University of Oslo
Department of Informatics

Digital compensation for static nonlinearity in $\Delta - \Sigma$ bitstream

Master thesis

Ehsan Arshad

6th November 2006

Acknowledgments

First of all I would like to thank my supervisor Dag T. Wisland for his excellent guidance and suppport throughout the writing of this thesis. I also want to thank you for giving me inspiration and motivation, and for the many hours of assistance.

I also wish to thank my good friend Mohammad Ali Saber for his encouragement, help with figure drawings and proofreading of this thesis.

Thanks to my family for always being there when support was needed.

Contents

List of Figures

List of Tables

Table 1: Abbreviations

Abbreviation:	Meaning:
VCO	Voltage Controlled Oscillator
FDSM	Frequency Delta Sigma Modulator
A/D	Analog to digital
A/F	Analog to frequency
F/D	Frequency to digital
ADC	Analog to digital converter
DAC	Digital to analog converter
DSP	Digital signal processing
OSR	Oversampling ratio
LO	Local oscillator
LSB	Least Significant Bit
MSB	Most Significant Bit
RCA	Ripple Carry Adder
RCAM	Ripple Carry Array multiplier
CSA	Carry Save Adder
CLA	Carry Lookahead Adder
PPM	partial product matrix

This table lists the meaning of common abbreviations and acronyms used throughout the thesis:

Chapter 1

Introduction

This chapter provides some background in order to explain the motivation for developing the linearization technique described later. An introduction is provided to the FDSM concept, for which the linearization technique is developed.

1.1 Background

An increasing part of the world is becoming digital. Since all the natural processes are analog and the computer is digital, an analog to digital conversion is necessary for any kind of digital signal processing. Due to the rapid increase in digital products, like digital radio(DAB), digital television, mp3 players etc, the demand for high quality A/D converters increases in parallel as well. In addition, wireless and battery driven equipments, being essential in the trendy tech world today, requires A/D converters which operates well with low supply voltages and also have low power consumption.

A novel analog to digital conversion technique, well suited for the products of today, is the Frequency Delta Sigma A/D conversion technique, so called FDSM [4] [5]. This is a high quality A/D converter, which operates well with low power supplies and also requires low power. The term “FDSM” does not have any clear definition. While referring to FDSM some interpret the Voltage Controlled Oscillator (VCO) as a part of the FDSM, whereas others will exclude the VCO from FDSM. In this thesis however, we will utilize the last mentioned interpretation of FDSM. The FDSM technique is mathematically equivalent to the conventional Δ - Σ technique [6]. The FDSM uses frequency modulation (FM) for converting

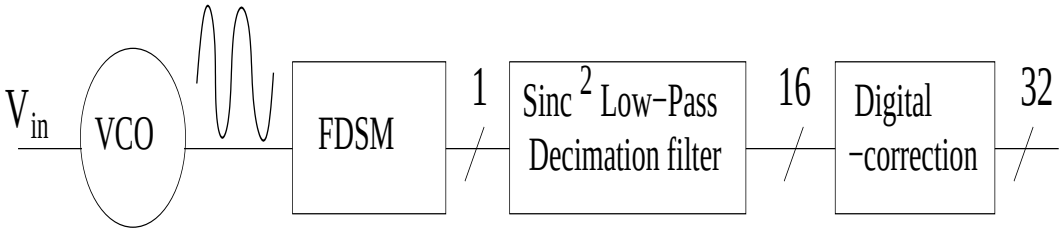

Figure 1.1: The complete FDSM system with non-linear LC-VCO input and digital correction

analog signals into digital. Another advantage of FDSM is of being fully digital as we have defined FDSM without the VCO. This will avoid the problems due to analog circuitry. In order to provide the input to the FDSM system, a VCO is utilized. As there is no feedback in the FDSM, linearity errors in the VCO provided input are not compensated.

For the A/D conversion a linear relationship between the analog input and digital output is necessary, otherwise the digital representation is heavily distorted. Therefore VCOs showing good linearity, such as the ring oscillator have always been used in the front end of the FDSM [7] [8]. However, the linearity of such ring oscillators is limited for low supply voltages. In consequence new solutions are being considered.

In this thesis another approach than the common ring oscillator method is developed, which will inherently focus on how good linearity is achieved. The technique is to use a LC-VCO, a systematic nonlinear VCO, and in attempt to obtain linearity we apply digital correction at the back end of the FDSM. The FDSM being a non-feedback system, results the non-linearity from the VCO pass through. This non-linearity can thus be corrected in the digital domain, namely at the back end of the FDSM. The developed technique in this thesis is mainly based on the square root characteristic of LC-VCO. Alternatively before the signal (analog) enters the FDSM, a correction could be made right after the VCO. Such a correction would have to be done in the analog domain, which is however a more difficult way to obtain the desired correction, and is therefore not preferable. The over all system will look something like shown in figure 1.1.

Other and more common use of the LC-VCO is in radio, and other wireless communication systems, where the LC-VCO is used for fre-

Table 1.1: LC-VCO vs Ring Oscillator

	Ring Oscillator	*LC − VCO*
V/F conversion	Good linear	Non-linear
Power consumption	Higher power consumption	Low power consumption
Phase Noise	Worse phase noise	Low phase noise
Tuning Range (relative)	Wide tuning range	Small tuning range
Area	Small area	Big area

quency up- and down conversion of signals. In this thesis the focus is at the use of LC-VCO in FDSM, thus our main concern is the linear characteristics of the LC-VCO. Linearity of the LC-VCO is rarely a subject of concern when being used for its normal applications.

There are both advantages and disadvantages when using LC-VCO compared to ring oscillator [9]. These are listed in table 1.1.

The main reason why we focus on LC-VCO, is that the main nonlinearity of LC-VCO is known à priori. This is due to square root dependency of the output frequency on the input voltage [10] that applies for most LC-VCOs. The linearization technique is also tended to correct for this non-linearity. Since we know that the non-linearity has the shape of a square root, it will be tried corrected by squaring.

This thesis is mainly based on the LC-VCO built by Bjørn Christian Paulseth [11], another Master student here at University of Oslo. He made a LC-VCO operating at frequencies around 5 GHz. His main focus was to get the phase noise as low as possible. Since he had a satisfactory square root response, we selected to use this VCO for our linearization process. He tuned the VCO from 0-1 V, and recorded the output frequencies. Those frequencies are used in this thesis.

To sum up, the goal of our research has been to assess the achieved linearity by using our developed technique on the frequency values obtained by Paulseth. By applying these values in our developed linearization technique, good results are obtained. Non-linearity as low as -42 dB or equivalent 7 bit is recorded.

1.2 Thesis layout

The layout of the thesis is as follows

- **Chapter 1** is an introduction, and presents the motivation for doing this thesis
- **Chapter 2** starts with the introduction of the source of non-linearity and the FDSM system. Thereafter some existing linearization techniques are presented, and finally the theory of the linearization technique developed is presented.
- **Chapter 3** is an analyze of the real data. Comparison of the obtained results with theory and other solutions is done. A discussion around the results is presented.
- **Chapter 4** starts with some theory about the implementation of the squarer circuit. The different parts used in the VHDL squarer code are presented, and finally results from the implementation are presented
- **Chapter 5** sums up the work done, gives a conclusion and suggestions for further work

Chapter 2

Digital Linearization

In this chapter an introduction to the source of non-linearity is given, followed by a brief presentation of the FDSM system. A few existing linearization techniques are introduced. Finally, the theory of our linearization technique is presented.

Before taking a detailed look at the linearization technique developed in this thesis, an introduction is provided on the LC-VCO and the FDSM. It is the LC-VCO which is the source of non-linearity in our A/D converter.

2.1 Source of non-linearity

The main operation of a VCO is to generate frequencies that are proportional to some tuning voltage. The LC-VCOs are most frequently tuned by reverse biased pn junctions or MOS varactors . Due to the characteristic of the varactors that are applied and also the tank circuit, the LC-VCO gain is quite non-linear.

For our purpose the LC-VCO functions as an integrated part of an A/D converter. An analog signal received by the VCO is used in order to generate some frequency proportional to this input signal. Further, this frequency is provided as input to the FDSM system. Therefore the VCO can be considered an analog to frequency (A/F) converter. One consideration to keep in mind when using the LC-VCO is that the proportional factor which regulates the voltage-frequency relationship, is not constant. When applying higher voltages to the LC-VCO, the change in frequency per unit change in the input voltage is decreased, thus making the LC-VCO non-linear.

Usually, the non-linearity of the LC-VCO is unwanted in the FDSM application. However, if the characteristics of the non-linearity is known a priori, such as the usual square root function of LC-VCOs, then there is the possibility to try to digitally correct the FDSM output in order to obtain linearity. We will try to get the linearity by squaring the FDSM output values, which has the form of a square root function. As better square root output the LC-VCO has, thus better linearization is achieved and thus simpler, faster, and less area occupying would the digital correction circuit be. In order to correct a square root characteristics, only a squaring circuit is needed. If there are any other non-linearities, additional correction circuitry is needed.

For further details on LC-VCO, I will refer to the thesis of Bjørn Christian Paulseth [11].

2.2 Frequency Delta-sigma modulator

For our purpose, the FDSM is just a black box. Taking analog frequencies as its input, and giving digital values as its output. A brief introduction is given here to provide a better understanding of the complete system.

As mentioned earlier, we will focus on the non-linearity of the VCO in FDSM. Compared to conventional delta sigma converters, the FDSM has a major advantage that it do not have feedback, thus avoiding non-linearities which appear in feedback DAC's in traditional $\Delta - \Sigma$ converters. The lack of this feedback in turn, makes the FDSM very sensitive to non-linearities in the VCO. As a consequence of the FDSM being a non feedback device, any non-linearities from the VCO will pass right through[8]. The linearity of the A/D conversion is thus limited by the linearity of the VCO. A highly linear VCO is therefore important. The VCO might by a linear VCO, or a non-linear VCO with correction.

The FDSM is superior to conventional Nyquist converters. First of all the FDSM provides the same advantage as conventional $\Delta - \Sigma$ modulators, of noise shaping. By using high oversampling ratio, e.g much higher sampling frequency than the minimum required nyquist sampling rate, the quantization noise is pushed out of band to frequencies higher than the frequency range of the signal of interest. Another advantage of the FDSM is of being a fully digital converter, as defined without the VCO, thus avoiding the analog domain problems. In addition the FDSM also

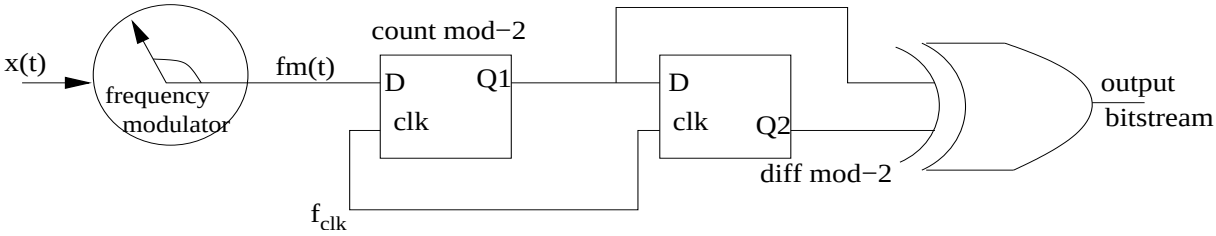

Figure 2.1: A simple first order FDSM (ref[1])

have some advantages compared to traditional Delta-Sigma modulators, these are

- multibit quantization with no feedback DAC
- very simple implementation in standard digital CMOS
- very high sampling frequency potential
- suited for low power supply voltage operation
- potential of low power consumption [10]

2.3 Linearization techniques

There exist diffferent kind of linearization techniques. Some techniques are specific for the device targeted, examples of which are found in [12], while other are more general [13] [14] [15]. A lineariztion technique for VCO linearization and one linearization technique developed for corrrecting non-linearity in ADC are shortly presented here.

One linearization technique, developed for VCOs is “Linearization by frequency feedback” [14]. This technique uses frequency feedback to linearize the non linear output of the VCO. The authors also assume that this technique may also potentially reduce the phase noise, but this was not observed in the realized structure.

Another linearization technique is described in [15], where a digital correction scheme for real-time applications is described. This technique has been proposed to correct for Integral Nonlinearity in any kind of ADCs. One potential advantage from using this technique, is the possibility to correct for offset errors. The price to pay for these benefits is

that an identical extra ADC must be included together with some extra digital signal processing (DSP) circuitry.

Since for our purpose, the linearization is to be done in the digital domain, placed at the back end of the FDSM, any technique requiring feedback to the VCO could not be used. We also wanted a simple but effective technique, without any complex signal processing. The desired technique should also use as few components as possible, to keep the power consumption low. The technique that is developed in this thesis is linearization by squaring. Unlike the other techniques, this technique is based on à priori knowledge about the non-linearity. Since the known non-linearity has the shape of a square root, a squaring in theory gives perfect linearization. But we are working with real components, and as known nothing is ideal in the real world. Deviations thus both from an ideal square root at the output of the VCO, and from perfect linearity at the squared output is expected. These deviations will be analyzed in the next chapter.

Regarding implementation, the squaring can be achieved by a multiplier device, since squaring is indeed a special case of multiplication. Squaring is multiplication in the special case where the multiplicator and the multiplicand are the same. A lot of literature is found on multipliers. Since we only needed squaring, and not multiplication in general, we wanted to make a squarer circuit. The reason for this is that a squarer is more compact than a multiplier, thereby reducing the area requirement and power consumption, two major factors in todays world. More about the implementation is found in chapter 4.

