0.0188	-0.0188
aluwaves 1000 μs	4.5301	0.6515	0.1513	0.0275	-0.0275
aluwaves 2000 μs	4.6163	0.7947	0.2000	0.0388	-0.0388
aluwaves 5000 μs	4.3454	0.9910	0.2783	0.0633	-0.0633
plane 10 μs	2.7801	0.1271	0.0751	0.0473	-0.0473
plane 20 μs	3.9801	0.1521	0.0892	0.0689	-0.0689
plane 40 μs	4.3454	0.2068	0.1077	0.0896	-0.0896
plane 80 μs	4.4915	0.3199	0.1383	0.1110	-0.1110
plane 160 μs	4.6173	0.5428	0.1981	0.1242	-0.1242
plane 320 μs	4.4996	0.8871	0.3017	0.1444	-0.1444
plane 640 μs	4.3454	0.9900	0.3939	0.1707	-0.1707
sphere 100 μs	3.9507	0.1684	0.0320	0.0074	-0.0074
sphere 200 μs	4.3091	0.2048	0.0397	0.0106	-0.0106
sphere 500 μs	4.3444	0.2845	0.0573	0.0144	-0.0144
sphere 1000 μs	4.3091	0.3343	0.0703	0.0171	-0.0171
sphere 2000 μs	4.3082	0.4535	0.0981	0.0209	-0.0209
sphere 5000 μs	4.2737	0.6567	0.1467	0.0272	-0.0272
weld 10 μs	4.6164	0.1457	0.0396	0.0152	-0.0152
weld 100 μs	4.5301	0.2521	0.0655	0.0221	-0.0221
weld 500 μs	4.3454	0.5154	0.1303	0.0379	-0.0379
weld 1000 μs 1	4.3454	0.6357	0.1694	0.0480	-0.0480
weld 1000 μs 2	4.6173	0.6021	0.1398	0.0306	-0.0306
weld 2000 μs	4.3454	0.6892	0.2007	0.0604	-0.0604
weld 5000 μs	4.3454	0.7490	0.2507	0.0848	-0.0847
white plane 20 μs	2.7497	0.2931	0.1055	0.0566	-0.0566
white plane 50 μs	3.9507	0.4127	0.1617	0.0813	-0.0813
white plane 100 μs	4.3454	0.5599	0.2172	0.1265	-0.1265
white plane 200 μs	4.3454	0.7483	0.2915	0.1726	-0.1726
white plane 500 μs	4.3454	0.9799	0.4157	0.2240	-0.2240
white plane 1000 μs	4.3454	0.9874	0.4710	0.2548	-0.2547
white plane 2000 μs	4.0320	0.9715	0.5306	0.2976	-0.2976

Table 6.9: Error evaluation of the AM-ER 2nd order differentiation filter (no smoothing)

	ED	ER (SW)	ER (HW)	MED	Mean err
Average	4.2582	0.5293	0.1784	0.0789	-0.0789
Max	4.6467	0.9910	0.5306	0.2976	-0.2976

Table 6.10: Error evaluation of the AM-ER 2nd order differentiation filter (no smoothing)

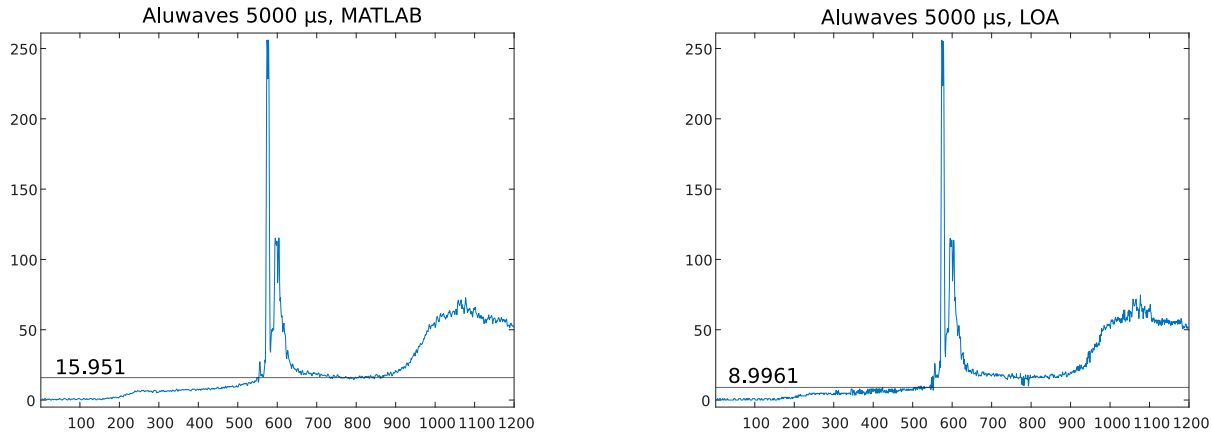
As expected, the DF-2ND is so accurate that the maximum ED is also due to the discrepancy between the minima. Table 6.9 highlights the metrics for this filter. The average MED in Table 6.10 is very low. The same general considerations discussed before for the other AM-ER filters also apply to the DF-S. Table 6.11 shows similar results to Table 6.8 but with a higher MED.

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	11.452	0.2520	0.2397	0.0538	-0.0536
aluwaves 500 μs	13.727	0.4654	0.4488	0.1194	-0.1162
aluwaves 1000 μs	13.871	0.7128	0.6899	0.1934	-0.1842
aluwaves 2000 μs	14.281	0.8414	0.8214	0.2818	-0.2633
aluwaves 5000 μs	12.241	0.9938	0.9824	0.4608	-0.4197
plane 10 μs	7.0154	0.1830	0.1730	0.1647	-0.1637
plane 20 μs	15.051	0.2067	0.1962	0.2219	-0.2207
plane 40 μs	13.417	0.2685	0.2544	0.2816	-0.2805
plane 80 μs	12.924	0.3924	0.3734	0.3155	-0.3144
plane 160 μs	11.959	0.6394	0.6129	0.3935	-0.3918
plane 320 μs	11.315	0.9223	0.9081	0.5224	-0.5190
plane 640 μs	11.439	0.9978	0.9882	0.6788	-0.6658
sphere 100 μs	13.492	0.2524	0.2300	0.0423	-0.0423
sphere 200 μs	11.868	0.2982	0.2731	0.0541	-0.0540
sphere 500 μs	12.121	0.3864	0.3583	0.0738	-0.0734
sphere 1000 μs	12.226	0.4304	0.4035	0.0898	-0.0890
sphere 2000 μs	13.095	0.5448	0.5185	0.1191	-0.1173
sphere 5000 μs	11.819	0.7307	0.7055	0.1753	-0.1710
weld 10 μs	11.853	0.1967	0.1832	0.0637	-0.0635
weld 100 μs	13.407	0.3052	0.2895	0.0981	-0.0978
weld 500 μs	14.512	0.5537	0.5382	0.1866	-0.1844
weld 1000 μs 1	13.848	0.6622	0.6483	0.2527	-0.2453
weld 1000 μs 2	12.725	0.6921	0.6653	0.1725	-0.1715
weld 2000 μs	12.036	0.7103	0.6983	0.3390	-0.3202
weld 5000 μs	12.270	0.7867	0.7717	0.4957	-0.4615
white plane 20 μs	5.3469	0.4109	0.3806	0.1924	-0.1920
white plane 50 μs	10.991	0.5266	0.4973	0.3070	-0.3059
white plane 100 μs	14.565	0.6771	0.6467	0.4390	-0.4372
white plane 200 μs	12.449	0.8340	0.8106	0.5792	-0.5767
white plane 500 μs	11.846	0.9928	0.9858	0.7911	-0.7853
white plane 1000 μs	12.551	0.9999	0.9948	0.9637	-0.9505
white plane 2000 μs	12.428	1.0000	0.9955	1.1760	-1.1442
Average	12.317	0.5896	0.5714	0.3218	-0.3149
Max	15.051	1.0000	0.9955	1.1760	-1.1442

Table 6.11: Error evaluation of the AM-ER differentiation filter (after smoothing)

6.1.1.3 LOA filter

The LOA is the first approximate adder tested. The total number of adders and subtractors in a filter surpasses by far that of the multipliers. For this reason, regardless of the approximation strategy, filters containing approximate adders tend to be less accurate than those based on inexact multipliers. Due to the one's complement LOA present in the differentiation filters, the average ER is 1 (see Table 5.3).



(a) MATLAB implementation

(b) Approximate circuit

Figure 6.6: Results of one column provided by the exact smoothing filter from MATLAB vs. the LOA approximate circuit implemented using VHDL

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	6.0633	0.1900	0.1898	0.0622	-0.0257
aluwaves 500 μs	6.5024	0.3925	0.3920	0.2464	-0.1751
aluwaves 1000 μs	6.7813	0.6180	0.6170	0.4800	-0.3681
aluwaves 2000 μs	6.7731	0.7658	0.7639	0.6786	-0.5320
aluwaves 5000 μs	6.9550	0.9921	0.9889	1.1281	-0.9136
plane 10 μs	5.8519	0.1021	0.1020	0.0599	-0.0453
plane 20 μs	5.7685	0.1243	0.1240	0.0779	-0.0572
plane 40 μs	6.2347	0.1750	0.1727	0.1175	-0.0895
plane 80 μs	6.2370	0.2775	0.2683	0.1585	-0.1217
plane 160 μs	6.8229	0.4841	0.4721	0.2597	-0.1906
plane 320 μs	7.5669	0.8484	0.8307	0.4980	-0.2952
plane 640 μs	7.7517	0.9910	0.9718	0.9411	-0.7134
sphere 100 μs	5.9470	0.1370	0.1369	0.0405	-0.0330
sphere 200 μs	6.3291	0.1683	0.1681	0.0538	-0.0427
sphere 500 μs	6.7402	0.2371	0.2366	0.0858	-0.0672
sphere 1000 μs	6.7685	0.2861	0.2854	0.1180	-0.0874
sphere 2000 μs	6.6621	0.3989	0.3983	0.1806	-0.1252
sphere 5000 μs	6.6865	0.6143	0.6136	0.3196	-0.1943
weld 10 μs	6.7810	0.1230	0.1228	0.0437	-0.0302
weld 100 μs	6.4290	0.2208	0.2201	0.0863	-0.0560
weld 500 μs	6.9115	0.4944	0.4924	0.2635	-0.1303
weld 1000 μs 1	6.7110	0.6182	0.6146	0.4474	-0.3219
weld 1000 μs 2	6.8791	0.5422	0.5403	0.2185	-0.1171
weld 2000 μs	6.7895	0.6807	0.6769	0.7436	-0.6283
weld 5000 μs	7.8011	0.7396	0.7352	0.9644	-0.8309
white plane 20 μs	6.1181	0.2395	0.2393	0.1028	-0.0810
white plane 50 μs	6.0125	0.3544	0.3540	0.1725	-0.1275
white plane 100 μs	6.2938	0.4908	0.4896	0.2550	-0.1758
white plane 200 μs	6.8050	0.6758	0.6643	0.3934	-0.2716
white plane 500 μs	6.9542	0.9534	0.9336	0.7162	-0.4660
white plane 1000 μs	10.3065	0.9916	0.9728	1.0611	-0.8285
white plane 2000 μs	14.266	0.9929	0.9740	1.5804	-1.4335

Table 6.12: Error evaluation of the LOA smoothing filter

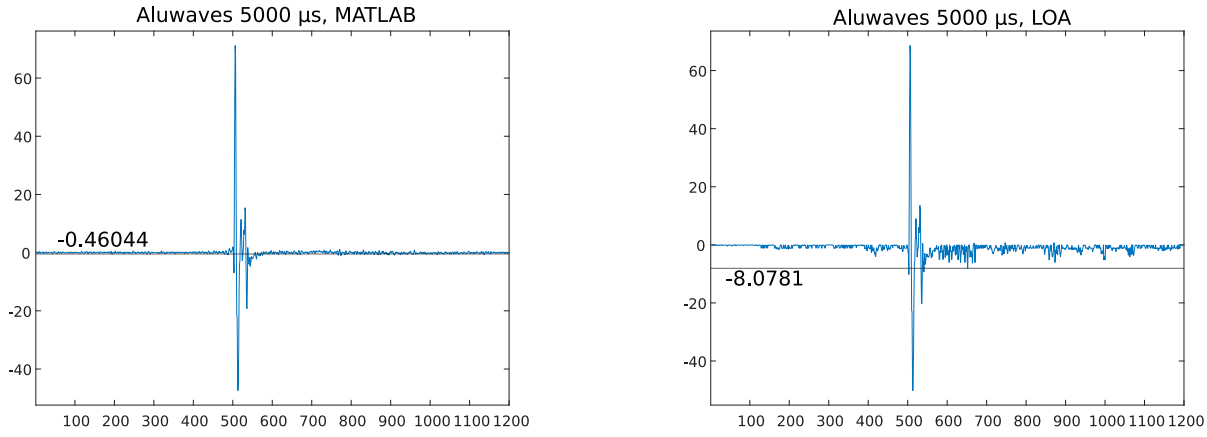
	ED	ER (SW)	ER (HW)	MED	Mean err
Average	6.9844	0.4975	0.4926	0.3923	-0.2992
Max	14.266	0.9929	0.9889	1.5804	-1.4335

Table 6.13: Error evaluation of the LOA smoothing filter

Compared to Figure 6.2, the effects of approximation are more visible in Figure 6.6. Naturally, the large ER does not affect the final result significantly. It is important to underline that the graphs shown in these sections compare columns that are noisier than average. This means that normally, images such as Figure 6.6b are not heavily altered by noise and show better similarities with the software results. The numbers listed in Tables 6.12 and 6.13 are similar to the metrics characterizing the previous designs.
The LOA DF-NS's accuracy is consistent with the previous designs. This time though, in some cases, the peak present in each column of the software filter appears to be higher than the approximate one. However, from the maximum ED shown in Table 6.14, it follows that this problem is much less relevant than before.

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	6.6537	1.0000	0.9997	0.1031	-0.1022
aluwaves 500 μs	6.2531	1.0000	0.9991	0.2364	-0.2351
aluwaves 1000 μs	7.6177	1.0000	0.9988	0.3751	-0.3731
aluwaves 2000 μs	7.6177	1.0000	0.9985	0.5572	-0.5553
aluwaves 5000 μs	7.6177	1.0000	0.9976	0.9102	-0.9089
plane 10 μs	6.4255	1.0000	0.9993	0.2018	-0.2015
plane 20 μs	6.3916	1.0000	0.9993	0.2235	-0.2233
plane 40 μs	6.4311	1.0000	0.9993	0.2661	-0.2656
plane 80 μs	6.0986	1.0000	0.9994	0.3129	-0.3119
plane 160 μs	6.9366	1.0000	0.9990	0.3859	-0.3839
plane 320 μs	6.5921	1.0000	0.9976	0.6021	-0.5998
plane 640 μs	7.0579	1.0000	0.9957	1.0156	-1.0146
sphere 100 μs	5.7930	1.0000	0.9999	0.0824	-0.0813
sphere 200 μs	6.5251	1.0000	0.9999	0.0929	-0.0917
sphere 500 μs	6.3208	1.0000	0.9998	0.1137	-0.1121
sphere 1000 μs	6.4795	1.0000	0.9997	0.1365	-0.1347
sphere 2000 μs	6.4165	1.0000	0.9994	0.1794	-0.1772
sphere 5000 μs	6.7733	1.0000	0.9986	0.2870	-0.2845
weld 10 μs	6.3986	1.0000	0.9998	0.1044	-0.1036
weld 100 μs	6.2970	1.0000	0.9997	0.1421	-0.1409
weld 500 μs	6.5010	1.0000	0.9990	0.3105	-0.3091
weld 1000 μs 1	7.3357	1.0000	0.9987	0.4737	-0.4723
weld 1000 μs 2	7.1004	1.0000	0.9989	0.2465	-0.2439
weld 2000 μs	7.3534	1.0000	0.9986	0.6669	-0.6658
weld 5000 μs	7.6177	1.0000	0.9983	0.8918	-0.8908
white plane 20 μs	6.2088	1.0000	0.9989	0.2572	-0.2561
white plane 50 μs	6.2475	1.0000	0.9991	0.3430	-0.3414
white plane 100 μs	6.1669	1.0000	0.9988	0.4377	-0.4358
white plane 200 μs	7.0548	1.0000	0.9984	0.5852	-0.5830
white plane 500 μs	6.9997	1.0000	0.9956	0.9601	-0.9586
white plane 1000 μs	6.5131	1.0000	0.9941	1.3526	-1.3521
white plane 2000 μs	7.4128	1.0000	0.9986	1.6975	-1.6970
Average	6.7253	1.0000	0.9987	0.4547	-0.4533
Max	7.6177	1.0000	0.9999	1.6975	-1.6970

Table 6.14: Error evaluation of the LOA differentiation filter (no smoothing)



(a) MATLAB implementation

(b) Approximate circuit

Figure 6.7: Results of one column provided by the exact differentiation (no smoothing) filter from MATLAB vs. the LOA approximate circuit implemented using VHDL

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	3.2852	1.0000	0.9951	0.0539	-0.0466
aluwaves 500 μs	3.4560	1.0000	0.9877	0.0739	-0.0601
aluwaves 1000 μs	3.4438	1.0000	0.9804	0.1013	-0.0804
aluwaves 2000 μs	3.4931	1.0000	0.9748	0.1408	-0.1115
aluwaves 5000 μs	3.3773	1.0000	0.9658	0.2274	-0.1826
plane 10 μs	3.2862	1.0000	0.9996	0.1133	-0.0891
plane 20 μs	3.3898	1.0000	0.9994	0.1227	-0.0980
plane 40 μs	3.3404	1.0000	0.9988	0.1381	-0.1103
plane 80 μs	3.2570	1.0000	0.9973	0.1530	-0.1197
plane 160 μs	3.1908	1.0000	0.9930	0.1738	-0.1318
plane 320 μs	3.4107	1.0000	0.9778	0.2203	-0.1616
plane 640 μs	3.3266	1.0000	0.9691	0.2887	-0.2149
sphere 100 μs	3.1007	1.0000	0.9988	0.0492	-0.0445
sphere 200 μs	3.1896	1.0000	0.9983	0.0526	-0.0470
sphere 500 μs	3.5740	1.0000	0.9968	0.0592	-0.0514
sphere 1000 μs	3.3581	1.0000	0.9951	0.0649	-0.0554
sphere 2000 μs	3.3904	1.0000	0.9913	0.0753	-0.0620
sphere 5000 μs	3.4883	1.0000	0.9825	0.0954	-0.0759
weld 10 μs	3.2682	1.0000	0.9985	0.0601	-0.0512
weld 100 μs	3.3716	1.0000	0.9959	0.0701	-0.0583
weld 500 μs	3.4255	1.0000	0.9836	0.1013	-0.0816
weld 1000 μs 1	3.3607	1.0000	0.9805	0.1293	-0.1033
weld 1000 μs 2	3.4268	1.0000	0.9861	0.0903	-0.0691
weld 2000 μs	3.3591	1.0000	0.9797	0.1733	-0.1398
weld 5000 μs	3.2747	1.0000	0.9822	0.2650	-0.2194
white plane 20 μs	3.3841	1.0000	0.9987	0.1382	-0.1049
white plane 50 μs	3.4522	1.0000	0.9971	0.1739	-0.1342
white plane 100 μs	3.3007	1.0000	0.9944	0.2017	-0.1536
white plane 200 μs	3.3602	1.0000	0.9901	0.2469	-0.1791
white plane 500 μs	3.3091	1.0000	0.9812	0.3470	-0.2457
white plane 1000 μs	3.3924	1.0000	0.9816	0.4459	-0.3176
white plane 2000 μs	3.3758	1.0000	0.9880	0.5969	-0.4792

Table 6.15: Error evaluation of the LOA 2nd order differentiation filter (no smoothing)

	ED	ER (SW)	ER (HW)	MED	Mean err
Average	3.3568	1.0000	0.9887	0.1639	-0.1275
Max	3.5740	1.0000	0.9996	0.5969	-0.4792

Table 6.16: Error evaluation of the LOA 2nd order differentiation filter (no smoothing)

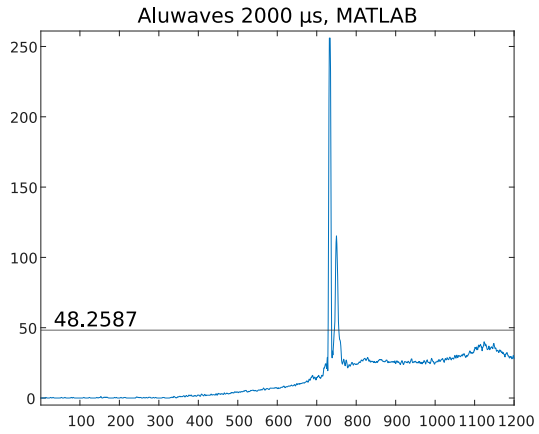
Given its simplicity, the LOA DF-2ND is again the most accurate LOA filter as confirmed by the comparison of Tables 6.15 and 6.16 with Tables 6.12 and 6.14. In this case, the maximum ED is produced by the difference between the maxima of the two columns, as explained before. This problem affects all the LOA differentiation filters.
The coupling of LOA smoothing and DF-S filters generates unexpected metric: this design is more accurate than the LOA smoothing filter alone. What it means is that error compensation occurs at some point along the chain. As a consequence, the MED and the mean error reduce by more than 50%. The maximum ED also decreases. Thus, the combination of the smoothing filter and the differentiation filter creates a design that is even more accurate the the DF-2ND filter. The results are reported in Table 6.17.

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	2.0311	1.0000	0.9925	0.0363	-0.0105
aluwaves 500 μs	2.5299	1.0000	0.9902	0.0790	-0.0140
aluwaves 1000 μs	2.8655	1.0000	0.9866	0.1275	-0.0166
aluwaves 2000 μs	2.8855	1.0000	0.9877	0.1766	-0.0182
aluwaves 5000 μs	3.0504	1.0000	0.9919	0.2657	-0.0226
plane 10 μs	2.1361	1.0000	0.9900	0.0428	-0.0098
plane 20 μs	2.0654	1.0000	0.9896	0.0511	-0.0097
plane 40 μs	2.2172	1.0000	0.9878	0.0637	-0.0092
plane 80 μs	2.1822	1.0000	0.9860	0.0778	-0.0113
plane 160 μs	2.4590	1.0000	0.9819	0.1145	-0.0134
plane 320 μs	2.5240	1.0000	0.9883	0.1981	-0.0192
plane 640 μs	2.8766	1.0000	0.9917	0.2970	-0.0177
sphere 100 μs	2.1280	1.0000	0.9884	0.0251	-0.0086
sphere 200 μs	2.5990	1.0000	0.9871	0.0299	-0.0089
sphere 500 μs	2.5173	1.0000	0.9851	0.0414	-0.0095
sphere 1000 μs	2.6325	1.0000	0.9856	0.0519	-0.0103
sphere 2000 μs	2.4956	1.0000	0.9852	0.0722	-0.0118
sphere 5000 μs	2.6245	1.0000	0.9859	0.1157	-0.0152
weld 10 μs	2.2286	1.0000	0.9921	0.0287	-0.0088
weld 100 μs	2.3767	1.0000	0.9910	0.0447	-0.0104
weld 500 μs	2.5427	1.0000	0.9906	0.1026	-0.0148
weld 1000 μs 1	3.0894	1.0000	0.9916	0.1429	-0.0161
weld 1000 μs 2	2.5483	1.0000	0.9840	0.0992	-0.0134
weld 2000 μs	2.7081	1.0000	0.9922	0.1876	-0.0161
weld 5000 μs	3.0652	1.0000	0.9910	0.2408	-0.0135
white plane 20 μs	1.9907	1.0000	0.9795	0.0604	-0.0114
white plane 50 μs	2.0368	1.0000	0.9810	0.0855	-0.0121
white plane 100 μs	2.4816	1.0000	0.9789	0.1179	-0.0130
white plane 200 μs	2.5924	1.0000	0.9826	0.1689	-0.0153
white plane 500 μs	2.7314	1.0000	0.9912	0.2816	-0.0158
white plane 1000 μs	2.7912	1.0000	0.9904	0.3586	-0.0126
white plane 2000 μs	2.9332	1.0000	0.9953	0.4278	-0.0077
Average	2.5293	1.0000	0.9879	0.1317	-0.0130
Max	3.0894	1.0000	0.9953	0.4278	-0.0226

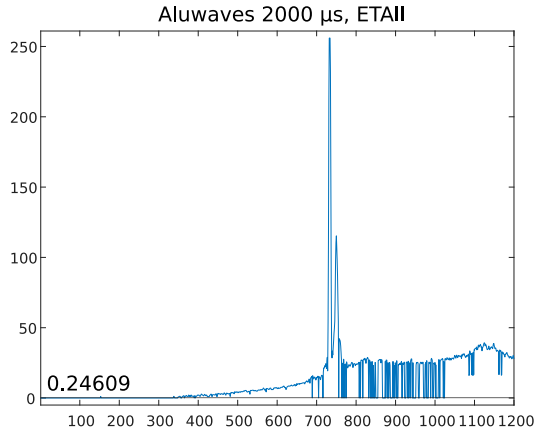
Table 6.17: Error evaluation of the LOA differentiation filter (after smoothing)

6.1.1.4 ETAII filter

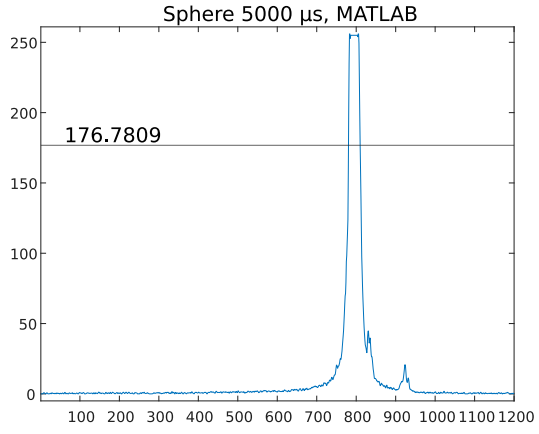
As explained in Chapter 5, the fully approximated ETAII adder has no practical purpose due to its large ED. This is especially true when the bit-width is small. Because of this, a lot of versions for this circuit have only been partially approximated to keep the error under control. Nonetheless, the effects of this approximation are clearly visible in the ETAII smoothing filter. This algorithm suffers from a problem that affects the MSBs. It happens when a carry bit is generated at the least significant group of bits and it has to propagate all the way to the MSBs. As the carry chain is split, this bit never makes it to the leftmost block unless another carry is created along the path. When this does not happen, the MSBs contain an error, which due to their weight, is also the highest possible. The example in Figure 6.8b shows that a lot of samples go down to zero, while others occupy a lower position compared to their surroundings. The return to zero happens due to the correction applied by the saturation circuit for negative numbers. This happens when a lot of '0's separate the MSB block from the LSB block. Depending on where the leading '1' is, the result can be either zero or a number much smaller than the actual result. For instance, if the correct result is "00010000011110010" and the approximate one is "00000000011110010", the resulting number is at least 10 times smaller than the original. When the smoothing filter performs the final subtraction, the result is negative rather than positive. This in turn means that the output will be saturated to '0'. If the approximate number is larger instead, the subtraction yields a positive number, which is however not even close to the values of surrounding samples. Both cases of this drooping effect can be seen in Figure 6.8. In the different situation presented in Figure 6.8d, a sagging phenomenon is visible when the laser line is very wide.



