NTNU

Norwegian University of
Science and Technology

Implementation of Empirical Mode Decomposition on an FPGA using Fixed Point Arithmetic

Lars Haugseng Andersen

Master of Science in Electronics
Submission date: July 2017
Supervisor: Bjørn B. Larsen, IES

Norwegian University of Science and Technology
Department of Electronic Systems

Contents

List of Figures

List of Tables

Abstract

This report will test how well the empirical mode decomposition (EMD) algorithm performs on a field-programmable gate array (FPGA), using solely fixed point arithmetic. The implementation is designed to receive a data set of N 16 bits data points, and return the different intrinsic mode function (IMF) components and the residue. The implementation is based on the naive EMD, using cubic spline interpolation to generate the envelopes when finding the different IMF components. With fixed point precision, there will be a certain error when storing a decimal number. By applying precision bits, effectively multiplying the number by a factor of 2 for every bit applied, this error decreases. The implementation is designed to be able to specify the amount of precision bits wanted for the cubic spline calculations, with a maximum amount of 22. The implementation is simulated and tested for different amount of bits, and the resulting IMF components are compared to software implementation results using floating point precision. For a data set containing 2048 samples of uniform distributed noise, the number of precision bits would need to be greater than 13. Fewer precision bits did not produce any meaningful result due to the error for the later IMF component being too large. The error is defined as the difference between the floating point, and the fixed point precision results. This error is compared to the floating point precision to gain a signal to noise ratio (SNR). For every new IMF extracted, the SNR decreases. The main reason for this, is because the cubic splines are made out of third order polynomials, which also applies to the error. This means that the error would escalate quickly when the distance between the extrema becomes large. This results in more precision bits needed for later IMF components, in order to keep the SNR constant. For real-time purposes, the input signal sampling frequency for this design can not be higher than 159 kHz for $N = 2048$. This sampling frequency becomes less for bigger window size, N, which was expected due to the complexity of the EMD algorithm.

Sammendrag

Denne raporten tester hvor godt empirical mode decomposition (EMD) fungerer på en field-programmable gate array (FPGA) ved kun å bruke fastkomma aritmetikk. Implementasjonen er designet for å ta i mot et datasett bestående av N 16 bits datapunkter, og returnere intrinsic mode function (IMF) komponentene pluss resten. Implementasjonen er basert på den naive EMD algoritmen, som bruker kubisk spline interpolering til å generere omslagene som blir brukt til å finne IMF komponentene. Med fastkomma presisjon, vil det kunne forekomme en avrundingsfeil når man skal representere desimaltall. Ved å legge til presisjons bits, vil verdien øke med en faktor på 2 for hvert ekstra bit. Dette vil føre til en mindre avrundingsfeil, som igjen fører til bedre presisjon. Antall presisjons bits for implementasjonen kan spesifiseres, maksimalt lik 22. Implementasjonen er simulert og testet for forskjellig antall presisjonsbits, og de resulterende IMF komponentene er sammenlignet med programvare implementering som bruker flyttalls presisjon. For et datasett med 2048 punkter med uniformt fordelt støy, må man ha mer enn 13 presisjonsbits for å få meningsfulle resultat for de siste IMF komponentene. Feilen på IMF komponentene er definert som forskjellen mellom fastkomma og flyttalls resultater. Denne feilen sammenlignes så med flyttalls resultatene for å finne et signal-til-støy-forhold (SNR). Hver IMF kompnent vil ha en lavere SNR enn den forrige. Hovedgrunnen til dette er at kubisk spline er bygget opp av tredje ordens polynomer, noe som også gjelder for feilen. Denne feilen vil eskalere raskt når avstanden mellom punktene blir stor. Dette betyr at flere presisjonsbits må legges til for å holde en konstant SNR, ettersom flere og flere IMF komponenter er funnet. For sanntid, må punktprøvingsfrekvensen for inngangssignalet være mindre enn 159 kHz ved en vindusstørrelse på 2048 punkter. Denne grensen blir mindre når antall punkter i inngangssignalet øker, noe som er forventet grunnet kompleksiteten til EMD algoritmen.

1. Introduction

This report presents a hardware implementation of the EMD [5] for a FPGA. It's an experimental approach to test how well the EMD performs relying solely on fixed point precision arithmetic in terms of precision, speed and area usage. EMD is a powerful tool for extracting IMF components from any arbitrary waveform. The algorithm is effectively a dynamic filter, and if performed on white noise, the IMF components become normally distributed, and their respective Fourier spectra becomes identical [12]. The EMD was introduced by Nordon E. Huang and his group at NASA in 1998 [5]. It was developed to extract information from non-linear and non-stationary data sets. EMD has been used in different fields, such as physics [3], image analysis [14], biomedicine [4, 7, 6, 8], mechanical health diagnosis [15] and others. The motivation behind implementing the algorithm in hardware, is to reduce the time consumption when performing EMD. Because fixed point operations perform much better in terms of speed compared to floating point operations, it will be used throughout the whole design. The downside of this, is that fixed point representation requires external bits in order to get a certain precision. The implementation will be tested using different amount of precision bits. The results will be compared to fixed point precision results as an SNR. Testing, synthesizing and simulations will be done in Quartus, and ModelSim respectively. The whole hardware implementation is written in Verilog, and tested in ModelSim using a custom made test bench. The hardware implementation will be using the integer arithmetic intellectual property (IP) core [2] multiplication, division, and addition/subtraction when synthesizing the design. Resource usage will be measured in number of Adaptive Logic Modules (ALM), registers, memory bits and digital signal processing (DSP) blocks used and a speed test will be performed for different window size, N. First off, the theory behind the EMD algorithm and the cubic spline interpolation will be presented. This will be the foundation of the whole implementation throughout the report. Following in

Chapter 3, presenting the approach on how to implement the algorithm in terms of arithmetic operations needed, how the data flows through the system, and how the numbers are represented in terms of bits. Lastly in Chapter 5, the implementation will be tested and the results compared to a software implementation with floating point precision.

2. EMD background and theory

This chapter will give an introduction to the theory behind the EMD algorithm [5]. Given an input signal $x[n] = x(nT_s)$, T_s being the sampling period, $x \in \mathbb{R}$, $n \in \mathbb{Z}$ and $n = [0 : N-1]$, EMD decomposes a signal $x[n]$ into K IMF components, $C_k[n]$, $k = [1 : K]$. The signal can be described by the sum of its IMF components and the leftovers referred to as the residue, $r_{K+1,0}[n]$. Equation 2.1 shows the result when EMD is applied to a signal $x[n]$

$$x[n] = \sum_{k=1}^{K} C_k[n] + r_{K+1,0}[n]. \tag{2.1}$$

For any given signal with N (also referred to as the window size) number of samples, the number of IMF components contained in the signal will be[13],

$$K \leq \log_2(N) \tag{2.2}$$

An IMF is an oscillating function that can have a time varying frequency and amplitude. It is defined by the following conditions as stated in [5, ch. 4]

- **1:** in the whole data set, the number of extremum and the number of zero crossings must either equal or differ at most by one
- **2:** at any point, the mean value of the envelope defined by the local maxima and the envelope defined by the local minima is zero.

The residue, can never be an IMF, and will either equal zero or be a monotonically increasing or decreasing function. EMD uses a process called sifting to identify the different IMF components. The process involves finding a curve that best describes the center of the signal $x[n]$ at any point. This is done by calculating an upper and a lower envelope, $u[n]$

and $l[n]$, that encapsulates the signal, and then calculating the mean, $\mu[n]$, of the two envelopes.

$$\mu[n] = \frac{u[n] - l[n]}{2} \tag{2.3}$$

Figure 2.1 illustrates a signal $x[n]$ and its respective upper and lower envelopes and the mean, $\mu[n]$.

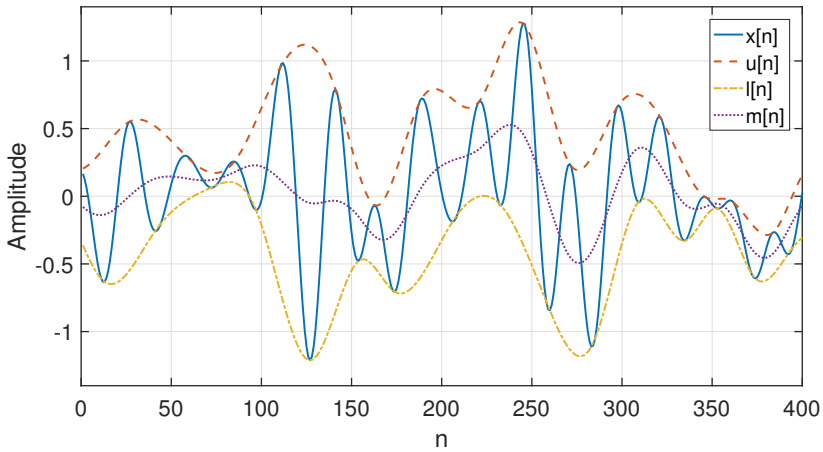

Figure 2.1.: Upper, lower and mean envelopes, $u[n]$, $l[n]$ and $\mu[n]$, of a signal $x[n]$.

$\mu[n]$ gives a good approximation of the middle of the signal $x[n]$. The envelopes are calculated by first identifying all the extrema in $x[n]$, and then interpolate between all the maxima to gain the upper envelope, and all the minima to gain the lower envelope. The goal of the interpolation, is to get a curve that represents the outer boundaries of the signal as accurate as possible. This report will be using cubic spline interpolation for this purpose. After the mean envelope is computed, it's subtracted from $x[n]$ to produce a signal $r_{1,1}[n]$ more symmetric with respect to zero,

$$r_{1,1}[n] = r_{1,0}[n] - \mu_{1,0}[n] \tag{2.4}$$

$r_{1,1}[n]$ is the result after one iteration of the sifting process, for finding the first IMF component, shown in Figure 2.2.

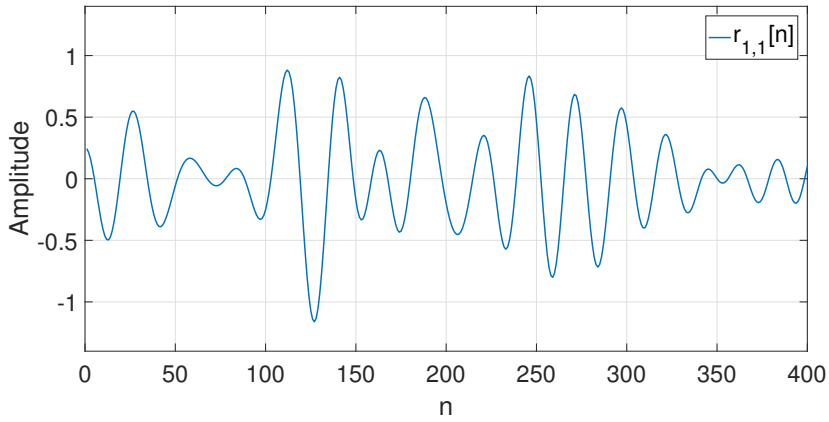

Figure 2.2.: The result after performing one iteration of the sifting process on $x[n]$, $r_{1,1}[n]$.