2.4 The linearization technique

The technique developed in this thesis is somewhat different from previous mentioned techniques. The main difference that can also be considered an advantage is that the non-linearity we are working with here is static and known à priori. Unlike the other techniques where the non-linearity is unknown, the à priori knowledge about the non-linearity simplifies the linearization. Since we know the shape of the non-linearity, which in our case is the square root function, we can make a circuit to correct for exact that non-linearity, The non-linearity being static, gives us also the advantage that we can make a static correction, instead of

some complex dynamic correction.

The linearization technique is meant to be implemented with the FDSM system. To keep the circuit size small, and also save power, a simple technique requiring few components was needed. Multiplication is an option to solve this problem. However a squarer was selected to make the implementation even more compact. Since fewer components are needed to implement a squarer, compared to traditional multipliers, power is also saved.

By linearization here in this thesis, is meant to have a linear relation between the output of the digital correction circuit, and the input to the VCO. Some of the internal results may be non-linear, but the final result is to be linear.
The concept can be easier explained by an example. Let us assume that three different signals are applied to the VCO, where the amplitude of the first signal is $A1$. The amplitude of the second signal, $A2$, is $A1 + \Delta V1$, and the amplitude of the third signal is $A3 = A2 + \Delta V2$, and $\Delta V1 = \Delta V2$ The desired output is of course that the VCO has a linear response so that the second oscillation frequency is $f2 = f1 + \Delta f1$, and the third frequency is $f3 = f2 + \Delta f2$, such that $\Delta f1 = \Delta f2$. This is the desired and also the correct output. The desired relationship can be described mathematically as,

$$A1 \rightarrow f1$$

$$A2 = A1 + \Delta V1 \rightarrow f2 = f1 + \Delta f1$$
$$A3 = A2 + \Delta V2 \rightarrow f3 = f2 + \Delta f2$$
$$\Delta V1 = \Delta V2 \rightarrow \Delta f1 = \Delta f2$$

However, since the VCO is non-linear, we get that $\Delta f2 \neq \Delta f1$.

At the next step, the FDSM frequency to digital conversion, these frequencies are simply converted to digital values. The FDSM performs only an F/D conversion, converting one frequency at the time, without changing their internal non-linear relationship. As a result, any non-linearity that appears at the FDSM input, would pass right through with no change. Finally the FDSM outputs a bit-stream representing the input frequency. The bitstream then passes through a $sinc^2$ low pass decimation filter that provides 16 bit digital values. The final result is that for a linear analog voltage input to the VCO, the digital 16 bit representation

is non-linear and totally different from the original analog signal. AS the non-linearity has passed unchanged through the FDSM, a digital correction can be done here, after the $sinc^2$ filter.

To find the over all response, we can start at the beginning and analyze the individual responses. The first conversion of the analog signal is done by the VCO, going from an analog voltage signal to a frequency. As mentioned above, the VCO has a non-linear response. The next step is the FDSM. The FDSM takes the frequency output of the VCO and converts the frequency into a digital representation. The FDSM has a linear response. For the overall response of a system to be linear, all the individual responses must be linear. In our case however, a non-linear response is followed by a linear response, so that the overall response is nonlinear.

$$H_{sys} = H_{vco} * H_{FDSM} = H_{nonlinear} * H_{linear} = H_{nonlinear}$$

However, even though the VCO has a non-linear response, the response is known a priori. Thus this non-linearity can be corrected by applying a correcting circuitry with a inverse response to the response of the VCO,

$$H_{sys} = H(vco)_{nonlinear} * H(FDSM)_{linear} * \frac{1}{H(vco)_{(}nonlinear)} = H_{linear}$$

As mentioned above for a system to be linear all the individual responses have to be linear. This is not the case in our final solution. We do not have only one, but two non-linear responses. The VCO response and the inverse of this response. How can the result than be linear? The fact is that the non-linear responses are inverse to each other, with the result that the non-linear response is canceled. Figure 2.1-2.3 shows how the signal is converted from being a linear analog signal at the input of the VCO, to becoming a nonlinear digital representation at the output of the $sinc^2$ filter.

The technique we will demonstrate here consists of some simple mathematics manipulation of the output values from the $sinc^2$ filter. As mentioned before, these values appear in a square root manner and by simply squaring all the values, a theoretical good linearization may be achieved. How good linearity we get, depends at how good square root shape the values appear in. Under ideal conditions, with ideal components, perfect linearity is achieved. Under such ideal conditions, the VCO would have a perfect square root output. This perfect square root

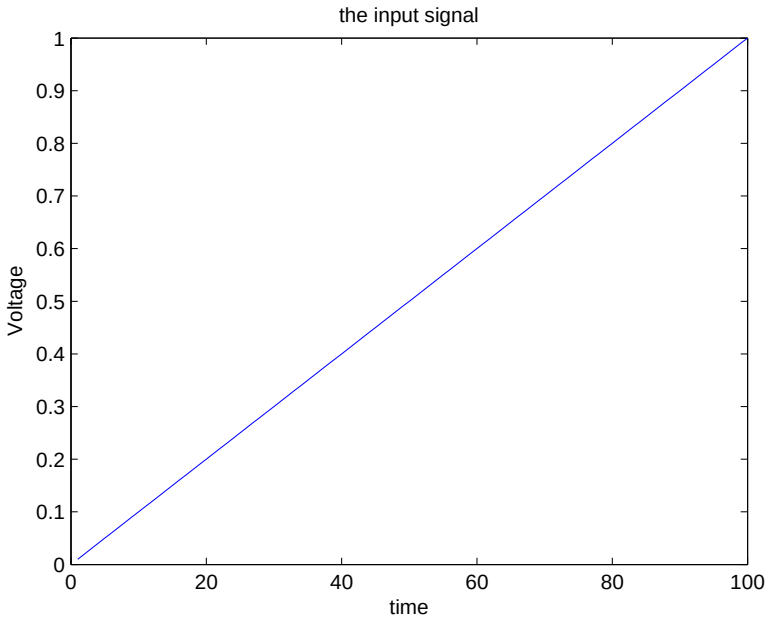

Figure 2.2: Input to the LC-VCO

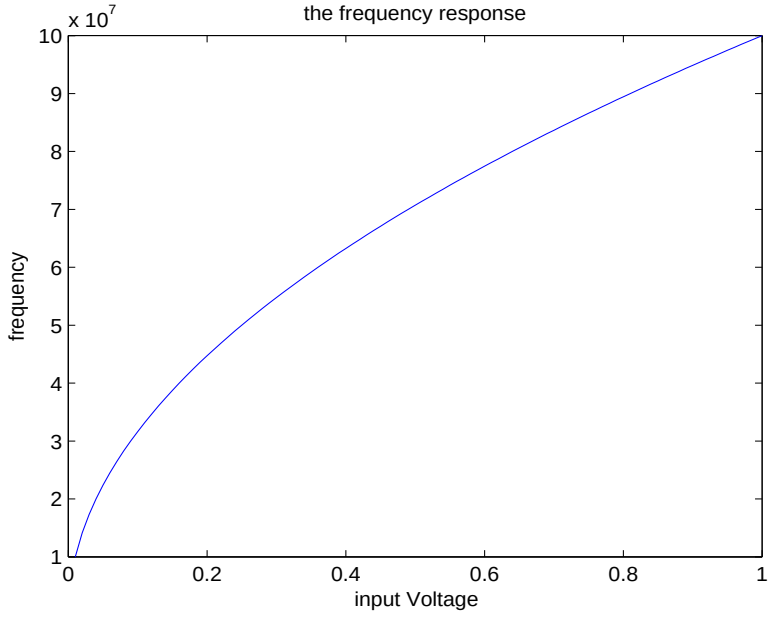

Figure 2.3: Voltage-Frequency conversion in the LC-VCO

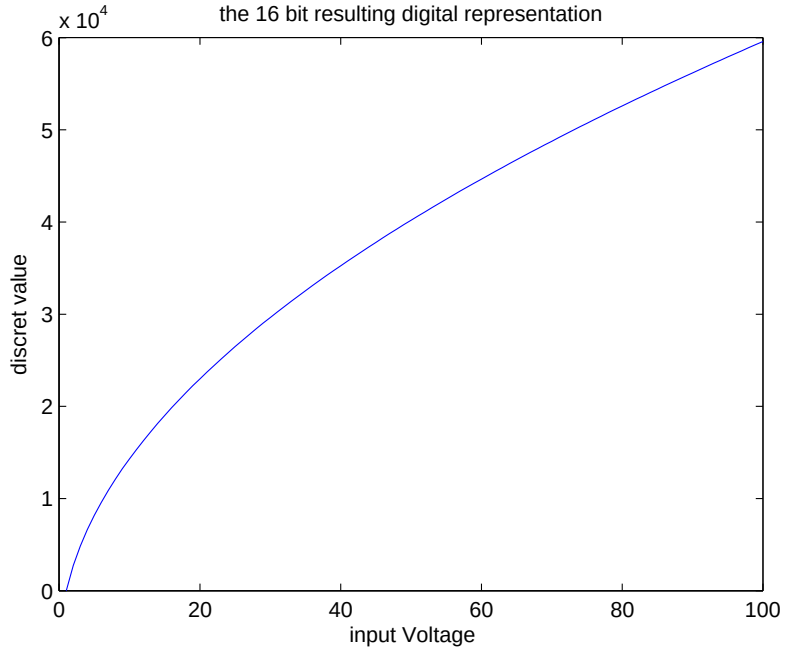

Figure 2.4: The resulting 16 bit representation

would pass unchanged through the FDSM(the non-linearity having the shape of a square root will pass unchanged, the values will get a F/D conversion). At the back-end of the FDSM, a squaring would be applied to this perfect square root, giving a perfect linear result.

However under actual conditions, and not ideal, things are different. The digital correction consisting of squaring, is applied at the back-end of the FDSM. The correction circuitry consists simply of a digital squaring circuit. Since it is now assumed real conditions, the VCO will not have a perfect square root output. As a consequence, the final result, the output of the correction circuit, will have some deviations from a perfect linear output.

As mentioned earlier, the change in frequency per unit voltage change, is much lower for high voltages than for low voltages. A way of representing this change is to use the derivative, which is a measure for change per unit. This can be described mathematically as:

$$f'(v_1) > f'(v_2), \qquad given \quad v_1 < v_2$$

However, it already exists a well defined mathematics description of the resonance frequency of the LC-VCO. This description is given as

$$F = \frac{1}{2\pi\sqrt{LC}} \tag{2.1}$$

where L is the inductance of the VCO, and C the capacitance. A capacitance can be represented as charge per voltage as

$$C = \frac{q}{V} \tag{2.2}$$

and by combining equation 2.1 and 2.2 we get the result,

$$F = \frac{\sqrt{V}}{\sqrt{qL}} \tag{2.3}$$

As can be seen from equation 2.3, the frequency is proportional to the square root of the voltage scaled with a constant factor $1/\sqrt{qL}$, where q is the electron charge, and L represents the inductance of the varactor. These are the values that appears at the output of the VCO. A linear relationship can be achieved by multiplication with the inverse function, in this case a squaring function as shown by Equation 2.4. Here comes one of the strengths of this technique. It can be used with any LC-VCO who has this square root relationship, independent of all other parameters.

$$F = \frac{\sqrt{V}}{\sqrt{qL}} => \left(\frac{\sqrt{V}}{\sqrt{qL}}\right)^2 = \frac{V}{qL} \tag{2.4}$$

As a result of non-ideal components, the output deviates from the ideal square root function. These deviations will be further analyzed by using toolboxes in Matlab in the next chapter. The result will also be used to consider if any other correction than the squaring is necessary to achieve satisfactory results.

Chapter 3

Data Analysis

In this chapter simulations are done in order to verify the theory established in the preceding chapter, and comparison is made between the two. Both the non-linear output of the VCO used, and the linear output values of the correction circuit will be analyzed. The results obtained are compared to results obtained using other types of oscillators.

All the analysis in this chapter are based at LC-VCO output values which we got from Bjørn Christian Paulseth, another student here at University of Oslo.

Yet another student here at University of Oslo, Mohammad Ali Saber, also worked with a LC-VCO [2]. Unlike Paulseth, Saber made the VCO specifically to be used in a FDSM application. Because of this special purpose of the VCO, Saber's VCO did not have the square root output that LC-VCO's usually have. Saber's target was to design a LC-VCO with linear regions. The whole output needed not to be linear, but it was sufficient to have some linear portions. The intention was to improve on the VCO's linear characteristics in order to avoid any post processing. The linearization technique developed in this thesis, is therefore not applied to this VCO. The VCO does not display the square root output characteristics, therefore it is not meaningful to apply squaring for this VCO's output values. A plot of the output of this VCO is shown in figure 3.1.