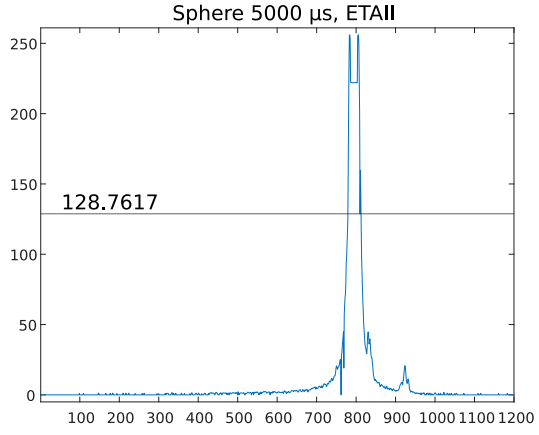
(a) MATLAB implementation



(b) Approximate circuit



(c) MATLAB implementation



(d) Approximate circuit

Figure 6.8: Results of one column provided by the exact smoothing filter from MATLAB vs. the ETAII approximate circuit implemented using VHDL

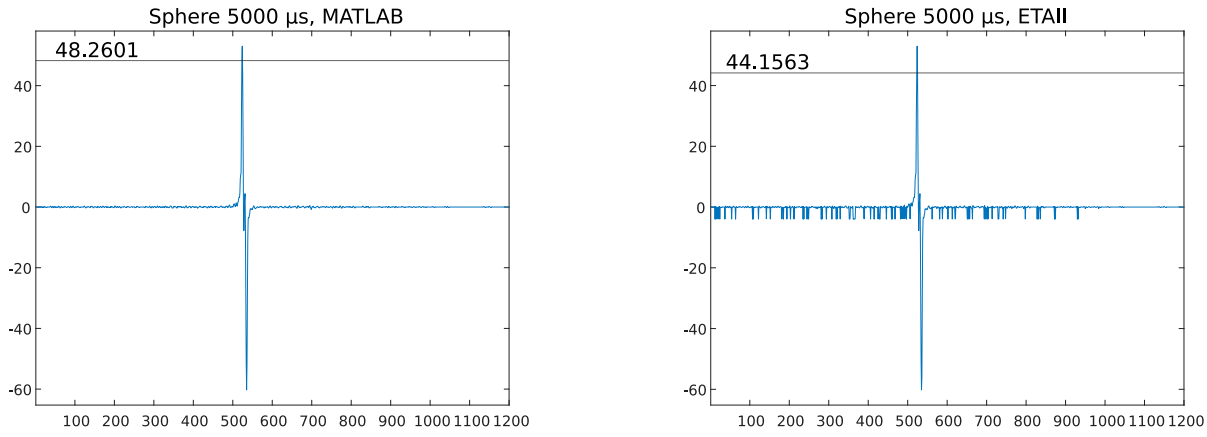
This is caused by the saturation applied on top combined with a wide laser line that generates several constant

consecutive samples. Although difficult to prove, this issue does not seem to affect the signal's derivative around the peak. As shown later in Figure 6.10 featured in the discussion surrounding the DF-S, the shape of the derivative looks similar to the accurate one but with sharper and wider transitions due to the fast variation in the original signal. These spikes and fluctuations do not pose a threat to the integrity of the signal as long as the region around the zero crossing is able to retain its shape. Table 6.18 shows the error metrics for the ETAII smoothing filter. Despite the high ED, this filter exhibits satisfactory ER and MED.

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	32.961	0.1899	0.1291	0.0587	-0.0586
aluwaves 500 μs	48.062	0.3925	0.1738	0.0938	-0.0934
aluwaves 1000 μs	48.027	0.6180	0.2618	0.1718	-0.1713
aluwaves 2000 μs	48.013	0.7658	0.2629	0.3028	-0.3021
aluwaves 5000 μs	48.070	0.9921	0.1819	0.4960	-0.4950
plane 10 μs	32.980	0.1019	0.0503	0.0550	-0.0542
plane 20 μs	48.028	0.1240	0.0619	0.0755	-0.0735
plane 40 μs	48.080	0.1749	0.0892	0.0962	-0.0932
plane 80 μs	48.067	0.2775	0.1560	0.1253	-0.1239
plane 160 μs	33.069	0.4841	0.2916	0.1778	-0.1761
plane 320 μs	33.043	0.8484	0.3700	0.2820	-0.2803
plane 640 μs	33.048	0.9910	0.1256	0.3711	-0.3693
sphere 100 μs	48.014	0.1370	0.1215	0.0373	-0.0372
sphere 200 μs	48.050	0.1683	0.1459	0.0472	-0.0470
sphere 500 μs	48.044	0.2371	0.2000	0.0731	-0.0730
sphere 1000 μs	48.018	0.2861	0.2276	0.1241	-0.1239
sphere 2000 μs	48.019	0.3989	0.2969	0.2063	-0.2062
sphere 5000 μs	48.019	0.6143	0.3674	0.3730	-0.3727
weld 10 μs	47.967	0.1230	0.0932	0.0381	-0.0378
weld 100 μs	33.083	0.2208	0.1584	0.0739	-0.0735
weld 500 μs	48.055	0.4944	0.2041	0.1485	-0.1477
weld 1000 μs 1	48.031	0.6182	0.1916	0.1816	-0.1809
weld 1000 μs 2	48.059	0.5422	0.3656	0.1771	-0.1767
weld 2000 μs	48.065	0.6807	0.1592	0.2651	-0.2641
weld 5000 μs	48.102	0.7396	0.1500	0.6589	-0.6575
white plane 20 μs	31.953	0.2392	0.1613	0.0630	-0.0627
white plane 50 μs	48.027	0.3543	0.2268	0.1520	-0.1507
white plane 100 μs	48.059	0.4907	0.2957	0.2074	-0.2046
white plane 200 μs	48.029	0.6758	0.3669	0.2710	-0.2690
white plane 500 μs	48.013	0.9534	0.3037	0.3638	-0.3616
white plane 1000 μs	48.048	0.9916	0.1235	0.7388	-0.7363
white plane 2000 μs	48.059	0.9929	0.1134	1.4972	-1.4941
Average	44.724	0.4974	0.2008	0.2501	-0.2490
Max	48.102	0.9929	0.3700	1.4972	-1.4941

Table 6.18: Error evaluation of the ETAII smoothing filter

The problem affecting the smoothing filter has an impact on the differentiation filters as well. However, its effects are much smaller. The possibility to deal with negative numbers and the absence of the saturation circuit prevent the output signal from being set to '0'. A drooping effect is still present but is does not influence the subsequent zero-crossing detection. Figure 6.9 displays this phenomenon. Tables 6.19 and 6.20 contain the error metrics related to the DF-NS. The numbers for this filter are particularly good. The average hardware ER is one of the lowest encountered so far.



(a) MATLAB implementation

(b) Approximate circuit

Figure 6.9: Results of one column provided by the exact differentiation (no smoothing) filter from MATLAB vs. the ETAII approximate circuit implemented using VHDL

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	4.1432	0.2136	0.0190	0.0624	-0.0623
aluwaves 500 μs	4.1721	0.4207	0.0478	0.1588	-0.1584
aluwaves 1000 μs	4.1598	0.6557	0.0773	0.2549	-0.2544
aluwaves 2000 μs	4.1609	0.8000	0.1015	0.3313	-0.3307
aluwaves 5000 μs	4.1471	0.9938	0.1389	0.4508	-0.4500
plane 10 μs	4.0742	0.1274	0.0095	0.0241	-0.0235
plane 20 μs	4.1585	0.1524	0.0151	0.0456	-0.0439
plane 40 μs	4.1546	0.2074	0.0225	0.0721	-0.0690
plane 80 μs	4.1600	0.3215	0.0330	0.1065	-0.1023
plane 160 μs	4.1529	0.5468	0.0538	0.1731	-0.1685
plane 320 μs	4.1430	0.8919	0.1071	0.3498	-0.3452
plane 640 μs	4.1390	0.9967	0.1338	0.4272	-0.4226
sphere 100 μs	4.1239	0.1698	0.0074	0.0235	-0.0234
sphere 200 μs	4.1248	0.2067	0.0101	0.0322	-0.0320
sphere 500 μs	4.1124	0.2877	0.0180	0.0583	-0.0580
sphere 1000 μs	4.1150	0.3389	0.0257	0.0836	-0.0833
sphere 2000 μs	4.1051	0.4608	0.0434	0.1421	-0.1418
sphere 5000 μs	4.1039	0.6657	0.0742	0.2448	-0.2445
weld 10 μs	4.1564	0.1468	0.0101	0.0317	-0.0314
weld 100 μs	4.1563	0.2544	0.0220	0.0701	-0.0696
weld 500 μs	4.1467	0.5199	0.0579	0.1898	-0.1888
weld 1000 μs 1	4.1503	0.6401	0.0817	0.2672	-0.2660
weld 1000 μs 2	4.1678	0.6078	0.0638	0.2089	-0.2082
weld 2000 μs	4.1519	0.6913	0.0954	0.3089	-0.3077
weld 5000 μs	4.1419	0.7571	0.1035	0.3243	-0.3230
white plane 20 μs	4.0300	0.2946	0.0221	0.0668	-0.0667
white plane 50 μs	4.0986	0.4152	0.0375	0.1163	-0.1149
white plane 100 μs	4.1372	0.5637	0.0561	0.1797	-0.1762
white plane 200 μs	4.1363	0.7525	0.0893	0.2867	-0.2812
white plane 500 μs	4.1308	0.9831	0.1344	0.4251	-0.4192
white plane 1000 μs	4.1258	0.9991	0.1376	0.4210	-0.4152
white plane 2000 μs	4.1221	0.9998	0.1200	0.3427	-0.3369

Table 6.19: Error evaluation of the ETAII differentiation filter (no smoothing)

	ED	ER (SW)	ER (HW)	MED	Mean err
Average	4.1345	0.5338	0.0615	0.1963	-0.1943
Max	4.1721	0.9998	0.1389	0.4508	-0.4500

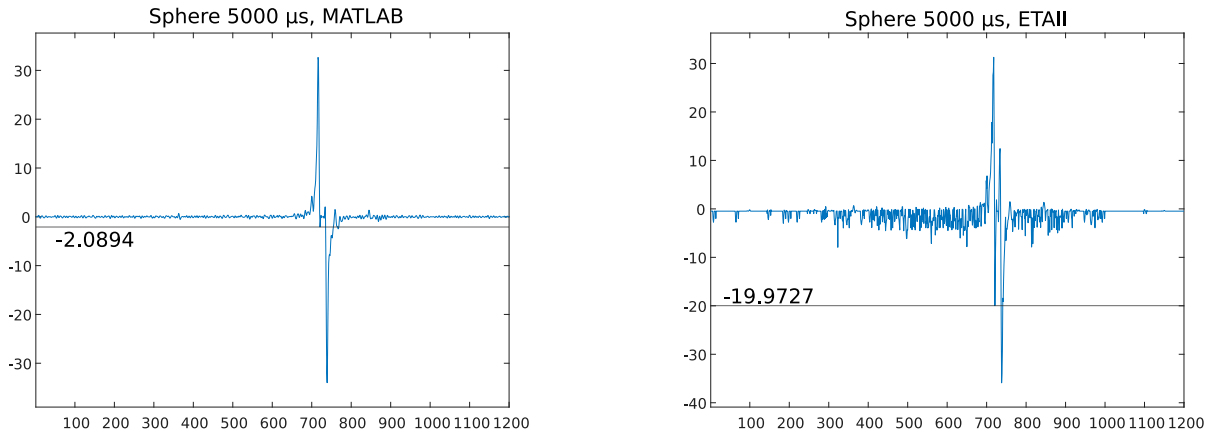
Table 6.20: Error evaluation of the ETAII differentiation filter (no smoothing)

The ETAII DF-2ND is extremely accurate. The error metrics shown in Table 6.21 do not represent the inaccuracy of the approximate algorithm but they simply highlight the presence of quantization noise that affects all filters. It is caused by the difference in precision between the software version and the hardware circuit. This becomes clear when comparing the two ERs. They indicate that this circuit is actually exact. This is because, due to the drooping effect created by the ETAII adders, approximations had to be avoided to ensure the filter works.

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	0.0121	0.2118	0.0000	0.0002	-0.0002
aluwaves 500 μs	0.0120	0.4178	0.0000	0.0004	-0.0004
aluwaves 1000 μs	0.0120	0.6509	0.0000	0.0006	-0.0006
aluwaves 2000 μs	0.0121	0.7940	0.0000	0.0008	-0.0007
aluwaves 5000 μs	0.0120	0.9899	0.0000	0.0010	-0.0009
plane 10 μs	0.0055	0.1266	0.0000	0.0002	-0.0001
plane 20 μs	0.0094	0.1518	0.0000	0.0003	-0.0001
plane 40 μs	0.0118	0.2064	0.0000	0.0005	-0.0002
plane 80 μs	0.0121	0.3195	0.0000	0.0007	-0.0003
plane 160 μs	0.0121	0.5419	0.0000	0.0010	-0.0005
plane 320 μs	0.0118	0.8852	0.0000	0.0013	-0.0008
plane 640 μs	0.0115	0.9879	0.0000	0.0015	-0.0009
sphere 100 μs	0.0091	0.1683	0.0000	0.0002	-0.0001
sphere 200 μs	0.0104	0.2047	0.0000	0.0002	-0.0002
sphere 500 μs	0.0108	0.2842	0.0000	0.0003	-0.0003
sphere 1000 μs	0.0107	0.3340	0.0000	0.0003	-0.0003
sphere 2000 μs	0.0105	0.4531	0.0000	0.0004	-0.0004
sphere 5000 μs	0.0103	0.6562	0.0000	0.0006	-0.0006
weld 10 μs	0.0120	0.1454	0.0000	0.0002	-0.0001
weld 100 μs	0.0120	0.2517	0.0000	0.0003	-0.0002
weld 500 μs	0.0120	0.5148	0.0000	0.0005	-0.0005
weld 1000 μs 1	0.0120	0.6351	0.0000	0.0007	-0.0006
weld 1000 μs 2	0.0120	0.6015	0.0000	0.0006	-0.0005
weld 2000 μs	0.0120	0.6883	0.0000	0.0007	-0.0006
weld 5000 μs	0.0118	0.7476	0.0000	0.0008	-0.0007
white plane 20 μs	0.0034	0.2921	0.0000	0.0003	-0.0003
white plane 50 μs	0.0090	0.4116	0.0000	0.0005	-0.0004
white plane 100 μs	0.0115	0.5587	0.0000	0.0009	-0.0005
white plane 200 μs	0.0117	0.7465	0.0000	0.0013	-0.0007
white plane 500 μs	0.0116	0.9758	0.0000	0.0016	-0.0009
white plane 1000 μs	0.0112	0.9815	0.0000	0.0016	-0.0009
white plane 2000 μs	0.0109	0.9668	0.0000	0.0016	-0.0009
Average	0.0109	0.5282	0.0000	0.0007	-0.0005
Max	0.0121	0.9899	0.0000	0.0016	-0.0009

Table 6.21: Error evaluation of the ETAII 2nd order differentiation filter (no smoothing)

When combining ETAII smoothing and differentiation filters together, the output signal is extremely noisy. While the shape is preserved, the fluctuations are steeper. The zero-crossing detector should still be able to find the zero. However, its position could be particularly inaccurate. Figure 6.10 shows the derivative of the signal in Figure 6.8d. All the error metrics listed in Tables 6.22 and 6.23 significantly increase.



(a) MATLAB implementation

(b) Approximate circuit

Figure 6.10: Results of one column provided by the exact differentiation (after smoothing) filter from MATLAB vs. the ETAII approximate circuit implemented using VHDL

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	11.005	1.0000	0.9941	0.5778	-0.5723
aluwaves 500 μs	13.112	1.0000	0.9662	0.8265	-0.8141
aluwaves 1000 μs	15.868	1.0000	0.9426	1.0442	-1.0153
aluwaves 2000 μs	16.658	1.0000	0.9136	1.2760	-1.1876
aluwaves 5000 μs	20.023	1.0000	0.8600	1.7730	-1.5829
plane 10 μs	12.060	1.0000	0.9899	0.5154	-0.4835
plane 20 μs	13.258	1.0000	0.9877	0.5419	-0.4996
plane 40 μs	14.593	1.0000	0.9825	0.5626	-0.5132
plane 80 μs	13.551	1.0000	0.9717	0.5937	-0.5371
plane 160 μs	12.886	1.0000	0.9520	0.6613	-0.6045
plane 320 μs	12.767	1.0000	0.9000	1.0619	-0.9854
plane 640 μs	15.195	1.0000	0.8059	1.5662	-1.3726
sphere 100 μs	12.576	1.0000	0.9957	0.4815	-0.4756
sphere 200 μs	13.797	1.0000	0.9939	0.4899	-0.4824
sphere 500 μs	16.687	1.0000	0.9909	0.5125	-0.5000
sphere 1000 μs	19.175	1.0000	0.9883	0.5621	-0.5312
sphere 2000 μs	17.948	1.0000	0.9819	0.6435	-0.6013
sphere 5000 μs	17.883	1.0000	0.9654	0.9108	-0.8531
weld 10 μs	14.426	1.0000	0.9943	0.5009	-0.4927
weld 100 μs	15.591	1.0000	0.9870	0.5463	-0.5323
weld 500 μs	16.533	1.0000	0.9551	0.9367	-0.9038
weld 1000 μs 1	16.651	1.0000	0.9241	1.1272	-1.0739
weld 1000 μs 2	16.814	1.0000	0.9703	0.7345	-0.7095
weld 2000 μs	16.887	1.0000	0.8986	1.2292	-1.1291
weld 5000 μs	20.634	1.0000	0.8769	1.3322	-1.0629
white plane 20 μs	12.802	1.0000	0.9867	0.5184	-0.4981
white plane 50 μs	10.830	1.0000	0.9758	0.5914	-0.5286
white plane 100 μs	12.907	1.0000	0.9571	0.6544	-0.5724
white plane 200 μs	14.671	1.0000	0.9252	0.7611	-0.6614
white plane 500 μs	14.890	1.0000	0.8412	1.1015	-0.9506
white plane 1000 μs	18.102	1.0000	0.7592	1.4334	-1.0008
white plane 2000 μs	19.108	1.0000	0.7413	1.5381	-0.7734

Table 6.22: Error evaluation of the ETAII differentiation filter (after smoothing)

	ED	ER (SW)	ER (HW)	MED	Mean err
Average	15.309	1.0000	0.9367	0.8627	-0.7657
Max	20.634	1.0000	0.9957	1.7730	-1.5829

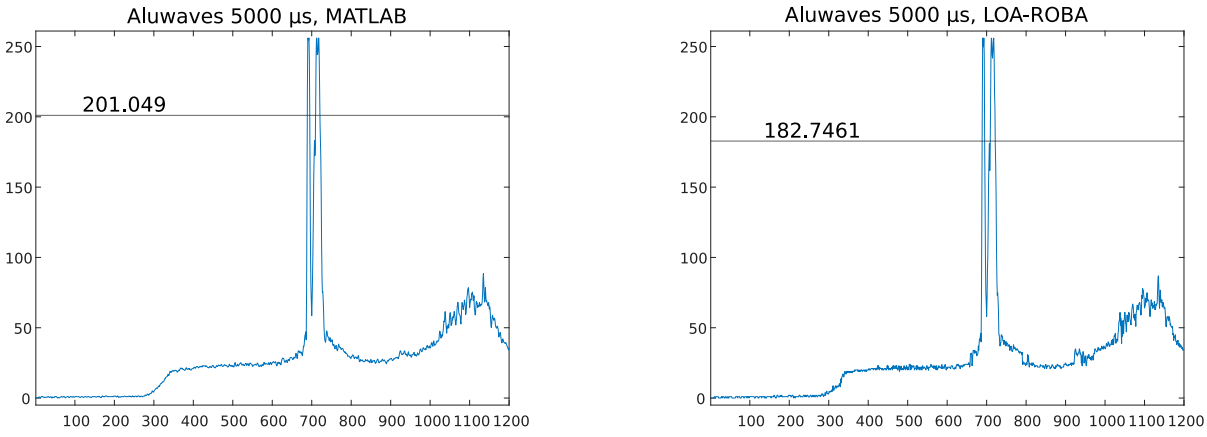
Table 6.23: Error evaluation of the ETAII differentiation filter (after smoothing)

6.1.1.5 LOA-RoBA filter

The LOA-RoBA filter is obtained by replacing all the multipliers in the filter with RoBA multipliers and all the adders and subtractors with LOAs. Figure 6.11 depicts an example of the effect the approximation has on an image processed by the smoothing filter. By looking at the previous set of results, it is assumed that the mean source of error is the LOA. Coupling these two approximation methods together, however, does not deteriorate the output signal. The error metrics for this design are shown in Table 6.24. Similarly to the RoBA smoothing filter, the maximum ED is greater than expected. This, however, does not affect the final image as this phenomenon is only local.

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	14.652	0.1900	0.1898	0.0638	-0.0257
aluwaves 500 μs	17.918	0.3925	0.3919	0.2463	-0.1640
aluwaves 1000 μs	17.844	0.6180	0.6168	0.4799	-0.3502
aluwaves 2000 μs	18.278	0.7658	0.7636	0.7013	-0.5314
aluwaves 5000 μs	18.303	0.9921	0.9884	1.1681	-0.8899
plane 10 μs	8.7405	0.1022	0.1019	0.0742	-0.0481
plane 20 μs	13.931	0.1243	0.1240	0.1185	-0.0574
plane 40 μs	18.185	0.1750	0.1724	0.2038	-0.1350
plane 80 μs	19.113	0.2775	0.2678	0.2399	-0.1471
plane 160 μs	19.322	0.4841	0.4708	0.3199	-0.1944
plane 320 μs	17.587	0.8484	0.8292	0.5594	-0.3042
plane 640 μs	17.830	0.9910	0.9700	0.9839	-0.6701
sphere 100 μs	14.477	0.1370	0.1369	0.0433	-0.0324
sphere 200 μs	19.569	0.1683	0.1681	0.0604	-0.0426
sphere 500 μs	19.419	0.2371	0.2363	0.0909	-0.0651
sphere 1000 μs	18.361	0.2861	0.2852	0.1212	-0.0835
sphere 2000 μs	18.839	0.3989	0.3980	0.1828	-0.1179
sphere 5000 μs	18.758	0.6143	0.6133	0.3190	-0.1798
weld 10 μs	18.252	0.1230	0.1227	0.0489	-0.0310
weld 100 μs	17.911	0.2208	0.2201	0.0963	-0.0578
weld 500 μs	18.440	0.4944	0.4921	0.2823	-0.1299
weld 1000 μs 1	17.955	0.6182	0.6142	0.4668	-0.3013
weld 1000 μs 2	19.904	0.5422	0.5400	0.2315	-0.1144
weld 2000 μs	18.527	0.6807	0.6762	0.7623	-0.5947
weld 5000 μs	19.060	0.7396	0.7344	1.0424	-0.8260
white plane 20 μs	6.0950	0.2395	0.2392	0.1008	-0.0728
white plane 50 μs	12.089	0.3545	0.3540	0.2090	-0.1369
white plane 100 μs	17.894	0.4908	0.4894	0.3647	-0.2068
white plane 200 μs	18.409	0.6758	0.6626	0.5135	-0.2997
white plane 500 μs	19.184	0.9534	0.9303	0.8181	-0.4637
white plane 1000 μs	18.126	0.9916	0.9689	1.1362	-0.7681
white plane 2000 μs	19.296	0.9929	0.9704	1.6595	-1.3306
Average	17.258	0.4975	0.4918	0.4284	-0.2929
Max	19.904	0.9929	0.9884	1.6595	-1.3306

Table 6.24: Error evaluation of the LOA-RoBA smoothing filter



(a) MATLAB implementation

(b) Approximate circuit

Figure 6.11: Results of one column provided by the exact smoothing filter from MATLAB vs. the LOA-RoBA approximate circuit implemented using VHDL

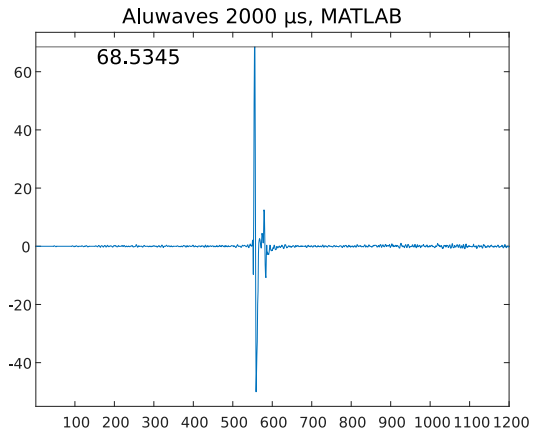
Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	7.4976	0.2136	0.1471	0.0483	-0.0477
aluwaves 500 μs	8.7473	0.4207	0.3223	0.1479	-0.1453
aluwaves 1000 μs	8.6977	0.6557	0.5095	0.2559	-0.2514
aluwaves 2000 μs	8.6009	0.8000	0.6546	0.3968	-0.3901
aluwaves 5000 μs	8.5485	0.9938	0.8744	0.6729	-0.6641
plane 10 μs	6.7879	0.1274	0.1105	0.1371	-0.1363
plane 20 μs	7.9446	0.1524	0.1320	0.1610	-0.1582
plane 40 μs	7.9631	0.2074	0.1729	0.2026	-0.1951
plane 80 μs	7.6543	0.3215	0.2520	0.2410	-0.2222
plane 160 μs	8.2641	0.5468	0.4157	0.3021	-0.2829
plane 320 μs	8.8349	0.8919	0.7392	0.4668	-0.4448
plane 640 μs	8.7979	0.9967	0.9211	0.7746	-0.7456
sphere 100 μs	7.0416	0.1698	0.0957	0.0338	-0.0336
sphere 200 μs	8.4469	0.2067	0.1195	0.0442	-0.0431
sphere 500 μs	7.9542	0.2877	0.1753	0.0636	-0.0615
sphere 1000 μs	8.6657	0.3389	0.2181	0.0821	-0.0802
sphere 2000 μs	8.0353	0.4608	0.3154	0.1171	-0.1152
sphere 5000 μs	7.4320	0.6657	0.4936	0.1999	-0.1968
weld 10 μs	7.5593	0.1468	0.0975	0.0508	-0.0499
weld 100 μs	8.4635	0.2544	0.1758	0.0840	-0.0820
weld 500 μs	8.8330	0.5199	0.4092	0.2089	-0.2036
weld 1000 μs 1	8.1979	0.6401	0.5298	0.3356	-0.3281
weld 1000 μs 2	9.1607	0.6078	0.4425	0.1707	-0.1657
weld 2000 μs	8.6201	0.6913	0.5994	0.4935	-0.4850
weld 5000 μs	8.8966	0.7571	0.6796	0.6842	-0.6757
white plane 20 μs	5.7010	0.2946	0.2138	0.1778	-0.1768
white plane 50 μs	6.3905	0.4152	0.3085	0.2670	-0.2650
white plane 100 μs	8.2484	0.5637	0.4341	0.3531	-0.3425
white plane 200 μs	9.2699	0.7525	0.6144	0.4870	-0.4590
white plane 500 μs	8.5904	0.9831	0.8958	0.7725	-0.7336
white plane 1000 μs	8.6274	0.9991	0.9647	1.0850	-1.0475
white plane 2000 μs	8.7385	0.9998	0.9636	1.4135	-1.3838

Table 6.25: Error evaluation of the LOA-RoBA differentiation filter (no smoothing)

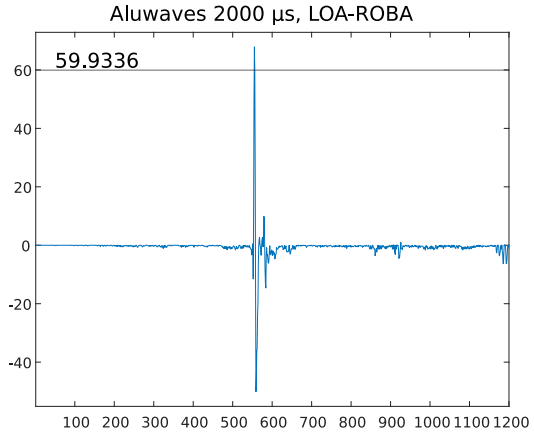
	ED	ER (SW)	ER (HW)	MED	Mean err
Average	8.1629	0.5338	0.4374	0.3416	-0.3316
Max	9.2699	0.9998	0.9647	1.4135	-1.3838

Table 6.26: Error evaluation of the LOA-RoBA differentiation filter (no smoothing)

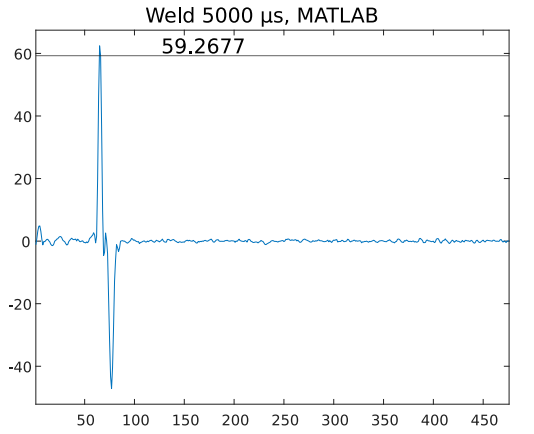
The LOA-RoBA DF-NS exhibits an interesting property: it is more accurate than the LOA DF-NS. Surprisingly, by increasing the approximation depth, the accuracy also grows. This means that by combining the LOA circuits and the RoBA multipliers, some kind of compensation occurs. Although the maximum ED is higher, both the MED and the ER are down. This last parameter goes from 100% to roughly 50% as seen in Tables 6.25 and 6.26. Compensation might happen when signed numbers are used with a symmetric range. Figures 6.12 show some examples. Once again, the output signal is very similar to the one generated with MATLAB.