Clearly $r_{1,1}[n]$ has become more symmetric, and is one step closer to become an IMF. A flowchart of the EMD algorithm can be found in Figure 2.3. There are two iteration parameters, k and m. k indicates that the k'th IMF is being extracted from $x[n]$, and m is a parameter indicating how many sifting iterations performed. As the input signal, $x[n]$, enters the algorithm, it is assigned to $r_{1,0}[n]$, meaning the signal that the first IMF component is to be identified, and zero sifting iterations has been performed. $r_{1,0}[n]$ is then checked if it is monotonic or zero. If no, this means it contains at least one IMF, and the sifting process begins. The maxima and minima are identified, and the upper and the lower envelopes, $u_{1,0}[n]$ and $l_{1,0}[n]$ are calculated with cubic spline interpolation.

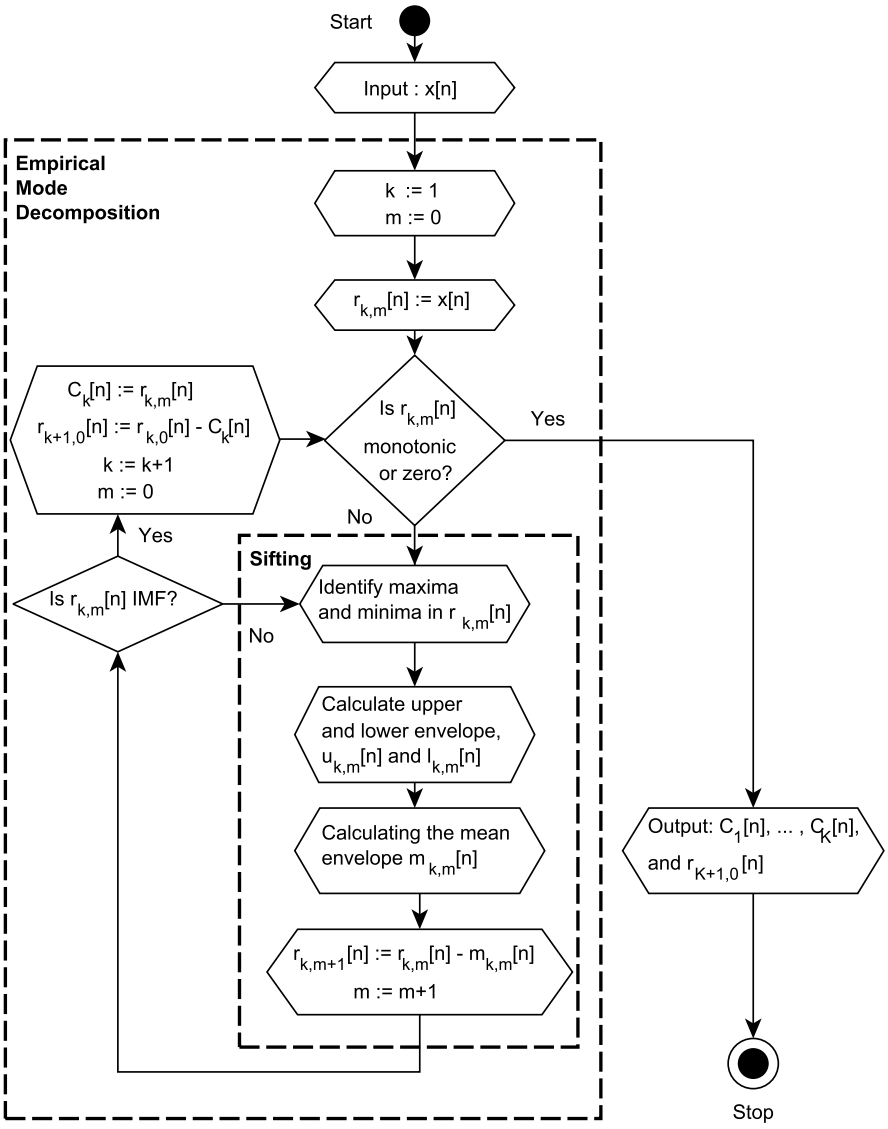

Figure 2.3.: Flowchart of the EMD algorithm.

The mean envelope $\mu_{1,0}[n]$ is calculated and then subtract from $r_{1,0}[n]$ to produce $r_{1,1}[n]$ as shown in Equation 2.4. m is incremented, and the first sifting iteration is finished. An IMF check is performed on $r_{1,1}[n]$, and if it does not satisfy the conditions, the second sifting iteration begins. After M_1 iterations the conditions are satisfied, and the sifting process is done. The obtained IMF components is then saved as $C_1[n]$. Now $r_{2,0}[n]$ is produced by subtracting $C_1[n]$ from $r_{1,0}[n]$ as shown in Equation 2.5.

$$r_{2,0}[n] = r_{1,0} - C_1[n] \tag{2.5}$$

The whole process is then repeated on $r_{2,0}[n]$ to find the next IMF component. After all the K IMF components has been identified, the leftovers $r_{K+1,0}[n]$ has either become a monotonic signal or zero, which is the residue.

2.1. Cubic Spline Interpolation

Interpolation is used to generate the envelopes in the sifting process. This section will present the theory behind Cubic spline interpolation as derived in[1]. A cubic spline is a curve designed to best fit a set of points with arbitrary spacing. It is constructed peacewise of I third-order polynomial curves, connecting a set of $I+1$ specified data points, $z_i = (x_i, y_i)$, $i = [0 : I]$. The characteristics of the cubic spline are[1]:

1. The spline passes through all specified data points.
2. First derivative continuity at interior point.
3. Second derivative continuity at interior point.
4. Boundary conditions specified at the free ends.

Figure 2.4 shows an example of 5 points and the splines connecting them all together. Here $I = 4$.

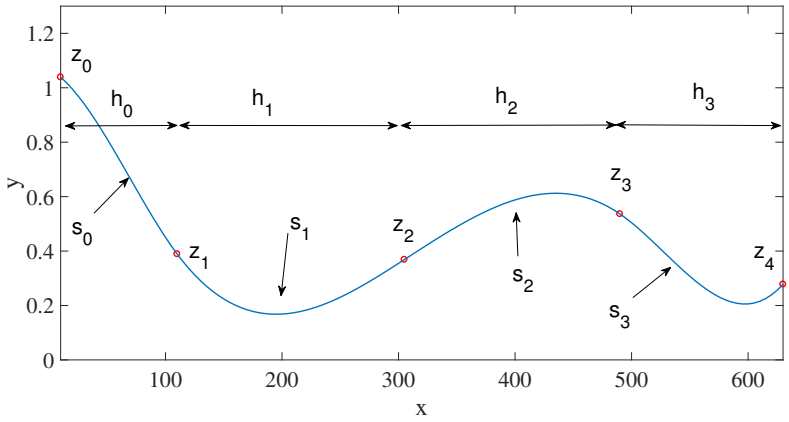

Figure 2.4.: Cubic spline connecting 5 data points, z_0 to z_4.

The different splines $s_i(x)$ are constructed by the third order polynomial equation in Equation 2.6.

$$s_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i \quad ; \; x \in [\,0, h_i\,) \tag{2.6}$$

where x_i is the x coordinate of point z_i, and a_i, b_i, c_i and d_i are the spline coefficients unique to each of the splines. Each spline connects two neighbour points, and the coefficients can be determined by the y coordinate of the two points, the spacing between them at the x-axis, $h_i = x_{i+1} - x_i$, and the second derivative at each point, y_i'' and y_{i+1}'' as shown in Equation 2.7 - 2.10.

$$a_i = \frac{(y_{i+1}'' - y_i'')}{6h_i} \tag{2.7}$$

$$b_i = \frac{y_i''}{2} \tag{2.8}$$

$$c_i = \frac{(y_{i+1} - y_i)}{h_i} - \frac{y_{i+1}'' h_i}{6} - \frac{y_i'' h_i}{3} \tag{2.9}$$

$$d_i = y_i \tag{2.10}$$

The second derivatives can be found by solving the governing equation for cubic splines [1], shown in Equation 2.11.

$$h_{i-1}y_{i-1}'' + 2(h_{i-1} + h_i)y_i'' + h_i y_{i+1}'' = 6[\frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}}] \tag{2.11}$$

To simplify the notation, A_i, B_i, C_i and D_i are substituted into the equation as shown in Equation 2.12,

$$A_i y_{i-1}'' + B_i y_i'' + C_i y_{i+1}'' = D_i \tag{2.12}$$

Where

$$A_i = h_{i-1} \tag{2.13}$$

$$B_i = 2(h_{i-1} + h_i) \tag{2.14}$$

$$C_i = h_i \tag{2.15}$$

$$D_i = 6\left[\frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}}\right] \tag{2.16}$$

This equation can be extended to a tridiagonal matrix, as shown in Equation 2.17, containing the information to calculate the second derivatives in every point z_i.

$$\begin{bmatrix} B_0 & C_0 & & & & & \\ A_1 & B_1 & C_1 & & & & \\ & A_2 & B_2 & C_2 & \ddots & & \\ & & & \ddots & & & \\ & & & & \ddots & C_{I-2} & \\ & & & & A_{I-1} & B_{I-1} & C_{I-1} \\ & & & & & A_I & B_I \end{bmatrix} \begin{bmatrix} y_0'' \\ y_1'' \\ \vdots \\ \vdots \\ \vdots \\ y_{I-1}'' \\ y_I'' \end{bmatrix} = \begin{bmatrix} D_0 \\ D_1 \\ \vdots \\ \vdots \\ \vdots \\ D_{I-1} \\ D_I \end{bmatrix} \tag{2.17}$$

Here B_0, C_0, A_I and B_I are initial conditions, and needs to be chosen. For this report, natural boundary conditions is used. The natural boundary conditions implies that $y_0'' = y_I'' = 0$, resulting in $B_0 = B_I = 1$, $C_0 = A_I = 0$, and $D_0 = D_I = 0$[1]. To solve this system, the Thomas algorithm [10] (also referred to as the tridiagonal matrix algorithm) is used[1]. The Thomas algorithm solves any tridiagonal matrix in $O(I)$ operations. This algorithm consists of two steps. The first step involves a forward sweep, calculating two new variables, $\hat{C}_i$ and $\hat{D}_i$, referred to as primes, where

$$\hat{C}_i = \begin{cases} \dfrac{C_i}{B_i} & ; i = 0 \\ \dfrac{C_i}{B_i - A_i \hat{C}_{i-1}} & ; i = 1, 2, ..., I-1 \end{cases} \tag{2.18}$$

and

$$\hat{D}_i = \begin{cases} \dfrac{D_i}{B_i} & ; i = 0 \\ \dfrac{D_i - A_i \hat{D}_{i-1}}{B_i - A_i \hat{C}_{i-1}} & ; i = 1, 2, ..., I \end{cases} \tag{2.19}$$

The second step is a backwards substitution to calculate the second derivatives as shown in Equation 2.20.

$$y_i'' = \begin{cases} \hat{D}_i & ; i = I \\ \hat{D}_i - \hat{C}_i y_{i+1}'' & ; i = I-1, I-2, ..., 0 \end{cases} \tag{2.20}$$

[1] A brief summary of the Thomas algorithm can be found at `https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm`

The spline coefficients can now easily be found by inserting the second derivative in Eq. 2.7, 2.8, 2.9 and 2.10, and the cibic spline can be calculated.