From here on Bjørn's VCO is used. This LC-VCO was tuned from 0-1V and the output frequency values were recorded in a file. These values are further explored in this thesis. These values were imported in Matlab, and plots were made. The main characteristics of the VCO are shown through these plots.
By tuning this VCO with voltages from 0-1 V, the VCO operates at fre-

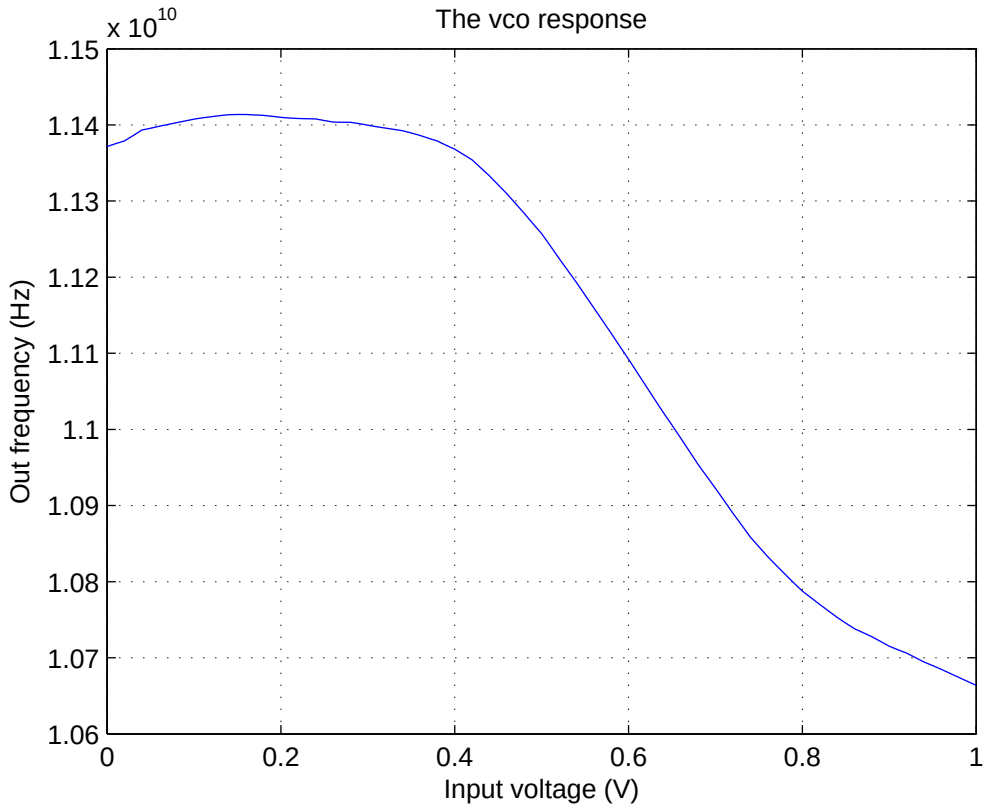

Figure 3.1: The output of saber,s VCO (ref:[2])

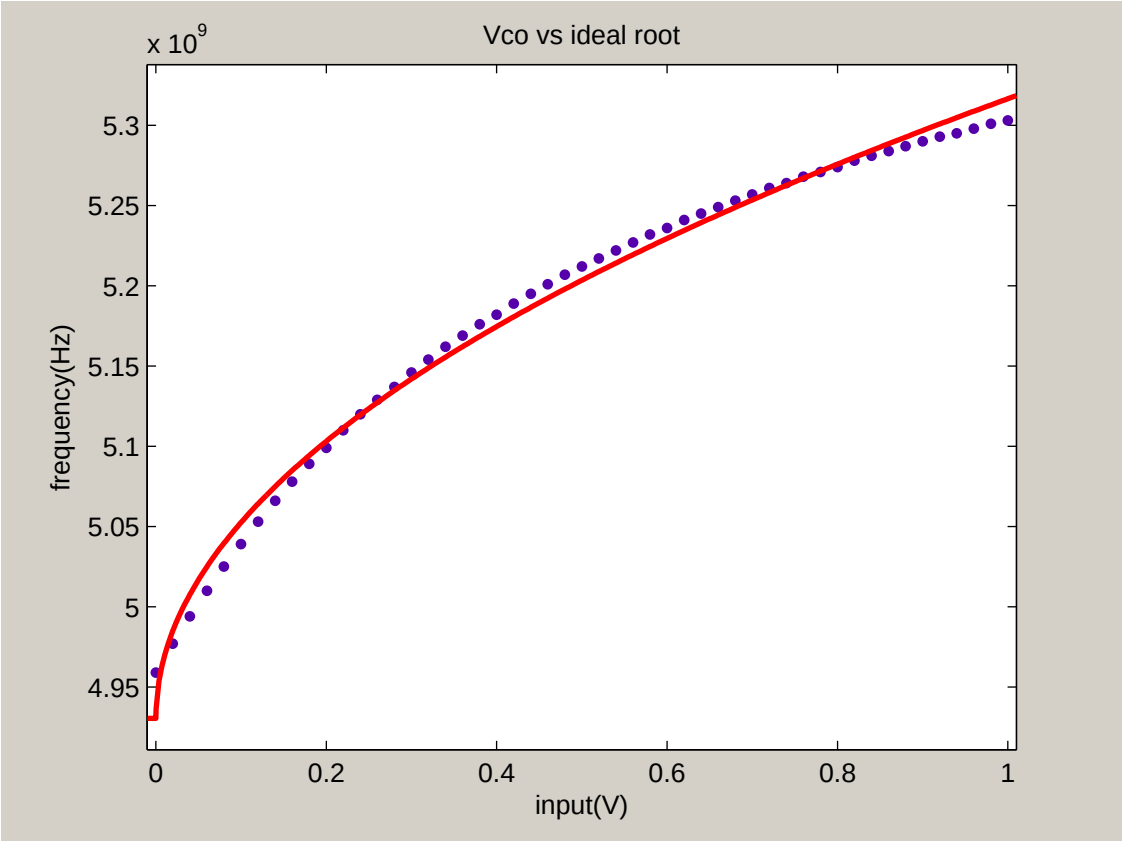

Figure 3.2: Cadence output values vs. fitting

quencies from 5- 5.3 GHz, providing a frequency range of approximately 300 MHz. The output frequencies are displayed with a resolution of 0.02V, giving 51 points of values. The resolution of 0.02V is assumed good enough for our purpose.

To further process these values, the Curve Fitting Toolbox in Matlab was used. Most of the analysis was done using this toolbox. The values were imported to this toolbox, where a fitting to an ideal square root function was applied to check how well these values match the ideal square root function. The original values together with the fitting are plotted in figure 3.2.

As seen by the plot, there is a slight deviation. The form of this deviation can be seen by plotting the residuals. This is done in figure 3.3. The normalized residuals are calculated according to equation 3.1

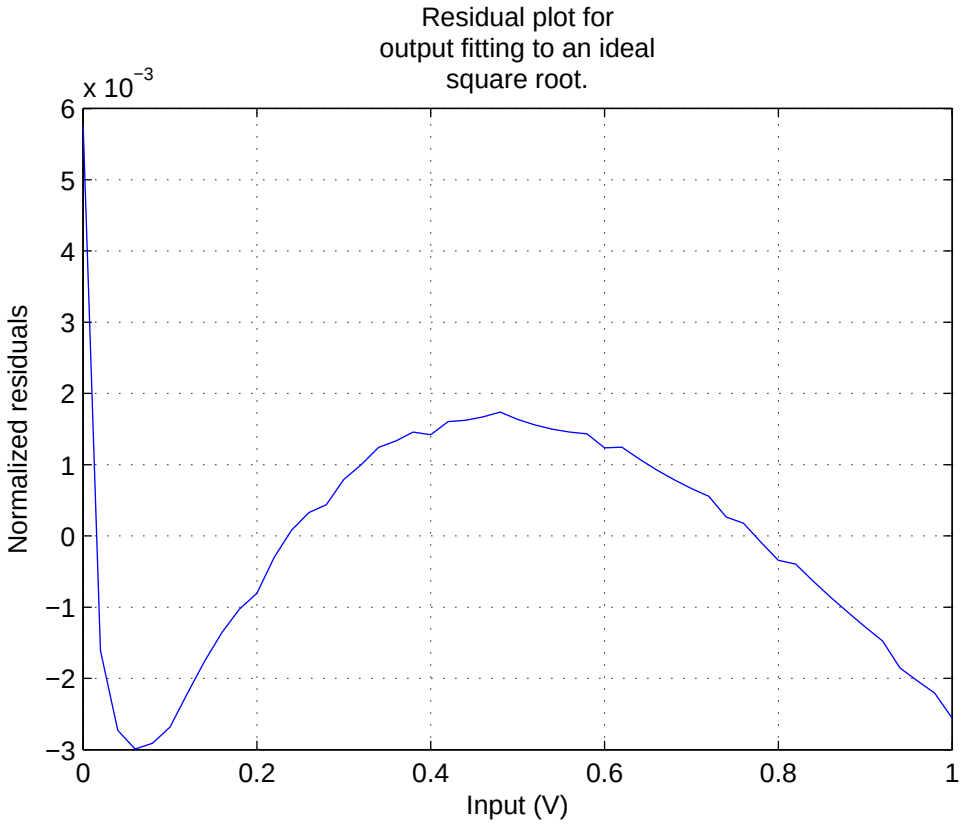

Figure 3.3: Residual plot. The deviation in data from an ideal square root function.

and 3.2.

$$residuals[i] = fitted[i] - orig[i] \tag{3.1}$$

$$norm\_residuals[i] = \frac{residuals[i]}{orig[i]} \tag{3.2}$$

3.1 Polynomial fitting

Curve fitting was applied to the output of the VCO. Fittings of different degree polynomials were applied to the square root output of the VCO to find out more on the error. The fittings were done from a first degree

Table 3.1: Polynomial fit

$polynomorder(V)$	$linearity$
1.order	92.86
2.order	99.51
3.order	99.97
4.order	~ 1

(linear fit), to a fourth degree fit. The different fits applied to the output can be seen in figure 3.4, while the residual-plot from these fittings is seen in figure 3.5. The fourth degree fitting is not shown in the fitting plot, because it totally overlapped the original data, making it impossible to see the original data in the figure. However, in the residual plot this fitting is plotted together with the other fittings.

As observed from figure 3.5, most of the non-linearities would be removed if the system was compensated by a fourth-order function. An ideal LC-VCO should only have second-order errors, but since the varactor used is not ideal, these non-linearities will add up and cause higher-order errors. The second-order error is however dominating. This is due to square function which is found naturally in the MOS transistors. As we are correcting for this second-order error, most of the error is compensated. We are actually only performing a squaring, thus not correcting all the second order errors, but a big part. This is because the second order error is not a pure square function. It also contains some first order coefficients and a constant. It is not of the form x^2, but is of the form $ax^2 + bx + c$.
The result would be even better if we compensated with a higher-order function. This is however not done here in this thesis. The correcting function we use, is not supposed to only be good mathematical. The correcting function is to be implemented in hardware. A high-order function would be rather complex in hardware, and is therefore not preferred.

3.2 Adjustments to the VCO output

The VCO, operating at frequencies around 5 GHz is normal for LC-VCO's. For the FDSM application such frequencies are too high to handle, because the FDSM can not operate at such high frequencies. Therefore, in

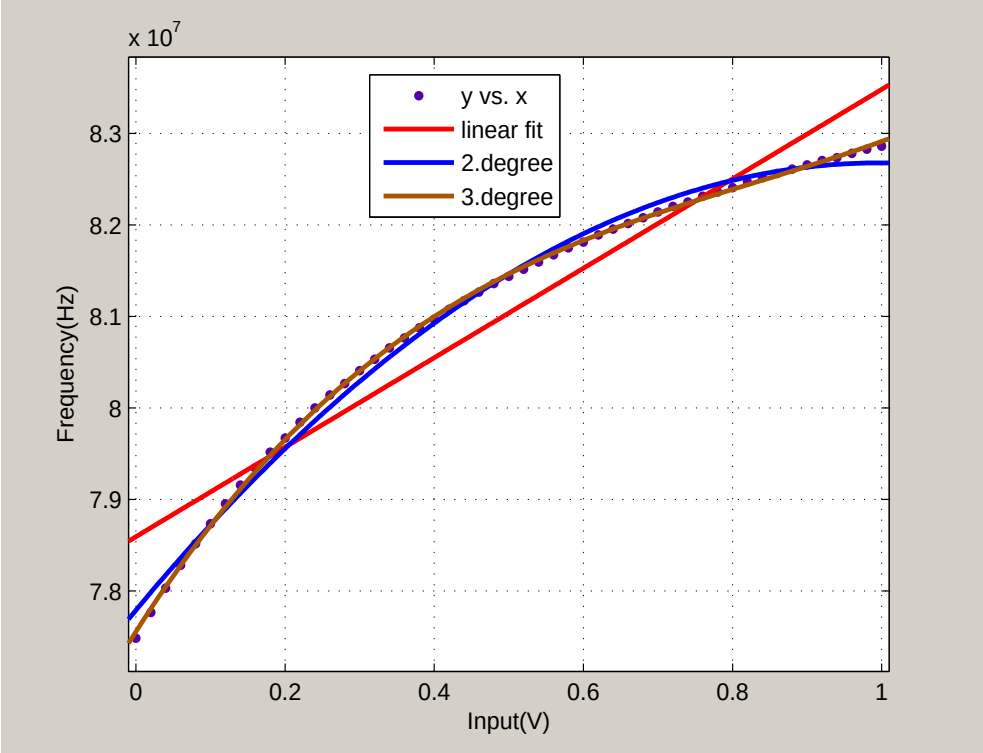

Figure 3.4: VCO output fitting to different polynomials

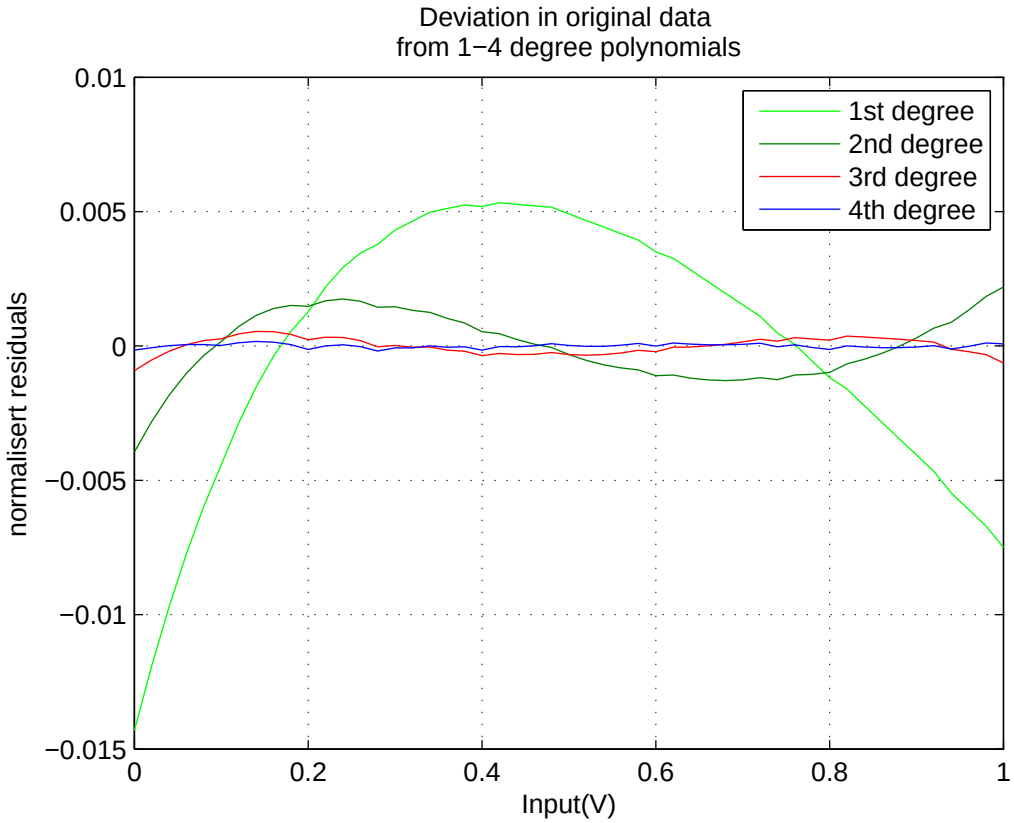

Figure 3.5: Residual plot for fitting of the LC-VCO output to different polynomials