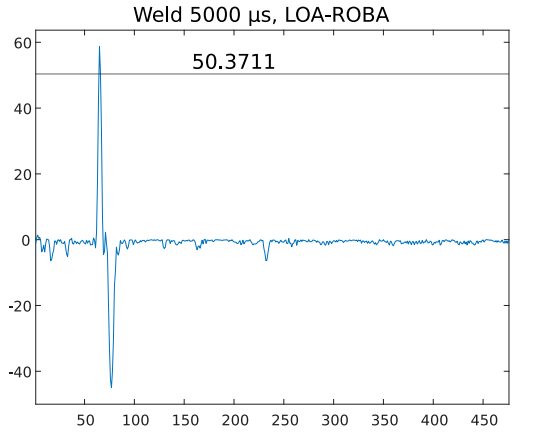
(a) MATLAB implementation



(b) Approximate circuit



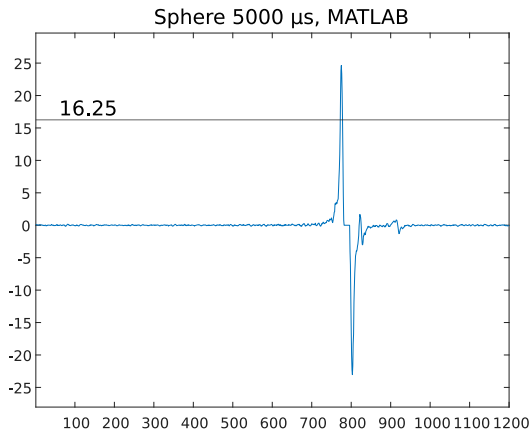
(c) MATLAB implementation



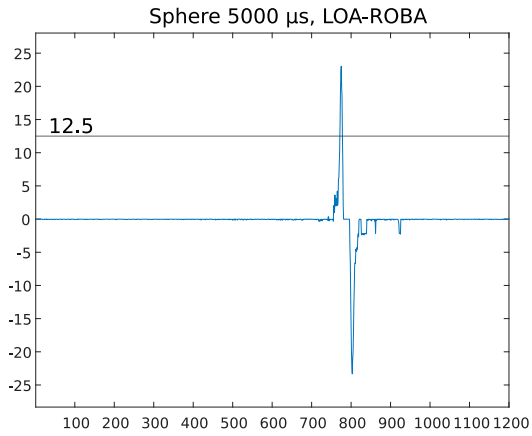
(d) Approximate circuit

Figure 6.12: Results of one column provided by the exact differentiation (no smoothing) filter from MATLAB vs. the LOA-RoBA approximate circuit implemented using VHDL

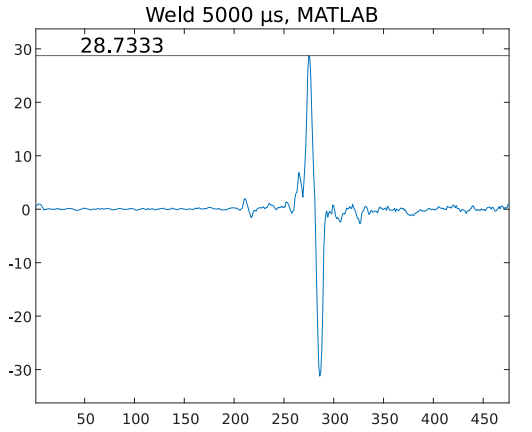
The LOA-RoBA DF-2ND displays metrics similar to the ones observed for the LOA filter. This is not surprising as the RoBA filter is very accurate. The results are reported in Table 6.27. Figure 6.13 shows an example. It can be seen that oscillations are less frequent but are more squared.



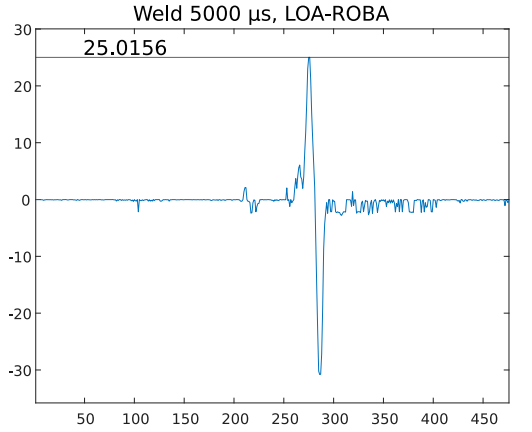
(a) MATLAB implementation



(b) Approximate circuit



(c) MATLAB implementation



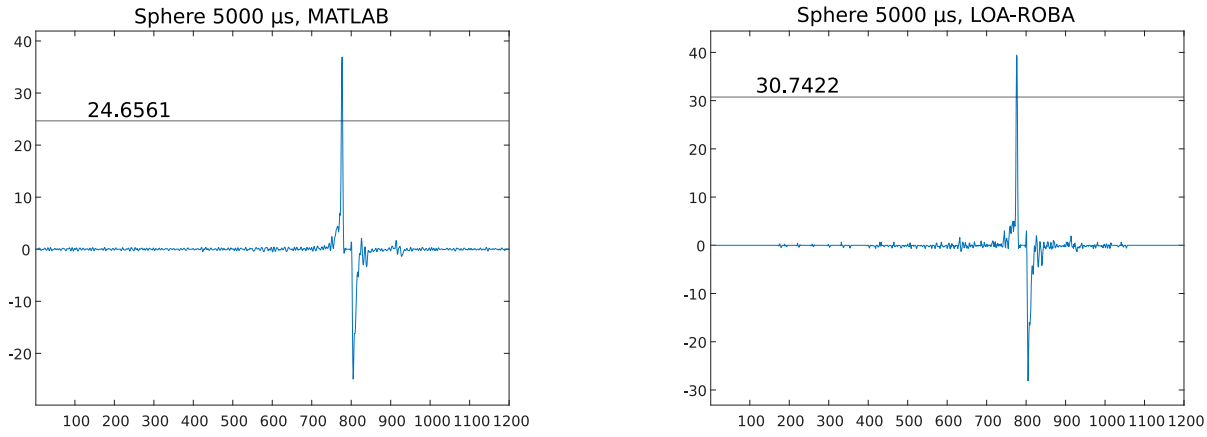
(d) Approximate circuit

Figure 6.13: Results of one column provided by the exact 2nd order differentiation filter from MATLAB vs. the LOA-RoBA approximate circuit implemented using VHDL

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	3.5000	1.0000	0.9945	0.0381	-0.0294
aluwaves 500 μs	3.8677	1.0000	0.9878	0.0601	-0.0432
aluwaves 1000 μs	3.6021	0.9999	0.9808	0.0894	-0.0637
aluwaves 2000 μs	3.7510	0.9999	0.9760	0.1302	-0.0949
aluwaves 5000 μs	3.8188	0.9999	0.9683	0.2184	-0.1664
plane 10 μs	3.3484	1.0000	0.9997	0.0984	-0.0734
plane 20 μs	3.7844	1.0000	0.9994	0.1093	-0.0828
plane 40 μs	3.8500	1.0000	0.9988	0.1238	-0.0946
plane 80 μs	3.7844	1.0000	0.9973	0.1414	-0.1055
plane 160 μs	3.5833	1.0000	0.9928	0.1651	-0.1185
plane 320 μs	3.5396	1.0000	0.9779	0.2148	-0.1478
plane 640 μs	3.6021	0.9999	0.9738	0.2845	-0.2011
sphere 100 μs	3.1188	1.0000	0.9987	0.0328	-0.0273
sphere 200 μs	3.5667	1.0000	0.9981	0.0365	-0.0299
sphere 500 μs	3.9021	1.0000	0.9963	0.0438	-0.0344
sphere 1000 μs	3.8854	1.0000	0.9944	0.0501	-0.0386
sphere 2000 μs	3.7188	1.0000	0.9898	0.0618	-0.0456
sphere 5000 μs	3.7500	0.9999	0.9807	0.0835	-0.0594
weld 10 μs	3.4365	1.0000	0.9983	0.0440	-0.0341
weld 100 μs	3.4688	1.0000	0.9956	0.0551	-0.0416
weld 500 μs	3.5854	0.9999	0.9833	0.0887	-0.0654
weld 1000 μs 1	3.4896	0.9999	0.9809	0.1178	-0.0874
weld 1000 μs 2	3.5688	1.0000	0.9848	0.0781	-0.0526
weld 2000 μs	3.6021	1.0000	0.9820	0.1626	-0.1242
weld 5000 μs	3.7177	1.0000	0.9852	0.2546	-0.2041
white plane 20 μs	3.3685	1.0000	0.9989	0.1232	-0.0875
white plane 50 μs	3.5792	1.0000	0.9969	0.1623	-0.1203
white plane 100 μs	3.2850	1.0000	0.9946	0.1940	-0.1416
white plane 200 μs	3.7521	1.0000	0.9905	0.2403	-0.1660
white plane 500 μs	3.5354	1.0000	0.9834	0.3438	-0.2342
white plane 1000 μs	3.6188	0.9999	0.9871	0.4437	-0.3075
white plane 2000 μs	3.8521	1.0000	0.9931	0.5944	-0.4717
Average	3.6198	1.0000	0.9894	0.1526	-0.1123
Max	3.9021	1.0000	0.9997	0.5944	-0.4717

Table 6.27: Error evaluation of the LOA-RoBA 2nd order differentiation filter (no smoothing)

For the LOA-RoBA DF-S, its ED and ER are really close to the metrics of the RoBA DF-S, whereas the MED is more akin to that of the LOA DF-S. Similarly to what happened with the DF-NS, the ER is halved. However, the MED increases. Figure 6.14 shows a comparison between exact and approximate designs. Although the differences are hard to spot, they are visible. In terms of accuracy, cascading two LOA-RoBA filters does not make the error explode. Tables 6.28 and 6.29 lists the error metrics.



(a) MATLAB implementation

(b) Approximate circuit

Figure 6.14: Results of one column provided by the exact differentiation (after smoothing) filter from MATLAB vs. the LOA-RoBA approximate circuit implemented using VHDL

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	6.4949	0.2520	0.2390	0.0285	-0.0056
aluwaves 500 μs	7.1021	0.4654	0.4480	0.0753	-0.0207
aluwaves 1000 μs	7.0621	0.7128	0.6884	0.1257	-0.0338
aluwaves 2000 μs	6.0376	0.8414	0.8200	0.1781	-0.0482
aluwaves 5000 μs	6.3440	0.9938	0.9821	0.2716	-0.0768
plane 10 μs	3.0175	0.1830	0.1733	0.0406	-0.0011
plane 20 μs	5.0392	0.2067	0.1964	0.0661	-0.0057
plane 40 μs	7.0230	0.2685	0.2553	0.1049	-0.0004
plane 80 μs	6.3085	0.3924	0.3735	0.1361	0.0051
plane 160 μs	5.9615	0.6394	0.6115	0.1724	-0.0111
plane 320 μs	5.8992	0.9223	0.9060	0.2493	-0.0472
plane 640 μs	6.7609	0.9978	0.9904	0.3513	-0.0835
sphere 100 μs	6.3609	0.2524	0.2295	0.0201	-0.0011
sphere 200 μs	6.9453	0.2982	0.2724	0.0279	-0.0016
sphere 500 μs	6.1693	0.3864	0.3571	0.0405	-0.0027
sphere 1000 μs	6.2414	0.4304	0.4020	0.0500	-0.0050
sphere 2000 μs	5.6234	0.5448	0.5164	0.0701	-0.0091
sphere 5000 μs	6.0861	0.7307	0.7029	0.1099	-0.0220
weld 10 μs	6.4197	0.1967	0.1827	0.0241	-0.0012
weld 100 μs	6.3166	0.3052	0.2886	0.0440	-0.0040
weld 500 μs	6.7976	0.5537	0.5367	0.1085	-0.0291
weld 1000 μs 1	6.0514	0.6622	0.6481	0.1547	-0.0411
weld 1000 μs 2	6.6627	0.6921	0.6625	0.0990	-0.0157
weld 2000 μs	6.2630	0.7103	0.6981	0.1998	-0.0433
weld 5000 μs	6.7312	0.7867	0.7719	0.2672	-0.0540
white plane 20 μs	2.2114	0.4109	0.3819	0.0533	-0.0015
white plane 50 μs	4.7733	0.5266	0.4975	0.1006	-0.0076
white plane 100 μs	7.4634	0.6771	0.6466	0.1822	-0.0136
white plane 200 μs	6.7419	0.8340	0.8093	0.2588	-0.0225
white plane 500 μs	6.1840	0.9928	0.9845	0.3662	-0.0567
white plane 1000 μs	6.2797	0.9999	0.9952	0.4477	-0.0694
white plane 2000 μs	6.4319	1.0000	0.9968	0.5279	-0.0643

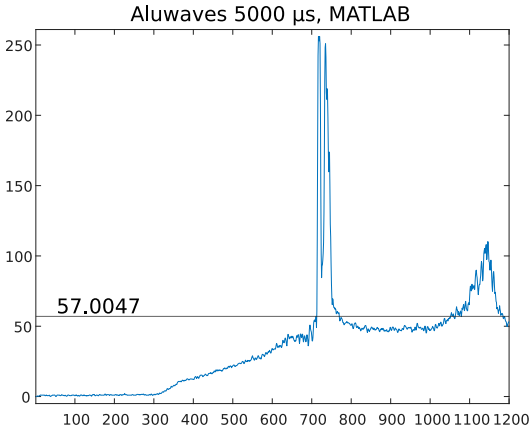
Table 6.28: Error evaluation of the LOA-RoBA differentiation filter (after smoothing)

	ED	ER (SW)	ER (HW)	MED	Mean err
Average	6.1189	0.5896	0.5708	0.1548	-0.0248
Max	7.4634	1.0000	0.9968	0.5279	-0.0835

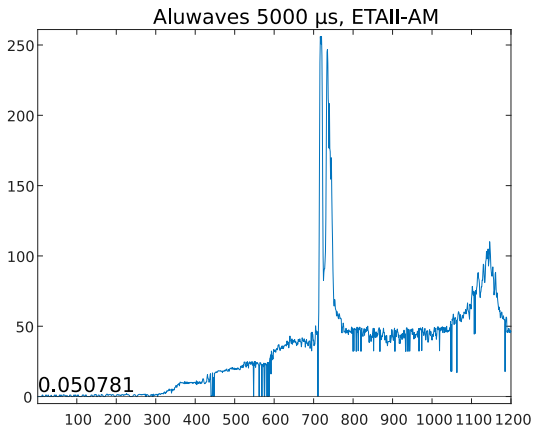
Table 6.29: Error evaluation of the LOA-RoBA differentiation filter (after smoothing)

6.1.1.6 ETAII-AM-ER filter

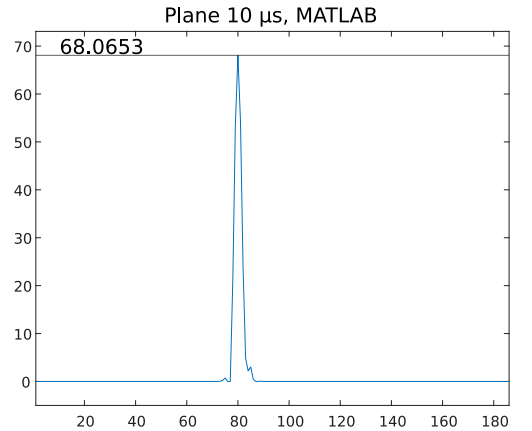
The ETAII-AM-ER filter combines the ETAII adders and subtractors with AM-ERs. While the accuracy of the different multipliers is stable, the ETAII circuits possess different sizes and approximation strategies. Thus, their error performance can vary significantly. Starting from the smoothing filter, it is clear that this circuit is affected by the same problem the ETAII filters suffer from. Tables 6.30 and 6.31 show that, while the ER is unchanged, the MED and ED increase when compared to the partially approximated filters (Tables 6.18 and 6.6). The images presented in Figure 6.15 and 6.16 indicate that the drooping effect is not negligible. While the results in Figure 6.15b might be acceptable, the error in Figures 6.15d and 6.16b affects the peak. Not only does this make the maximum plummet to a lower value, but it also reduces the overall height of the signal. For example, in Figure 6.15c, the peak is at 68, whereas the maximum in Figure 6.15d is placed below 50. Despite the loss in the signal intensity, the zero can still be correctly identified given the appropriate PT. When dealing with two consecutive peaks, one would expect to deal with two zeroes close together. This effect can be filtered by the zero-crossing detector after computing the average position. However, experimental results show that this disturbance does not affect the zero, which occurs once, but the shapes of the maximum and minimum around it. The zero is also in the correct position. An example is shown in Figure 6.17.



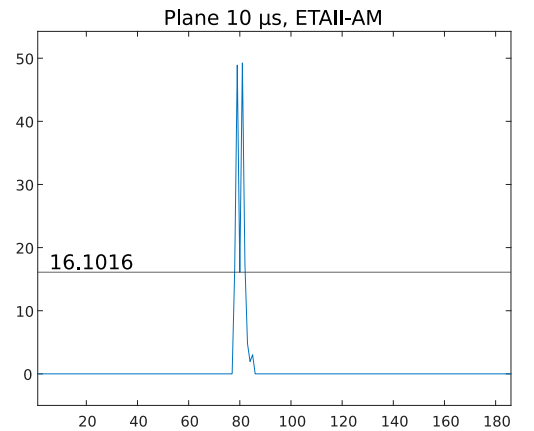
(a) MATLAB implementation



(b) Approximate circuit

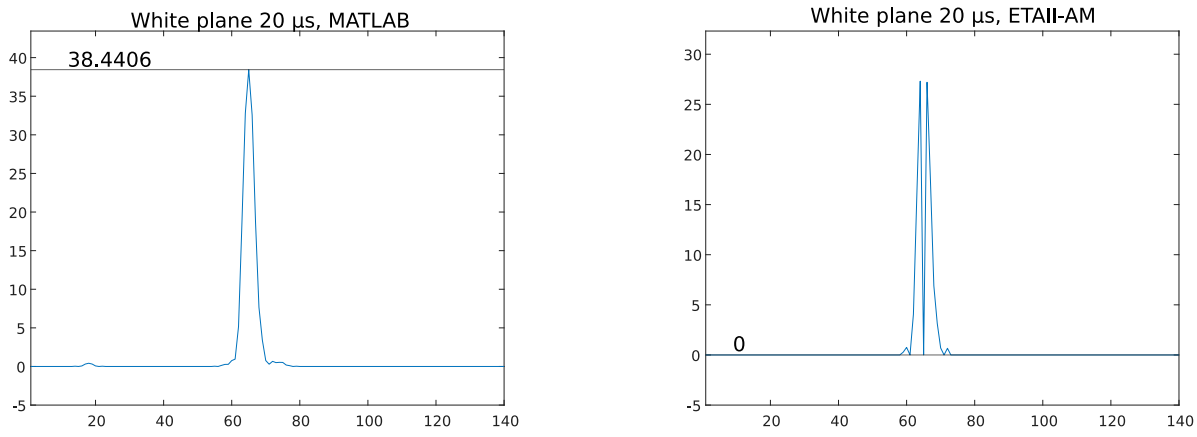


(c) MATLAB implementation



(d) Approximate circuit

Figure 6.15: Results of one column provided by the exact smoothing filter from MATLAB vs. the ETAII-AM-ER approximate circuit implemented using VHDL



(a) MATLAB implementation

(b) Approximate circuit

Figure 6.16: Results of one column provided by the exact smoothing filter from MATLAB vs. the ETAII-AM-ER approximate circuit implemented using VHDL

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	40.632	0.1899	0.1397	0.0743	-0.0742
aluwaves 500 μs	56.659	0.3925	0.2768	0.2003	-0.2001
aluwaves 1000 μs	55.162	0.6180	0.4494	0.4123	-0.4119
aluwaves 2000 μs	55.602	0.7658	0.5567	0.7356	-0.7346
aluwaves 5000 μs	56.954	0.9921	0.7055	1.4594	-1.4566
plane 10 μs	51.964	0.1019	0.0931	0.1472	-0.1468
plane 20 μs	54.192	0.1240	0.1131	0.2155	-0.2146
plane 40 μs	53.536	0.1749	0.1486	0.2724	-0.2706
plane 80 μs	55.612	0.2775	0.2283	0.3278	-0.3264
plane 160 μs	44.792	0.4841	0.3882	0.4249	-0.4236
plane 320 μs	46.219	0.8484	0.5502	0.5992	-0.5978
plane 640 μs	48.100	0.9910	0.6013	0.9023	-0.8991
sphere 100 μs	54.120	0.1370	0.1307	0.0574	-0.0574
sphere 200 μs	52.026	0.1683	0.1584	0.0774	-0.0772
sphere 500 μs	52.029	0.2371	0.2198	0.1122	-0.1120
sphere 1000 μs	53.461	0.2861	0.2553	0.1454	-0.1451
sphere 2000 μs	55.003	0.3989	0.3398	0.2112	-0.2108
sphere 5000 μs	56.001	0.6143	0.4558	0.3556	-0.3551
weld 10 μs	54.586	0.1230	0.1076	0.0652	-0.0651
weld 100 μs	50.126	0.2208	0.1846	0.1204	-0.1201
weld 500 μs	54.354	0.4944	0.2931	0.2687	-0.2681
weld 1000 μs 1	47.655	0.6182	0.4202	0.4384	-0.4375
weld 1000 μs 2	47.342	0.5422	0.4212	0.2522	-0.2517
weld 2000 μs	54.552	0.6807	0.5214	0.7396	-0.7383
weld 5000 μs	55.707	0.7396	0.6172	1.4041	-1.4005
white plane 20 μs	38.441	0.2392	0.2175	0.1277	-0.1277
white plane 50 μs	46.462	0.3543	0.3163	0.3501	-0.3492
white plane 100 μs	53.095	0.4907	0.4170	0.5277	-0.5258
white plane 200 μs	45.050	0.6758	0.5375	0.6685	-0.6659
white plane 500 μs	45.219	0.9534	0.6161	0.9405	-0.9364
white plane 1000 μs	54.212	0.9916	0.6668	1.2961	-1.2908
white plane 2000 μs	56.394	0.9929	0.8706	1.9704	-1.9649

Table 6.30: Error evaluation of the ETAII-AM-ER smoothing filter

	ED	ER (SW)	ER (HW)	MED	Mean err
Average	51.414	0.4975	0.3756	0.4969	-0.4955
Max	56.954	0.9929	0.8706	1.9704	-1.9649

Table 6.31: Error evaluation of the ETAII-AM-ER smoothing filter

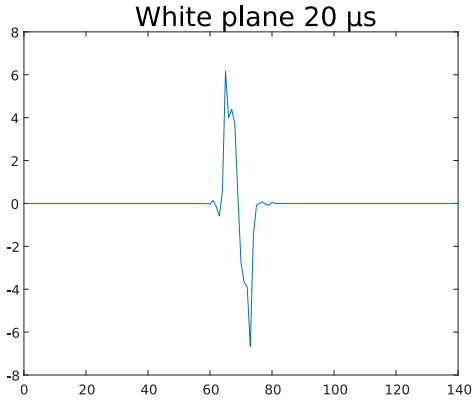
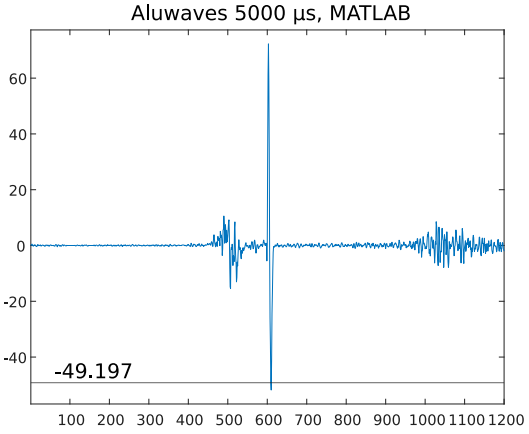
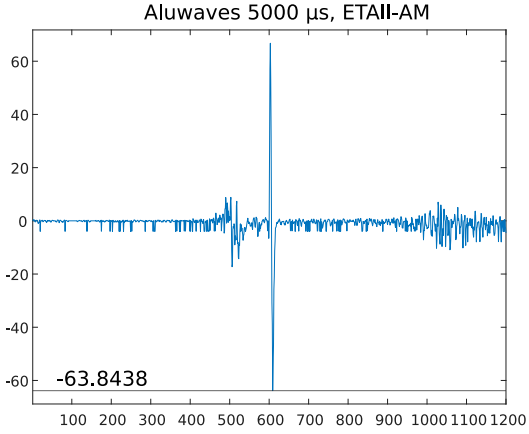


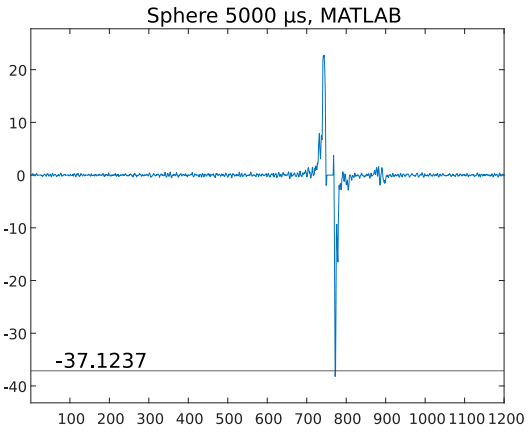
Figure 6.17: Derivative of a signal affected by a drooping effect at the peak



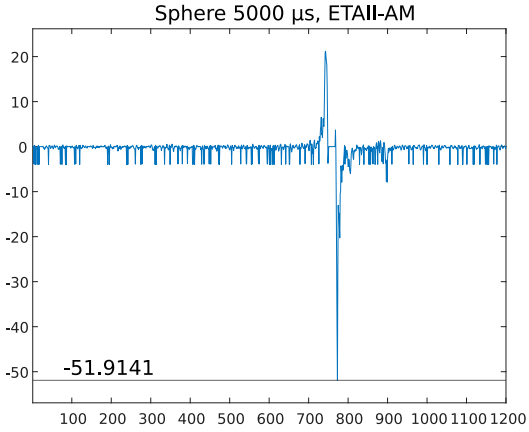
(a) MATLAB implementation



(b) Approximate circuit



(c) MATLAB implementation



(d) Approximate circuit

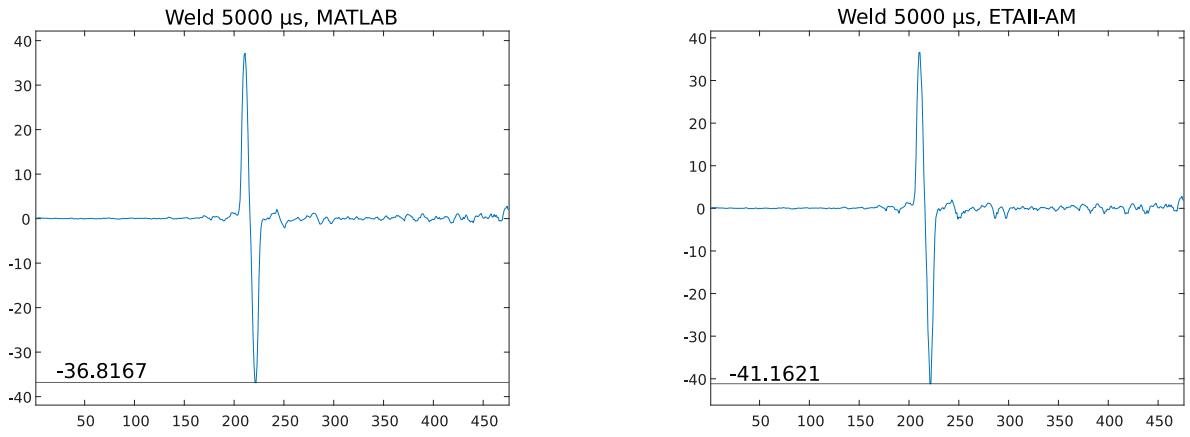
Figure 6.18: Results of one column provided by the exact differentiation (no smoothing) filter from MATLAB vs. the ETAII-AM-ER approximate circuit implemented using VHDL

The ETAII-AM-ER DF-NS exhibits aspects from both the ETAII and AM-ER filters. Both the drooping effect and a significantly lower minimum are visible in Figure 6.18. From Table 6.32 it can be seen that, as expected, the accuracy decreases compared to the partially approximated filters from Tables 6.19 and 6.8.