3. EMD implementation approach

The idea behind the implementation is shown in Figure 3.1. It consists of a sifting module which performs the sifting process, and a controller with access to block RAM. The implementation receives a data set $x[n]$, consisting of N 16-bits samples, N being a power of 2, and returns the IMF components and the residue at the output.

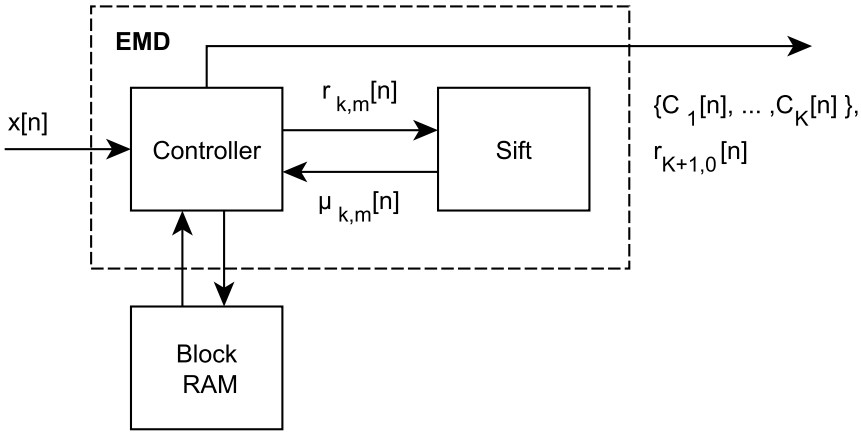

Figure 3.1.: Block diagram of the EMD implementation.

The sifting module performs the different steps within the sifting box illustrated by the flowchart in Figure 2.3, Chapter 2, except the last step involving subtracting the mean envelope. This effectively means that for an input signal $r_{k,m}[n]$, the module calculates the mean envelope, $\mu_{k,m}[n]$. The controller subtracts the mean envelope, and then checks if the new data set has become an IMF. If not, the controller sends it back to the sifting module to perform another sift iteration. As mentioned, the controller

has access to block RAM which is used to temporarily store the different $r_{k,m}[n]$ generated after each sift iteration. Each of the $r_{k,0}[n]$ is also stored, and accessed when $r_{k,m}[n]$ has become an IMF component. The memory is then updated by subtracting the newly discovered IMF from $r_{k,0}[n]$. Before the sifting process begins, the controller checks whether $r_{k,0}[n]$ is a monotonic component or zero. If so, this is the residue, which no more IMF component can be extracted. Whenever an IMF, $C_k[n]$, it is located, it's automatically sent to the output of the module. The implementation finishes when the residue is identified.

The sifting module is implemented as illustrated by Figure 3.2. First the extrema from the input data set $r_{k,m}[n]$, are identified. Then the maxima and minima are separated, and directed to their own cubic spline module to generate the upper and the lower envelopes, $u_{k,m}[n]$ and $l_{k,m}[n]$ respectively.

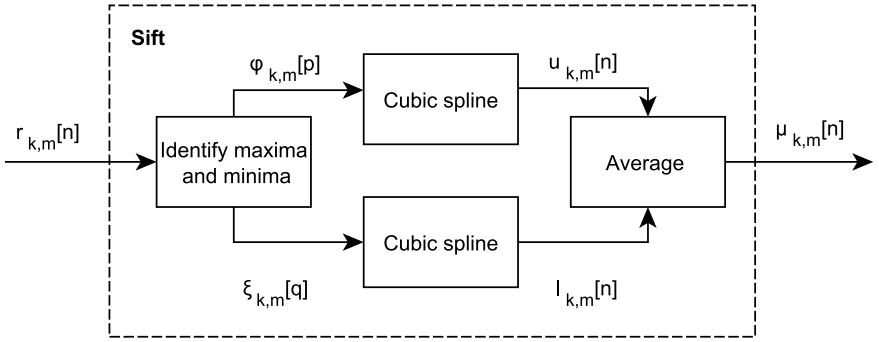

Figure 3.2.: Block diagram of the sift implementation.

$\varphi_{k,m}[p]$ contains the maxima of $r_{k,m}[n]$, and $\xi_{k,m}[q]$ contains the minima. To ensure that the splines generated are of length N, the first and the last sample of $r_{k,m}[n]$ are marked as both a maxima and a minima. p and q are variables ranging from 0 to $P_{m,k} - 1$, and 0 to $Q_{m,k} - 1$ where $P_{m,k}$ and $Q_{m,k}$ are the number of maxima and minima of $r_{k,m}[n]$. The cubic spline module consists of 5 steps as discussed in Section 2.1. Figure 3.3 shows the block diagram of the cubic spline implementation.

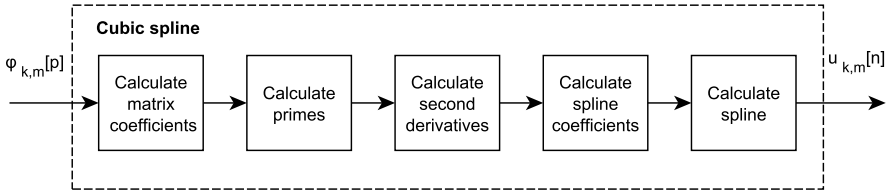

Figure 3.3.: Block diagram of the cubic spline implementation.

First the matrix coefficients needed for the matrix in Equation 2.17 are calculated. Then the tridiagonal matrix algorithm is performed by first calculating the primes in Equation 2.18 and 2.19, and the calculate the second derivatives as in Equation 2.20. Then the spline coefficients are calculated as in Equation 2.7 - 2.10, and used in the last step to calculate each of the splines as in Equation 2.6. Interpolation is costly in terms of speed, propagation delay and area usage, and will be the main optimization area. A more detailed implementation of the Cubic spline interpolation will be presented in the next section.

3.1. Cubic spline hardware implementation

This section will present a more detailed strategy on how the cubic spline interpolation is implemented in terms of hardware. The implementation is based on the theoretical presentation in Section 2.1. Every variable is represented as an array, $A[i]$, containing I coefficients, $A[i] = \{A_i\}_{i=0}^{I-1}$. Each of the splines $s_i(x)$ from Equation 2.6, are represented by a discrete variable w_i, as shown in Equation 3.1,

$$\begin{aligned} s_i[w_i] &= s_i(w_i T_s) \\ &= a_i(w_i T_s)^3 + b_i(w_i T_s)^2 + c_i(w_i T_s) + d_i; \\ &\quad w_i \in [0, 1, ..., h_i - 1] \end{aligned} \tag{3.1}$$

where T_s is the sampling period of the input signal $x[n]$. Here h_i is the number of samples between two neighbour points. For illustration, the cubic spline module for calculating the upper envelope $u_{k,m}[n]$ in figure 3.2, is used as an example. The two cubic spline modules are an exact

copy of each other, and thus only one of them is presented. To simplify the notation, the iteration parameters k and m are not noted.

The hardware implementation of the cubic spline is divided into five calculation steps, also referred to as modules. The different steps are shown in Figure 3.4. Here, the input, $\varphi[i]$, is an array containing $I+1$ maxima, which represents the y coordinate of the different data points $\{z_i\}_{i=0}^{I}$. $h[i]$ is an array containing the distance between each of the maxima in terms of samples. The output $u[n]$ is an array with I different splines concatenated, as shown in Equation 3.2.

$$\begin{aligned} u[n] = \{&s_0[0], s_0[1], ..., s_0[h_0 - 1], s_1[0], s_1[1], ..., \\ &s_1[h_1 - 1], ..., s_{I-1}[0], s_{I-1}[1], ..., s_{I-1}[h_{I-1}]\} \end{aligned} \tag{3.2}$$

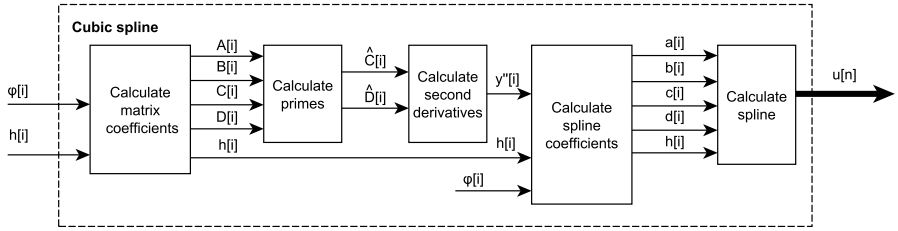

Figure 3.4.: Cubic spline hardware implementation overview.

The five calculation steps are:

1. Calculating $A[i]$, $B[i]$, $C[i]$, and $D[i]$ containing all the matrix coefficients (Equation 2.13 - 2.16).

2. Calculating the primes $\hat{C}[i]$ and $\hat{D}[i]$ from the tridiagonal matrix algorithm (Equation 2.18, 2.19).

3. Backward substitution to find the second derivatives $y''[i]$ (Equation 2.20).

4. Calculate the spline coefficients $a[i]$, $b[i]$, $c[i]$ and $d[i]$ (Equation 2.7 - 2.10).

5. Calculating the I splines from Equation 3.1, and concatenate them as in Equation 3.2 to receive the upper envelope, $u[n]$.

Each of these five calculation steps will be presented in terms of arithmetical operations needed to perform the calculations. Figure 3.5, 3.6, 3.7, 3.8 and 3.9 illustrates this respectively. The local critical path, meaning the longest path delay through each of the modules will be highlighted in red.

1) Calculating the matrix coefficients

Figure 3.5 shows the the different arithmetic operations needed to calculate the different matrix coefficients. The $<<$ represents bit shift to the left, effectively meaning multiplying by a factor of 2. The critical path within this module consists of 2 subtractions, 1 division and 1 multiplication.