order to make the output frequencies compatible with the FDSM application, they must be scaled down. The down-scaling can be achieved by applying a frequency divider or a pre-scaler on the VCO's output values. Such a frequency divider may consist of simple D flip flops, where each D flip-flop scales down the input frequency to half the input value. In order to scale down the frequency down to reasonable values for the FDSM, which is about 50-100 MHz, 6 flip flops are needed, resulting in a division by $2^6 = 64$, and the frequencies are than scaled down to about 80 MHz. Since all the values are divided by the same amount, the shape of the VCO output values are retained. A division do not only reduce the oscillation frequencies, but the frequency range is also reduced accordingly. This means that the frequency range is now only about 5 MHz, which is a relatively small range for a LC-VCO. However, the VCO is to be used in a FDSM application. Therefore the measure of interest is the absolute frequency range, rather than the relative. For the FDSM application a frequency range of approximately 1 MHz is sufficient. A frequency range of 5 MHz is thus well acceptable. This approach of using D flip-flops is assumed in the simulations later.

An alternative approach in order to down-scale the frequencies is to use a RF-mixer [16]. The advantage would be that the frequency range would stay the same, while the operating frequency would be down-scaled. To use this mixer solution, a local oscillator(LO) would be needed. The task of the LO is to generate some reference frequencies. The LO generated reference frequencies are to be multiplied with the incoming frequencies from th LC-VCO, in order to down-scale the LC-VCO output frequencies. The multiplication of the frequencies from the oscillators, would result in two new frequencies. We would get the sum and the difference of the frequencies of the two oscillators. Only one of the resulting frequencies is needed, the other can be filtered out. This local oscillator would be a part in addition to the mixer, both being analog parts. They would have big area requirement when implemented, and would also increase the power consumption. In addition, we would have to deal with analog parts, which is not preferred. These are the reasons for not recommending the mixer solution, although it would have been nice to have wide frequency range. This solution is also quite complex, compared to the simple D flip-flops.

The new frequency range, when 6 D flip flops are assumed used to down scale the VCO output frequencies, is seen in plot 3.6. As seen by the plot, the shape is retained. We say that 6 flip flops are *assumed*

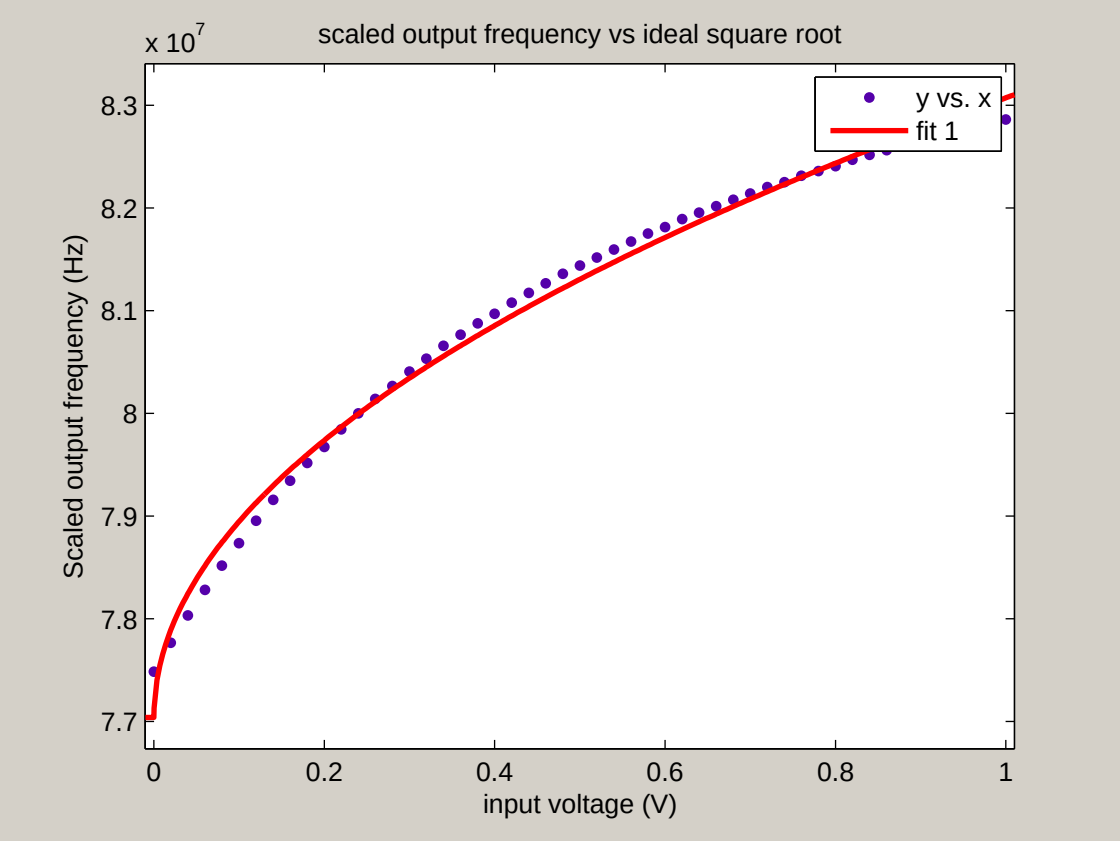

Figure 3.6: The down scaled output values vs. fitting. The new frequency range from 77-83 MHz.

used, because the flip flops were not actually made. Only the division by 64 was done in Matlab, to simulate the actual process.

3.3 A/D conversion

When the frequencies are down scaled, they still remain in an analog state. They must be converted to digital form before the digital correction can be applied. The values are quantized in time, since we only have 51 values, but their amplitude also have to be quantized. This quantization represents the A/D conversion that is done by the FDSM in a real system.
Since the values will be inside a certain range, and they will be above a minimum value (offset), the values can be adjusted for this offset, and normalized according to the range before they are converted, giving much better resolution. The conversion is done according to formula 3.3.

$$F = \frac{F_{in} - F_{min}}{F_{max} - F_{min}} \tag{3.3}$$

In the real process, the conversion is done by the FDSM , but we will use Matlab for simulation purpose, assuming ideal conversion. The actual FDSM outputs an oversampled bit-stream, where the oversampling-ratio (OSR) is given as

$$OSR = \frac{f_s}{f_{nyquist}} \tag{3.4}$$

where f_s is the sampling frequency of the FDSM. The$f_{nyquist}$ in the denominator of equation 3.4 stands for the Nyquist sampling frequency. The Nyquist sampling frequency is the theoretical minimum sampling frequency a signal with max frequency f_{max} can be sampled with without distorting the signal, and is given as

$$f_{nyquist} = 2 * f_{max}. \tag{3.5}$$

In actual Nyquist rate A/D converters also, samplings rate a little above the Nyquist rate is used. This is because no components are ideal.

The bit-stream out of the FDSM is then sent through a low pass decimation filter, which removes the noise that are pushed to the higher

frequencies by the FDSM. This filter also reduces the sampling frequency to the Nyquist rate, and outputs 16 bit word. To simulate this process, also in our simulation is chosen 16 bit representation. The result of using formula 3.3 is that the frequencies are now normalized within the range 0-1. To represent them with 16 bit, only a multiplication with 2^{16} , and the rounding of to the nearest integer is needed,

$$16\ bit = round(F * 2^{16}) \tag{3.6}$$

The last value in the array, $2^{16} = 65536$, is replaced with 65535. As known the highest value that can be represented with x bit is $2^x - 1$. The changing from 65536 to 65535 is done to keep us within 16 bits. In the real FDSM, the conversion may not be ideal, but our purpose is to focus at the non-linearity of the VCO, other sources of noise are disregarded. Although the FDSM reduces the quantization noise significant.

Since the 16 bit representation have the form of a square root function, a squaring function is applied, so that all the values in the array are squared. As seen by plot 3.7, the squared result is linear in the mid-region, but bends of at the corners. Since we have 5 MHz range, and we only need approximately 1 MHz, we have room for reduction. By choosing a smaller input voltage range of the plot, higher linearity may be achieved. The smaller area must still provide sufficient frequency range, so it can not be arbitrary small. When choosing a smaller voltage range, the resulting frequency range must be kept in regard, so that the frequency range do not become to small.

The whole 32 bit range of the output represents 5 MHz of frequency range. The 16 bit input is squared, thereby the 32 bit output. Shown by equation 3.7.

$$(2^{16})^2 = 2^{32} \tag{3.7}$$

It would seem possible that to get better linearity, since the whole range represent 5 MHz, any portion of $1/5$ of the output can be selected. This is not the case. The reason for this is found in plot 3.6, which is the input to the FDSM. Since the input is square-root, e.g non-linear, the curve do not have a constant derivative. The square root function has a steep slope in the beginning, but flattens out for higher values. The result is that for small voltage values, we can get sufficient frequency range by only a small input voltage range. As higher up we get in the input voltage, thus wider input voltage range do we have to select, to obtain

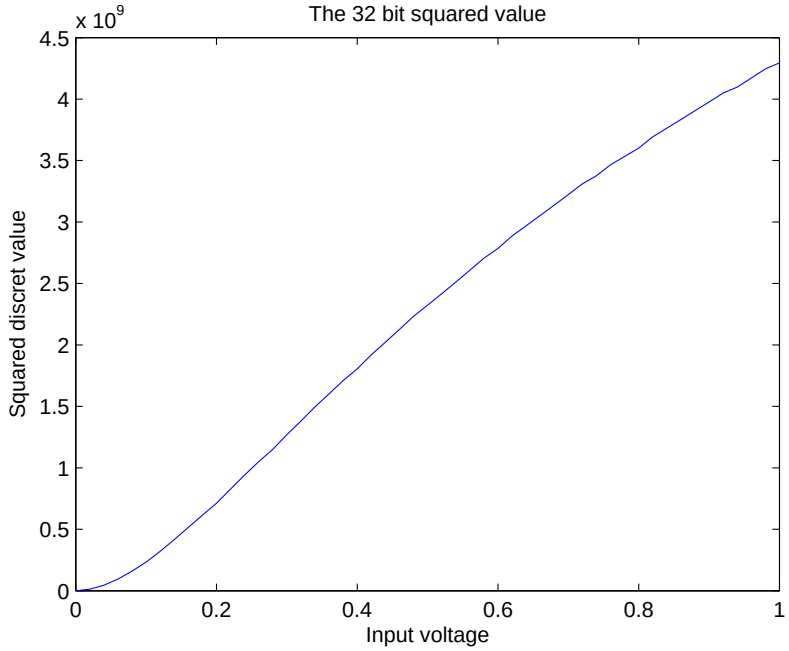

Figure 3.7: The squared values

sufficient frequency range. An arbitrary portion of $1/5$ can therefore not be selected. If an area of good linearity is selected from the $output$ plot shown in figure 3.7, one must go back to the $input$ plot shown in figure 3.6 to see if the selected voltage range has enough frequency range or not.

Matlab is used to simulate the whole process, from a linear voltage input to the VCO, through non-linearities, to the corrected output of the squaring circuit. With Matlab the results will be analyzed and we can get an idea of how good linearization is achievable before it is implemented in hardware.

To get better linearity, a smaller area can be selected. The choice of this smaller area can be done here in Matlab. Different ranges within the whole range can be selected, and than the range with best linearity can be selected.

Table 3.2: Linearity results

$In\ range(V)$	$Out\ range(MHz)$	$Linearity$	$Max\ deviation$	Bit
0-1	5.375	99.48	4.5%	4.47
0.14-0.3	1.25	99.92	2.75%	5.18
0.32-0.8	2.00	99.68	5.26%	4.25
0.44-0.66	0.844	99.92	0.8%	6.97
0.7-1	0.359	99.66	0.33%	8.24
0.2-0.36	1.09	99.98	0.7 %	7.12

3.4 Further improvements

As seen from the plots in the preceding section, the VCO output is not an ideal square root, with the consequence that the results do not become ideal linear when squared. The results from the squaring in the previous section will be further tried improved by selecting smaller input ranges. Different input ranges will be considered in this section, and their linearity will be compared towards finding the best linearity.