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	13.286	0.2136	0.1130	0.0868	-0.0868
aluwaves 500 μs	17.312	0.4207	0.2428	0.2029	-0.2029
aluwaves 1000 μs	14.522	0.6557	0.3880	0.3261	-0.3261
aluwaves 2000 μs	15.440	0.8000	0.5052	0.4516	-0.4516
aluwaves 5000 μs	14.647	0.9938	0.6798	0.6745	-0.6745
plane 10 μs	10.606	0.1274	0.0989	0.1757	-0.1757
plane 20 μs	14.910	0.1524	0.1173	0.2359	-0.2359
plane 40 μs	14.473	0.2074	0.1509	0.2997	-0.2997
plane 80 μs	15.981	0.3215	0.2174	0.3754	-0.3754
plane 160 μs	14.142	0.5468	0.3503	0.4807	-0.4807
plane 320 μs	13.395	0.8919	0.5912	0.6985	-0.6984
plane 640 μs	14.781	0.9967	0.7343	0.8750	-0.8750
sphere 100 μs	12.848	0.1698	0.0767	0.0520	-0.0520
sphere 200 μs	16.940	0.2067	0.0955	0.0689	-0.0689
sphere 500 μs	15.201	0.2877	0.1403	0.1034	-0.1034
sphere 1000 μs	14.600	0.3389	0.1740	0.1343	-0.1342
sphere 2000 μs	14.711	0.4608	0.2515	0.1972	-0.1972
sphere 5000 μs	14.790	0.6657	0.3862	0.3094	-0.3094
weld 10 μs	12.974	0.1468	0.0800	0.0767	-0.0767
weld 100 μs	14.448	0.2544	0.1423	0.1335	-0.1335
weld 500 μs	15.435	0.5199	0.3065	0.2928	-0.2928
weld 1000 μs 1	15.179	0.6401	0.4031	0.3999	-0.3999
weld 1000 μs 2	16.814	0.6078	0.3524	0.2876	-0.2876
weld 2000 μs	15.715	0.6913	0.4695	0.4993	-0.4993
weld 5000 μs	14.762	0.7571	0.5622	0.6540	-0.6540
white plane 20 μs	8.567	0.2946	0.1817	0.2238	-0.2238
white plane 50 μs	13.825	0.4152	0.2671	0.3777	-0.3777
white plane 100 μs	14.417	0.5637	0.3673	0.5486	-0.5486
white plane 200 μs	15.825	0.7525	0.5156	0.7647	-0.7647
white plane 500 μs	15.729	0.9831	0.7429	1.0495	-1.0495
white plane 1000 μs	15.423	0.9991	0.8240	1.2196	-1.2196
white plane 2000 μs	16.603	0.9998	0.8831	1.3515	-1.3515
Average	14.635	0.5338	0.3566	0.4259	-0.4258
Max	17.312	0.9998	0.8831	1.3515	-1.3515

Table 6.32: Error evaluation of the ETAII-AM-ER differentiation filter (no smoothing)

As the ETAII units in the DF-2ND are exact, this version is equivalent to the AM-ER DF-2ND and is thus pretty accurate as seen in Tables 6.33 and 6.34. Figure 6.19 hardly shows any visible differences.



(a) MATLAB implementation

(b) Approximate circuit

Figure 6.19: Results of one column provided by the exact 2nd order differentiation filter from MATLAB vs. the ETAII-AM-ER approximate circuit implemented using VHDL

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	4.3082	0.2119	0.0455	0.0094	-0.0094
aluwaves 500 μs	4.6467	0.4182	0.0966	0.0188	-0.0188
aluwaves 1000 μs	4.5301	0.6515	0.1513	0.0275	-0.0275
aluwaves 2000 μs	4.6163	0.7947	0.2000	0.0388	-0.0388
aluwaves 5000 μs	4.3454	0.9910	0.2783	0.0633	-0.0633
plane 10 μs	2.7801	0.1271	0.0751	0.0473	-0.0473
plane 20 μs	3.9801	0.1521	0.0892	0.0689	-0.0689
plane 40 μs	4.3454	0.2068	0.1077	0.0896	-0.0896
plane 80 μs	4.4915	0.3199	0.1383	0.1110	-0.1110
plane 160 μs	4.6173	0.5428	0.1981	0.1242	-0.1242
plane 320 μs	4.4996	0.8871	0.3017	0.1444	-0.1444
plane 640 μs	4.3454	0.9900	0.3939	0.1707	-0.1707
sphere 100 μs	3.9507	0.1684	0.0320	0.0074	-0.0074
sphere 200 μs	4.3091	0.2048	0.0397	0.0106	-0.0106
sphere 500 μs	4.3444	0.2845	0.0573	0.0144	-0.0144
sphere 1000 μs	4.3091	0.3343	0.0703	0.0171	-0.0171
sphere 2000 μs	4.3082	0.4535	0.0981	0.0209	-0.0209
sphere 5000 μs	4.2737	0.6567	0.1467	0.0272	-0.0272
weld 10 μs	4.6164	0.1457	0.0396	0.0152	-0.0152
weld 100 μs	4.5301	0.2521	0.0655	0.0221	-0.0221
weld 500 μs	4.3454	0.5154	0.1303	0.0379	-0.0379
weld 1000 μs 1	4.3454	0.6357	0.1694	0.0480	-0.0480
weld 1000 μs 2	4.6173	0.6021	0.1398	0.0306	-0.0306
weld 2000 μs	4.3454	0.6892	0.2007	0.0604	-0.0604
weld 5000 μs	4.3454	0.7490	0.2507	0.0848	-0.0847
white plane 20 μs	2.7497	0.2931	0.1055	0.0566	-0.0566
white plane 50 μs	3.9507	0.4127	0.1617	0.0813	-0.0813
white plane 100 μs	4.3454	0.5599	0.2172	0.1265	-0.1265
white plane 200 μs	4.3454	0.7484	0.2915	0.1726	-0.1726
white plane 500 μs	4.3454	0.9799	0.4157	0.2240	-0.2240
white plane 1000 μs	4.3454	0.9874	0.4710	0.2548	-0.2547
white plane 2000 μs	4.0320	0.9715	0.5306	0.2976	-0.2976

Table 6.33: Error evaluation of the ETAII-AM-ER 2nd order differentiation filter (no smoothing)

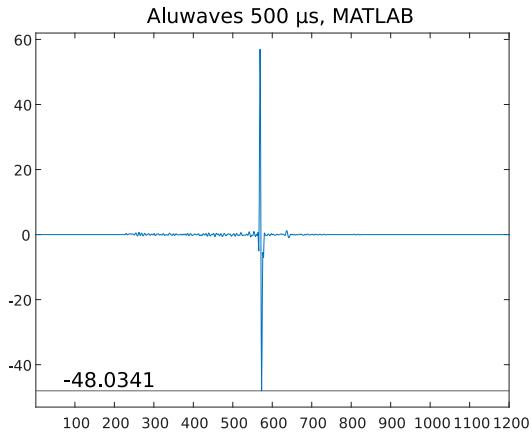
	ED	ER (SW)	ER (HW)	MED	Mean err
Average	4.2582	0.5293	0.1784	0.0789	-0.0789
Max	4.6467	0.9910	0.5306	0.2976	-0.2978

Table 6.34: Error evaluation of the ETAII-AM-ER 2nd order differentiation filter (no smoothing)

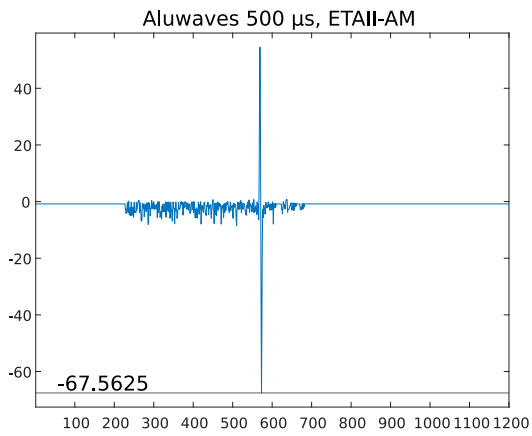
The same considerations made for the ETAII-AM-ER DF-NS apply to the DF-S. The only difference lies in its accuracy. While the graphs shown in Figure 6.20 present similar results to the DF-NS, Table 6.35 clearly highlights a deterioration of all the metrics. This is a direct consequence of the connection between two approximate filters.

Parameter	ED	ER (SW)	ER (HW)	MED	Mean err
aluwaves 100 μs	15.715	1.0000	1.0000	0.9659	-0.9638
aluwaves 500 μs	19.528	1.0000	0.9999	1.2540	-1.2496
aluwaves 1000 μs	20.324	1.0000	0.9997	1.5154	-1.5036
aluwaves 2000 μs	22.296	1.0000	0.9996	1.7814	-1.7396
aluwaves 5000 μs	23.789	1.0000	0.9994	2.3900	-2.2818
plane 10 μs	16.630	1.0000	1.0000	0.9949	-0.9809
plane 20 μs	16.973	1.0000	1.0000	1.0521	-1.0357
plane 40 μs	19.942	1.0000	1.0000	1.1239	-1.1077
plane 80 μs	21.780	1.0000	1.0000	1.1767	-1.1577
plane 160 μs	18.260	1.0000	0.9999	1.2948	-1.2726
plane 320 μs	20.037	1.0000	0.9997	1.8142	-1.7915
plane 640 μs	17.668	1.0000	0.9993	2.4070	-2.3661
sphere 100 μs	16.582	1.0000	1.0000	0.8613	-0.8593
sphere 200 μs	21.473	1.0000	1.0000	0.8773	-0.8746
sphere 500 μs	19.039	1.0000	1.0000	0.9056	-0.9019
sphere 1000 μs	20.692	1.0000	1.0000	0.9497	-0.9447
sphere 2000 μs	18.809	1.0000	0.9999	1.0372	-1.0292
sphere 5000 μs	20.621	1.0000	0.9998	1.3373	-1.3215
weld 10 μs	18.323	1.0000	1.0000	0.8987	-0.8955
weld 100 μs	19.529	1.0000	1.0000	0.9646	-0.9600
weld 500 μs	18.128	1.0000	0.9998	1.4253	-1.4153
weld 1000 μs 1	21.273	1.0000	0.9997	1.6484	-1.6304
weld 1000 μs 2	18.125	1.0000	0.9999	1.1806	-1.1741
weld 2000 μs	21.875	1.0000	0.9996	1.7720	-1.7367
weld 5000 μs	18.695	1.0000	0.9996	1.9335	-1.8175
white plane 20 μs	12.610	1.0000	1.0000	0.9965	-0.9935
white plane 50 μs	16.079	1.0000	1.0000	1.1484	-1.1267
white plane 100 μs	21.142	1.0000	1.0000	1.3152	-1.2871
white plane 200 μs	20.725	1.0000	0.9998	1.5406	-1.5076
white plane 500 μs	19.347	1.0000	0.9997	2.0940	-2.0471
white plane 1000 μs	20.555	1.0000	0.9994	2.3512	-2.2751
white plane 2000 μs	20.949	1.0000	0.9994	2.3327	-2.2086
Average	19.297	1.0000	0.9998	1.4169	-1.3893
Max	23.789	1.0000	1.0000	2.4070	-2.3661

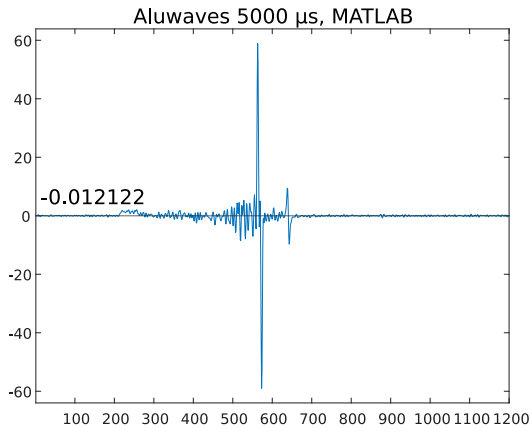
Table 6.35: Error evaluation of the ETAII-AM-ER differentiation filter (after smoothing)



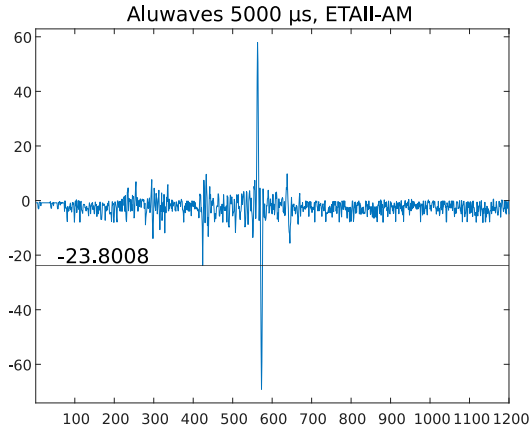
(a) MATLAB implementation



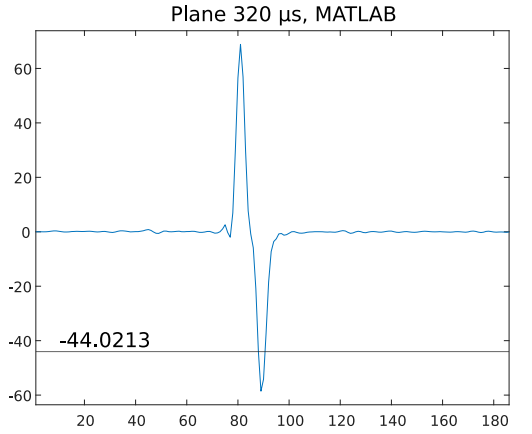
(b) Approximate circuit



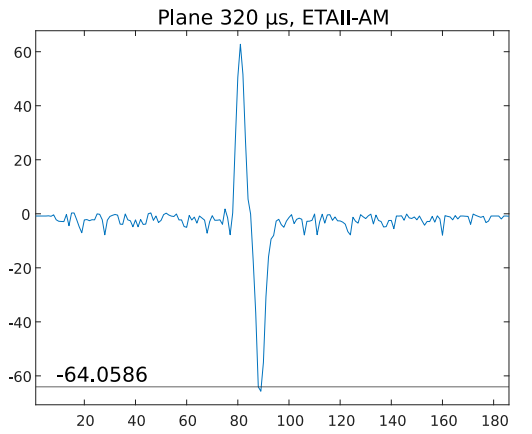
(c) MATLAB implementation



(d) Approximate circuit



(e) MATLAB implementation1



(f) Approximate circuit

Figure 6.20: Results of one column provided by the exact differentiation (after smoothing) filter from MATLAB vs. the ETAII-AM-ER approximate circuit implemented using VHDL

In Figures 6.21, 6.22, and 6.23, the results highlighted in the previous tables are shown in the form of bar graphs. In this way, it is easier to compare the error performance of the various filters already discussed and spot patterns efficiently.

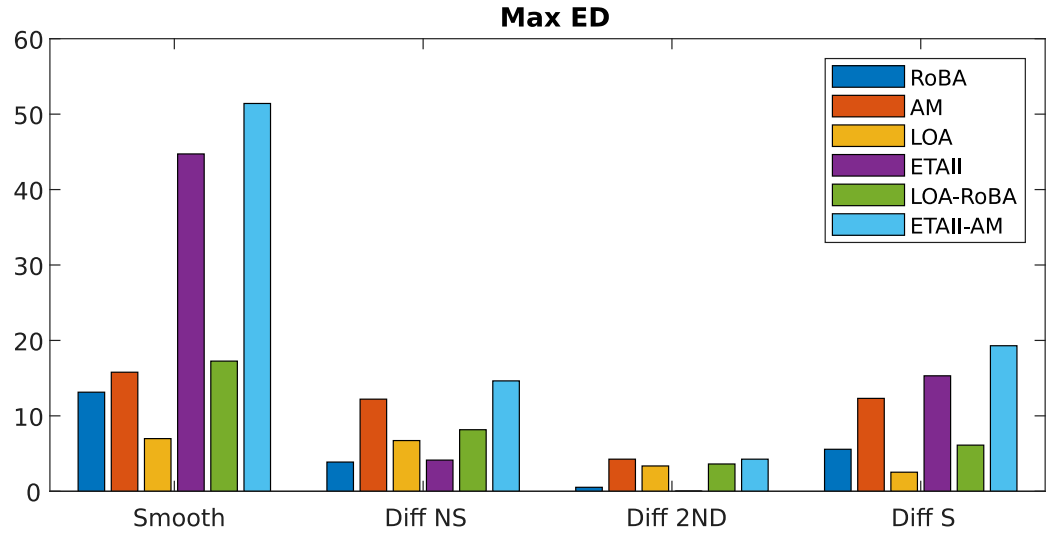


Figure 6.21: Bar chart summarizing the results shown in the tables: maximum ED

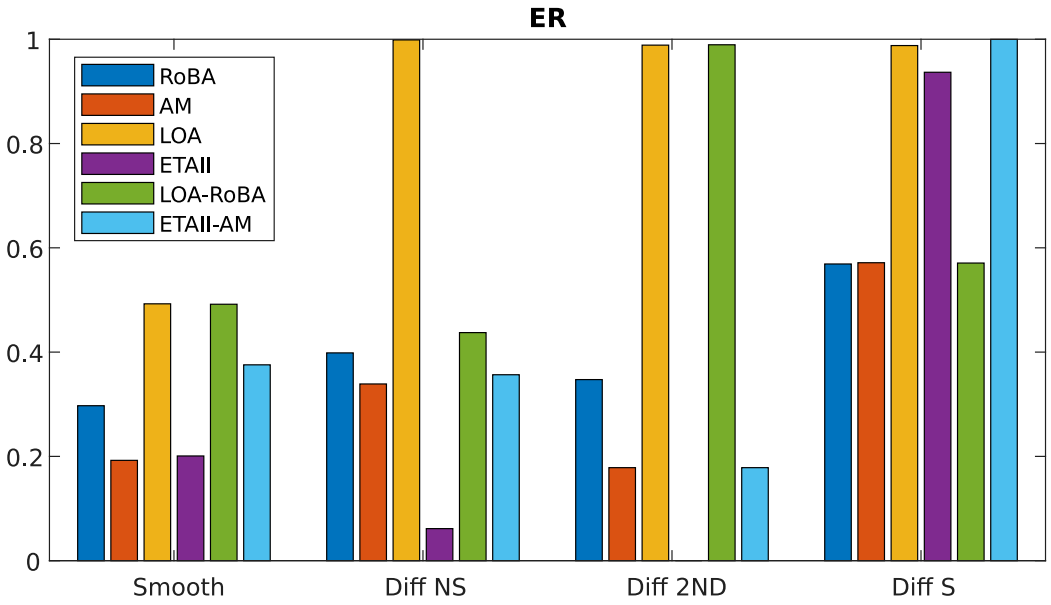


Figure 6.22: Bar chart summarizing the results shown in the tables: ER

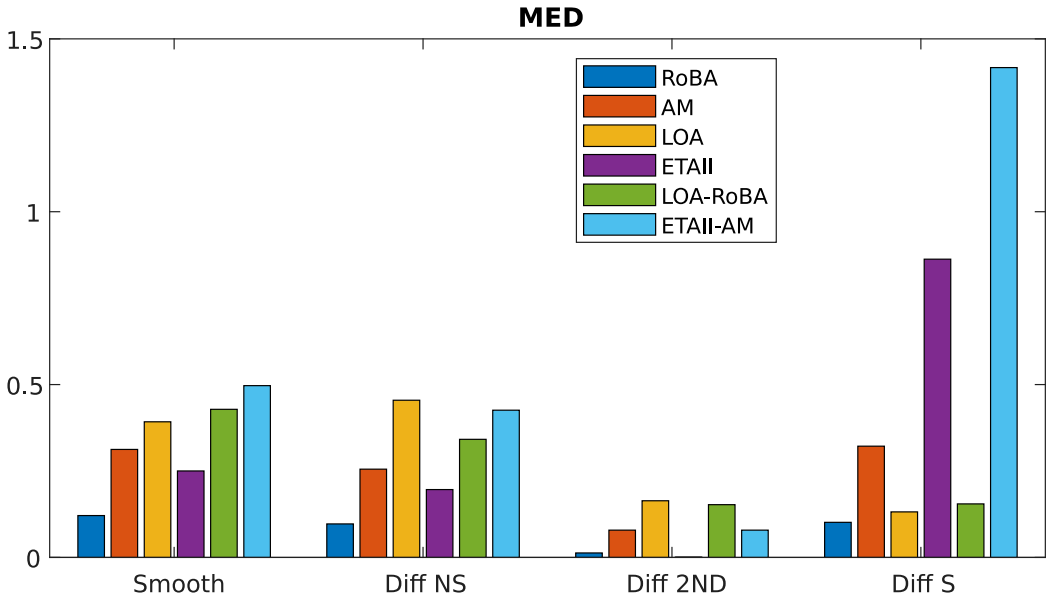


Figure 6.23: Bar chart summarizing the results shown in the tables: MED

6.1.2 Zero-crossing

The final block in the system's pipeline is the zero-crossing detection circuit. Four variants have been tested for accuracy assessment. The two main versions are: the zero-crossing with no average (ZC-NA), which is much simpler, and the zero-crossing with average (ZC-A) that features a 26-bit adder. The number of implementations is doubled by considering both exact and approximate adaptations of the two circuits. The approximate versions make use of the AXS4-based divider. The mean is obtained by means of a 26-bit LOA. Given the importance of a reliable computation of the zero position, an ETAII version of this adder was not created. The reasons behind the choice of the LOA is its higher approximation granularity: accuracy can be controlled bit by bit. This is not the case for the ETAII that works with clumps of bits. The 15-bit subtractor placed right before the divider is not approximated to avoid introducing too much error. Before synthesis, the zero-detection capability of the unit was tested. To do this, the design was first simulated on Modelsim. The testbench written for this task is able to send a result to the output file only when the signal that validates the circuit's output is active. This file is then read by a MATLAB script that converts the binary numbers back to decimal and plots each of them as a dot on a figure with the resolution of the tested image. If everything works smoothly, the final result should look like the original image. The most important thing is to ensure the continuity along the shape of the object. In other words, all the valid zeroes must be found. This consideration affects the choice of the best positive threshold (PT) for the circuit. A lower PT creates a very noisy image, wheres a higher one might mistake valid zeroes for noise. In this case, the first scenario, i.e. low PT, is the preferred one. Ideally, the number of zeroes in an image should be equal to or less than the horizontal resolution. In practice, they are a lot more due to noise. In the following sections, the zero-crossing circuits have been tested with two different PTs as a way to highlight the different behaviours. Additionally, comparisons between ZC-NA and ZC-A versions as well as between exact and approximate circuits are discussed.
Considering that the input data can come from two distinct kinds of approximate differentiation filters, i.e. LOA-RoBA and ETAII-AM-ER, and that three versions of these filters exist, the total number of simulations performed for a single image is 36. This also includes the three exact filters. While all 32 images are tested, only the most interesting ones are shown.

6.1.2.1 Comparison between exact and approximate designs

The following graphs in Figure 6.25 display the effects the approximations have on the final results. For this comparison, the PT was set to eight. This low number creates a more noisy image which is able to highlight the differences better. The fewer the dots, the better the design, as long as all the valid zeroes are correctly detected. The picture selected for this comparison is *Aluwaves* 5000μs. Ideally, with no noise present, this image should look like Figure 6.24. This noise-free image was created on MATLAB by locating the pixel with the maximum

intensity for each column. For this analysis, the images are generated by the ZC-NA. The ones created by the ZC-A exhibit similar profiles and are discussed at a later time. "Exact design" refers to the ZC-NA circuit that make use of accurate arithmetic blocks and receives its data from exact differentiation filters. The data coming from the approximate filters is instead analyzed by the detector mounting the approximate divider. Figure 6.25 shows the results.