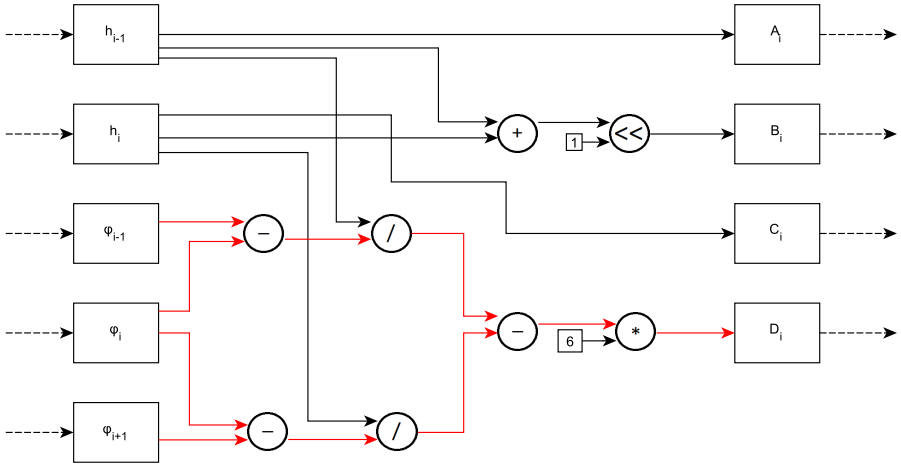

Figure 3.5.: Signal propagation for finding the matrix coefficients (Equation 2.13 - 2.16).

Division is a very costly operation in terms of hardware. It requires the most resources of all the arithmetic operations in this design, and has the most impact on the critical path. It turns out that for calculating the

spline coefficient, no division is needed at all. By multiplying by h_{i-1} and h_i to each side of the governing equation in Equation 2.11, the division disappears completely, resulting in new matrix coefficients, $\widetilde{A}_i$, $\widetilde{B}_i$, $\widetilde{C}_i$, $\widetilde{D}_i$.

$$\widetilde{A}_i = h_i h_{i-1}^2 \tag{3.3}$$

$$\widetilde{B}_i = 2(h_i h_{i-1}^2 + h_i^2 h_{i-1}) \tag{3.4}$$

$$\widetilde{C}_i = h_i^2 h_{i-1} \tag{3.5}$$

$$\widetilde{D}_i = 6\left[(\varphi_{i+1} - \varphi_i)h_{i-1} - (\varphi_i - \varphi_{i-1})h_i\right] \tag{3.6}$$

where φ represents y. Thus no division is required for calculating these new matrix coefficients, the design was still implemented as shown in Figure 3.5. This decision was made because the new matrix coefficients required too many bits to be represented. The amount of bits needed to represent a number will be discussed in Chapter 4. Also, there are two main reasons to try to avoid division in hardware. One is to reduce the total resource usage, and the other is to reduce the critical path of the whole system. This path is not found in this module, but in the module calculating the primes. If the new coefficients where to be used, the change would only results in an increase in bits needed to represent the coefficients. Overall this will result in an increased resource usage which would outweigh the resources saved from not doing the division.

2) Calculating the primes

The prime calculation is the bottleneck of this EMD implementation. The signal propagation path is shown in figure 3.6. The critical path of this module consists of 2 subtractions, one division and 2 multiplications, and is the longest path in the whole system.

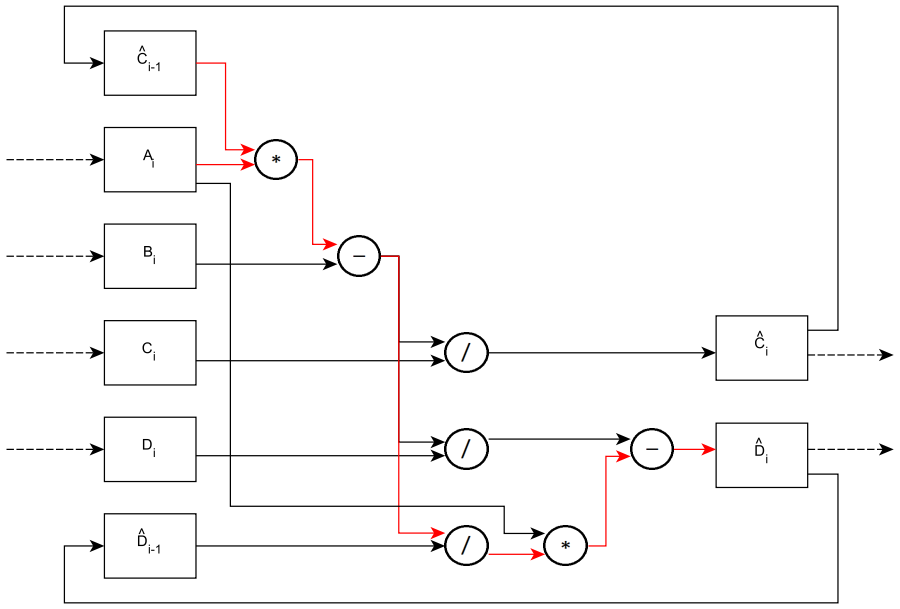

Figure 3.6.: Signal propagation for finding the primes (Equation 2.18, 2.19).

In this design an intellectual property (IP) named LPM_DIVIDE [2, pp. 29 - 34] is used for division. This divider can perform division in one clock cycle, for inputs up to 64 bits. Because of this input bit limit, it is preferred to keep the number of bits needed by the input, low. This makes room for more precision bits resulting in an increased precision when performing the division. Because of this, the setup in Figure 3.6 does not directly follow Equation 2.19 when calculating $\hat{D}_i$. Instead, the representation is split up into two divisions as shown in Equation 3.7

$$\hat{D}_i = \frac{D_i}{B_i - A_i\hat{C}_{i-1}} - \frac{\hat{D}_{i-1}}{B_i - A_i\hat{C}_{i-1}} \times A_i \tag{3.7}$$

The precision bits are necessary to maintain a certain precision when doing fixed point arithmetic, and will be discussed in Chapter 4. By splitting the equation into two divisions, A_n can be multiplied after the division is finished.

3) Calculating the second derivatives

For calculating the second derivatives, only one multiplication and one subtraction is needed.

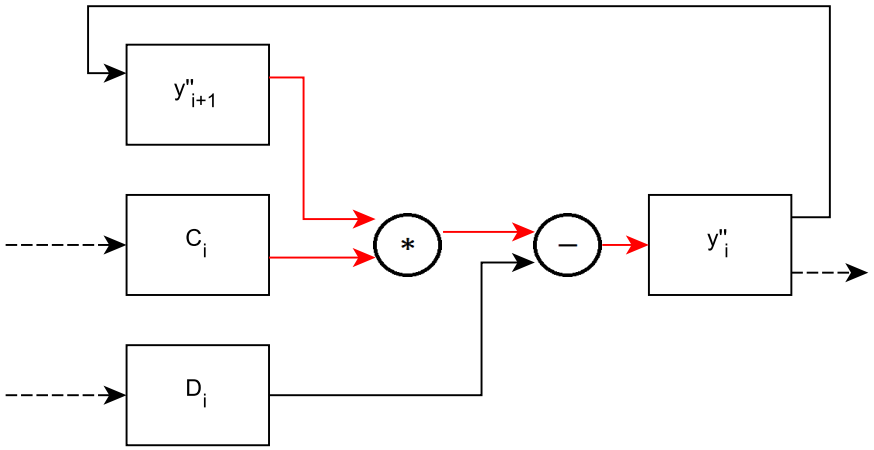

Figure 3.7.: Signal propagation for finding the second derivative (Equation 2.20).

4) Calculating the spline coefficients

Calculating the spline coefficients requires the most resources out of the five modules, shown in Figure 3.8. It consists of 4 subtractions, 4 divisions, 3 multiplications, and one bit shift right. The critical path of this module consists of 2 subtractions, one division and one multiplication. The $>>$ represents a bit shift to the right. This effectively divides the represented value by 2 for every right shift. The shifting operation is a so called free operation in hardware in terms of speed. This is because it only involves removing the least significant bit, and adding a new bit set to zero as the most significant bit. If the number to be shifted is odd, the result will be 0.5 less than the results from a real division by 2. This difference can be reduced by adding precision bits, which will be discussed in Chapter 4.

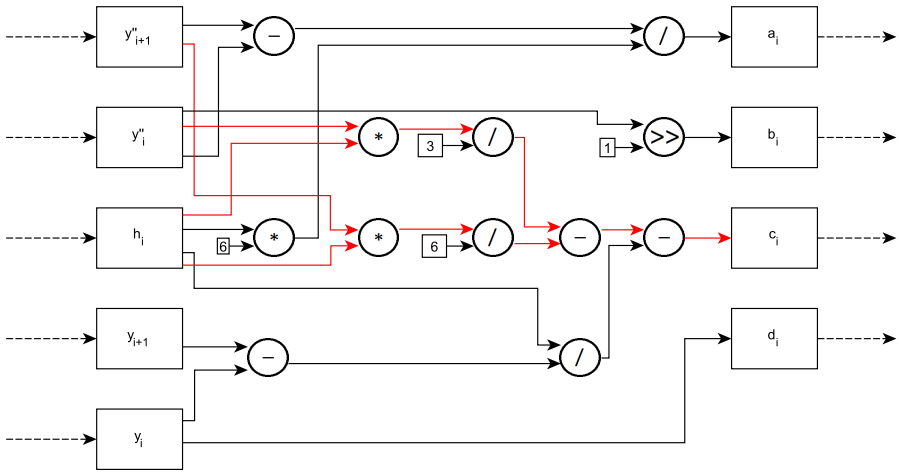

Figure 3.8.: Signal propagation for finding the spline coefficients (Equation 2.7 - 2.10).

5) Calculating the splines

The last part of the cubic spline module is the spline calculation. Every coefficient a_i, b_i, c_i, and d_i is used together to produce one unique spline $s[w_i]$, containing a total of h_i samples, where h_i is the number of samples between two maxima, φ_i and φ_{i+1}. To calculate each of these samples in $s[w_i]$, there has to be a counter counting through the interval $w_i = [0, 1, ..., h_i - 1]$. Figure 3.9 shows this counter which receives the spacing h_i, and outputs the array w_i. Every array is represented as a bold line, and all the multiplications performs element-wise multiplication. This means that for any two input arrays $\bar{A}_i = [\bar{A}_0, \bar{A}_1, ..., \bar{A}_{h_i-1}]$ and $\bar{B}_i = [\bar{B}_0, \bar{B}_1, ..., \bar{B}_{h_i-1}]$ the multiplication result will be $\bar{C}_i = [\bar{A}_0\bar{B}_0, \bar{A}_1\bar{B}_1, ..., \bar{A}_{h_i-1}\bar{B}_{h_i-1}]$.