To get an impression of the linearity, first the entire tuning range is selected. The goodness of linearity for the other ranges will than be compared to this one.
The squared values are plotted together with their best linear fit and the corresponding residuals in figure 3.4. The maximum normalized residual value is presented in table 3.2. The curve fitting tool only showed actual residuals, not relative. Therefore only the actual residuals are plotted. However, for comparison purpose the relative residuals were needed. The relative residuals are thus calculated according to equation3.2, and the maximum value of the residuals for the different ranges are presented in table 3.2.

The bit representation of the non-linearity is calculated according to 3.8

$$bit = \frac{20log(max\ deviation)}{6.02} \tag{3.8}$$

The $linearity$ column of table 3.2 shows how well the different ranges matches their best linear fit.

As seen from table 3.2, different ranges have their advantages and disadvantages. The whole input range gives the best frequency range, but suffers at all the other criteria, such as linearity and also max deviation. The range 0,44-0,66V has good linearity and also very low value

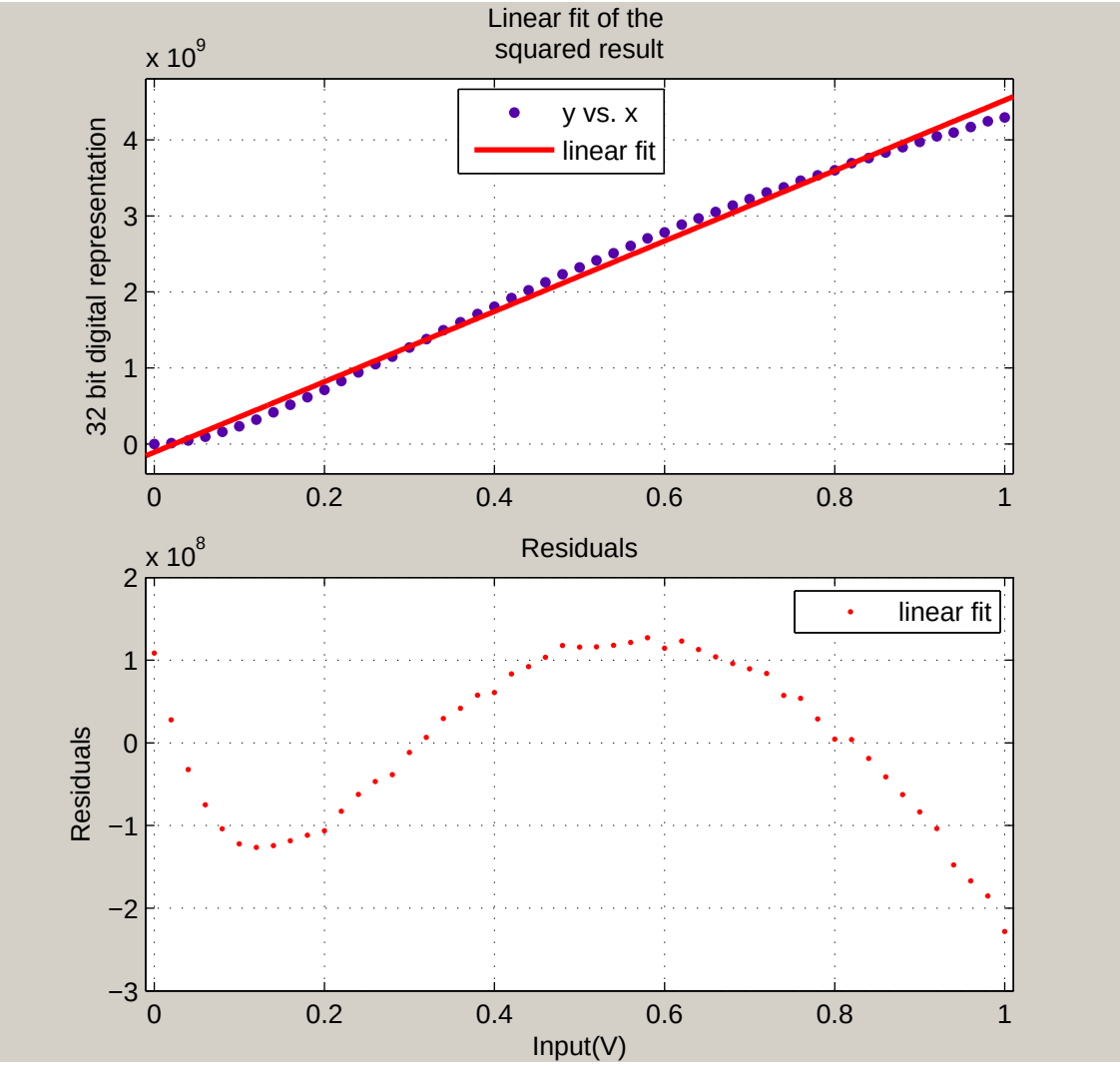

Figure 3.8: Linear fit for the full input range 0-1 V

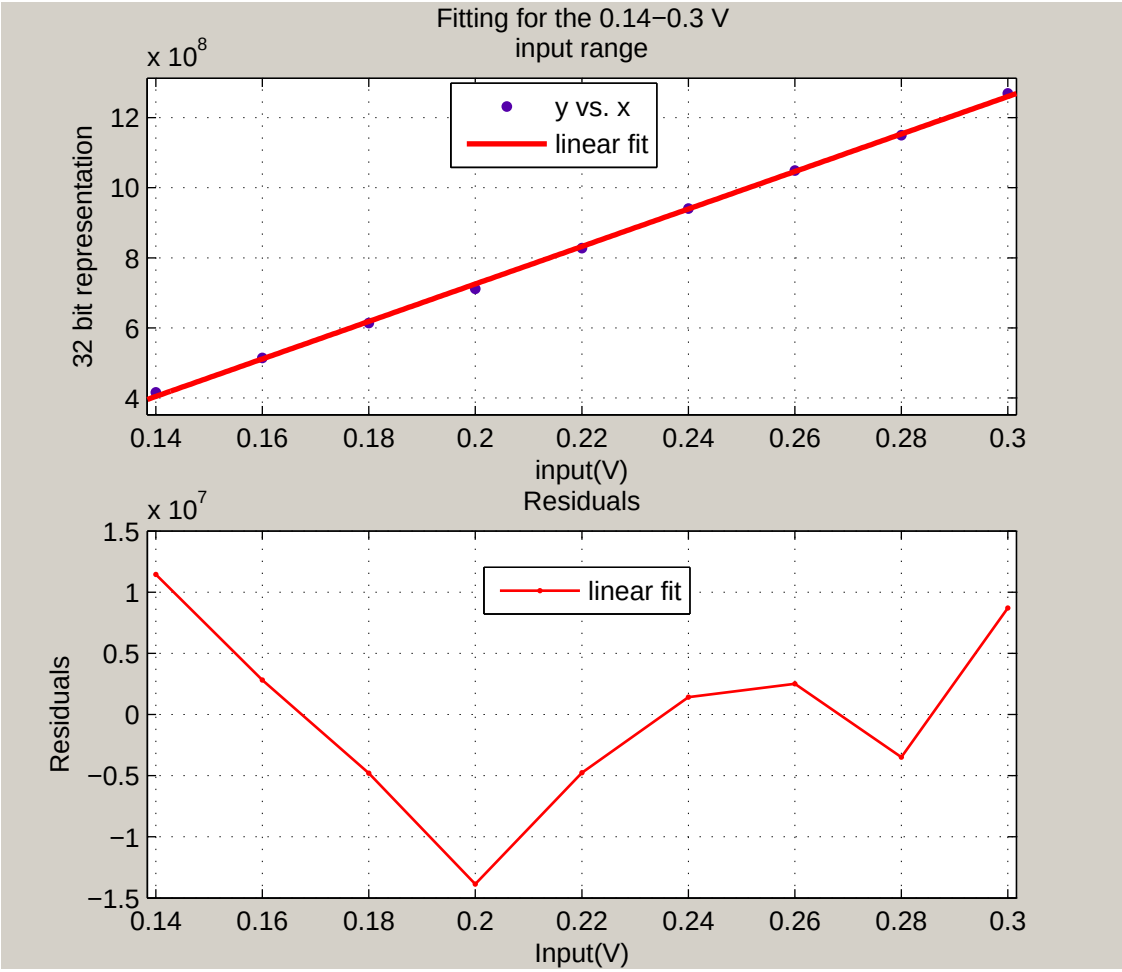

Figure 3.9: Linear fit for the range 0.14-0.3 V with corresponding residuals

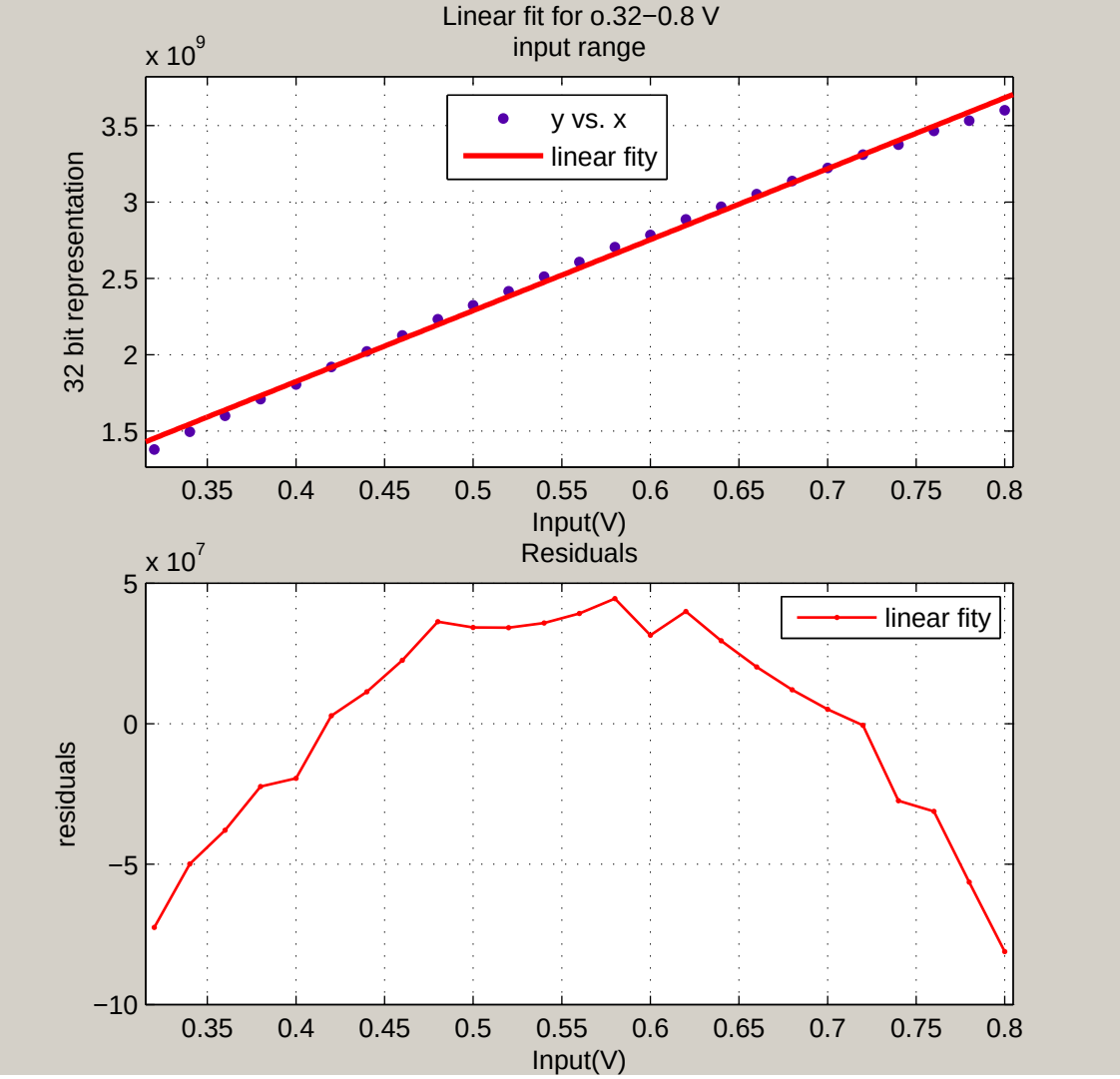

Figure 3.10: Linear fit for the range 0.32-0.8 V with corresponding residuals

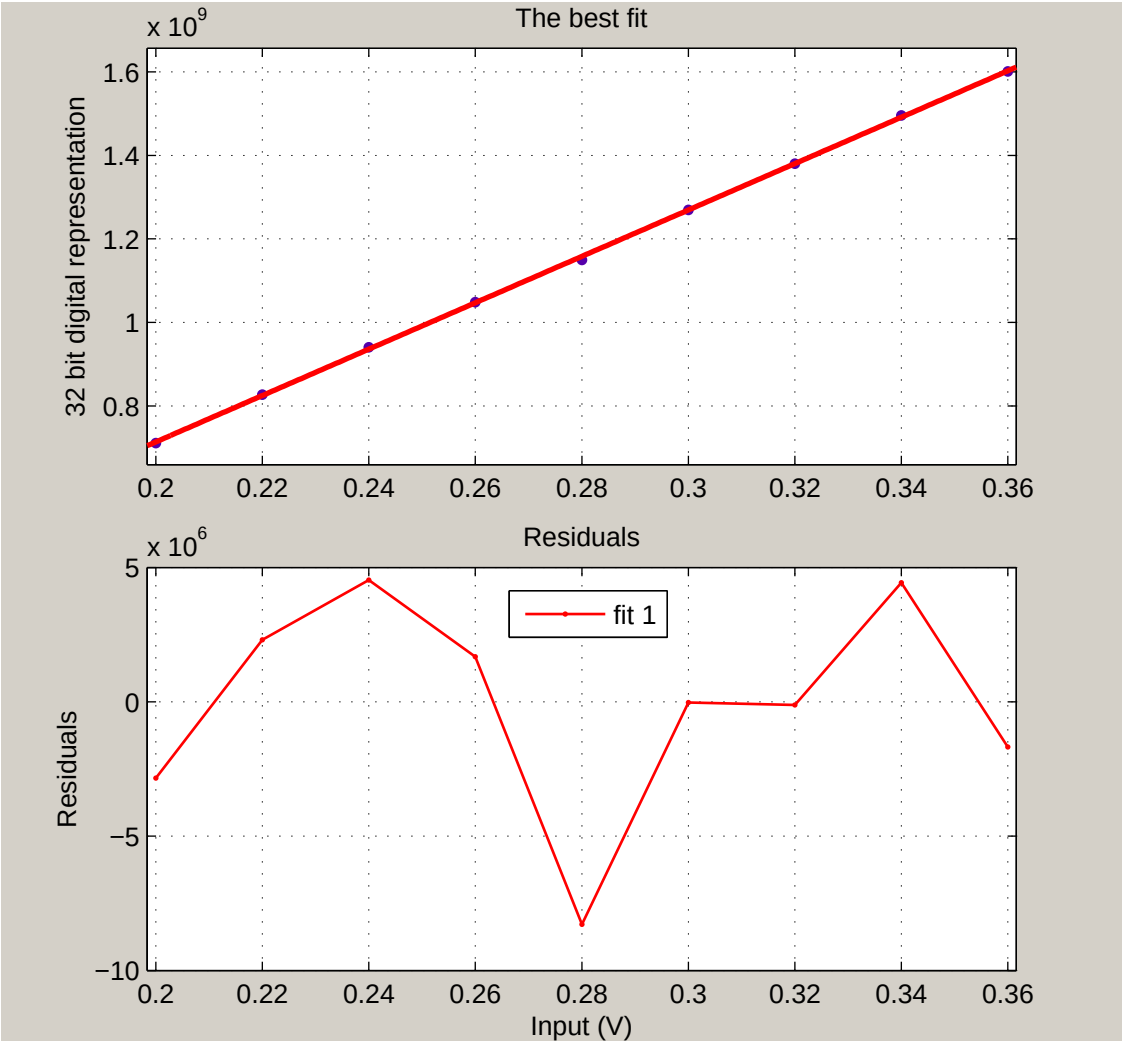

Figure 3.11: The best linear fit and corresponding residuals

for max deviation, but have poor frequency range. The 0,3 - 0,8 V range , has a very good frequency range and acceptable linearization, but has a very high value of max deviation. Another range with high linearity is the 0,14 - 0,3 V range, however the frequency range for this one is not so good. This range also has its disadvantage, namely that it do not have the lowest value for max deviation. The range in table 3.2, 0.7-1 V, has the lowest value for max deviation. It could thus have been the best range for our purpose. The problem is that this range only has a frequency range of 0.359 MHz. This is far below the approximately 1 MHZ required in th FDSM application. The last range in the table, 0.2-0.36 V, has the best linearity result and also lowest max deviation. However the frequency range is not the best. None of the ranges can thus be labeled as the over all best or worst.When selecting a tuning range, trade offs such as linearity, output range and max deviation have to be considered. The range ultimately selected, should depend on the specific application. Some of the ranges are plotted with their linear fit and corresponding residuals in figure 3.8 to 3.11.

3.5 Matlab results

As can be seen from the plots in the preceding chapter and table 3.1, many good results are achieved by the various simulations in Matlab. The original output of the VCO was very non-linear, a square root actually. By applying the linearization technique, a huge improvement in the linearity was obtained, with results from 99.4% match to a linear fit.

The whole input range do not show good linearity results, but there is still room for improvements. We have 5 MHz of range, while approximately 1 MHz is sufficient for the FDSM application. Therefore a smaller portion of the tuning range is selected in order to improve on the linearity. Table 3.1 shows the results over different tuning ranges.

As mentioned in the previous section, trade offs have to be considered when selecting a tuning range. Since our primary concern is linearity, the 0.2 - 0.36 V tuning range is considered as the best for our purpose. This is due to good linearity, while max deviation is also the lowest. A possible drawback of choosing this range could have been that this input voltage range do not result in the best frequency range, but a range of 1.09 MHz is well acceptable for FDSM application. This range is thus considered as the best.

The obtained results are satisfactory. We have to keep in mind, that the proposed technique is developed for a special purpose; to be used with the FDSM modulator. Before drawing any final conclusions about the good linearity, it is therefore necessary to compare the results with other results obtained. A lots of work have been done towards achieving the best linearity for the FDSM A/D converter. Many different approaches are tried out. The results obtained in this thesis is therefore compared to these other results in the next section. Until then, a final conclusion can not be drawn.

3.6 Comparison to other work

In the preceding section the results obtained were presented. To get a better picture of how good these results actually are, the results must be presented in a relative way. For our purpose the results will be compared with other work that is done with focus on getting lowest possible max deviation from a straight line.

One of this is the master thesis of a student here at University of Oslo, Jan Arne Leszczynski, who worked with another approach towards getting linear oscillator [17]. He worked with ring oscillator with focus on getting it as linear as possible. This oscillator was also meant to be used with the FDSM system. In his thesis the oscillator itself had to be linear, unlike our technique where the digital correction is providing improved linearity,

All the FDSM systems up to today have been using ring-oscillators at the input. Ring-oscillator have been selected because of its good linear response. Since the ring-oscillator can often have good linear response, no additional correction circuitry is needed. The linearity thus have been limited by the linearity of the ring oscillator. The main focus of this thesis was to find out if it is possible to achieve any better linearity by using LC-VCO with digital correction circuitry, than the regular ring-oscillator.

As the preceding results show, the LC-VCO achieved very good linearity. The results are compared with the results obtained using ring-oscillators in table 3.3. In [17] two technologies were used, 350nm and 90nm. Three types of oscillators were made with the 350 nm technology, and two with 90 nm. Only the best results from these two technologies are presented here in this table. Another approach towards getting highest possible linear ring oscillator is found in [8], in which

Table 3.3: Linearity comparison

$Oscillator$	$non-linearity(\%)$	Bit
LC-VCO	0.7%	7.12
Ring-oscillator 350nm [17]	0.12%	9,70
Ring-oscillator 90nm [17]	-0.19%	9,04
Ring oscillator, strong inversion [8]	0.5%	7.64
Ring oscillator weak inversion [8]	2.4%	5.34
Linear LC-VCO[2]	0.08%	10.3

the frequency tuning is performed from the bulk terminal of the MOS transistors. Two different approaches are tried here, with the transistors operating in strong- and weak inversion. Both results are presented in table 3.3. A piecewise-linear LC-VCO was also made in [2]. This VCO is also presented in the table.

As there are not presented measured result for all the VCOs, the comparison is done of the simulated results. This way of comparison is be the most fair.

As seen from this table, [2] which is a piecewise linear LC-VCO displays the best result. The results was also obtained without any post processing. LC-VCO is thus a good alternative to the common ring oscillator in FDSM applications. However, we must keep in mind that all results are simulated results, when measured, deviation may occur. To draw an absolute conclusion from simulated results, would be wrong. The results in the table are only meant to give an idea of how good results are obtained. These are not the final results.