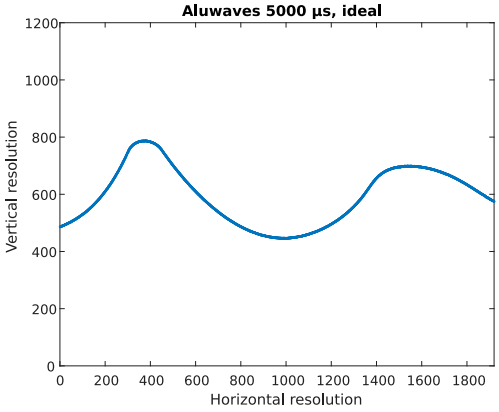
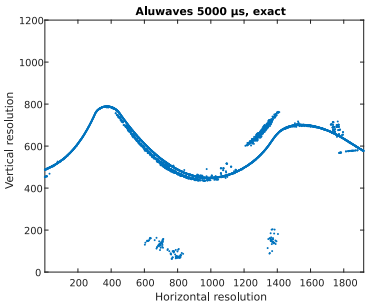
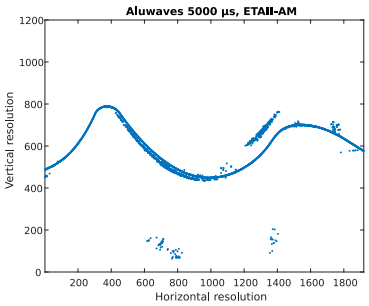


Figure 6.24: Output of zero-crossing detector if there were no noise

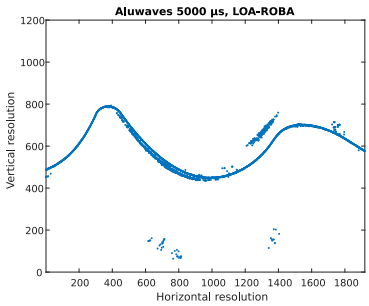
Not much difference is visible when comparing exact and approximate designs. This is expected since the region close the zero is normally not heavily influenced by the approximations as observed from several figures in Subsection 6.1.1. Furthermore, the surrounding noise exacerbated by the low accuracy of the inexact circuits does not usually surpass the PT. This is especially true for the ETAII-AM-ER filters that experience the drooping effect. Employing the approximate divider might shift the zero position. However, the small displacement is not visible from the figures. While the ETAII-AM-ER output image is similar to the one generated by the exact detector, the LOA-RoBA data produces a result affected by considerably less noise. This behaviour is observed in other images as well. By focusing on the columns rather than the rows, it can be seen that the DF-2ND is the least noisy when compared to the other two differentiation filters.
A quantitative analysis was performed as well. In particular, a MATLAB script was created that is able to process the output files generated by the zero-crossing detectors. Thus, the exact hardware version was compared to the approximate ones. The main problem with the extraction of this data is that the zeroes found by one circuit are not necessarily the same in number and position as the other one. This means that for a specific column of an image, the exact version might have no zeroes, whereas the approximate design might find one or more. The opposite is also true. To overcome this issue, the data for each column of the image is read by accessing an array that contains information on the column number. The amount of zeroes detected on that column depends on the two designs under test. For instance, if one circuit finds two zeroes and the other one returns three of them, the number considered is the smallest one of the two, i.e. two zeroes. The ED of all combinations of points (five in this example) are computed. The array containing the EDs for each pair of zeroes is sorted and the two smallest numbers are selected. This is done to avoid situation in which an exceedingly big ED is observed due to noise present only in one design. This method however does not always works. This is why sometimes large EDs are registered by the program. Most of the times these values can be discarded from the analysis as they are not representative of the system's accuracy. Thus, the contents of the ED array can be filtered to remove values above a certain threshold. Two different values are reported: the maximum ED among the samples whose ED is less than ten and the maximum ED of the samples whose ED is less than one. From this data, two MEDs are obtained. Another important metric is the ratio between the number of elements < 1 in the ED and the total number of elements. The higher this number, the better the design. This ratio is also reported for the < 10 case. For this task, two images have been tested: one with little to no noise and another with a lot of noise. These images are: Weld 1000 μs #2 (A.4g) and Aluwaves 5000 μs (A.1e).



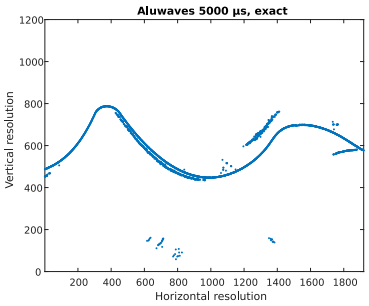
(a) Exact filter, no smoothing



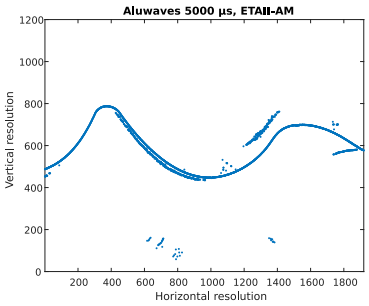
(b) ETAII-AM-ER filter, no smoothing



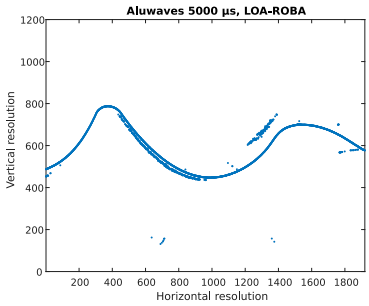
(c) LOA-RoBA filter, no smoothing



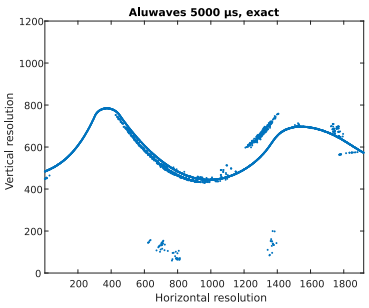
(d) Exact filter, second order



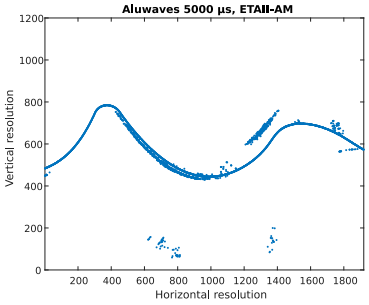
(e) ETAII-AM-ER filter, second order



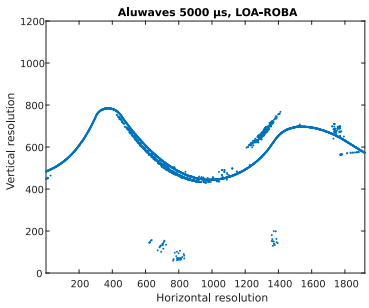
(f) LOA-RoBA filter, second order



(g) Exact filter, with smoothing



(h) ETAII-AM-ER filter, with smoothing



(i) LOA-RoBA filter, with smoothing

Figure 6.25: Comparison among exact and approximate systems

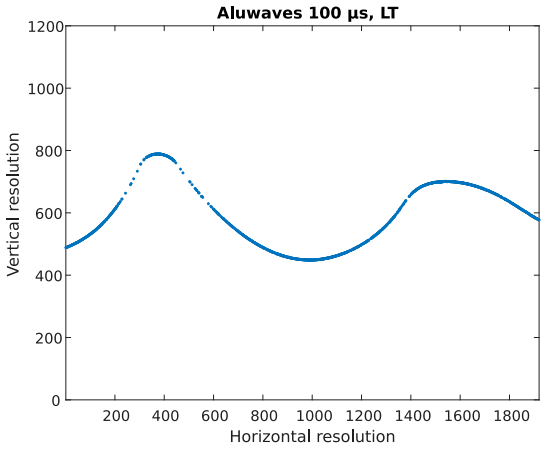
The results are shown in Table 6.36. In the table, a comparison is carried out between exact and approximate designs. The values depend on the PT and NT chosen. In this case 20 and -5 respectively. The zero-crossing system elaborates the data coming from the LOA-RoBA filter and the ETAII-AM-ER filter. The approximate detection circuit makes use of an inexact divider.

Metric	Alu LOA-RoBA	Alu ETAII-AM-ER	Weld LOA-RoBA	Weld ETAII-AM-ER
Maximum ED < 10	7.3624	7.5639	4.3214	4.3037
Maximum ED < 1	0.9955	0.9682	0.9588	0.9531
Ratio of EDs < 10	1.0000	1.0000	0.9918	0.9923
Ratio of EDs < 1	0.9866	0.9890	0.9852	0.9872
MED < 10	0.0985	0.0981	0.0892	0.0851
MED < 1	0.0688	0.0752	0.0766	0.0756

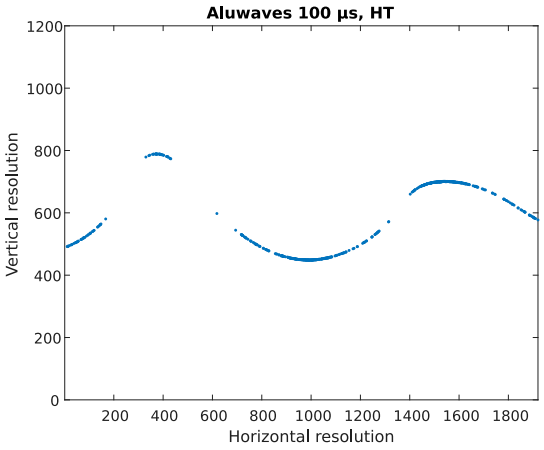
Table 6.36: Quantitative comparison between exact and approximate zero-crossing circuits

From the table, it can be seen that more than 95% of the EDs in the arrays is less than '1'. This is a very good result. Although the maximum ED is close to one, the MED is much lower.

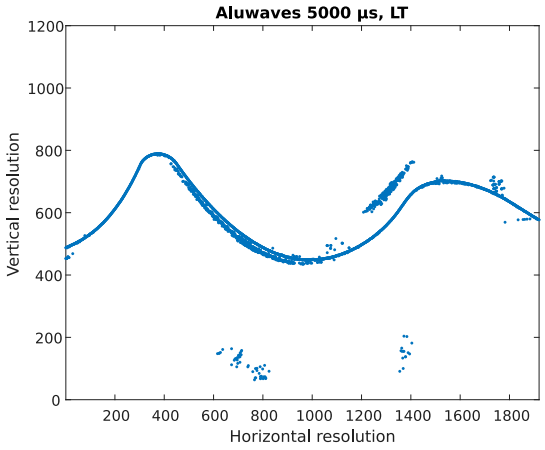
6.1.2.2 Comparison between low threshold and high threshold



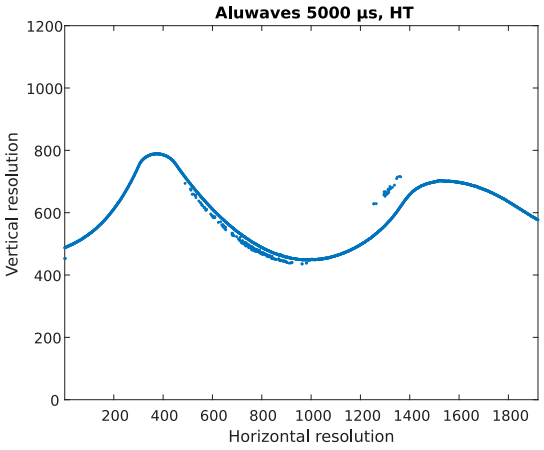
(a) Effect of PT = 8



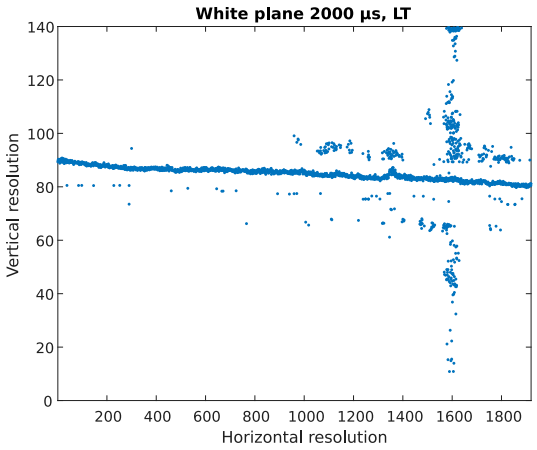
(b) Effect of PT = 20



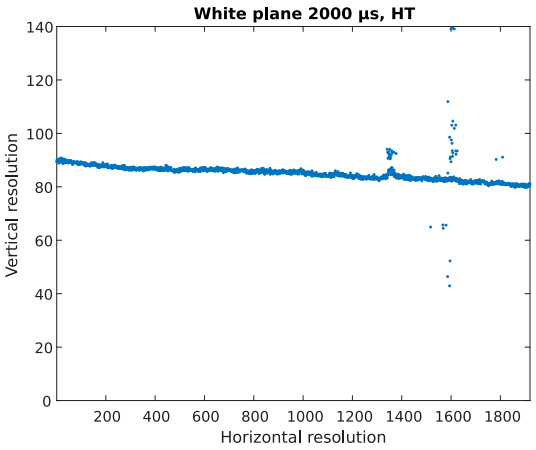
(c) Effect of PT = 8



(d) Effect of PT = 20



(e) Effect of PT = 8



(f) Effect of PT = 20

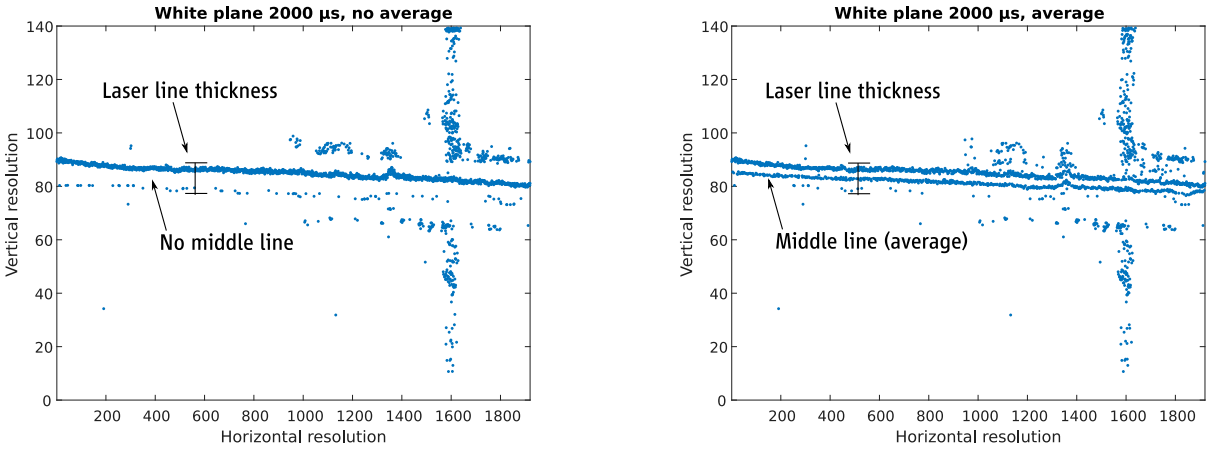
Figure 6.26: Comparison of the effects of low PT (left) and high PT (right) on the zero detection

This subsection is dedicated to the impact the PT has on the output of the zero-crossing detector. The NT found in the ZC-A, on the other hand, is fixed. The NT is normally a small negative number. Setting it to increasingly negative values entails that a lot of zeroes that are close together might get clumped into one by the average computation unit. In other words, since average calculation occurs more frequently, noise reduces. Similarly to the choice of PT, the NT can also contribute to noise filtering. However, with this last method, the

position of the zeroes might be exceedingly approximated due to the distance between the points that cross the x-axis. Thus, this subsection focuses on examining the effects of a PT variation. In the next subsection, it will be shown that the correct selection of the NT can be beneficial to images that absolutely require the average calculation.
Figure 6.26 shows three couples of images. The left column is the result of a low positive threshold (LT) equal to 8. For the column on the right, a higher positive threshold (HT) of 20 is employed. The results are tightly related to the exposure time. As expected, higher positive thresholds are ideal for noisy images resulting from a long exposure time. Both the *Aluwaves* 5000μs and the *White plane* show a large exposure time. It is interesting to see how much noise is removed just by increasing the PT. Further raising this value for this kind of images is possible and can eliminate even more invalid zeroes. A large PT is instead detrimental to *Aluwaves* 100μs, which is almost completely erased. In this case, a low PT is enough to perfectly capture the shape of the object depicted in this image.

6.1.2.3 Comparison between average and no average

The tests carried out on the ZC-NA and the ZC-A show interesting results. As expected, it was found that an average computation unit is useful when more than one zero is present near the valid one, i.e. when the laser line is thick. It follows that images with a short exposure time do not need this kind of feature. The difference between a ZC-A and a ZC-NA circuit is that for the former, the zeroes are not located at the edges of the laser line, but rather in the middle.
To better explain how the average calculation works, a column with two zeroes around the peak region is considered. This simple case is also the most common. The ZC-A detector processes the zeroes and returns only one of them, which is placed in the middle of the two original zeroes. This is guaranteed as long as the NT is not passed for the entire computation. However, no constraints are specified for the PT. Therefore, while the ZC-A circuit, whose output solely depends on the NT, always finds one zero, the ZC-NA might find either one or two. This is because the ZC-NA does not include a NT and its output relies on the only threshold available, which is the PT. In these circumstances, if the second and lower peak passes the PT, two zeroes are found. On the contrary, if the PT is not exceeded, only one zero is produced, which is erroneously placed at the first edge of the laser line encountered by the zero-crossing circuit. Either way, the location obtained by the ZC-NA is inaccurate. The ZC-A, however, only works well when the majority the columns of an image do not cross the NT. Figure 6.27 shows an example of the effect of the two circuits on the same picture.

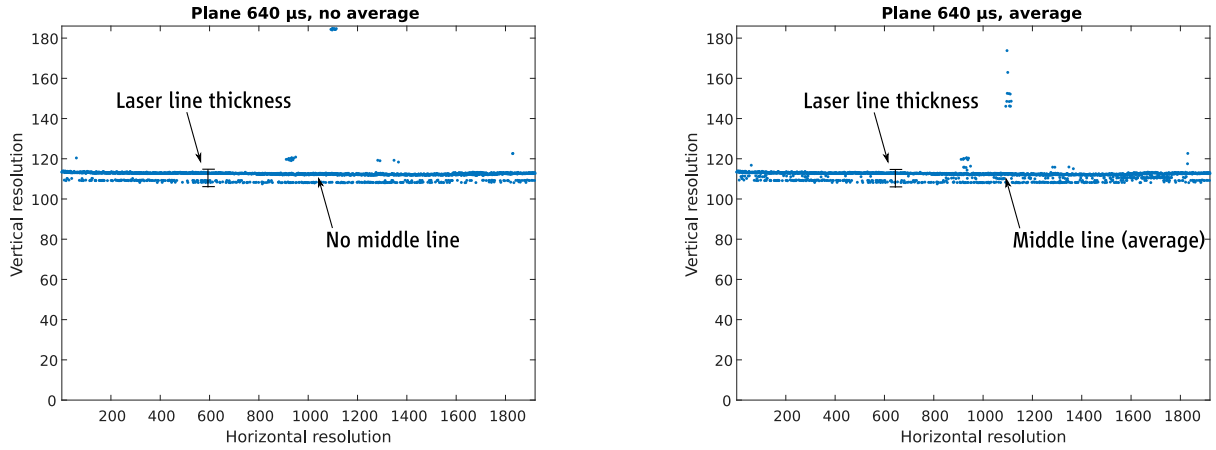


(a) White plane output, no average

(b) White plane output, with average

Figure 6.27: Effects of average computation on zero detection for a white plane image

Figure 6.27a displays the results of the ZC-NA circuit. Two rows of zeroes that match the edges of the laser line are visible. The first zero (top row) is always found as the PT is always passed the first time. The second zero is not always spotted and depends on the value of the PT. With a ZC-A circuit, on the other hand (Figure 6.27b), a row of zeroes starts forming in the middle. Given the wide laser line, this solution might provide a more accurate estimation. However, if NT is not small enough, this hardly happens. Figure 6.28 shows a different example.



(a) Plane output, no averagea

(b) Plane output, average

Figure 6.28: Effects of average computation on zero detection for a plane image

In Figure 6.28b, a line in the middle is present. However, it is erratic and not continuous. Hence, the zeroes in the middle cannot be used to identify the object scanned. This happens because the NT is passed in most cases and, as a consequence, average computation cannot start. Moreover, when dealing with images that are not straight lines, the average computation is rarely consistent and only some points actually benefit from it. In the worst scenario, this displacement can prove detrimental as it may disrupt the continuity of the main laser line. The method does not produce satisfactory results because it is too reliant on the NT. To fix the issue, the NT can be lowered. This correction actually improves the quality of the output image shown in Figure 6.29, as the middle line is now the only continuous one. In conclusion, given the complexity of the ZC-A circuit and its sensitivity to the NT, it should be considered only when high precision is required.

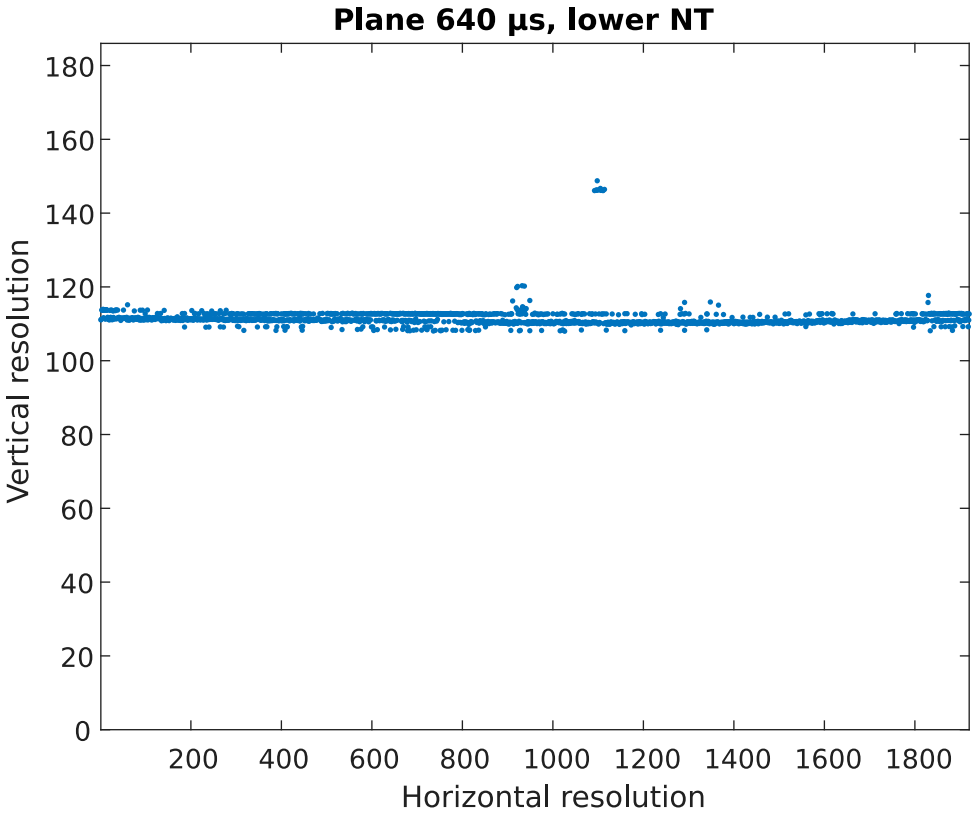


Figure 6.29: Average computation with NT = -10

6.2 Post-synthesis analysis and simulation

In this section, power, FPGA resources and delay are estimated for each unit starting from the building blocks to the bigger ones. Timing constraints are specified in an synopsys design constraints (SDC) file created for each circuit. In this project, only the clock period is specified in the file. To find the maximum frequency, a 1 ns period is used. The reason is that no design can reach this performance as the FPGA frequency is limited at around 700 MHz. The timing analysis that ensues after fitting verifies all timing requirements to make sure that no violation occurred. It provides the maximum frequency in MHz for different voltage-temperature corners (multicorner analysis). It lists the slowest paths and also verifies the compliance of the setup and hold times. The important figure for this project is the maximum frequency. As paths are calculated from a sequential element to the next one, the timing analysis works only for sequential circuits. Thus, when testing combinational circuits such as multipliers and adders, all input and output ports are equipped with registers.
The fitter also produces the information on the area of each design, which is extrapolated from the resource section of the final report. However, for an FPGA, this parameter is not given in μm^2 as it would be common practise for an ASIC design. Instead, the main metrics related to area are logic array blocks (LABs), adaptive logic modules (ALMs), and adaptive look-up tables (ALUTs) [87]. An ALUT is a block that implements the combinational logic present in the design and it can have up to seven inputs. The amount of ALUTs needed by the design and the number of inputs required by each of them can be found in the resource section of the fitter report. However, only the total number of ALUTs is reported in the project. In particular, the ones used for combinational logic. It is important to note that this parameter simply counts the number of logic functions required by the design and it is estimated even before synthesis. Thus, this value is not indicative of the actual occupied area. An ALM is a bigger block made of two ALUTs plus registers, full adders and multiplexers. It represents the basic building block in an FPGA. The number of ALMs used in the final placement over the ALMs available defines the logic utilization of a design. As ALMs can have different roles in the circuit, the resource report splits them into different categories. In particular, ALMs used for LUT logic, registers or memory. In this chapter, only the total number of ALMs is stated. The types of logic functions an ALM can realize are listed here: [88]. An array of ALMs is called LAB. A Cyclone V LAB can fit ten ALMs. In the resource report, a LAB might be fully exploited, when all of its ALMs are in use, or it can be exploited partially. LABs are surrounded by a matrix of interconnections that are also programmable through switches. In a standard island architecture, these wires occupy most chip area and represent the main contribution in power consumption mainly due to leakage current from the switches [89, 90, 91]. Another interesting parameter in the report is the number of registers, which are realized by ALMs as well. Finally, the difficulty in packing the design is a metric that depends on the packing algorithm required to fit a project. The harder it is to fit the design, the more aggressive is the algorithm chosen. As 3207 LABs are available on the selected FPGA, a “low” packing difficulty is expected for every unit in the project. As a consequence, this parameter is not reported in this thesis.
The last metric is power, which is estimated by the power analyzer provided by Quartus. In this project, only thermal power is considered and it represents the power dissipated through heat. It can be divided into three main contributions: dynamic power, static power, and I/O power [92]. Dynamic power is related to the internal switching, i.e. charging and discharging of gate capacitances. The core dynamic power takes into account both the logic elements and interconnections. This also includes the clock tree networks. A simple model is sufficient to compute dynamic power, provided that signal activities used for the power simulation are accurate enough. The static power, also known as standby power, stems from leakage currents which flow through transistors that are turned off. I/O power is the dynamic power dissipated externally due to switching of I/O pins. As the real FPGA is not actually programmed, no pins are assigned. Failure to assign the pins might have an effect on the final accuracy of the estimation for both power and delay. This is because of the length of the interconnections and the distance between the used pins that can generate parasitic capacitances. However, this error can be neglected.
Synthesis and fitting provide information on area and delay. However, to obtain power consumption estimates, gate level simulation is required and it is performed on the netlist generated during synthesis. Both timing and power analysis require the paths to be constrained, hence, registers have been placed on all input and output ports. The exact designs are always grouped with their approximated versions. For instance, an accurate 9-bit adder is always followed by all 9-bit inexact adders. In the next subsection, results are reported and discussed for arithmetic circuits. All the considerations clarified at the beginning of the chapter also apply here.