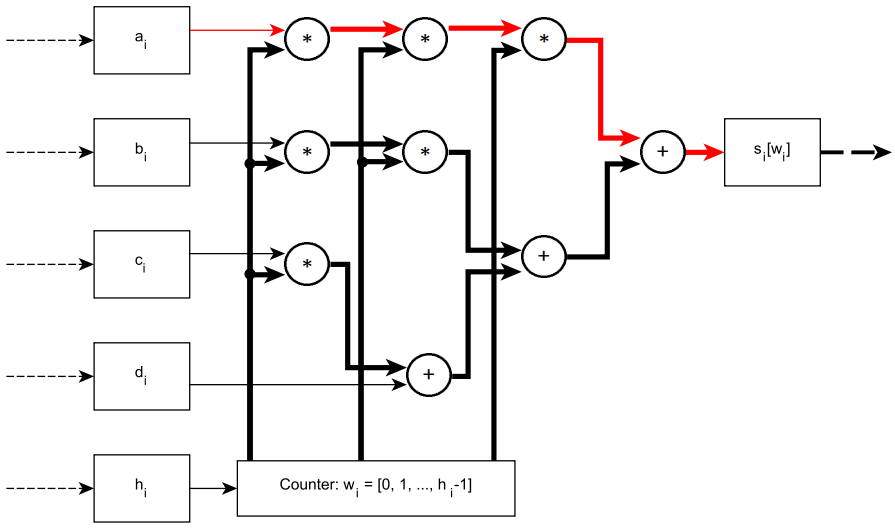

Figure 3.9.: Signal propagation for calculating the splines (Equation 2.6).

The critical path of this module consists of 3 multiplications and one addition.

3.1.1. Arithmetic operation requirement summary

Table 3.1 summarizes the arithmetic operations required by each of the modules, as well as the total requirement by the whole cubic spline module. Each of the five steps are referred to with a shortened name. The shortened name of step one to five follows respectively: CMC, CP, CSD, CSC, CS.

Operation	CMC	CP	CSD	CSC	CS	Total
Addition/Subtraction	4	2	1	4	3	14
Multiplication	1	2	1	3	6	13
Division	2	3	0	4	0	9
Bit shift left/right	1	0	0	1	0	2

Table 3.1.: Amount of arithmetic operations required by each of the five steps in the cubic spline calculation.

3.2. Data flow

This section will present how the data flows through the sifting module in figure 3.1. The data flow relies on registers, first in first out (FIFO) and first in last out (FILO) queues. The examples will only follow the upper envelope generation from Figure 3.2. For simplicity, the iteration parameters k and m are not noted in the examples. Figure 3.10 shows the three stages of computation within the sift module.

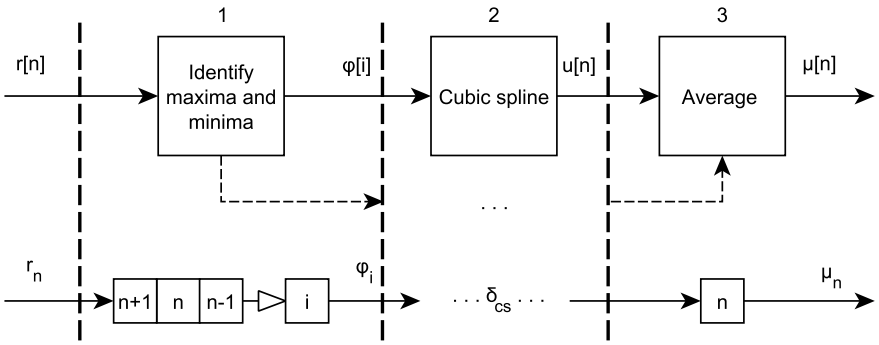

Figure 3.10.: Data flow through the sift module.

The implementation is pipelined, with each of the stages illustrated with the dotted lines. A pipelined implementation is a technique that allows multiple processes overlap in execution. More detailes about pipelining can be found in [9, ch. 4.5]. Each of the stages has different execution time. The blocks with the n indexation, represents a register containing the respective n'th sample, in case of stage one, the n'th sample of the input signal $r[n]$. The arrow pointing to i indicates an index change. The first stage stores 3 samples r_{n+1}, r_n and r_{n-1} and marks r_n as a maxima if the two neighbour samples are smaller. Every clock, the registers values are being shifted to the right, and a new sample enters. When a maxima, φ_i, is identified, it enters the cubic spline module in stage two. δ_{cs} indicates the total execution time for the cubic spline module, which will be presented later in this section. The implementation of the cubic spline module is also pipelined, as shown in Figure 3.11. Each of the five steps

presented in Section 3.1 are the different stages in the pipeline.

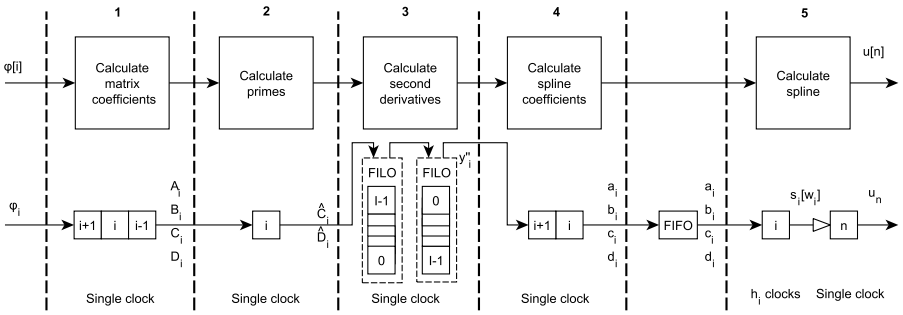

Figure 3.11.: Data flow through the cubic spline module.

Stage 1 uses a single clock cycle to produce the i'th matrix coefficients, A_i, B_i, C_i, D_i, after an initial delay of three cycles needed for the data to clock through the three registers. Calculating the primes in stage 2, has an initial delay of 1 clock cycle, before the primes, $\hat{C}_i$ and $\hat{D}_i$, enters the next stage one by one each respective clock cycle. Stage 3, calculating the second derivatives consists of 2 FILO queues. The first queue is needed to gather all the data points, in order to perform the backwards substitution in Equation 2.20. The FILOs reverse the order of the data points, thus a second queue is needed to reverse the order once more. There will be a hold of $2I$ clock cycles from the first primes, $\hat{C}_0$ and $\hat{D}_0$, enters, before y''_0 returns. The remaining second derivatives then follows one by one each respective clock cycle. Stage 4, calculating the spline coefficients, has the same behaviour as stage 1 and 2, with an initial delay of 2 clock cycles. Stage 5 calculates a spline segment, $s_i[w_i]$ (Equation 3.1), for each of the coefficients, a_i, b_i, c_i and d_i. Each spline $s_i[w_i]$ consists of h_i samples, and uses one clock cycle to produce each respective sample, h_i clocks in total. Each of the spline segments are returned as a part of $u[n]$, as shown in Equation 3.2. The initial delay of stage 5 is one clock cycle, before producing a sample of $u[n]$ every clock cycle respectively. FIFO queues are being used between stage 4 and 5 to temporarily store the spline coefficients when the different spline segments are being calculated. Because the FIFOs are used only as temporary storage, they're not considered a

stage of the pipeline. The FIFOs introduces a delay of 2 clock cycles for the first sample, and 1 clock cycle every time it outputs a sample.

Now how to find δ_{cs}, being the total execution time of the cubic spline module. Lets consider the last maxima entering the module, φ_I. Defining δ_{cs} as the time it takes from this sample enters, to the last sample of $u[n]$, u_{N-1}, leaves the module, it can be derived as followed. It will take a maximum of 4 clock cycles before $\hat{D}_I$ enters the first FILO in step 3. Then another I cycles to calculate the second derivatives, and store them in the second FILO. Lastly it will take 5 clock cycles before the first spline coefficients, a_0, b_0, c_0 and d_0 are stored in the spline calculation module in stage 5. The last sample of the upper envelope $u[n]$, u_{N-1}, will then leave the module after $N+I-1$ clock cycles (because the FIFO is accessed $I-1$ times after a_0, b_0, c_0 and d_0). Summing all the delays results in a δ_{cs} as shown in Equation 3.8

$$\delta_{cs} = 8 + 2I + N \tag{3.8}$$

This delay will occur for every cubic spline calculation. For this implementation, with N being a power of 2, and the first and last sample in the data set being a maxima, the maximum number of maxima contained in the data set $I_{\text{max}} = \frac{N}{2} + 1$. By inserting I_{max} in Equation 3.8 gives us the maximum delay for any data set $\delta_{cs\_\text{max}} = 2N + 10$ clock cycles.

4. Fixed point bit representation

When dealing with fixed point arithmetic, extra bits may be needed when performing multiplication and addition/subtraction to prevent overflow. Overflow happens when trying to represent a number with too few bits. For instance, trying to represent 16 as unsigned, 5 bits are needed (10000_2), but if there's only 4 available, the fifth bit will be chopped off, leaving behind the remaining 4, resulting in a value of 0 (0000_2). When two unsigned numbers are multiplied together, each number being represented with n and m bits respectively, the result would need $n+m$ bits in order to be able to represent all the possible results. When adding or subtracting, the result would need $n+1$ bits if $n > m$ or $m+1$ bits if $n < m$. The extra bits added in the result, will be referred to as extension bits. Fixed point numbers can only represent whole numbers, which can be a problem if decimal precision is needed. A way to increase the precision is to multiplying the represented number by 2, by adding an extra bit as the least significant bit (lsb), referred to as a precision bit. Figure 4.1 shows a 16 bit unsigned sample, with L extension bits and M precision bits respectively.

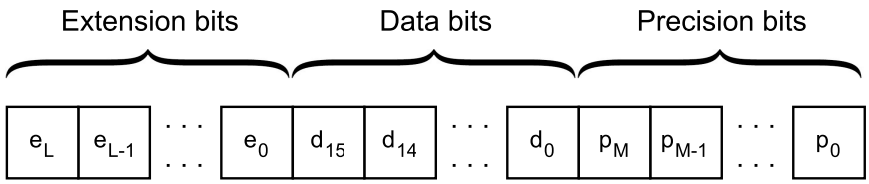

Figure 4.1.: Fixed point bit representation structure.

For every decimal number, A, represented as fixed point, A_{fix}, the precision will be within a certain range E, depending on the number of precision bits, M.

$$A_{\text{fix}} = A - E \tag{4.1}$$

where the error, E, being within the range shown in Equation 4.2.

$$\frac{1}{2^M} \geq E \geq 0 \tag{4.2}$$

This error will impact the result when doing multiplication, addition/-subtraction or division. More details about this can be found in Appendix A. For this implementation, it means that more precision bits gives better results, but at the cost of an increase in area usage. Table 4.1 gives an overview on how many bits that are used for each of the outputs of the different modules in Figure 3.4. The shortened version of the names used in Table 3.1 are used to represent the different modules.

CMC	Bits	CSC	Bits
$A[i]$	$\log_2(N)$	$a[i]$	$16 + 2M_{cs}$
$B[i]$	$\log_2(N) + 2$	$b[i]$	$16 + 2M_{cs}$
$C[i]$	$\log_2(N)$	$c[i]$	$16 + M_{cs}$
$D[i]$	$4 + 16 + M_m$	$d[i]$	$16 + M_m$
CSD	Bits	CP	Bits
$y''[i]$	$16 + M_{cs}$	$\hat{C}[i]$	M_{cs}
		$\hat{D}[i]$	$16 + M_{cs}$

Table 4.1.: Number of bits used to represent each of the outputs of the different modules in the cubic spline calculation.