The obtained results in this thesis was not better than the others, but they were close. Improvement is possible, if a LC-VCO with better square root output is made. Correcting for the higher-order errors is also a way of getting improved results. This means that LC-VCO is absolutely an alternative to the traditional ring oscillator considering linearity. Depending on your needs the one or the other can be selected out from the criteria listed in table 1.1, or any other criteria one may have.

In theory perfect linearization is achieved by squaring a square root function. The fact that we did not get perfect linearization was totally expected. None of the components are ideal, mismatch between theory and simulation does therefore exist. However, our final goal was

not simulations. The developed linearization technique was supposed to be implemented in hardware, which also is the reason for not applying complex higher-order mathematical corrections. More about the implementation is presented in the next chapter.

Chapter 4

Implementation of the squaring circuit

In this chapter theory regarding implementation is presented. Thereafter VHDL code for a squarer is written to verify the simulations carried out in the previous chapter. The VHDL simulation results are later compared with the Matlab simulation results obtained in the preceding chapter.

4.1 Digital squaring

The approach is to implement this squaring function in digital hardware like FPGA or ASIC. The code of a 16 bit squarer function is made in VHDL. A squaring requires fewer operations than multiplication, resulting in faster and more compact implementation. For the multiplication of two 16 bit words, A and B, it would be generated $16 * 16 = 256$ product-terms(PT),

$$PT(x) = a_i * b_j \qquad i, j = 0...15$$

For squaring, the A and B operand are the same, reducing the number of product-terms to 136.

4.1.1 Partial Product Matrix

The first step in any multiplication is the creation of the Partial Product Matrix (PPM). In order to obtain a small size and fast implementation of a squarer, the PPM should be optimized. As known, squaring is a special

case of multiplication, in which the multiplicand and the multiplier are the same. When the normal multiplication are carried out, for this case, where the multiplier and multiplicand are the same, a very clear pattern appear in the PPM, well visible. The product terms on the both sides of the diagonal are symmetric. The symmetry is a result of the multiplier and multiplicand being the same. Let i and j denote the bit number in the multiplier and multiplicand. To obtain all the PT's, all the a_i bits have to be multiplied with all the a_j bits. Since both are the same, we get the result

$$a_i a_j = a_j a_i.$$

The multiplication of two different numbers do not have this symmetry, because

$$a_i b_j \neq b_i a_j.$$

As a result of the symmetry, the number of PT's are reduced to approximately 50% of an ordinary PPM.
To further reduce the the PPM, another identity of binary multiplication can be used, namely $2 * a = leftshift$, yielding the result that all the PT's which appear twice in a column to be summed, are just shifted one column to the left. The result is that $2a$ in column i can be replaced with a in column $i + 1$
The third and last identity, is that for binary multiplication , we have that

$$a * a = a$$

The result from this is that the terms $a_i * a_i$ can be replaced with just a_i. The and operation for the bits of the same bit position can be avoided. An example of this is shown in equation 4.1.

$$a_3 \; and \; a_3 = a_3 \tag{4.1}$$

The result of the presented squaring special features is that the PPM can be created faster, since requiring fewer and operations and also more compact than a multiplier.

The final result takes the form of a up side down triangle as shown below.

$$
\begin{array}{c}
a_{15}a_{14}a_{13}.....a_2a_1a_0 * a_{15}a_{14}a_{13}.....a_2a_1a_0 \\
\hline
a_{15}a_0 \quad a_{14}a_0.....a_2a_0 \quad a_1a_0 \quad a_0 \\
a_{15}a_1 \quad a_{14}a_1.....a_2a_1 \quad a_1 \quad a_0a_1 \\
a_{15}a_2 \quad a_{14}a_2.....a_2 \quad a_1a_2 \quad a_0a_2 \\
..... \\
..... \\
..... \\
a_{15}a_{14} \quad a_{14}.....a_2a_{14} \quad a_1a_{14} \quad a_0a_{14} \\
a_{15} \quad a_{14}a_{15}.....a_2a_{15} \quad a_1a_{15} \quad a_0a_{15} \\
\hline
a_{15}a_{14} \quad a_{15}a_{13}.....a_1a_0 \quad 0 \quad a_0 \\
..... \\
..... \\
..... \\
..... \\
a_8
\end{array}
$$

In ordinary multiplication, Booth recording is a common used method to reduce the PPM. However, as ¨ shown above the PPM for a squarer is well reduced compared to compared to a multiplier for the same size input. Booth recording is therefore not utilized in the realization of the squarer.

4.2 Adder

When the PPM is optimized, a good way to sum all the PP's is needed. Finding a good algorithm to sum the PP's, can give the biggest reduction in delay. Thus the type of adder to be used, has to be selected carefully. Following, a few adders are presented, with their Pro and Cons, whereas one or more of them are selected.

4.2.1 Ripple Carry Adder

Amongst numerous adders, Ripple Carry Adder (RCA) is a very basic and straightforward adder. Before two bits at position i can be summed, they have to wait until the carry from the previous position, $i-1$, arrives. Thus, before the summation at the Most Significant Bit(MSB) can be done, in worst case the carry must ripple through all the adders from the Least Significant Bit (LSB) to MSB. The carry delay is thus proportional to n, where n is the word length.

$$C_d = n * A_d$$

C_d is the carry delay, n is the word length and A_d the delay of one adder.

Implementing the RCA in hardware is also straightforward with no no complex routing. For the summation of two n bits word, where n is small, the RCA can be the first choice. As long as n is small, the difference in delay, compared to other adder structures is not significant. When choosing any other adder structure than the RCA, the time delay that is saved compared to the RCA must be evaluated against the increased complexity. For small n, the RCA may even be faster than the other adders, because of the complex wiring of the other adders. The carry delay of RCA is proportional to n.

However for the summation of partial products in a multiplier, the version that would be used is the Ripple carry Array Mutiplier (RCAM). This has a worst case carry delay of $2n$ [18].

4.2.2 Carry Save Adder

The Carry Save Adder(CSA) is another type of adder [3]. This adder is faster than the RCA in that manner that each carry bit do not have to ripple through all the adders. Rather the carry bits are stored in a carry array, thereby the name CSA, and the sum bits are stored in a sum array, whereas those two arrays are summed together to produce the final output. In the final step where the two last vectors are accumulated, the type of adder must be selected carefully. A RCA used at this stage, minimizes, or in worst case removes the time saved using CSA. For the final accumulation, a Carry Lookahead Adder is used, which is further explained in the next section. Unlike the linear increase in delay proportional to word length for the RCAM, for which the delay is given as 2n, the CSA delay increase as a logarithmic function citenettet.

$$C_d = n + log(n)$$

[18].