6.2.1 Arithmetic circuits

Table 6.37 shows the results for adders and subtractors. Regarding the two slow corners, the frequency that appears in the tables is the lowest of the two. To obtain these figures, no SDC file was provided to the compiler. The final Quartus report applies a restriction to the maximum frequency limiting it at 717 MHz. Thus, it is assumed that this is the highest frequency achievable with this FPGA. Because of this, extremely fast designs have their speed limited when synthesized. The percentage change highlights the difference between the approximate circuit and the respective exact implementation. Ideally, all inexact designs should be faster and smaller.
The logic utilization is given in terms of ALUTs and ALMs. This last column contains two quantities. The first one represents the number of ALMs actually used. The amount in brackets symbolizes the minimum number of ALMs needed. This value can be achieved by the compiler when performing area optimization.

Design	Maximum Frequency (MHz)	ALUTs	ALMs	Used for
9-bit BKA	373.00	17	15 (12)	Smoothing (exact)
9-bit LOA	697.35 (+87 %)	11	14 (10)	Smoothing
9-bit ETAII	424.09 (+14 %)	16	15 (12)	Smoothing
17-bit BKA (subtractor)	285.71	40	31 (25)	Smoothing (exact)
17-bit LOA (subtractor)	408.66 (+43 %)	27	26 (21)	Smoothing
17-bit ETAII (subtractor)	430.29 (+51 %)	42	34 (29)	Smoothing
Wallace tree w/ 16-bit BKA	226.5	75	57 (48)	Smoothing (exact)
Wallace tree w/ 16-bit LOA	272.93 (+20 %)	63	48 (41)	Smoothing
Wallace tree w/ 16-bit ETAII	273.00 (+20 %)	81	55 (48)	Smoothing
9-bit KSA (subtractor)	412.71	22	17 (13)	Not used
9-bit BKA (subtractor)	428.27	17	15 (13)	Diff., no smoothing (exact)
9-bit LOA (subtractor)	684.46 (+60 %)	11	14 (10)	Diff., no smoothing
9-bit ETAII (subtractor)	444.05 (+4 %)	16	15 (12)	Diff., no smoothing
Wallace tree w/ 15-bit BKA	233.7	75	50 (47)	Diff. (exact)
Wallace tree w/ 15-bit LOA	261.78 (+12 %)	63	48 (43)	Diff.
Wallace tree w/ 15-bit ETAII	243.78 (+4 %)	88	58 (53)	Diff.
Wallace tree w/ 11-bit BKA	207.51	76	55 (48)	Diff. 2nd order (exact)
Wallace tree w/ 11-bit LOA	300.48 (+45 %)	62	48 (42)	Diff. 2nd order
Wallace tree w/ 11-bit ETAII	230.04 (+11 %)	74	60 (49)	Diff. 2nd order
17-bit BKA (subtractor)	319.08	40	30 (25)	Diff., after smoothing (exact)
17-bit LOA (subtractor)	370.92 (+16 %)	25	27 (20)	Diff., after smoothing
17-bit ETAII (subtractor)	413.91 (+30 %)	42	31 (28)	Diff., after smoothing

Table 6.37: Maximum frequency and area for adders and subtractors

From Table 6.37, it is apparent that all the approximate circuits perform better than the exact ones in terms of frequency. The most impressive improvement are exhibited by the adders based on the LOA algorithm. These units also require less area and can be more accurate than ETAII-based architectures. A 9-bit KSA was also tested. Its results are similar to the BKA. Naturally, in absolute terms, the fastest designs are the ones with the smallest bit-width.
Figures 6.30 and 6.31 transform the results from Table 6.37 into bar charts for better comparison. For each version of the adders and subtractors in the filters, the exact implementation and the two approximate circuits are examined side by side.

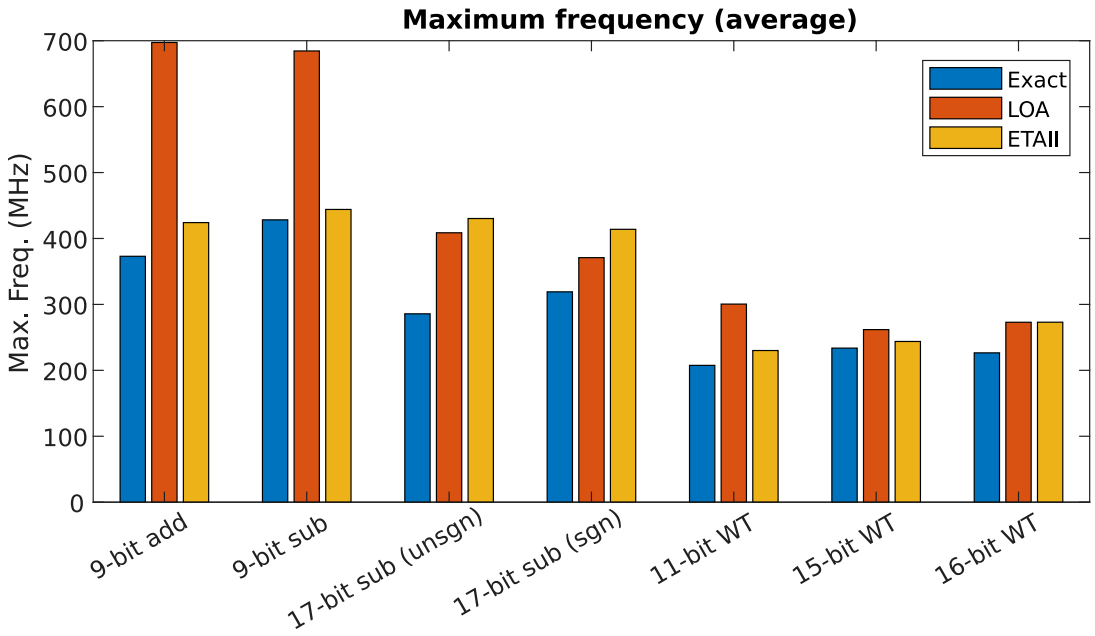


Figure 6.30: Bar chart summarizing the results shown in the tables: adders' maximum frequency

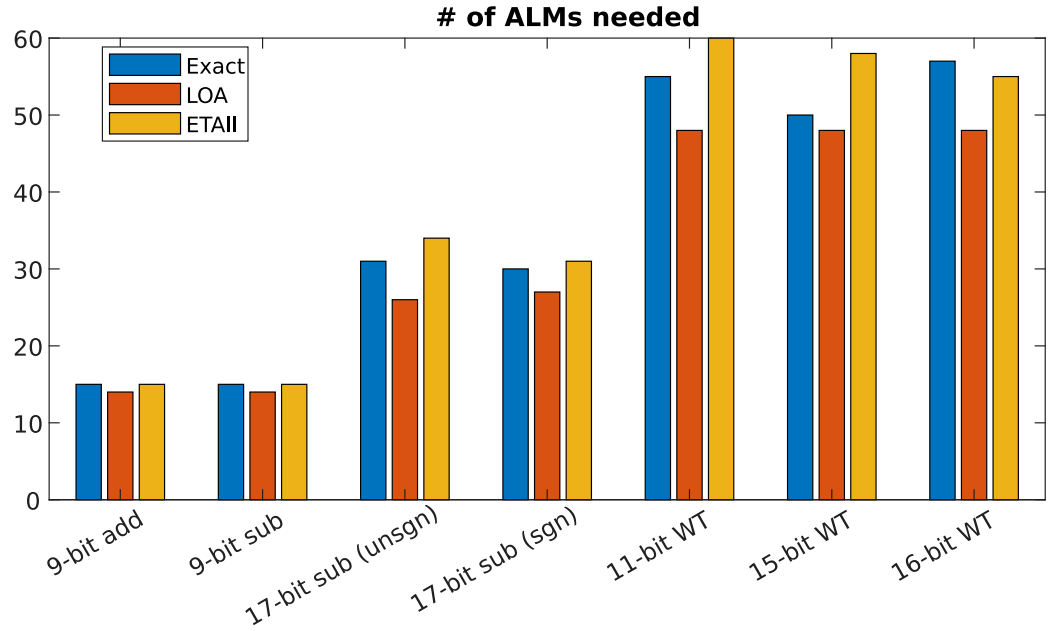


Figure 6.31: Bar chart summarizing the results shown in the tables: adders' ALMs

The multipliers' results are reported in Table 4.20. The parallelism contained in their names refers to the input length rather than the output one, which is always 15.

Design	Maximum Frequency (MHz)	ALUTs	ALMs	Used for
9-bit WTM 18	437.25	17	13 (12)	Smoothing (exact)
9-bit RoBA 18	224.37 (-49 %)	41	32 (31)	Smoothing
9-bit AM-ER 18	771.60 (+76 %)	11	12 (9)	Smoothing
9-bit WTM 66	447.83	15	15 (12)	Smoothing (exact)
9-bit RoBA 66	222.37 (-50 %)	46	29 (29)	Smoothing
9-bit AM-ER 66	797.45 (+78 %)	6	13 (8)	Smoothing
9-bit WTM 36	462.96	17	14 (12)	Smoothing (exact)
9-bit RoBA 36	230.68 (-50 %)	41	32 (30)	Smoothing
9-bit AM-ER 36	707.71 (+53 %)	11	12 (9)	Smoothing
9-bit WTM 161	350.26	26	19 (17)	Smoothing (exact)
9-bit RoBA 161	201.29 (-42 %)	54	35 (34)	Smoothing
9-bit AM-ER 161	659.63 (+88 %)	14	14 (11)	Smoothing
9-bit WTM 214	309.02	31	26 (25)	Smoothing (exact)
9-bit RoBA 214	260.21 (-16 %)	46	35 (34)	Smoothing
9-bit AM-ER 214	477.55 (+55 %)	21	15 (13)	Smoothing
9-bit BWM 74	291.55	34	28 (24)	Diff. (exact)
9-bit RoBA 74	166.58 (-43 %)	59	38 (37)	Diff.
9-bit AM-ER 74	583.43 (+100 %)	18	15 (13)	Diff.
9-bit BWM 122	300.66	34	26 (25)	Diff. (exact)
9-bit RoBA 122	226.91 (-25 %)	55	38 (36)	Diff.
9-bit AM-ER 122	405.19 (+35 %)	26	19 (18)	Diff.
9-bit BWM 166	296.47	35	27 (26)	Diff. (exact)
9-bit RoBA 166	184.16 (-38 %)	62	40 (39)	Diff.
9-bit AM-ER 166	415.45 (+40 %)	26	19 (17)	Diff.
9-bit BWM 109	297.62	37	29 (28)	Diff. (exact)
9-bit RoBA 109	225.17 (-24 %)	58	40 (39)	Diff.
9-bit AM-ER 109	385.80 (+30 %)	29	21 (18)	Diff.
13-bit BWM 546	193.42	70	51 (50)	Diff. 2nd order (exact)
13-bit RoBA 546	162.65 (-16 %)	82	55 (51)	Diff. 2nd order
13-bit AM-ER 546	575.71 (+198 %)	22	19 (16)	Diff. 2nd order
17-bit BWM 74	213.63	71	54 (53)	Diff. after smoothing (exact)
17-bit RoBA 74	171.97 (-20 %)	70	45 (45)	Diff. after smoothing
17-bit AM-ER 74	503.02 (+135 %)	24	19 (17)	Diff. after smoothing
17-bit BWM 122	228.15	74	57 (55)	Diff. after smoothing (exact)
17-bit RoBA 122	170.18 (-25 %)	73	50 (46)	Diff. after smoothing
17-bit AM-ER 122	411.18 (+80 %)	32	25 (25)	Diff. after smoothing
17-bit BWM 166	195.69	74	62 (59)	Diff. after smoothing (exact)
17-bit RoBA 166	166.17 (-15 %)	76	54 (49)	Diff. after smoothing
17-bit AM-ER 166	445.83 (+128 %)	31	24 (23)	Diff. after smoothing
17-bit BWM 109	188.5	80	64 (62)	Diff. after smoothing (exact)
17-bit RoBA 109	164.64 (-13 %)	80	54 (49)	Diff. after smoothing
17-bit AM-ER 109	371.33 (+97 %)	39	25 (23)	Diff. after smoothing

Table 6.38: Maximum frequency and area for multipliers

The numbers in Table 6.38 confirm that the AM-ER is superior. With improvements up to 200%, its speed and resource usage are unrivalled. On the other hand, the RoBA multiplier performs way worse than the exact WTM and it is also bulkier. However, when the bit-width increases, the BWM grows faster in complexity than the RoBA. This explains why the resources needed by a BWM surpass those reserved for a RoBA multiplier of the same size. In this case, the fastest architectures are found in the AM-ER multipliers employed by the smoothing filter. As explained earlier, the optimizations made on these circuits are able to reduce their

complexity to that of an adder. To match this speed, the circuits could also be implemented by using LOA adders. However, their maximum frequency would be nearly identical.
Bar charts of the values from Table 6.38 are also created. In this case, the number of types of multipliers is smaller than that of the adders. Given the number of coefficients, an average was computed for each category. Thus, for instance, to obtain the graph for the signed 9-bit BWM used in the differentiation filter, the data from the four coefficients is averaged. They are shown in Figures 6.32 and 6.33.

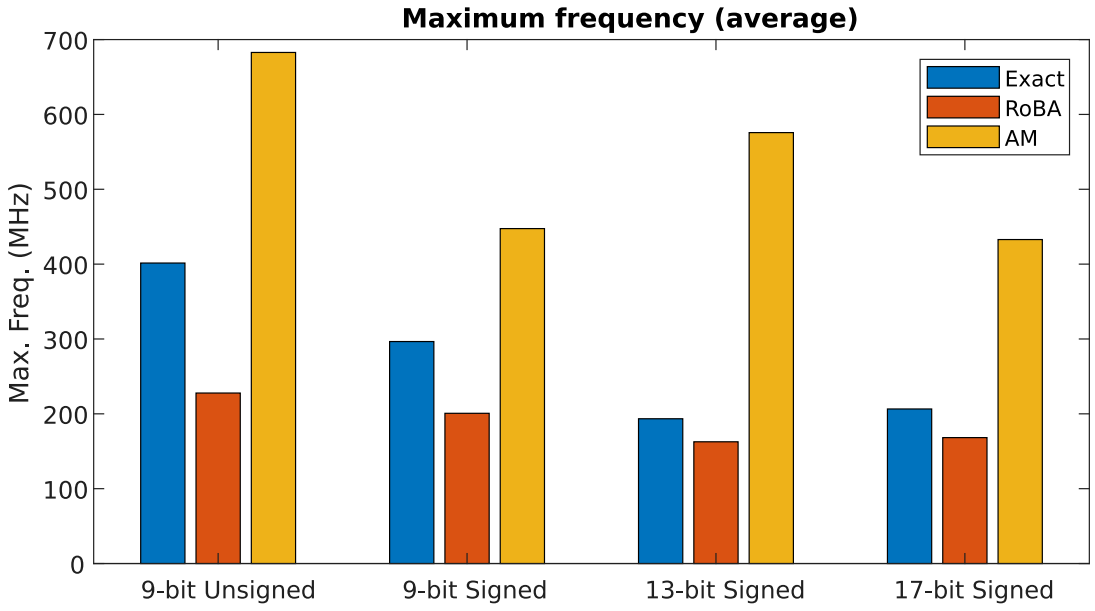


Figure 6.32: Bar chart summarizing the results shown in the tables: multipliers' maximum frequency

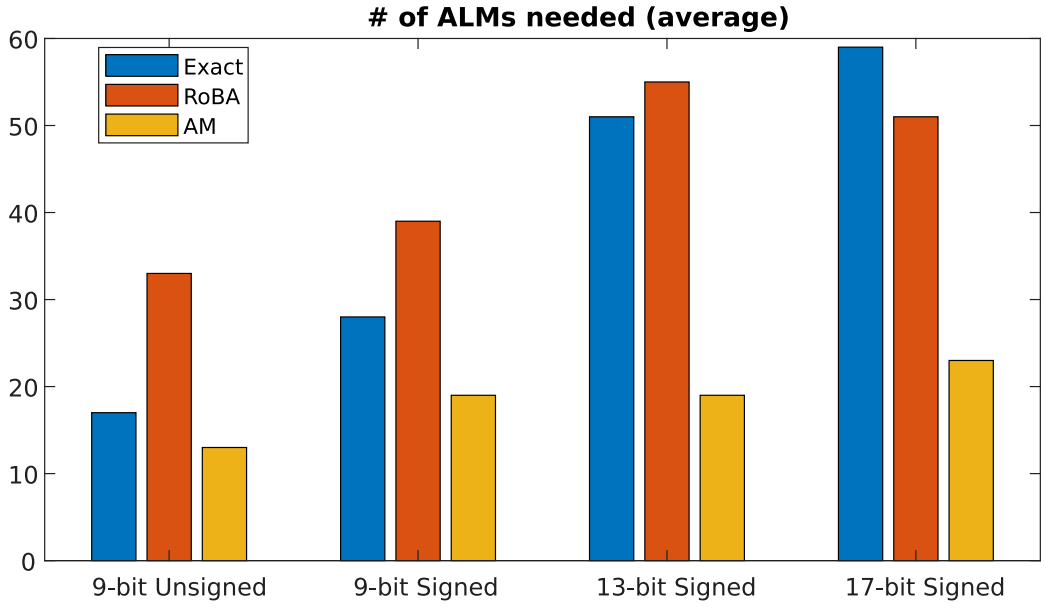


Figure 6.33: Bar chart summarizing the results shown in the tables: multipliers' ALMs

Finally, Table 6.39 shows the performance of the arithmetic circuits required by the zero-crossing detector.

Design	Maximum Frequency (MHz)	ALUTs	ALMs
Exact divider	22.21	393	290 (275)
Exact divider (pipeline)	99.69 (+348 %)	396	287 (271)
AXS1	25.92 (+16 %)	385	287 (275)
AXS2	22.58 (+2 %)	412	292 (283)
AXS3	22.83 (+3 %)	375	272 (261)
AXS4	27.44 (+24 %)	361	266 (248)
AXS4 (pipeline)	102.5 (+273 %)	299	222 (196)
26-bit BKA	250.88	64	49 (41)
26-bit LOA	328.62 (+31 %)	42	43 (33)
15-bit BKA (subtractor)	322.89	34	26 (22)

Table 6.39: Maximum frequency and area for zero-crossing arithmetic circuits

It can be seen that the divider is the slowest circuit in the system and also one of the biggest. As explained earlier, the replacement strategy employed by the approximate divider does not reduce the critical path as it is designed to keep the error below a certain threshold. Thus, speed does not improve significantly. When pipelining is applied to the circuit, its maximum frequency increases dramatically. However, when using this version in the zero-crossing detector, its FSM must be adjusted for the new timing introduced by the additional registers. Given its performance, the AXS4 cell was chosen to replace the exact divider cells. Despite the underwhelming speed results, the area required by the circuit is greatly reduced. This is partly because the last row of subtractor cells is completely removed rather than replaced. AXS2 and AXS3 are the most accurate cells but they also show little to no improvements. AXS1 is worse than AXS4 and also less accurate. For this reason it was discarded. Overall, the results regarding the dividers are disappointing. The reason is that the approximations implemented were designed for integrated circuits at transistor level.
The remaining rows in the table show the 26-bit adders and the 15-bit subtractor, which is not approximated. In the next few tables, the power report for the same circuits is listed. To obtain reliable figures, the same frequency, the input vectors and the simulation time were applied whenever possible. The power is expressed in mW. By analyzing the report, it was found that the majority of the total power ($> 80\%$) is I/O power. However, this number depends strongly on the input and output activities and thus on the parallelism of the circuit and on the simulation time. For this reason, the total power required by the combinational cells is also reported in brackets. This value is much lower and commensurate with the size of the circuits. Other types of power present in the report are related to the register and the clock routing power consumption. However, these values are not reported since they do not change among similar designs. Table 6.40 displays the results obtained by analyzing the adders and subtractors. As a lot of different bit-widths and number of input operands exist, the input vectors generated were different. Nonetheless, circuits that share the same input port description are provided with the same input file. The simulation time is kept constant and is obtained by multiplying the amount of input vectors by the clock period. For arithmetic circuits, 100 000 random vectors were generated. The clock frequency was fixed at 80 MHz.

Design	Total power (mW)	Block dynamic power (mW)	Block static power (mW)	Routing dynamic power (mW)
9-bit BKA	14.77 (0.16)	12.50	1.56	0.71
9-bit LOA	13.40 (0.08)	11.23	1.36	0.80
9-bit ETAII	14.82 (0.18)	12.55	1.56	0.71
17-bit BKA (subtractor)	27.08 (0.35)	23.54	2.56	0.97
17-bit LOA (subtractor)	24.42 (0.19)	21.01	2.56	0.85
17-bit ETAII (subtractor)	27.22 (0.41)	23.58	2.59	1.05
Wallace tree w/ 16-bit BKA	33.07 (0.96)	26.32	4.87	1.88
Wallace tree w/ 16-bit LOA	31.16 (0.74)	24.30	4.87	1.99
Wallace tree w/ 16-bit ETAII	33.44 (1.04)	26.35	4.86	2.22
9-bit BKA (subtractor)	14.70 (0.15)	12.53	1.56	0.61
9-bit LOA (subtractor)	13.18 (0.08)	11.23	1.36	0.58
9-bit ETAII (subtractor)	14.47 (0.17)	12.53	1.36	0.58
Wallace tree w/ 15-bit BKA	31.24 (1.02)	24.81	4.56	1.88
Wallace tree w/ 15-bit LOA	29.32 (0.78)	22.84	4.56	1.92
Wallace tree w/ 15-bit ETAII	31.41 (1.10)	24.89	4.56	1.97
Wallace tree w/ 11-bit BKA	27.20 (1.03)	20.85	4.18	2.17
Wallace tree w/ 11-bit LOA	24.77 (0.81)	19.20	4.18	1.40
Wallace tree w/ 11-bit ETAII	26.76 (1.05)	20.89	4.18	1.70
17-bit BKA (signed subtractor)	27.34 (0.37)	23.54	2.56	1.25
17-bit LOA (signed subtractor)	24.44 (0.20)	21.01	2.56	0.87
17-bit ETAII (signed subtractor)	27.19 (0.41)	23.59	2.56	1.04

Table 6.40: Power consumption of adders and subtractors

Unexpectedly, the LOA circuits consume far less power than both the BKA and ETAII architectures. In fact, the ETAII adders and subtractors often dissipate more power than the exact versions. This is because some of these ETAII circuits are also exact. The Wallace tree with a 16-bit adder is the unit that requires more power. This is expected as it operates on four 16-bit inputs.
Similarly to frequency and FPGA resources, bar charts regarding the power consumption of the combinational cells are created. Table 6.40 is summarized in Figure 6.34.

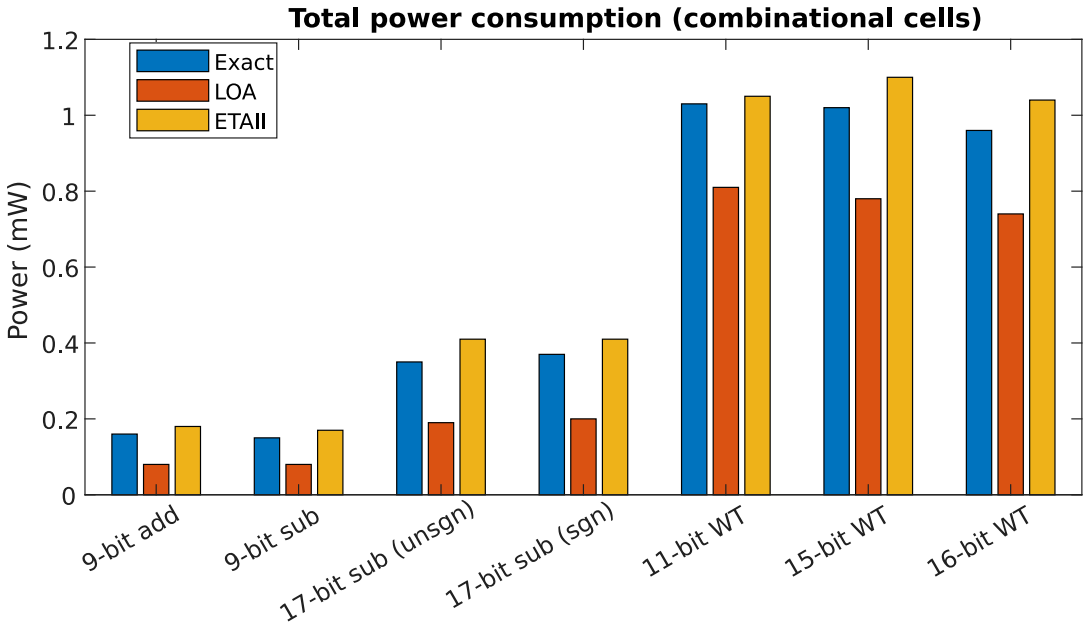


Figure 6.34: Bar chart summarizing the results shown in the tables: adders' total power

Table 6.41 shows the power results for the multipliers. The clock frequency and the number of vectors does

not change.