This implementation assumes that $x[n]$ consists of 16 bit samples. In Figure 4.1, this means that the $Data\ bits$ are 16 bits wide. In Table 4.1 this is listed as $'' + 16''$ in the "Bits" column. Every number before the $'' + 16''$ is the extension bits, and the number after is the number of precision bits. There are two kinds of precision bits in Table 4.1, M_{cs} and M_m. M_{cs} is the amount of precision bits specified in the test bench, used for the cubic spline calculation. This means for every decimal number generated inside the module, M_{cs} precision bits are being used. M_m is the amount of precision bits used by the block ram in Figure 3.1. This ups the precision of the temporary storing, and is needed because $\mu_{k,m}[n]$ contains decimal

numbers. The only reason why M_m is specified, and not just set to equal M_{cs}, is to reduce the total memory requirement. The temporary storing does not need the same precision as needed when calculating the cubic spline parameters.

Now, let's go back to Section 3.1. Consider Equation 3.7 , and how it differs from Equation 2.19. Of course these equations are the same, but re-ordered in terms of operations. This re-ordering has an impact on the number of bits required to perform the different operations. For instance consider the numerator and the denominator in Equation 2.19 and call them σ, and τ respectively. $\sigma = D_i - A_i\hat{D}_{i-1}$ and $\tau = B_i - A_i\hat{C}_{i-1}$. In terms of hardware, both σ and τ would be the input of a division circuit. To represent σ, a total of $1+\log_2(N)+16+M_{cs}$ bits is needed for $N \geq 16$. τ would need $\log_2(N) + 2 + M_{cs}$ bits. The precision of the result after a fixed point division, M_{result}, is determined by the difference in precision bits of the numerator, M_σ and denominator, M_τ.

$$M_{\text{result}} = M_\sigma - M_\tau \tag{4.3}$$

Because M_σ and M_τ both equals M_{cs}, the resulting precision bits would equal 0. For M_{result} to be equal M_{cs}, σ would need $1+\log_2(N)+16+2M_{cs}$ bits. With the limitation on the LPM_DIVIDE IP core divider, of a maximum of 64 bits at the input, this will give a maximum of 18 bits with a window size $N = 2048$, $64 = 1 + 11 + 16 + 2 \cdot 18$. This precision will of course decrease with bigger N.

Now consider Equation 3.7. Here the denominator are the same as τ, but the numerators only requires $4 + 16 + M_m$ bits. To get M_{cs} precision after the division, $D[i]$ would need $4 + 16 + 2 \cdot M_{cs}$ bits. This means that M_{cs} could be a maximum of 22 bits, independent of the window size N, $64 = 4 + 16 + 2 \cdot 22$. The number of bits required for the denominator still depends on N, but since N has to be huge in order to reach the limit, $\log_2(N) + 2 + 22 = 64$, this is not of any concern.

5. Testing and results

The testing is performed using Matlab to generate ideal EMD results with floating point precision, Quartus Prime to analyse and synthesize the Verilog hardware description language (HDL) code, and ModelSim to run a register transfer level (RTL) simulation with specified precision. The whole testing environment and setup is shown in Figure 5.1

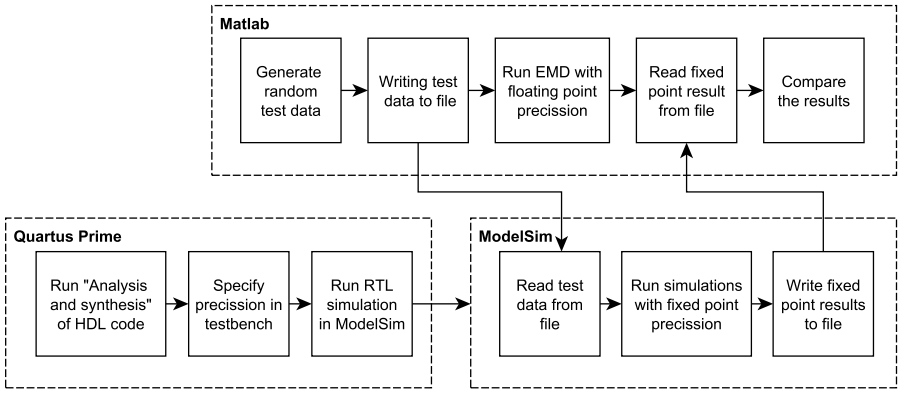

Figure 5.1.: Testing environment for receiving the floating point precision IMF results, fixed point precision simulation results, and comparing them to each other.

The whole simulation in ModelSim is defined by a test bench. The test bench contains information about how many precision bits the module is going to use, the input data for the module, and what results to store. First off, the HDL code is analyzed and synthesized in Quartus. Then the RTL simulation is called and executed in ModelSim. Meanwhile, a random data set is generated in Matlab, and saved to a $.dat$ file. This file is accessed in the RTL simulation, and the resulting IMF components

are written to another $.dat$ file. The result file is then read in Matlab and compared to the floating point precision IMF components.

5.1. IMF results

The data set $x_{\text{test}}[n]$ was used for testing, and consists of 2048 samples of uniform distributed noise, shown in Figure 5.2.

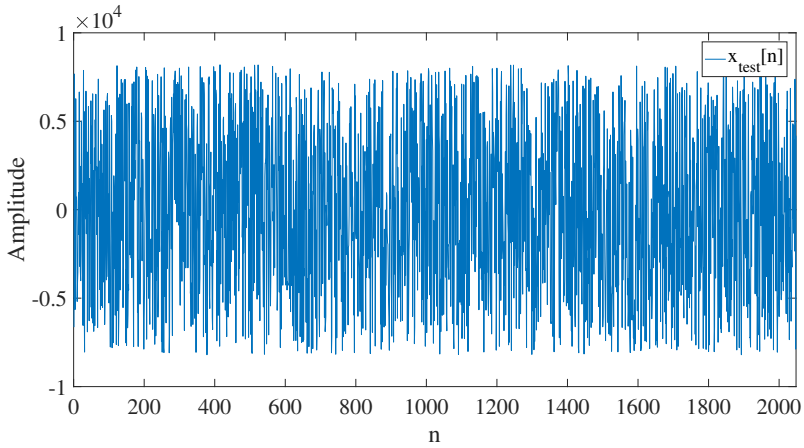

Figure 5.2.: Test data, $x_{\text{test}}[n]$, consisting of 2048 samples of uniformly distributed noise.

The data has a maximum amplitude of 2^{13} which is within the range of a 16 bit signed representation ranging from $2^{15} - 1$ to -2^{15}. Each IMF component received from the RTL simulation is noted as $\widetilde{C}_k[n]$. The floating point precision results, fixed point precision results and the difference between them are shown in Figure 5.3. The simulation was performed using 22 precision bits, $M_{cs} = 22$, and 21 memory bits, $M_m = 21$ (will not change for any test results throughout the rest of this report), which is the maximum for this deign.

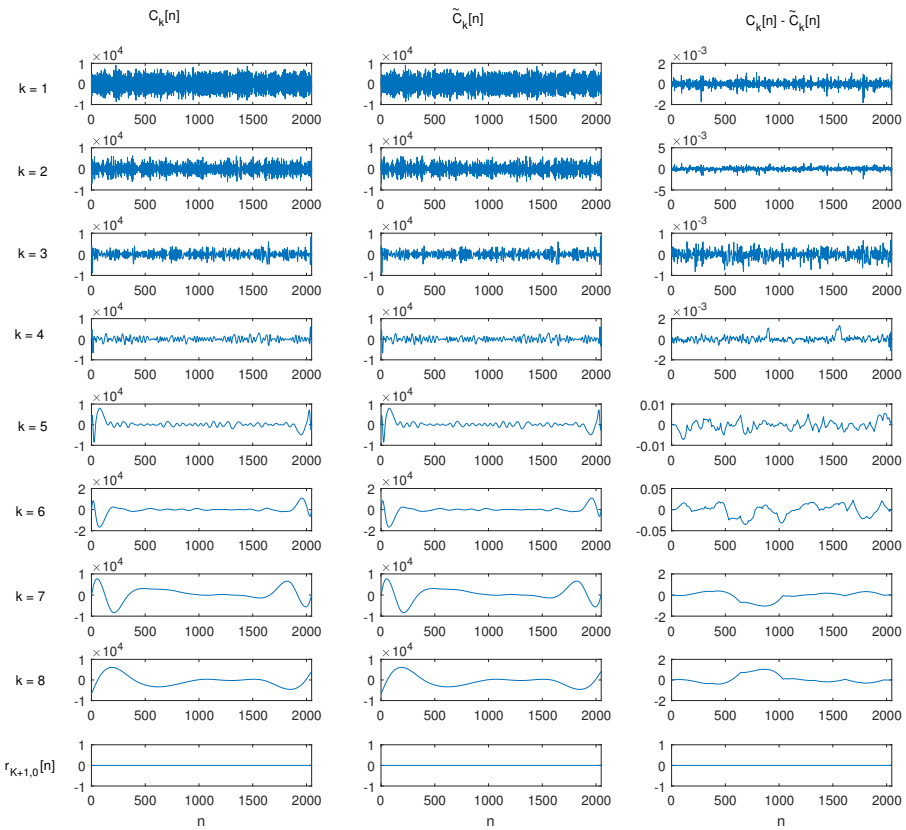

Figure 5.3.: IMF components of $x_{\text{test}}[n]$. Floating point precision on the left, fixed point precision in the middle, and the difference between them on the right.

To measure how much the fixed point precision results, equals the floating point results, SNR_k, is calculated for each of the IMF components, where

$$\text{SNR}_k = \frac{\sum_{n=1}^{N}(C_k[n])^2}{\sum_{n=1}^{N}(C_k[n] - \widetilde{C}_k[n])^2} \tag{5.1}$$

Figure 5.4 shows the different SNR in decibel for every IMF component, with different precision bits, M_{cs} ranging from 13 to 22.

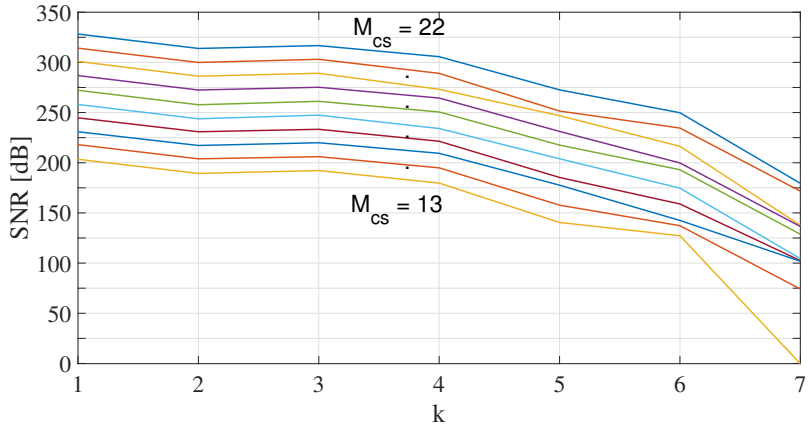

Figure 5.4.: SNR for all the IMF components, with precision bits, M_{cs}, ranging from 13 to 22.