For multiplication, and thereby also squaring, the CSA is most frequently used. This is because unlike the other adders, which outputs one sum bit at the same position as the input bit, and a carry bit to the next position, the CSA outputs two bits at the same position as the input, both a carry and a sum bit. The CSA is a 3:2 compressor, meaning that

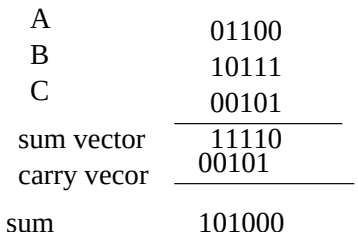

Figure 4.1: carry Save Adder

it takes 3 bits as its input, and reduces them to 2. For the summation of the partial products of a squarer, which has more than 3 levels, 3 and 3 levels are reduced to 2 and 2, until finally two vectors remain, one sum and one carry. There also exists some 4:2 compressor versions of the CSA, but they are not used here as they become very complex. The CSA adder is more complex and requires complex routing, compared to the RCAM. However, we get the benefit of reduced delay at low cost.

When we have more than three input, as the partial products in our squarer, the CSA's are set together in a tree structure. The combination of CSA's in this fashion is referred to as a Wallace Tree. A very simple CSA is shown in figure 4.1.

4.2.3 Carry Lookahead Adder

The partial products is by the Wallace Tree reduced to only two vectors, the sum vector and the carry vector. For the accumulation of this two vectors, the Carry Lookahead Adder (CLA) is used.
CLA has its strength in being very fast, but is rather complex. The complexity is the reason for not using CLA for the partial product summation. Since many summations are required, the whole system would end up being too complex. CLA is used for the final accumulation. As only two vectors are to be summed, the sum and carry vector output from the CSA, the gain in speed is more the loss in form of increased complexity. The delay of a CLA increase logarithmic with the word length [19].

CLAs for more than 4 bits word become highly complex. Addition of words longer than 4 bits is therefore achieved by combibing several 4 bits CLAs.

Two main words for the CLA is *propagate* and *generate*. They hold information about whether there will be generated a carry, or a

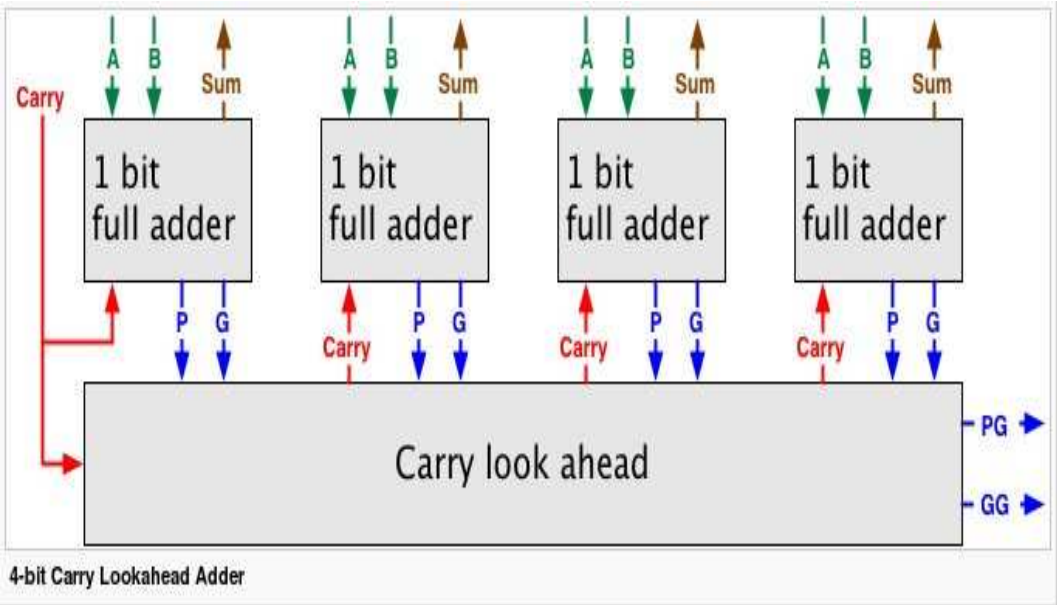

Figure 4.2: A 4 bit CLA adder ([3])

carry will just be propagated at bit position i. Generate equals 1 whenever the addition of two bits a and b is such that they will generate a carry regardless of whether or not the previous stage generated a carry, this represents the binary and operation,

A stage is called propagate when $only$ one of the bits equal 1, representing the binary xor operation. It means that the stage has the potential to generate a carry, but the ganeration of a carry out depends if there is a carry in or not.

Whether the stage will generate a carry or not, depends on whether the stage generates a carry itself, or if it propagates the carry from the previous significant bit position. The equations for these three operations are given below. A four bit CLA adder is shown in figure 4.2.

$$G(A,B) = A\ and\ B$$

$$P(A,B) = A\ xor\ B$$

$$C_i = G_i + P_i C_{i-1}$$

4.2.4 Adder conclusion

The conclusion drawn from the preceding adder presentations is that the CSA is used to sum up the partial products. The reason for this is that the CSA is less complex than the CSA, although it is slower than the CLA. A tree structure is needed for the summation and a tree built up by CLA's would be very complex.

To sum up the final carry- and sum vector, the CLA is preferred. This is due to the high speed of the CLA. The complexity is limited because only two vectors are to be summed

4.3 VHDL code

The VHDL code is made in accordance to the model of [20]. The purpose of this code is to take an 16 bit digital word as its input, square it and output the 32 bit as the final result. The VHDL code is attached in appendix B. The SQ entity is the top entity of the squarer. Within this entity there are several other entities, used as components. The top entity inside the SQ entity is the multiplier. The different components of the squarer are explained further in the following sections. Figure 4.3 shows a plot of the signal flow.

4.3.1 Multiplier

The wallace Tree, Squarebit and DBLC are all used as components within this entity.The multiplier provides the input to the squarebit and get the result back from the DBLC adder.

4.3.2 Squarebit

This is a entity inside the multiplier. What the squarer does is to take the 16 bit input, and produce the Partial Product Patrix (PPM). The PPM consists of 136 bit. The output is sent to the Wallace Tree.

4.3.3 Wallace Tree

The input to the Wallace Tree entity is the partial product bits, which the squarebit outputs. The Wallace tree takes this partial product matrix, consisting of 136 bit, and sums them up. The output of the Wallace tree

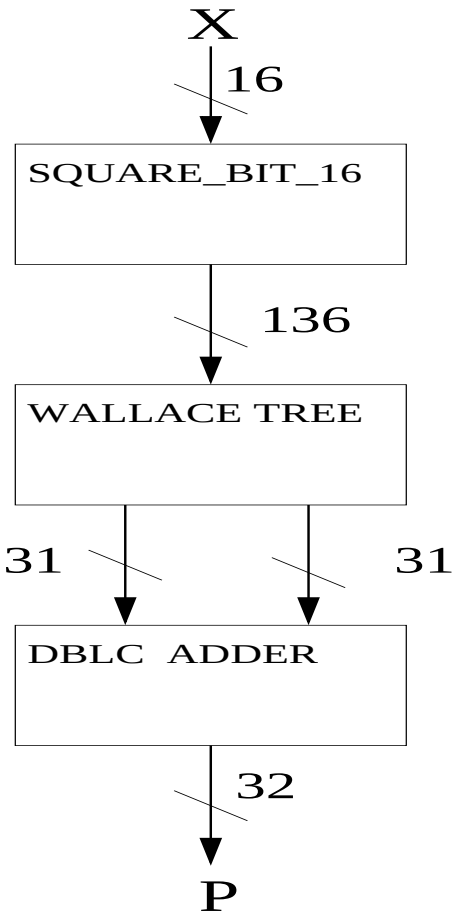

Figure 4.3: Representation of the digital squaring in VHDL

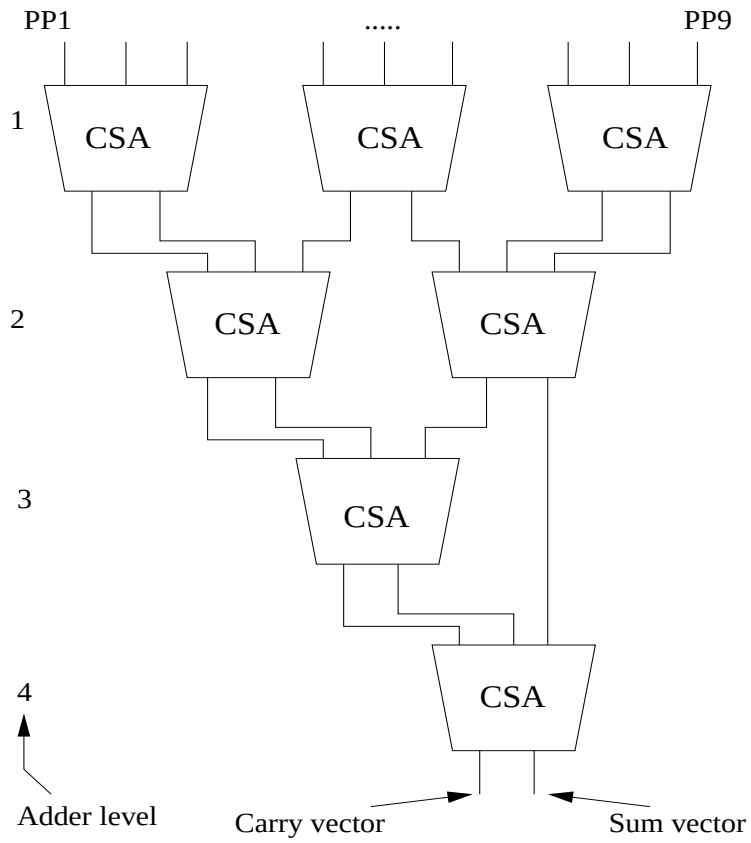

Figure 4.4: Wallace Tree

is two 31 bit vectors, a sum vector and a carry vector. A structure of the wallace Tree is shown in figure 4.3.3

4.3.4 DBLC adder

The DBLC (Distibuted Binary Carry Lookahead Adder) adder is a tree consisting of the CLA adders. This adder takes the two output vectors from the Wallace Tree as its input, sums them up, and produces the final 32 bit result.

4.4 VHDL testbench

A VHDL testbench is made to be able to simulate the VHDL code and to check that it works properly. The Matlab file $dig\_val.m$ (see appendix

A)is used as input to the testbench. This is because the values in this file represents the 16 bit output of the $sinc$ 2 LP-filter at the back end of the FDSM. The content of the file is in decimal form and must therefore be converted to a format that hardware can handle, before the values are used in the testbench. The type conversion is done in the testbench, where these values are type changed from natural to std_logic_vector. The testbench inserts these values to the SQ, which is the Unit Under Test (UUT) and the result is written to another output file. The signal "Q" reprsents the 32 bit squared result. The type of "Q" is std_logic_vector. This type is not supported by the "write" procedure in VHDL. The type of "Q" is therefore changed to bit_vector before it is written to the output file. The output file thus contains an array of binary values. These values can be changed to decimal form in Matlab. This squared result is finally compared with the results of simulation from previous chapter.

4.5 Physical implementation

Only the VHDL code is written in this thesis. This is because our primary concern has been the linearity. Any other simulations to get any area or delay results for the written code is not done. The area and timing constraaints dependens heavily on the device on which the code is implemented. Some devices are optimized for speed, while others are optimized for area. To get one general result for area and time delay is thus impossible, as long we do not specify any target device on which the code is to be implemented.

4.6 VHDL simulations

A waveform plot of the squarer output is presented here to verify the results, and also to verify that the code actually works properly.

4.7 VHDL results

The results from the VHDL testbench are written to the file $out.m$. These are,as expected exactly the same as obtained in Matlab. Since the code only performs a squaring, no deviations exist. Both the Matlab and VHDL results are plotted in figure 4.7. They totally overlap each other. To make it easier to separate between the two, diffferent colors are used