Design	Total power (mW)	Block dynamic power (mW)	Block static power (mW)	Routing dynamic power (mW)
9-bit WTM 18	17.59 (0.14)	16.06	0.91	0.62
9-bit RoBA 18	14.35 (0.29)	12.59	1.03	0.73
9-bit AM-ER 18	17.03 (0.08)	15.63	0.91	0.49
9-bit WTM 66	19.65 (0.12)	18.13	0.93	0.58
9-bit RoBA 66	14.00 (0.27)	12.39	0.97	0.64
9-bit AM-ER 66	19.46 (0.05)	17.65	1.03	0.78
9-bit WTM 36	17.49 (0.14)	16.03	0.93	0.52
9-bit RoBA 36	14.15 (0.28)	12.62	0.91	0.62
9-bit AM-ER 36	17.04 (0.07)	15.63	0.91	0.50
9-bit WTM 161	21.66 (0.27)	20.06	0.91	0.69
9-bit RoBA 161	15.51 (0.40)	13.62	1.03	0.87
9-bit AM-ER 161	20.85 (0.11)	19.29	0.91	0.65
8-bit WTM 214	20.77 (0.41)	19.28	0.84	0.65
8-bit RoBA 214	14.60 (0.39)	13.07	0.84	0.69
8-bit AM-ER 214	19.68 (0.17)	18.22	0.85	0.61
9-bit BWM 74	21.13 (0.43)	19.43	1.02	0.68
9-bit RoBA 74	21.21 (0.50)	19.46	0.90	0.85
9-bit AM-ER 74	20.26 (0.16)	18.77	0.90	0.58
9-bit BWM 122	21.11 (0.43)	19.45	0.95	0.71
9-bit RoBA 122	21.19 (0.48)	19.40	0.92	0.88
9-bit AM-ER 122	20.25 (0.27)	18.67	0.94	0.65
9-bit BWM 166	21.11 (0.45)	19.48	0.90	0.72
9-bit RoBA 166	21.40 (0.60)	19.50	0.90	1.00
9-bit AM-ER 166	19.96 (0.25)	18.44	0.90	0.62
9-bit BWM 109	21.13 (0.50)	19.52	0.90	0.71
9-bit RoBA 109	21.34 (0.58)	19.52	0.90	0.91
9-bit AM-ER 109	20.10 (0.28)	18.48	0.90	0.71
13-bit BWM 546	23.64 (0.96)	21.28	1.20	1.17
13-bit RoBA 546	22.40 (0.70)	19.79	1.34	1.27
13-bit AM-ER 546	21.71 (0.23)	19.79	1.20	0.72
17-bit BWM 74	23.00 (0.99)	20.45	1.48	1.07
17-bit RoBA 74	22.54 (0.61)	20.07	1.49	0.97
17-bit AM-ER 74	21.11 (0.20)	18.97	1.47	0.66
17-bit BWM 122	23.13 (0.96)	20.36	1.48	1.30
17-bit RoBA 122	22.47 (0.57)	20.06	1.48	0.93
17-bit AM-ER 122	21.17 (0.38)	18.94	1.47	0.75
17-bit BWM 166	23.38 (1.10)	20.52	1.48	1.38
17-bit RoBA 166	22.82 (0.72)	20.14	1.48	1.21
17-bit AM-ER 166	20.82 (0.31)	18.60	1.47	0.74
17-bit BWM 109	23.29 (1.24)	20.62	1.48	1.20
17-bit RoBA 109	23.33 (0.83)	20.13	1.64	1.56
17-bit AM-ER 109	20.63 (0.39)	18.39	1.47	0.77

Table 6.41: Power consumption of multipliers

At first glance, by examining the column related to the total power, it seems as if the RoBA multiplier consumes less power than the AM-ER and the exact multipliers. However, the power dissipated by the combinational cells inside the brackets tells a whole different story. The AM-ER consumes way less than the RoBA which is also more power-hungry than the exact architectures. This trend changes when complexity increases. It appears that for larger parallelisms, i.e. 13 and 17 bits, the RoBA does not require as much power as the BWM. This might be due to the fact that the BWM complexity increases more rapidly with parallelism.
Figure 6.35 shows a graphical representation of the data contained in Table 6.41.

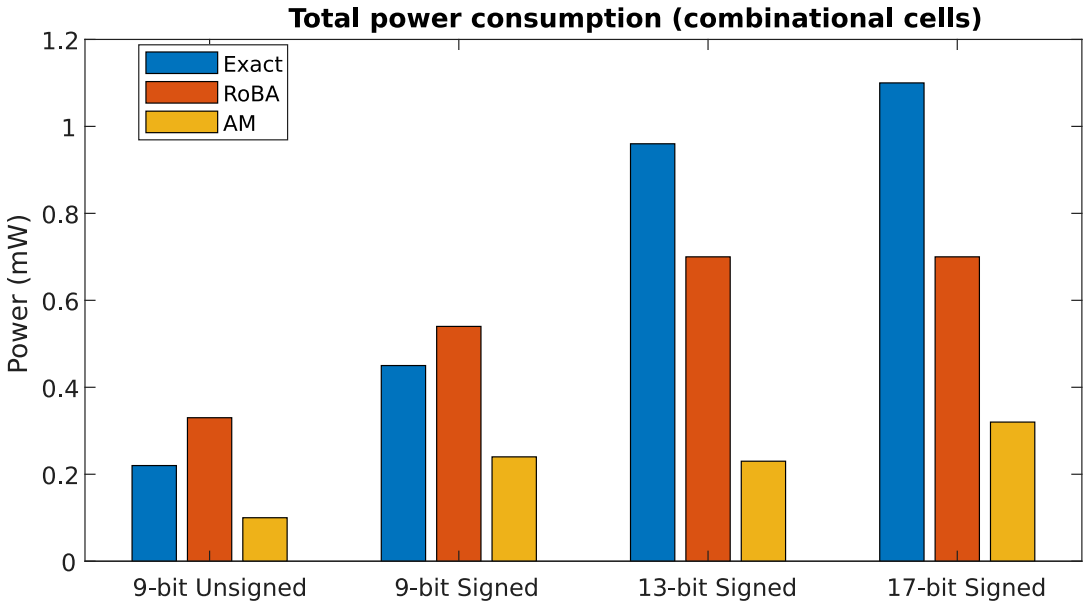


Figure 6.35: Bar chart summarizing the results shown in the tables: multipliers' total power

Design	Total power (mW)	Block dynamic power (mW)	Block static power (mW)	Routing dynamic power (mW)
Exact divider	9.78 (1.66)	5.96	2.40	1.41
Exact divider (pipeline)	9.82 (1.40)	6.00	2.40	1.42
AXS4	9.61 (1.56)	5.85	2.40	1.36
AXS4 (pipeline)	9.30 (1.05)	5.72	2.40	1.17
26-bit BKA	13.41 (0.15)	8.95	4.05	0.40
26-bit LOA	12.44 (0.09)	8.00	4.05	0.39
15-bit BKA (subtractor)	7.89 (0.08)	5.21	2.40	0.28

Table 6.42: Maximum frequency and area for zero-crossing arithmetic circuits

Table 6.42 shows the power consumption of the arithmetic circuits found in the zero-crossing detector. Due to the complexity of the dividers, only 10 000 vectors were tested. This was done to avoid an excessively long simulation time. Moreover, while the samples were randomly generated, the dividend always had to be smaller than the divisor. In this case, a clock frequency of 20 MHz was selected. The results suggest a low total power consumption overall. However, this is due to the size of the input vectors' files. Instead, the power required by the combinational cells is much higher than what was seen in the previous circuits.
It appears as though the approximate divider consumes less. In addition, pipelining helps reduce the power consumption. It is important to note that the power figures related to the registers increased by 61% when the exact divider is pipelined, while they increased by 45% for the approximate divider. The same frequency of 20 MHz is used for both the adders and subtractors. However, as the number of vectors is again 100 000, the total power is higher than that of the dividers. By looking at the total combinational power, the numbers are much more reasonable. The 26-bit LOA cuts power consumption almost in half. Register power is not interesting for other designs as it is similar.

6.2.2 Filters

In this section, the filters are tested for power, delay and area. Table 6.43 gives an overview of the maximum frequency and resource utilization of both the intermediate and final filters, i.e. LOA-RoBA and ETAII-AM-ER. Given the higher speed of the LOA compared to the ETAII, it was paired to the RoBA which is even slower than the exact one. Table 6.44 instead deals with power consumption. However, only the final filters are listed. All the filters are tested at a frequency of 80 MHz. The input vectors are part of an image, which is also one of the most noisy. Due to the long simulation time, an entire image could not be simulated. However, the results provided by the power analyzer show a high estimation confidence. The DF-S takes the input from the smoothing filter of the same kind for the same section of the image tested.

Design	Maximum Frequency (MHz)	ALUTs	ALMs
Exact smoothing filter	105.22	272	183 (175)
LOA smoothing filter	125.22 (+19 %)	220	151 (142)
ETAII smoothing filter	113.93 (+8 %)	265	177 (166)
RoBA smoothing filter	80.98 (-23 %)	478	324 (316)
AM-ER smoothing filter	108.14 (+3 %)	229	155 (144)
LOA-RoBA smoothing filter	88.79 (-16 %)	398	264 (250)
ETAII-AM-ER smoothing filter	117.45 (+12 %)	229	159 (151)
Exact diff. filter, no smoothing	130.51	249	179 (175)
LOA diff. filter, no smoothing	150.31 (+15 %)	228	169 (158)
ETAII diff. filter, no smoothing	138.37 (+6 %)	261	192 (181)
RoBA diff. filter, no smoothing	93.44 (-28 %)	390	270 (252)
AM-ER diff. filter, no smoothing	134.57 (+3 %)	219	162 (154)
LOA-RoBA diff. filter, no smoothing	104.20 (-20 %)	355	247 (232)
ETAII-AM-ER diff. filter, no smoothing	143.02 (+10 %)	242	175 (163)
Exact 2nd order diff. filter	96.09	217	154 (145)
LOA 2nd order diff. filter	113.46 (+18 %)	172	125 (117)
ETAII 2nd order diff. filter	99.44 (+3 %)	210	151 (145)
RoBA 2nd order diff. filter	83.21 (-13 %)	225	154 (143)
AM-ER 2nd order diff. filter	125.39 (+30 %)	170	124 (111)
LOA-RoBA 2nd order diff. filter	94.77 (-1 %)	173	122 (112)
ETAII-AM-ER 2nd order diff. filter	129.77 (+35 %)	172	129 (122)
Exact diff. filter after smoothing	91.52	542	398 (377)
LOA diff. filter after smoothing	113.78 (+24 %)	464	355 (342)
ETAII diff. filter after smoothing	97.71 (+7 %)	537	368 (356)
RoBA diff. filter after smoothing	78.89 (-14 %)	535	381 (351)
AM-ER diff. filter after smoothing	120.45 (+32 %)	362	262 (243)
LOA-RoBA diff. filter after smoothing	91.55 (+0 %)	447	305 (285)
ETAII-AM-ER diff. filter after smoothing	131.54 (+44 %)	326	216 (209)

Table 6.43: Maximum frequency and area for filters

All filters except the ones containing the RoBA multiplier end up with a higher frequency. Coupling the RoBA multiplier with the LOA actually increases the maximum frequency, although not significantly. As expected, the ETAII-AM-ER filter shows the best improvements. The same is true resources-wise. The more rapid increase in complexity of the BWM compared to the RoBA discussed before is also visible here in the DF-S and the DF-2ND which show a smaller penalty in terms of delay and use less resources.
Figures 6.36 and 6.37 illustrate the results from Table 6.43 by means of a bar chart.

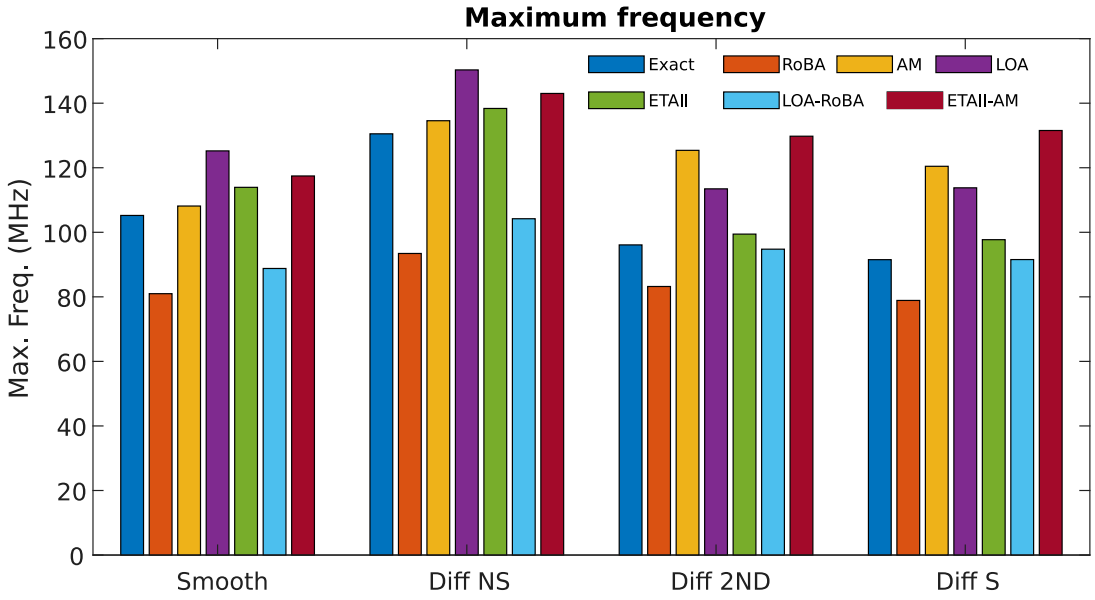


Figure 6.36: Bar chart summarizing the results shown in the tables: filters' maximum frequency

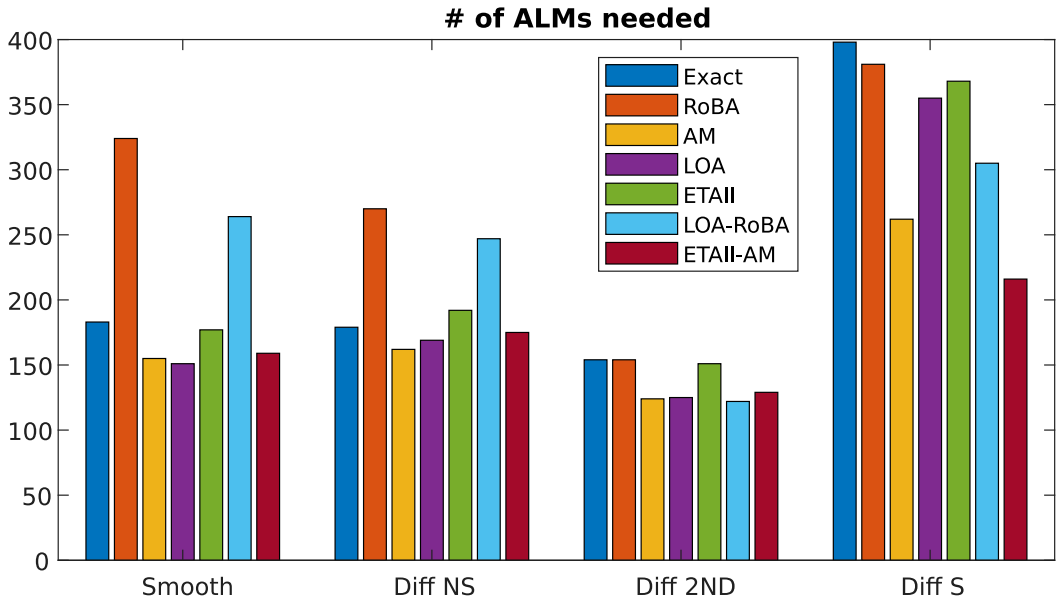


Figure 6.37: Bar chart summarizing the results shown in the tables: filters' ALMs

Design	Total power (mW)	Block dynamic power (mW)	Block static power (mW)	Routing dynamic power (mW)
Exact smoothing filter	17.64 (2.34)	14.28	0.94	2.42
LOA-RoBA smoothing filter	10.51 (1.60)	7.56	0.95	2.00
ETAII-AM-ER smoothing filter	17.66 (2.00)	14.54	0.95	2.17
Exact diff. filter, no smoothing	21.87 (3.57)	17.89	0.84	3.14
LOA-RoBA diff. filter, no smoothing	18.66 (2.61)	15.00	0.95	2.72
ETAII-AM-ER diff. filter, no smoothing	21.81 (2.95)	18.05	0.92	2.83
Exact 2nd order diff. filter	18.06 (2.02)	14.94	0.95	2.17
LOA-RoBA 2nd order diff. filter	12.85 (0.80)	10.57	0.93	1.35
ETAII-AM-ER 2nd order diff. filter	17.53 (1.82)	14.62	0.95	1.96
Exact diff. filter after smoothing	27.44 (7.32)	19.62	1.57	6.25
LOA-RoBA diff. filter after smoothing	22.13 (3.51)	16.94	1.53	3.66
ETAII-AM-ER diff. filter after smoothing	24.07 (4.23)	17.83	1.41	4.83

Table 6.44: Power consumption of filters

Table 5.5 shows the power consumption for the filters. Interestingly, the LOA-RoBA actually requires less power to function and this applies also to the power dissipated by the combinational cells. The LOA adders and subtractors might be responsible for this reduction. The power consumed by the ETAII-AM-ER is comparable to the exact design.
The power dissipated by the combinational cells in the filters is summarized in Figure 6.38. The numbers are borrowed from Table 5.5.

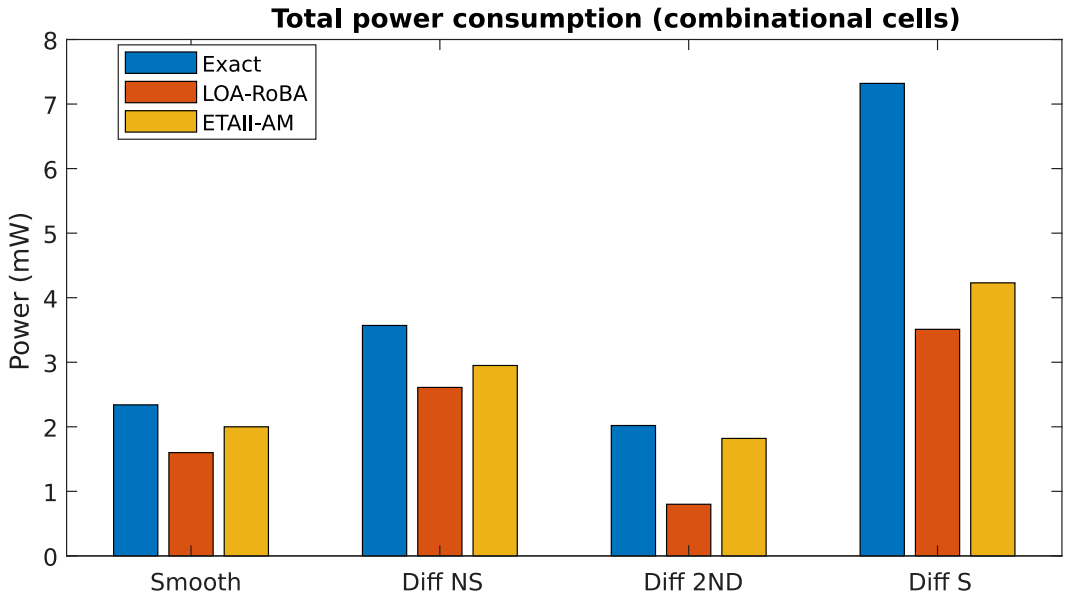


Figure 6.38: Bar chart summarizing the results shown in the tables: filters' power

6.2.3 Zero-crossing

In this last section, the zero-crossing detector is analyzed. Table 6.45 lists the result for the four variants, with no pipeline. It is clear that the divider represents the bottle neck not only of this circuit but also of the entire system. However, the results can be greatly improved by pipelining the dividers. The approximate designs show improvements in both frequency and resource usage.

Design	Maximum Frequency (MHz)	ALUTs	ALMs
Exact., no avg	20.21	464	358 (321)
Approx., no avg	25.37 (+26 %)	434	321 (296)
Exact., pt	20.82	528	395 (363)
Approx., pt	25.72 (+24 %)	487	358 (329)

Table 6.45: Maximum frequency and area for the zero-crossing detection circuit

Power analysis was performed using a 10 MHz clock frequency. This is due to the limitation posed by the divider. The input vectors come from the output of the differentiation filters, which processed the noisy image in the filter analysis. Table 6.46 shows the power results, however, the reduction in power is minimal.

Design	Total power (mW)	Block dynamic power (mW)	Block static power (mW)	Routing dynamic power (mW)
Exact, no avg	3.81 (0.06)	0.42	3.06	0.34
Approx., no avg	3.57 (0.04)	0.41	2.93	0.23
Exact, avg	4.21 (0.06)	0.37	3.51	0.33
Approx., avg	4.14 (0.05)	0.36	3.50	0.28

Table 6.46: Power consumption of the zero-crossing detection circuit

In this chapter, the results of the designs implemented during the project were reported. Each unit in the pipeline was considered separately. The full system can be put together based on the needs of the designer simply by combining the blocks with the desired performance. As several possible combinations exist, this analysis was not carried out for time reasons.

CHAPTER 7

Conclusion

Systems that process digital images, among others, can greatly benefit from approximate computing. In this thesis, approximation methods were applied to arithmetic circuits, i.e. adders, multipliers and dividers. For each of them several architectures and possible implementations exist in the literature. However, given the novelty of this paradigm, virtually no research has examined the application of approximate computing on laser line systems. the amount of inexact strategies is still relatively small today due to the novelty of this paradigm. Research has focused a lot on approximate adders. These circuits have been extensively studied and a lot of new algorithms are available in the literature. This is because the adder is a fundamental piece present in almost every digital design. Moreover, it is the main building block of tree multipliers. In this project, two designs have been evaluated, i.e. the LOA and the ETAII. Both of these adders aim at tackling the carry propagation overhead which turns out to be the critical path in a simple RCA. The carry propagation chain is cut and thus shortened. In this way, the circuits can be faster but they are no longer accurate. The approximation strategy for the LOA is simple and straightforward. Because it modifies full adders singularly, the desired accuracy can be obtained with a high granularity by establishing the bit size of the inexact part. On the other hand, the ETAII groups n bits in m accurate blocks. The bigger the blocks, the more accurate the design is. As the name suggests, this circuit is a revised version of a previous implementation called ETAI. Both of these designs suffer from accuracy issues that can at times make them unusable. To avoid unreasonable errors, the simplifications should only affect the LSBs. The ETAII adder does not guarantee this requirement and because of this it needs corrections. The synthesis results have shown that the LOA is the winner under all aspects. Given the approximation strategy, the LOA inexact part was always close to half the total bit-width. This means that, when parallelism grows, the rate at which the frequency increases compared to the exact version slows down. On the contrary, the ETAII has a fixed block size that does not increase with the bit-width. Thus, given a block size, the length of the critical path does not change, unless error corrections are required. In other words, if the ETAII block size uses fewer bits than the exact part of a LOA of the same parallelism, the ETAII will be faster. This normally happens for large words. Moreover, if no error correction affecting the critical path is necessary, this circuit should be faster than the LOA of the same size. Despite the considerations on frequency, the LOA always occupies a smaller area and consumes less power. For filters containing only approximate adders, the LOA filter wins on all counts with the exception of error analysis. However, this is due to the error correction applied to the ETAII units.

Approximate multipliers have also been studied. While, many different techniques have been proposed in the literature, a large part of the approximations revolves around the adders inside them. The RoBA multiplier and the AM-ER are the designs chosen for this project. In this case, the AM-ER is the absolute winner in terms of area, frequency and power. The RoBA, on the contrary, shows unsatisfactory results. The only positive side to it is its accuracy. However, given its considerably low maximum frequency and its large area, at its best, the RoBA managed to obtain the same performance as the exact counterpart. This happened only for larger parallelisms nevertheless. The underwhelming performance observed for the RoBA might be a consequence of the optimizations applied to the exact multipliers. Although the RoBA is also simplified, the optimizations take place on two different levels of abstraction. In the accurate architectures, i.e. the BWM and the WTM, the full adders dealing with the partial products are reduced by eliminating the unnecessary rows. As for the RoBA, multiplication is converted into two additions and a subtraction and no partial products are generated. This means that the RoBA works on the original operands and does not expand them into partial products.

The AM-ER, instead, employs a rather different approximation method. Similarly to the inexact adders, the AM-ER allows approximation of the LSBs. It also has a bit-level granularity thanks to which the error can be better controlled. Moreover, when optimized with a specific coefficient, it is able to reach speeds close to those of adders. As no partial products are present in the RoBA, these multipliers are less dependent on coefficients and their parameters are more stable.
Dividers, on the other hand, do not appear in literature as often as other circuits. Due to the complexity of these units, they are implemented in hardware only when speed is important. This might be the reason why not much time has been dedicated to the development of approximate versions of these circuits. The approximate subtractor cells studied here and applied to the divider are designed for VLSI circuits as the approximation is better visualized at transistor level. Combinational dividers intended for an FPGA design were not found. As a consequence, approximate array dividers do not show great improvement compared to the exact ones. A different replacement strategies can be tested depending on the constraint of the design. Moreover, the length of subpixeling can be reduced. This produces a number with a smaller fractional part, but the divider and all the circuits after that in the zero-crossing detection unit will be also smaller. Similarly to the multipliers, the bit-width reduction of an array divider corresponds to a quadratic reduction of the area.
The study of the filters also showed interesting results. The distribution of the error when combining different approximate designs and also when cascading multiple filters is unpredictable and in some cases error compensation was achieved. By comparing the results, the LOA filter performed the best in terms of frequency and was on par with the AM-ER filter in terms of resources. The filters featuring the RoBA multipliers are the most accurate, although given the results, an exact multiplier might be the best choice if accuracy is important. When considering the whole system, the simplest implementation is obtained by a single differentiation filter and a ZC-NA. This is enough to obtain meaningful results. Depending on the design constraints, the different approximation strategies proposed can be used. Of course, much more can be done as explained in the following section.

7.1 What more can be done?

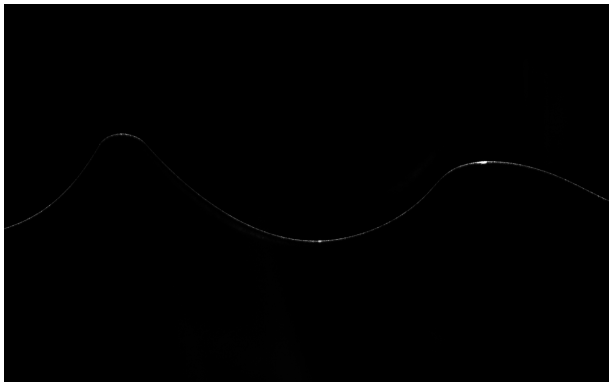
For such a project, a lot of different paths can be explored. While several design solution were tested, many other are waiting to be uncovered. Certainly, modelling and testing new approximate circuits is something that can be done. In particular, circuits specifically designed for FPGA. Of course, new exact architectures can also be implemented for comparison. The design exploration space can be greatly widened. In the project, the single arithmetic circuits have not been tested for different approximation depths. Although these results are present in literature, the number of approximate bits can be varied to see how it affects the blocks in the pipeline in terms of accuracy, as well as speed, area, and power. In the case of the LOA and the AM-ER, it is possible to either extend or shrink the approximate part simply by changing its bit-width. As for the ETAII, the same is done with blocks of different lengths. The size does not have to be the same for each block. Not much can be done with the RoBA multiplier. Further increasing the total word-length might produce a design that is faster and smaller than an exact equivalent. However, the variation of parameters such as frequency with the number of bits is unknown. The behaviour of the RoBA vs. the WTM for larger bit-widths is also not clear as only the 9-bit version was implemented. Both the BWM and WTM approximations depend strongly on the coefficient. This is also true for the RoBA but in a less significant way. After all, the number of blocks for the single operand remains the same. This is probably the reason why its complexity does not increase as quickly as the other multipliers. Adding extra bits extends the internal parallelism but the number of units remains the same. Extending the word-length of the BWM or the WTM creates entire new rows of full-adders, i.e. partial products, and also expands the columns. With reference to Figure 5.11, adding a bit enlarges both sides of the square. Thus, it is safe to assume that the complexity of the exact multiplier increases quadratically. This growth in resources might be instead close to linear in the RoBA, which does not deal with partial products. Similarly, complexity reduces at a higher rate when optimizing exact designs, whereas the RoBA does not exhibit significant improvements. Given the high accuracy of the RoBA, it is possible to approximate the BKA inside. This comes at a cost of a greater error. Although this gimmick will not significantly improve the RoBA multiplier, it might make it worth taking it into consideration.
SG filters with different cut-off frequencies can be tested. Their word-lengths can be increased or shortened by adjusting the fractional part. Finally, the approximate filters that were not discussed can be implemented. These are the LOA-AM-ER and the ETAII-RoBA. As the LOA and AM-ER filters displayed the best performance, combining the two will probably result in an even better filter than those tested. However, it is expected that

the ETAII-RoBA filter will yield inadequate results. All units in the filters can be pipelined to increase speed and contain power consumption. Both coarse- and fine-grain pipelining are feasible. The last technique is especially suitable for multipliers and multi-operand adders. Various filter structures can be designed other than the direct from seen here. Additionally, if the coefficients in a specific application have only two '1's, it is possible to implement optimized versions of the multiplier by means of approximate two-operand adders.
Further optimization can be applied to the zero-crossing detection circuit. For this unit, the divider represents the bottleneck. When sticking to the array architecture, the fastest divider is implemented by instantiating a row of registers for each row of subtractor cells. Given the high latency of this implementation, a sequential solution might produce similar results but with much less resources. However, the error generated from the approximation of the only adder might be too large given that it grows at every cycle. Naturally, testing other divider architectures as well as different division algorithms is a viable option. Finally, the effect of the threshold on the zero-crossing detection circuit can be further analyzed by providing additional threshold values at the input. The method used in this project to find the zero is not unique. Several solutions are conceivable to compute the zero position. In the ZC-A circuit, the LOA was chosen for the average computation. The reasons not to use the ETAII are justified by its unreliable error performance that also depends on the bit-width. However, an ETAII version of this circuit is worth trying. Metrics such as area-delay product and energy-delay product can also be estimated to understand more clearly which design is most suitable to meet a specific constraint.
Given the size of the design space, it is no surprise that much more can be done. In the meantime, the solutions proposed in this thesis represent an initial but valid step on which future improved implementations can rely.