The 8'th IMF is not plotted because it turned out that no sifting was performed to find it. With $M_{cs} = 13$, the energy of the error from the 7'th IMF, is almost as big as the IMF itself. With an error that big, this means that the IMF does have any meaningful value. By increasing the number of precision bits, the SNR increases by about 14 dB for every extra precision bit. $M_{cs} = 22$ is the highest precision available in this design, and has a SNR of about 175 dB for the 7'th IMF. An interesting observation is that it's not the amount of sifting performed on the signal that has the most impact on the SNR, it's all about the distance between the extrema when performing sift. Figure 5.5 shows the average number of samples between the maxima, $h_{\text{avg}}[k] = \{h_{\text{avg},k}\}_{k=1}^{K}$, and the total number of sift iterations performed for every IMF component, $m[k] = \{m_k\}_{k=1}^{K}$, where

$$h_{\text{avg},k} = \frac{\sum_{i=0}^{I_k-1} h_{i,k}}{I_k - 1} \tag{5.2}$$

and I_k being the total number of maxima in each IMF component.

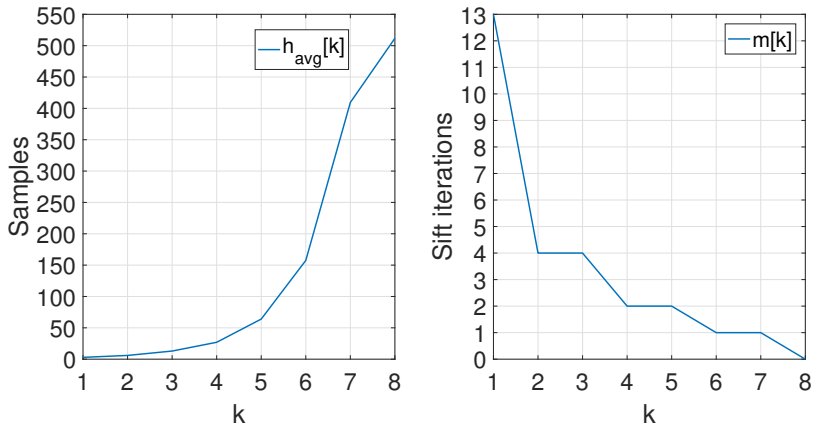

Figure 5.5.: To the left, the average number of samples between the maxima for every IMF component, $h_{\text{avg}}[k]$, and to the right, the number of sift iterations performed to find them, $m[k]$.

The reason why the distance is the leading factor for decreasing SNR, has to do with the cubic splines. Since the splines are constructed as a third order polynomial, the error will also be a third order polynomial. This means that better precision is needed for the spline coefficients, a_i, b_i, c_i, and d_i, (Equation 2.7 - 2.10) in order to maintain the same SNR as the distance between the extrema increases.

5.2. Area usage

The area usage is measured by number of adaptive logic modules (ALMs), registers, memory bits and digital signal processing units (DSPs) used when compiling the design in Quartus Prime. The usage will be presented in terms of resources per sample, $R'[N]$,

$$R'[N] = R[N]/N \tag{5.3}$$

where $R[N]$ is the total usage of resources for different window size N. Figure 5.6 shows the resource usage per sample for ALMs, registers and memory bits, $R'_{\text{ALM}}[N]$, $R'_{\text{reg}}[N]$ and $R'_{\text{mem}}[N]$.

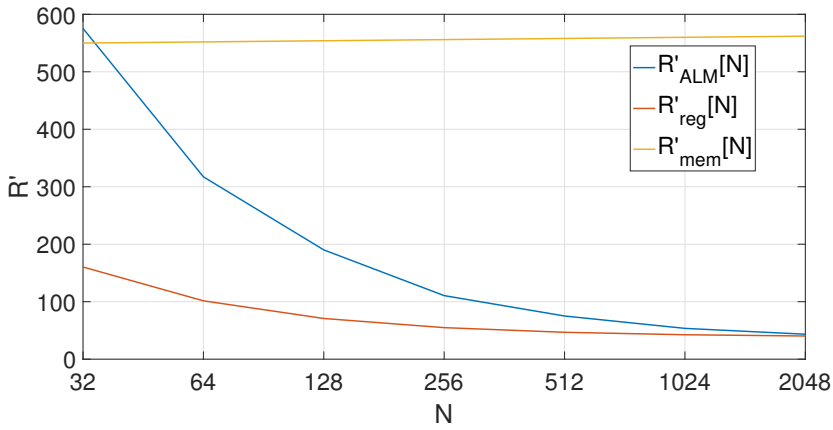

Figure 5.6.: Resource usage per sample for different input window size, N.

The resource usage per sample for the registers and the ALMs, R'_{reg} ans R'_{ALM}, seems to be converging to a constant, ν, $R'[N] \to \nu$ when $N \to \infty$. This property makes Equation 5.4, a good approximation for the two curves.

$$R'[N] = \frac{R_{\text{init}}}{N} + \nu \tag{5.4}$$

where R_{init} is a constant resource requirement unaffected by N. By experimenting in Matlab, it turns out that Equation 5.5 and 5.6 gives the best fit to the curves respectively.

$$R'_{\text{ALM}}[N] \approx \frac{17250}{N} + 35 \tag{5.5}$$

$$R'_{\text{reg}}[N] \approx \frac{3950}{N} + 38 \tag{5.6}$$

The memory bits per sample, R'_{mem}, is almost constant for different N.

$$R'_{\text{mem}}[N] = 540 + 2 \cdot \log_2(N) \tag{5.7}$$

The reason why it's not fully constant, is because of the FIFO memory for storing the cubic spline parameters. The parameters increases in num-

ber of bits for bigger N, as shown in Table 4.1 in Chapter 4. The total DSP requirement where constant for different N

$$R_{\text{DSP}} = 52 \tag{5.8}$$

5.3. Speed

All the speed testings is done by multiplying the total number of clocks used by the implementation to finish, by the minimum clock frequency received from TimeQuest analysis in Quartus[1]. The total amount of time used by the implementation to finish for different input window size, $T_f[N]$, is shown in Figure 5.7.

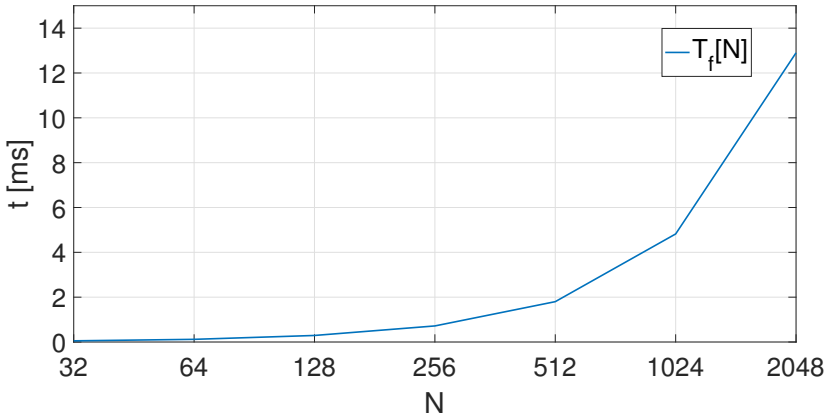

Figure 5.7.: Execution time for different input window size, N.

Now lets look at the average time it takes per sample for the implementation to finish, $T'_f[N]$, shown in Figure 5.8.

$$T'_f[N] = \frac{T_f[N]}{N} \tag{5.9}$$

[1]Reading Fmax in the Slow 900mV 100C Modle. This module assumes extreme conditions, meaning a low input voltage of 900 mV at a temperature of 100 °C

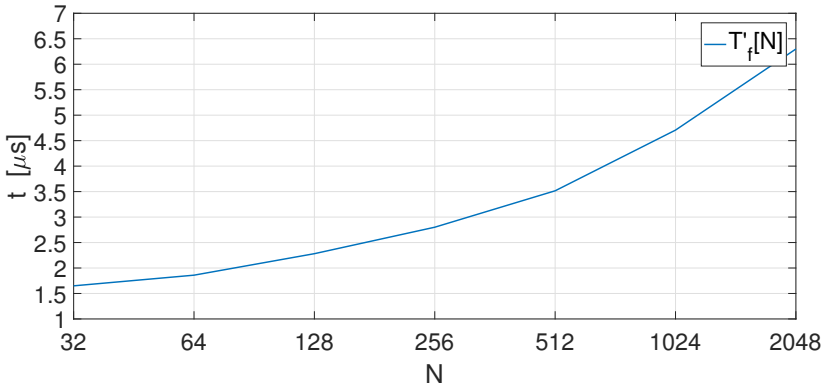

Figure 5.8.: Execution time per sample for different input window size, N.

The result shows that the the time per sample increases as N becomes larger. This behaviour is expected because it is known from [11] that the time complexity of the EMD algorithm is $O(N \log N)$. For real-time performance, this means that the maximum sampling frequency for the input signal, F_s, decreases when N increases. The maximum limit for the sampling frequency,

$$F_{s\_\text{lim}} = T_f^{-1} \tag{5.10}$$

is shown in Figure 5.9.

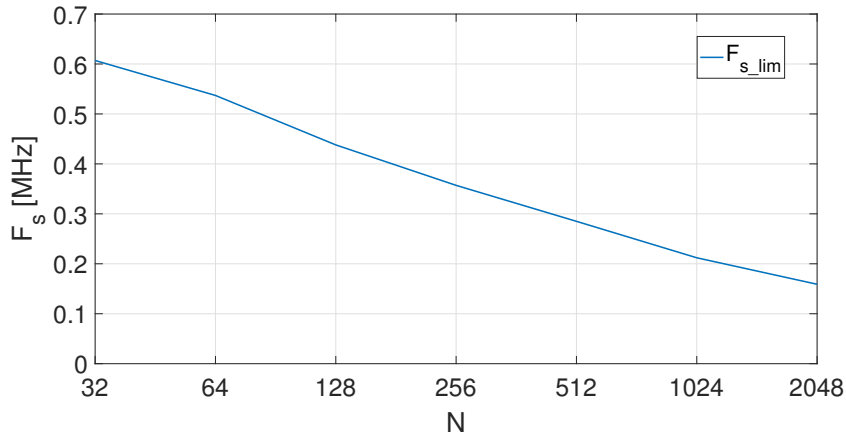

Figure 5.9.: Maximum input sampling frequency for real-time performance, $F_{s\_\mathrm{lim}}[N]$, for different input window size, N.

Figure 5.10 shows how the maximum clock frequency $f_{\mathrm{clock\_max}}$ changes with increasing N.

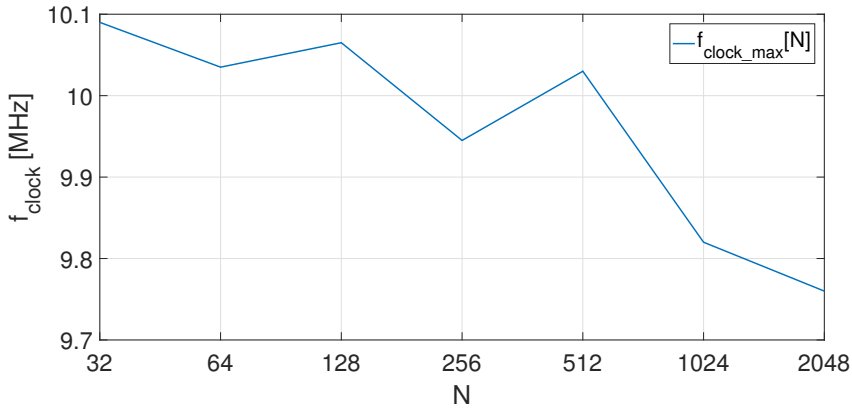

Figure 5.10.: Maximum clock frequency for the design, for different input window size, N.

Every time the design is compiled in Quartus, the gate mapping is con-

structed from scratch. This may lead to small variations on the critical path, which then leads to small varyations in $f_{\text{clock\_max}}$. Despite this, it seems to be decreasing as N increases. From Section 3.1 it is known that calculating the primes gives the longest critical path of the whole system. Since the input for the prime calculation, $A[i]$, $B[i]$, $C[i]$ and $D[i]$ requires more bits depending on the window size (shown in Table 4.1, Chapter 4), the critical path will become slightly longer due to how the arithmetic operations are generated in terms of the number of bits at the input.

6. Further improvements

This designed used an IP named LPM_DIVIDE, because it only uses one clock cycle to perform the division. Unfortunately it has a limit of 64 bits at the input. This sets the limit of 22 precision bits. In order to increase limit, a smaller division circuit using more than one clock cycle can be designed and used instead. This will increase the overall number of clock cycles used for the calculation, but may also reduce the propagation delay. This implementation did not consider the quality of the actual IMF components, and there are several ways to improve this. The boundary conditions has a lot of impact for bigger N. This hardware implementation uses natural boundary conditions, and also sets the end points of the data set to equal both a maximum and a minimum. This tend to make the cubic spline swing at the start and the end of the IMF as can be seen in Figure 5.3 for IMF 5 to 8. One approach to reduce this behaviour, can be to use linear interpolation a the end of the data set, and also not consider the endpoints as maxima and minima for the cubic spline calculation. More registers can be inserted to reduce the critical path of the system. By inserting a register between each of the arithmetic operations in Figure 3.6 for calculating the primes, the critical path could be reduced to only one division.

7. Conclusion

This report has proven that the EMD can be implemented using only fixed point precision. By applying external bits (referred to as precision bits) to the samples, effectively multiplying the value by 2, the EMD where able to produce better results closer to the floating point precision results. The energy of the difference was compared to the floating point result energy to get a SNR. By adding an extra precision bits, the SNR increased by about 14dB. Using 2048 samples of uniformly distributed noise as a testing data set, containing 8 IMF component, the SNR decreased for every new IMF component identified. Because the cubic spline interpolation is constructed by third order polynomials, the error will also behave as a third order polynomial. Since the IMF components contains less and less extrema for every IMF discovered, the average distance between them also increases. This increase in distance will make the third order polynomial error escalate which implies that an increasing number of precision bits is needed in order to keep the same SNR for every IMF component. For the uniformly distributed noise test data, 13 precision bits where not able to produce the last two IMF components although the first IMF had an SNR of about 200 dB. By increasing the amount of bits to 22, the first IMF had a SNR of about 325 dB, and the last of about 175 dB. For real-time purposes the maximum sampling frequency for the input signal tends to decrease as the window size, N, becomes larger. With a window size of 2048, the maximum sampling frequency for real-time performance equals about 159 kHz.

Bibliography

[1] Cubic spline tutorial. https://www.physicsforums.com/attachments/cubic-spline-tutorial-pdf.10531/. Accessed: 2017-07-26.

[2] Integer arithmetic ip cores user guide. https://www.altera.com/en_US/pdfs/literature/ug/ug_altmult_add.pdf. Accessed: 2017-06-04.

[3] Amir Bashan, Ronny Bartsch, Jan W. Kantelhardt, and Shlomo Havlin. Comparison of detrending methods for fluctuation analysis. *Physica A: Statistical Mechanics and its Applications*, 387(21):5080 – 5090, 2008.

[4] Kun Hu, C.K. Peng, Norden E. Huang, Zhaohua Wu, Lewis A. Lipsitz, Jerry Cavallerano, and Vera Novak. Altered phase interactions between spontaneous blood pressure and flow fluctuations in type 2 diabetes mellitus: Nonlinear assessment of cerebral autoregulation. *Physica A: Statistical Mechanics and its Applications*, 387(10):2279 – 2292, 2008.

[5] Norden E. Huang, Zheng Shen, Steven R. Long, Manli C. Wu, Hsing H. Shih, Quanan Zheng, Nai-Chyuan Yen, Chi Chao Tung, and Henry H. Liu. The empirical mode decomposition and the Hilbert spectrum for nonlinear and non-stationary time series analysis. *Proceedings: Mathematical, Physical and Engineering Sciences*, 454(1971):917–923, 1998.

[6] Men-Tzung Lo, Kun Hu, Yanhui Liu, C.-K. Peng, and Vera Novak. Multimodal pressure-flow analysis: Application of hilbert huang transform in cerebral blood flow regulation. *EURASIP Journal on Advances in Signal Processing*, 2008(1):785243, May 2008.

[7] Men-Tzung Lo, Lian-Yu Lin, Wan-Hsin Hsieh, Patrick Chow-In Ko, Yen-Bin Liu, Chen Lin, Yi-Chung Chang, Cheng-Yen Wang, Vincent Hsu-Wen Young, Wen-Chu Chiang, Jiunn-Lee Lin, Wen-Jone Chen, and Matthew Huei-Ming Ma. A new method to estimate the amplitude spectrum analysis of ventricular fibrillation during cardiopulmonary resuscitation. *Resuscitation*, 84(11):1505 – 1511, 2013.

[8] Men-Tzung Lo, Vera Novak, C.-K. Peng, Yanhui Liu, and Kun Hu. Nonlinear phase interaction between nonstationary signals: A comparison study of methods based on hilbert-huang and fourier transforms. *Phys. Rev. E*, 79:061924, Jun 2009.

[9] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[10] L. H. Thomas. Elliptic Problems in Linear Differential Equations over a Network. Technical report, Columbia University, 1949.

[11] Yung-Hung Wang, Chien-Hung Yeh, Hsu-Wen Vincent Young, Kun Hu, and Men-Tzung Lo. On the computational complexity of the empirical mode decomposition algorithm. 2014.

[12] Zhaohua Wu and Norden E. Huang. A study of the characteristics of white noise using the empirical mode decomposition method. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 460(2046):1597–1611, 2004.

[13] Zhaohua Wu and Norden E Huang. On the filtering properties of the empirical mode decomposition. *Advances in Adaptive Data Analysis*, 2(04):397–414, 2010.

[14] Zhaohua Wu, Norden E Huang, and Xianyao Chen. The multi-dimensional ensemble empirical mode decomposition method. *Advances in Adaptive Data Analysis*, 1(03):339–372, 2009.

[15] Jian Zhang, Ruqiang Yan, Robert X. Gao, and Zhihua Feng. Performance enhancement of ensemble empirical mode decomposition.

Mechanical Systems and Signal Processing, 24(7):2104 – 2123, 2010. Special Issue: ISMA 2010.

Appendices

A. Fixed point arithmetic error propagation

When trying to represent a decimal number with a fixed point precision, the value may be distorted by some error. For any decimal number A, represented as a fixed point A_{fix},

$$A_{fix} = A + E_a \tag{A.1}$$

where

$$\frac{1}{2^M} \geq E_a \geq 0 \tag{A.2}$$

and M being the amount of precision bits used. This error will have different behaviour when different arithmetic operations is performed. Table A.1 gives an overview of how the fixed point representation error propagates through different arithmetic operations.

Operation	Equation	Resulting error
Addition	$A_{\text{fix}} + B_{\text{fix}}$	$E_a + E_b$
Subtraction	$A_{\text{fix}} + B_{\text{fix}}$	$E_b - E_a$
Multiplication	$A_{\text{fix}} \cdot B_{\text{fix}}$	$AE_b + BE_a - E_aE_b$
Division	$\frac{A_{\text{fix}}}{B_{\text{fix}}}$	$\left(\frac{1+\frac{E_a}{A}}{1+\frac{E_b}{B}} - 1\right)\frac{A}{B}$

Table A.1.: How fixed point representation error propagates through different arithmetic operations.

A.1. Addition/subtraction error propagation

For any two numbers, A and B, the fixed sum $C_{fix} = A_{fix} + B_{fix}$ can be rewritten as

$$\begin{aligned} C + E_c &= (A + E_a) + (B + E_b) \\ C + E_c &= (A + B) + (E_a + E_b) \end{aligned} \tag{A.3}$$

and the resulting error after the summation will be

$$E_c = E_a + E_b \tag{A.4}$$

A.2. Multiplication error propagation

For any two numbers, A and B, the fixed multiplication $C_{fix} = A_{fix} \cdot B_{fix}$ can be rewritten as

$$\begin{aligned} C + E_c &= (A + E_c)(B + E_b) \\ C + E_c &= AB + AE_b + BE_a + E_bE_a \end{aligned} \tag{A.5}$$

and the resulting error after the multiplication will be

$$E_c = AE_b + BE_a + E_bE_a \tag{A.6}$$

A.3. Division error propagation

For any two numbers, A and B, the fixed division $C_{fix} = \frac{A_{fix}}{B_{fix}}$ can be rewritten as

$$\begin{aligned} C + E_c &= \frac{A + E_a}{B + E_b} \\ E_c &= \frac{A + E_a}{B + E_b} - \frac{A}{B} \end{aligned} \tag{A.7}$$

and the resulting error after the division will be

$$E_c = \left(\frac{1 + \frac{E_a}{A}}{1 + \frac{E_b}{B}} - 1 \right) \frac{A}{B}$$
$$E_c = \left(\frac{1 + \frac{E_a}{A}}{1 + \frac{E_b}{B}} - 1 \right) C \tag{A.8}$$

This means that the fixed representation will be the real result times some division factor depending on how big the error in A and B is with respect to itself.

$$C_{fix} = C \left(\frac{1 + \frac{E_a}{A}}{1 + \frac{E_b}{B}} \right) \tag{A.9}$$