Appendices

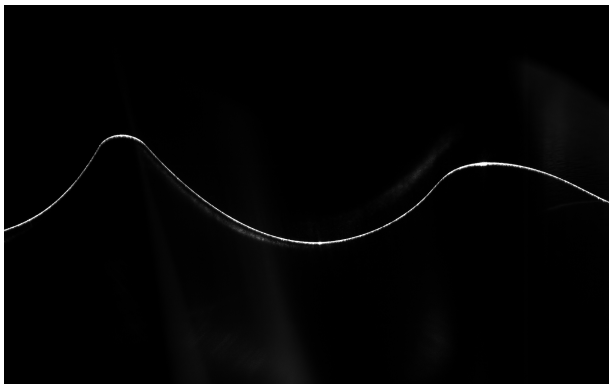
APPENDIX A

Tested images

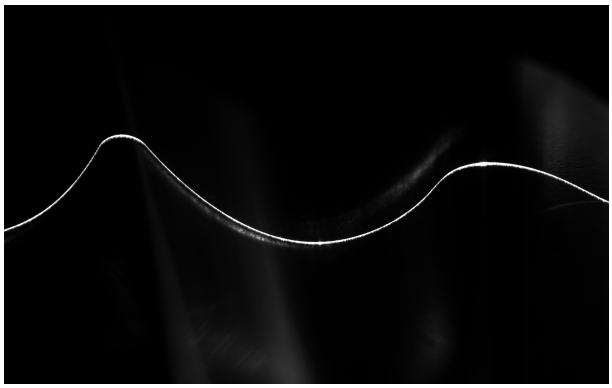


(a) Aluwaves 100 μs

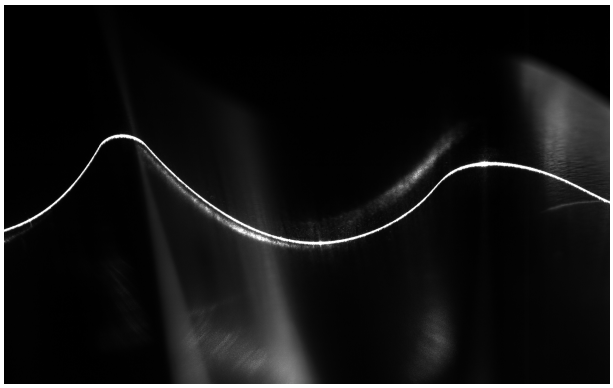
(b) Aluwaves 500 μs



(c) Aluwaves 1000 μs

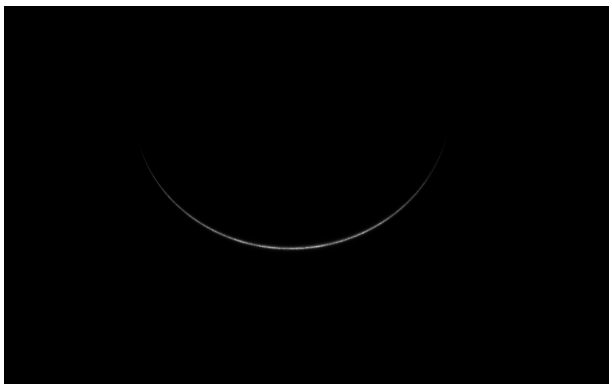


(d) Aluwaves 2000 μs

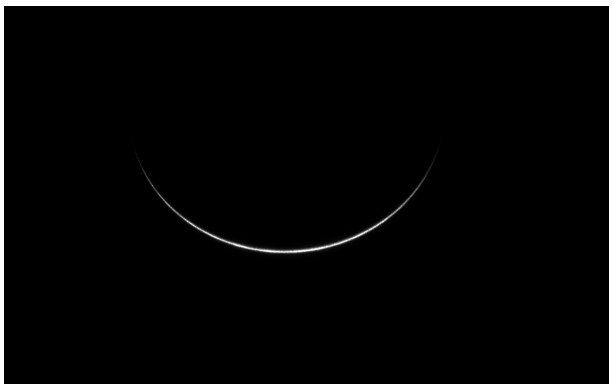


(e) Aluwaves 5000 μs

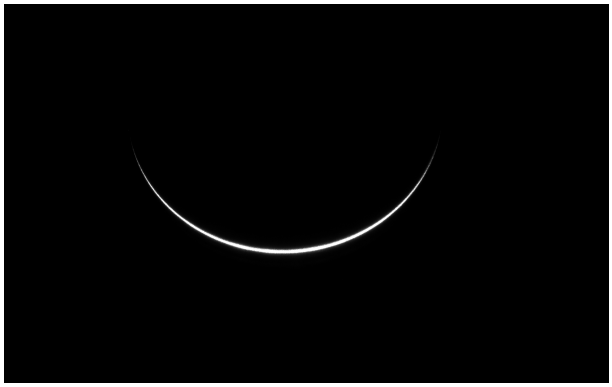
Figure A.1: Aluwaves for different exposure times



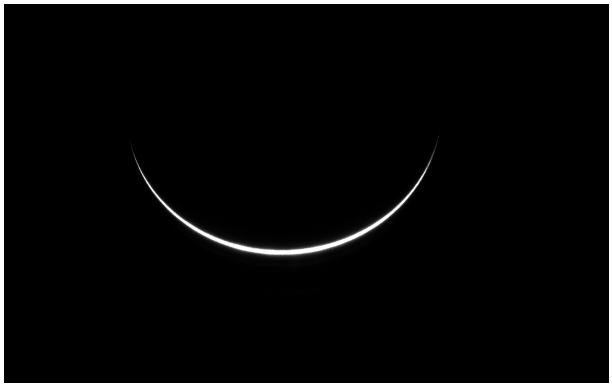
(a) Sphere 100 μs



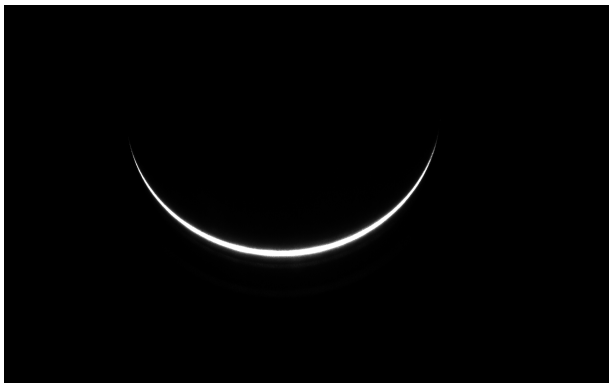
(b) Sphere 200 μs



(c) Sphere 500 μs



(d) Sphere 1000 μs



(e) Sphere 2000 μs

(f) Sphere 5000 μs

Figure A.2: Sphere for different exposure times

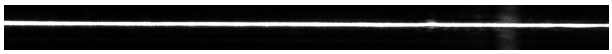
(a) White plane 20 μs

(b) White plane 50 μs

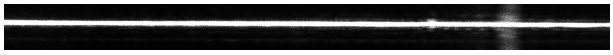
(c) White plane 100 μs

(d) White plane 200 μs

(e) White plane 500 μs

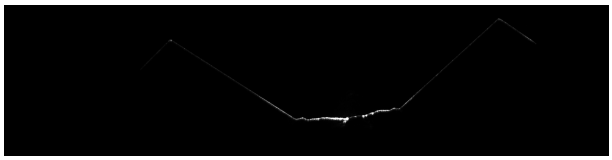


(f) White plane 1000 μs



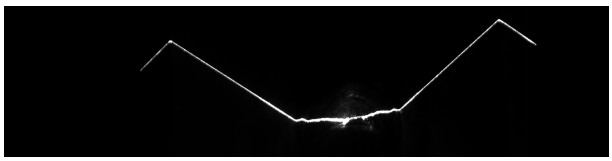
(g) White plane 2000 μs

Figure A.3: White plane for different exposure times

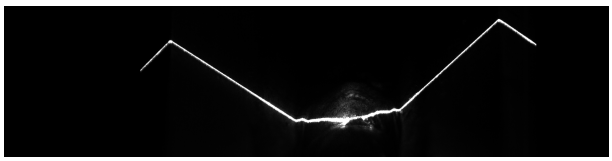


(a) Weld 10 μs

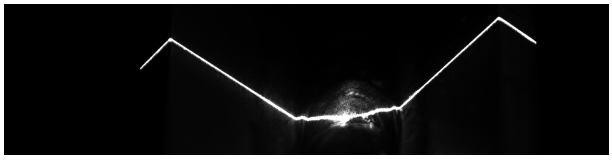
(b) Weld 100 μs



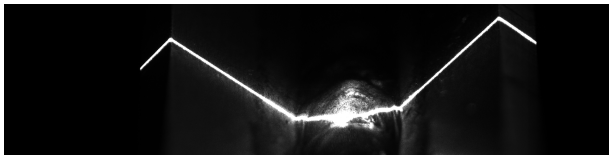
(c) Weld 500 μs



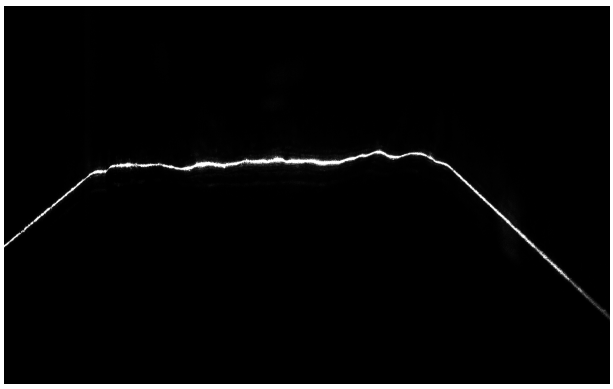
(d) Weld 1000 μs



(e) Weld 2000 μs



(f) Weld 5000 μs



(g) Weld 1000 μs

Figure A.4: Weld for different exposure times

(a) Plane 10 μs

(b) Plane 20 μs

(c) Plane 40 μs

(d) Plane 80 μs

(e) Plane 160 μs

(f) Plane 320 μs

(g) Plane 640 μs

Figure A.5: Plane for different exposure times

APPENDIX B

Bibliography

[1] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," *2013 18th IEEE European Test Symposium (ETS)*, pp. 1–6, 2013.

[2] MoviMED. What is laser triangulation? [Online]. Available: https://www.movimed.com/knowledgebase/what-is-laser-triangulation/

[3] T. S. Huang, "Computer vision: Evolution and promise," 1996.

[4] A. Klipfel. Improving the 3d scan precision of laser triangulation. Automation Technology GmbH. [Online]. Available: https://www.visiononline.org/userAssets/aiaUploads/file/25-Improvingthe3DScanPrecisionofLaserTriangulation-Dr_AthinodorosKlipfel.pdf

[5] R. A. Jarvis, "A perspective on range finding techniques for computer vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1983.

[6] D. Harrison and M. Weir, "High-speed triangulation-based 3-d imaging with orthonormal data projections and error detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 4, pp. 409–416, 1990.

[7] R. I. Hartley and P. Sturm, "Triangulation," *Computer Vision and Image Understanding*, 1997.

[8] J. Franca, M. Gazziro, A. Ide, and J. Saito, "A 3d scanning system based on laser triangulation and variable field of view," *IEEE International Conference on Image Processing 2005*, vol. 1, pp. I–425, 2005.

[9] W.-G. Wang, Z.-Y. Hu, and Z.-Y. Shun, "3d shape measurement based on computer vision," *Proceedings. International Conference on Machine Learning and Cybernetics*, vol. 2, pp. 910–914 vol.2, 2002.

[10] W. Lei, B. Mei, G. Jun, and O. ChunSheng, "A novel double triangulation 3d camera design," *2006 IEEE International Conference on Information Acquisition*, pp. 877–882, 2006.

[11] W. Latimer. (2015) Understanding laser-based 3d triangulation methods. Vision Systems Design. [Online]. Available: https://www.vision-systems.com/cameras-accessories/article/16738248/understanding-laserbased-3d-triangulation-methods

[12] MTIInstruments. Laser triangulation sensors. [Online]. Available: https://mtiinstruments.com/technology-principles/laser-triangulation-sensors/

[13] S. C. Park, M. K. Park, and M. G. Kang, "Super-resolution image reconstruction: a technical overview," *IEEE Signal Processing Magazine*, vol. 20, no. 3, pp. 21–36, 2003.

[14] W. Liu, X. Yang, Y. Wang, X. Yang, and W. Lin, "Subpixel segmentation for ceramic defects," *IECON 2021 – 47th Annual Conference of the IEEE Industrial Electronics Society*, pp. 1–5, 2021.

[15] L. Baboulaz and P. L. Dragotti, "Local feature extraction for image super-resolution," *2007 IEEE International Conference on Image Processing*, vol. 5, pp. V – 401–V – 404, 2007.

[16] A. J. Tabatabai and O. R. Mitchell, "Edge location to subpixel values in digital imagery," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-6, no. 2, pp. 188–201, 1984.

[17] E. Lyvers, O. Mitchell, M. Akey, and A. Reeves, "Subpixel measurements using a moment-based edge operator," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 12, pp. 1293–1309, 1989.

[18] Q. Wang, P. M. Atkinson, and W. Shi, "Fast subpixel mapping algorithms for subpixel resolution change detection," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 53, no. 4, pp. 1692–1706, 2015.

[19] J. Zeng, L. Fang, J. Pang, H. Li, and F. Wu, "Subpixel image quality assessment syncretizing local subpixel and global pixel features," *IEEE Transactions on Image Processing*, vol. 25, no. 12, pp. 5841–5856, 2016.

[20] E. Psarakis and G. Evangelidis, "An enhanced correlation-based method for stereo correspondence with subpixel accuracy," *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, vol. 1, pp. 907–912 Vol. 1, 2005.

[21] S. A. Stankevich, S. V. Shklyar, and V. M. Tyagur, "Subpixel resolution satellite imaging technique," *The International Conference on Digital Technologies 2013*, pp. 55–58, 2013.

[22] Y. Sun, J. Moura, and C. Ho, "Subpixel registration in renal perfusion mr image sequence," *2004 2nd IEEE International Symposium on Biomedical Imaging: Nano to Macro (IEEE Cat No. 04EX821)*, pp. 700–703 Vol. 1, 2004.

[23] X. Lu, A. Jain, and D. Colbry, "Matching 2.5d face scans to 3d models," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 1, pp. 31–43, 2006.

[24] N. Nahi, "Role of recursive estimation in statistical image enhancement," *Proceedings of the IEEE*, vol. 60, no. 7, pp. 872–877, 1972.

[25] J. Biemond and J. J. Gerbrands, "An edge-preserving recursive noise-smoothing algorithm for image data," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 9, no. 10, pp. 622–627, 1979.

[26] G. Fenu and T. Parisini, "Nonparametric kernel smoothing for model-free fault symptom generation," *1999 European Control Conference (ECC)*, 1999.

[27] K. Kai, L. Tingting, X. Xianchun, Z. Guoquan, and Z. Jianxin, "Study of infrared image denoising algorithm based on steering kernel regression image guided filter," pp. 1–3, 2019.

[28] W. Yang and I. G. Zurbenko, "A semiadaptive smoothing algorithm in bispectrum estimation," *IEEE Transactions on Signal Processing*, vol. 56, no. 11, pp. 5369–5375, 2008.

[29] Y. Ohtake, A. Belyaev, and I. Bogaevski, "Polyhedral surface smoothing with simultaneous mesh regularization," *Proceedings Geometric Modeling and Processing 2000. Theory and Applications*, pp. 229–237, 2000.

[30] H. Hou and H. Andrews, "Cubic splines for image interpolation and digital filtering," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 26, no. 6, pp. 508–517, 1978.

[31] Y. Zheng and R. Xu, "An adaptive exponential smoothing approach for software reliability prediction," *2008 4th International Conference on Wireless Communications, Networking and Mobile Computing*, pp. 1–4, 2008.

[32] A. Savitzky and M. J. E. Golay, "Smoothing and differentiation of data by simplified least squares procedures," *Analytical Chemistry*, vol. 36, pp. 1627–1639, 1964.

[33] R. W. Schafer, "What is a savitzky-golay filter? [lecture notes]," *IEEE Signal Processing Magazine*, vol. 28, no. 4, pp. 111–117, 2011.

[34] H. Jiang, F. J. H. Santiago, H. Mo, L. Liu, and J. Han, "Approximate arithmetic circuits: A survey, characterization, and recent applications," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2108–2135, 2020.

[35] M. Jung, D. M. Mathew, C. Weis, and N. Wehn, "Invited: Approximate computing with partially unreliable dynamic random access memory — approximate dram," *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–4, 2016.

[36] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 25–36, 2013.

[37] A. Raha, H. Jayakumar, S. Sutar, and V. Raghunathan, "Quality-aware data allocation in approximate dram," *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pp. 89–98, 2015.

[38] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan, "Macaco: Modeling and analysis of circuits for approximate computing," *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 667–673, 2011.

[39] H. Jiang, C. Liu, L. Liu, F. Lombardi, and J. Han, "A review, classification, and comparative evaluation of approximate arithmetic circuits," *ACM Journal on Emerging Technologies in Computing Systems*, 2017.

[40] M. Gligoric, S. Khurshid, S. Misailovic, and A. Shi, "Mutation testing meets approximate computing," *2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER)*, pp. 3–6, 2017.

[41] A. G. M. Strollo and D. Esposito, "Approximate computing in the nanoscale era," *2018 International Conference on IC Design Technology (ICICDT)*, pp. 21–24, 2018.

[42] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," *2011 24th Internatioal Conference on VLSI Design*, pp. 346–351, 2011.

[43] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas, "Bio-inspired imprecise computational blocks for efficient vlsi implementation of soft-computing applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 57, no. 4, pp. 850–862, 2010.

[44] K. Y. Kyaw, W. L. Goh, and K. S. Yeo, "Low-power high-speed multiplier for error-tolerant application," *2010 IEEE International Conference of Electron Devices and Solid-State Circuits (EDSSC)*, pp. 1–4, 2010.

[45] D. R. Kishor and V. K. Bhaaskaran, "Low power divider using vedic mathematics," *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 575–580, 2014.

[46] K. Manikantta Reddy, M. Vasantha, Y. Nithin Kumar, and D. Dwivedi, "Design of approximate dividers for error tolerant applications," *2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 496–499, 2018.

[47] L. Chen, J. Han, W. Liu, and F. Lombardi, "On the design of approximate restoring dividers for error-tolerant applications," *IEEE Transactions on Computers*, vol. 65, no. 8, pp. 2522–2533, 2016.

[48] J. Y. L. Low and C. C. Jong, "Non-iterative high speed division computation based on mitchell logarithmic method," *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 2219–2222, 2013.

[49] R. Zendegani, M. Kamal, A. Fayyazi, A. Afzali-Kusha, S. Safari, and M. Pedram, "Seerad: A high speed yet energy-efficient rounding-based approximate divider," *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1481–1484, 2016.

[50] O. Akbari, M. Kamal, A. Afzali-Kusha, and M. Pedram, "Dual-quality 4:2 compressors for utilizing in dynamic accuracy configurable multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 4, pp. 1352–1361, 2017.

[51] H. Pishro-Nik, *Introduction to Probability, Statistics, and Random Processes*. Kappa Research, LLC, 2014.

[52] S. Glen. Mean squared error: Definition and example. From StatisticsHowTo.com: Elementary Statistics for the rest of us! [Online]. Available: https://www.statisticshowto.com/probability-and-statistics/statistics-definitions/mean-squared-error/

[53] C. Liu, J. Han, and F. Lombardi, "An analytical framework for evaluating the error characteristics of approximate adders," *IEEE Transactions on Computers*, vol. 64, no. 5, pp. 1268–1281, 2015.

[54] J. Liang, J. Han, and F. Lombardi, "New metrics for the reliability of approximate and probabilistic adders," *IEEE Transactions on Computers*, vol. 62, no. 9, pp. 1760–1771, 2013.

[55] M. K. Ayub, O. Hasan, and M. Shafique, "Statistical error analysis for low power approximate adders," *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2017.

[56] S. Mazahir, O. Hasan, R. Hafiz, M. Shafique, and J. Henkel, "Probabilistic error modeling for approximate adders," *IEEE Transactions on Computers*, vol. 66, no. 3, pp. 515–530, 2017.

[57] T. Coe, T. Mathisen, C. Moler, and V. Pratt, "Computational aspects of the pentium affair," *IEEE Computational Science and Engineering*, vol. 2, no. 1, pp. 18–30, 1995.

[58] T. Coe and P. T. P. Tang, "It takes six ones to reach a flaw [pentium processor]," *Proceedings of the 12th Symposium on Computer Arithmetic*, pp. 140–146, 1995.

[59] J. O. Smith III, *Introduction to Digital Filters: with Audio Applications*. W3K Publishing, 2007.

[60] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-power digital signal processing using approximate adders," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 1, pp. 124–137, 2013.

[61] Z. Yang, A. Jain, J. Liang, J. Han, and F. Lombardi, "Approximate xor/xnor-based adders for inexact computing," *2013 13th IEEE International Conference on Nanotechnology (IEEE-NANO 2013)*, pp. 690–693, 2013.

[62] N. Zhu, W. L. Goh, and K. S. Yeo, "An enhanced low-power high-speed adder for error-tolerant application," *Proceedings of the 2009 12th International Symposium on Integrated Circuits*, pp. 69–72, 2009.

[63] D. Mohapatra, V. K. Chippa, A. Raghunathan, and K. Roy, "Design of voltage-scalable meta-functions for approximate computing," *2011 Design, Automation Test in Europe*, pp. 1–6, 2011.

[64] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," *DAC Design Automation Conference 2012*, pp. 820–825, 2012.

[65] X. Yang, Y. Xing, F. Qiao, Q. Wei, and H. Yang, "Approximate adder with hybrid prediction and error compensation technique," *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 373–378, 2016.

[66] J. Miao, K. He, A. Gerstlauer, and M. Orshansky, "Modeling and synthesis of quality-energy optimal approximate adders," *2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 728–735, 2012.

[67] R. Zendegani, M. Kamal, M. Bahadori, A. Afzali-Kusha, and M. Pedram, "Roba multiplier: A rounding-based approximate multiplier for high-speed yet energy-efficient digital signal processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 2, pp. 393–401, 2017.

[68] C.-H. Lin and I.-C. Lin, "High accuracy approximate multiplier with error correction," *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pp. 33–38, 2013.

[69] D. Baran, M. Aktan, and V. G. Oklobdzija, "Energy efficient implementation of parallel cmos multipliers with improved compressors," *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pp. 147–152, 2010.

[70] A. Momeni, J. Han, P. Montuschi, and F. Lombardi, "Design and analysis of approximate compressors for multiplication," *IEEE Transactions on Computers*, vol. 64, no. 4, pp. 984–994, 2015.

[71] C. Liu, J. Han, and F. Lombardi, "A low-power, high-performance approximate multiplier with configurable partial error recovery," *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–4, 2014.

[72] L. Chen, J. Han, W. Liu, P. Montuschi, and F. Lombardi, "Design, evaluation and application of approximate high-radix dividers," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, no. 3, pp. 299–312, 2018.

[73] S. Ullah, S. S. Murthy, and A. Kumar, "Smapproxlib: Library of fpga-based approximate multipliers," *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2018.

[74] F. Vaverka, R. Hrbacek, and L. Sekanina, "Evolving component library for approximate high level synthesis," *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 1–8, 2016.

[75] Brent and Kung, "A regular layout for parallel adders," *IEEE Transactions on Computers*, vol. C-31, no. 3, pp. 260–264, 1982.

[76] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Transactions on Computers*, vol. C-22, no. 8, pp. 786–793, 1973.

[77] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *JOURNAL OF THE ACM*, vol. 27, no. 4, pp. 831–838, 1980.

[78] T. Han and D. A. Carlson, "Fast area-efficient vlsi adders," *1987 IEEE 8th Symposium on Computer Arithmetic (ARITH)*, pp. 49–56, 1987.

[79] Aoki Laboratory. Hardware algorithms for arithmetic modules. Tohoku University. [Online]. Available: http://www.aoki.ecei.tohoku.ac.jp/arith/mg/algorithm.html

[80] R. K C and U. S. Kumar, "A comprehensive comparative analysis of parallel prefix adders for asic implementation," *Proceedings of the International Conference on Systems, Energy Environment (ICSEE) 2019*, 2019.

[81] K. Vitoroulis and A. J. Al-Khalili, "Performance of parallel prefix adders implemented with fpga technology," *2007 IEEE Northeast Workshop on Circuits and Systems*, pp. 498–501, 2007.

[82] J. G. Earle, "Latched carry save adder circuit for multipliers," U.S. Patent 3 340 388, 1965-07-12.

[83] Y. Pan, X. Jia, Z. Cheng, P. Ouyang, X. Wang, J. Yang, and W. Zhao, "An stt-mram based reconfigurable computing-in-memory architecture for general purpose computing," *CCF Transactions on High Performance Computing*, vol. 2, pp. 272–281, 2020.

[84] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, no. 5, pp. 349–356, 1965.

[85] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, no. 1, pp. 14–17, 1964.

[86] C. Baugh and B. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Transactions on Computers*, vol. C-22, no. 12, pp. 1045–1047, 1973.

[87] Fitter resource usage summary report. Intel. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/mapIdTopics/mwh1465496451103.htm

[88] Adaptive logic module (alm) definition. Intel. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/def_alm.htm

[89] F. Li, Y. Lin, and L. He, "Vdd programmability to reduce fpga interconnect power," *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, pp. 760–765, 2004.

[90] J. Anderson and F. Najm, "Active leakage power optimization for fpgas," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 3, pp. 423–437, 2006.

[91] G. Wang, S. Sivaswamy, C. Ababei, K. Bazargan, R. Kastner, and E. Bozorgzadeh, "Statistical analysis and design of harp fpgas," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 2088–2102, 2006.

[92] Power overview. Intel. [Online]. Available: https://www.intel.com/content/www/us/en/support/programmable/support-resources/power/pow-overview.html