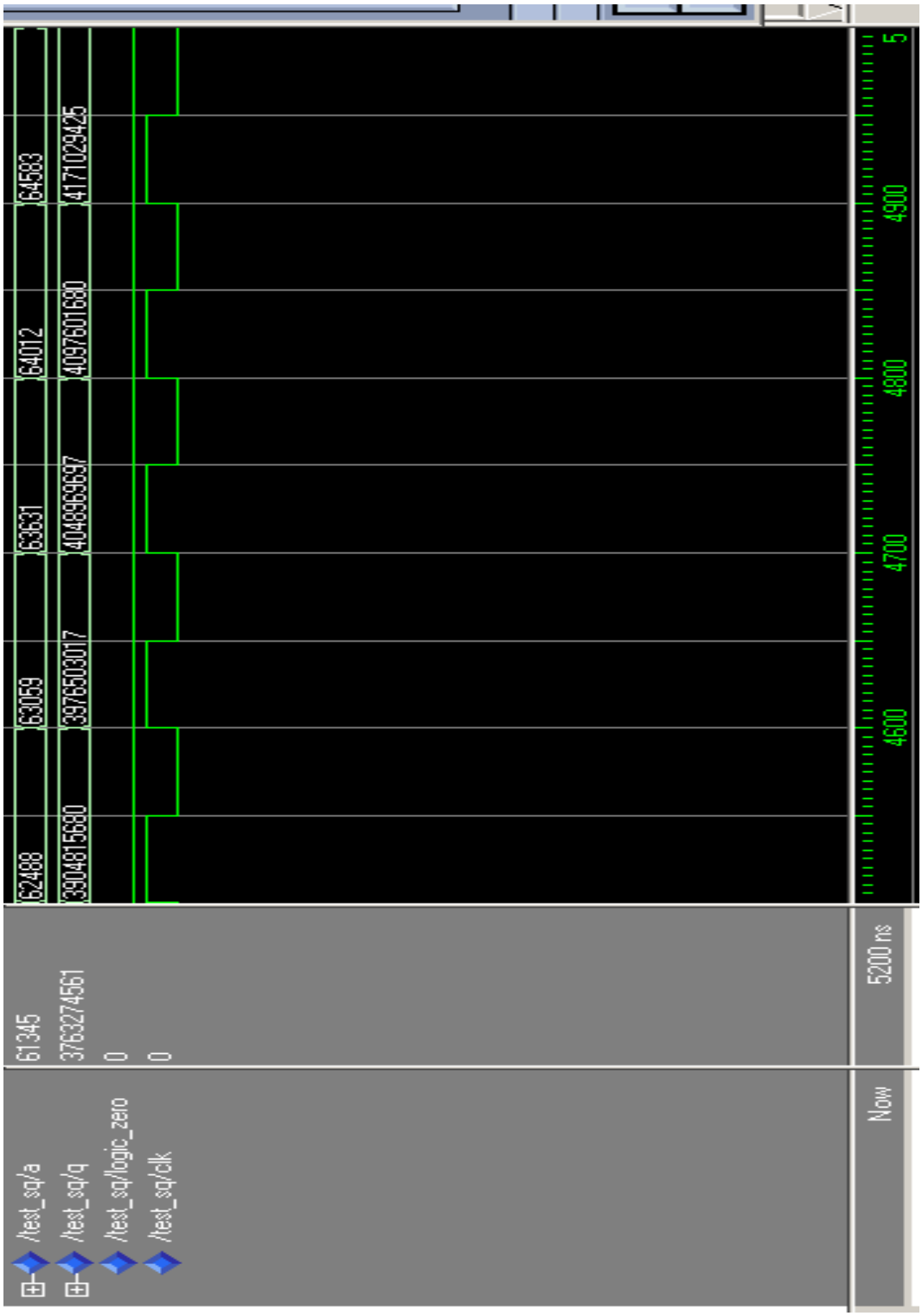

Figure 4.5: The waveform, showing the output of the digital squarer

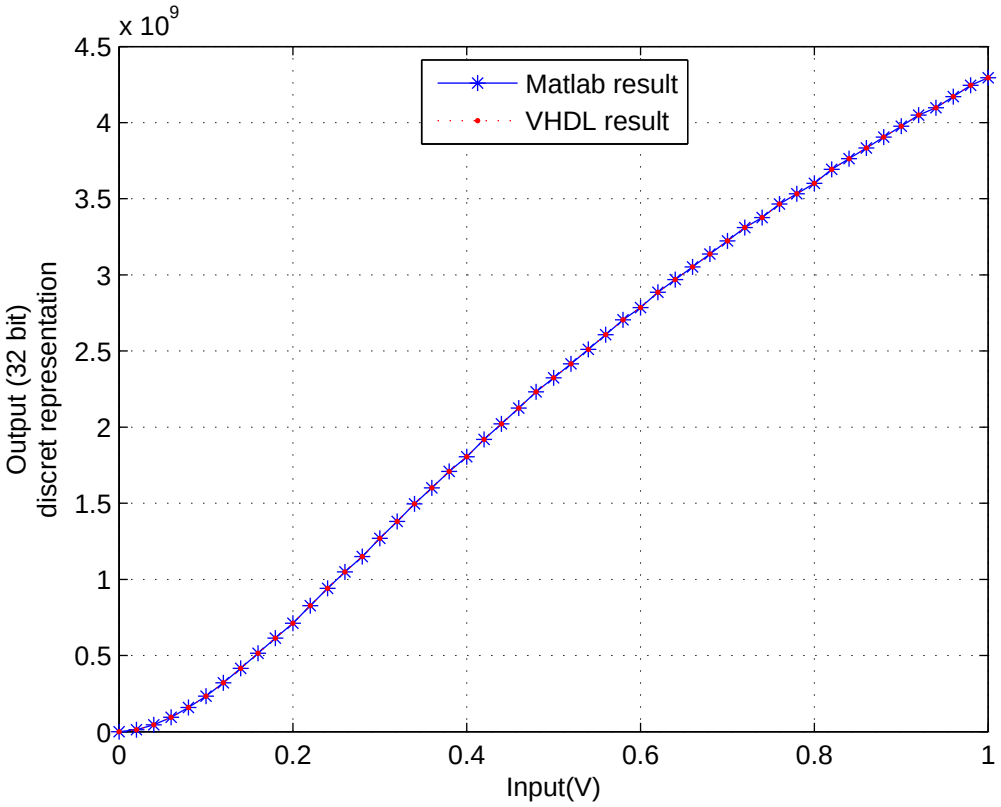

Figure 4.6: The Matlab and VHDL result plotted together

together with different marks. The figure confirms that the code works properly and also that the developed technique is possible to implement in hardware.

Chapter 5

Conclusion

5.1 Conclusion

The focus of this thesis has been to investigate the possibilities to use a LC-VCO in the FDSM application. A tradition have been to always use the ring-oscillator as the FM modulator in front of the FDSM system. We wanted to try some other oscallitors in our search for improved linearity

The new idea that we have been working with is to use LC-VCO in front of the FDSM, with digital correction, instead of the regular ring-oscillator, and see if any better linearity is obtained. . This thesis have somehow been an theoretical introduction to this new idea. Considering linearity good results are obtained by the proposed technique. A non-linear VCO was improved significant. We have to keep in mind the fact that this technique was developed to be used in the FDSM application. Therefor the results must be seen together with other results obtained in the same field. In this context the results were not the best.

The theoretical foundation is build, besides analysis and simulations verifying the theory. A working hardware code is also written which shows that the technique is possible to implement in hardware. It is shown that this new idea is absolutely competitive to the ring-oscillator based FDSM. It is therefore recommended to take this idea further.
Both the Matlab simulation results and VHDL results shows good linearity. Nonlinearity as low as 0.007 is obtained. This is not better than the best reported result for the ring oscillator, but is very close. The fact that the obtained results are not better than the ring oscillator, do not mean that the results are bad.

The linearization technique depends on the LC-VCO output characteristic. For the technique to be optimal the LC-VCO must have a very good square root characteristic. The LC-VCO we have been working with in this thesis was not developed with that focus. It is thought that with a LC-VCO created with especially focus at getting as fine square root characteristic as possible, better results may be expected. A way of doing this may be to get the voltage to capacitance characteristic as linear as possible. We have only done a squaring to obtain the results. By correcting the higher-order errors, improvements are possible. This will be at the cost of increased complexity for the hardware implementation.

5.2 Future work

As the conclusion is drawn that the idea of using LC-VCO in the FDSM application certainly has a future, some words must be said about what work is to be done further. In this thesis is done the theory, analysis and simulations. Only VHDL code is written. The next natural step would be an actual implementation in hardware of the VHDL code, FPGA or ASIC, and then check the result. When implementing, parameters as delay and area should be kept in mind. Maybe some changes in the code are needed to get satisfactory values for the area and delay. Power consumption should also be measured. Since one of the strengths of the FDSM is low power consumption, it should be focus on keeping the power consumption low.

The next step could be to set together the include an actual FDSM in between the LC-VCO and the digital correction, namely set up the whole system. This will be the most interesting part, namely to check the results when the whole system is set together. In this thesis an ideal FDSM have been assumed, it will be interesting to see how results are with the real system. It will also be interesting to see how much the real results deviate from our results, which are based on an ideal FDSM.

Some other linearization techniques may also be tried out. Maybe some kind of adaptive linearization. The proposed technique in this thesis may also produce some better results if a LC-VCO with focus on the square root characteristic is built.

As shown in the Matlab results section, better results are obtained by using higher degree of correction. A suggestion is also to make a

complex circuit which corrigates for errors up to 4th or 5th degree. It will be a complex circuit, but the linearity will also be improved.

Appendix A

Matlab script

```
% The matlab code for the process from the VCO output, via
%frequency dividing and A/D conversion, to squaring.

%The original values
figure()
plot(bjorn_orig_value(:,1),bjorn_orig_value(:,2))
title('The vco response')
xlabel('Input voltage')
ylabel('Out frequency')



% divide the vco output values, to get values in a
% range suitable for the FDSM.

Freq_div_val(:,1)= bjorn_orig_value(:,1);
Freq_div_val(:,2)= bjorn_orig_value(:,2)./(2^6);
figure()
% column1= input voltage
% column2= output frequencies

plot(Freq_div_val(:,1),Freq_div_val(:,2))
title('The new frequency range')
xlabel('Input voltage')
ylabel('Out frequency')

% Remove the offset
a=min(Freq_div_val(:,2));
b=max(Freq_div_val(:,2));
offset_rem(:,1)= Freq_div_val(:,1);
offset_rem(:,2)= Freq_div_val(:,2)- a;
freq_range= max(offset_rem(:,2))


% Scale to the region 0-1
```

```
normalized(:,1)= Freq_div_val(:,1);
normalized(:,2)= offset_rem(:,2)./(b-a);

% 16 bit rpresentation
dig_val(:,1)= normalized(:,1);
dig_val(:,2)= round (normalized(:,2).*(2^16));

figure()
plot(dig_val(:,1),dig_val(:,2))
title('The 16 bit digital representation')
xlabel('Input voltage')
ylabel('Discret value')


% linearisation, by squaring. giving 32 bit result
finale(:,1)= dig_val(:,1);
finale(:,2)= dig_val(:,2).^2;

figure()
plot(finale(:,1),finale(:,2))
title('The 32 bit squared value')
xlabel('Input voltage')
ylabel('Squared discret value')
```

Appendix B

Top entity

```
library ieee;
use ieee.std_logic_1164.all;


entity SQ is
  port(X: in std_logic_vector(15 downto 0);
       CLK: in std_logic;
       -- CLK only used with buffering/pipelining/accumulate
       P: out std_logic_vector(31 downto 0));
end SQ;

library ieee;

use ieee.std_logic_1164.all;
architecture A of SQ is
  component MULTIPLIER_16_16
    port(MULTIPLICAND: in std_logic_vector(0 to 15);
         PHI: in std_logic;
         RESULT: out std_logic_vector(0 to 31));
  end component;
  signal A: std_logic_vector(0 to 15);
  signal Q: std_logic_vector(0 to 31);
  signal LOGIC_ZERO: std_logic;
begin
  LOGIC_ZERO <= '0';
  U1: MULTIPLIER_16_16 port map(A,CLK,Q);
  -- std_logic_vector reversals to incorporate decreasing vectors
  A(0) <= X(0);
  A(1) <= X(1);
  A(2) <= X(2);
  A(3) <= X(3);
  A(4) <= X(4);
  A(5) <= X(5);
  A(6) <= X(6);
  A(7) <= X(7);
  A(8) <= X(8);
```

```
  A(9) <= X(9);
  A(10) <= X(10);
  A(11) <= X(11);
  A(12) <= X(12);
  A(13) <= X(13);
  A(14) <= X(14);
  A(15) <= X(15);
  P(0) <= Q(0);
  P(1) <= Q(1);
  P(2) <= Q(2);
  P(3) <= Q(3);
  P(4) <= Q(4);
  P(5) <= Q(5);
  P(6) <= Q(6);
  P(7) <= Q(7);
  P(8) <= Q(8);
  P(9) <= Q(9);
  P(10) <= Q(10);
  P(11) <= Q(11);
  P(12) <= Q(12);
  P(13) <= Q(13);
  P(14) <= Q(14);
  P(15) <= Q(15);
  P(16) <= Q(16);
  P(17) <= Q(17);
  P(18) <= Q(18);
  P(19) <= Q(19);
  P(20) <= Q(20);
  P(21) <= Q(21);
  P(22) <= Q(22);
  P(23) <= Q(23);
  P(24) <= Q(24);
  P(25) <= Q(25);
  P(26) <= Q(26);
  P(27) <= Q(27);
  P(28) <= Q(28);
  P(29) <= Q(29);
  P(30) <= Q(30);
  P(31) <= Q(31);
end A;
```

Multiplier

```
library ieee;
use ieee.std_logic_1164.all;


entity MULTIPLIER_16_16 is
port
(
        MULTIPLICAND: in std_logic_vector(0 to 15);
        PHI: in std_logic;
        RESULT: out std_logic_vector(0 to 31)
);
end MULTIPLIER_16_16;


library ieee;
use ieee.std_logic_1164.all;
architecture MULTIPLIER of MULTIPLIER_16_16 is
component SQUARE_BIT_16
port
(
                OPA: in std_logic_vector(0 to 15);
                SUMMAND: out std_logic_vector(0 to 136)
);
end component;
component WALLACE_16_16
port
(
        SUMMAND: in std_logic_vector(0 to 136);
        CARRY: out std_logic_vector(0 to 30);
        SUM: out std_logic_vector(0 to 30)
);
end component;
component DBLCADDER_32_32
port
(
        OPA:in std_logic_vector(0 to 31);
        OPB:in std_logic_vector(0 to 31);
        CIN:in std_logic;
        PHI:in std_logic;
```

```
        SUM:out std_logic_vector(0 to 31)
);
end component;

signal PPBIT:std_logic_vector(0 to 136);
signal INT_CARRY: std_logic_vector(0 to 32);
signal INT_SUM: std_logic_vector(0 to 31);
signal LOGIC_ZERO: std_logic;

begin -- Architecture

LOGIC_ZERO <= '0';
S:SQUARE_BIT_16
        port map
        (
                OPA(0 to 15) => MULTIPLICAND(0 to 15),
                SUMMAND(0 to 136) => PPBIT(0 to 136)
        );
W:WALLACE_16_16
        port map
        (
                SUMMAND(0 to 136) => PPBIT(0 to 136),
                CARRY(0 to 30) => INT_CARRY(1 to 31),
                SUM(0 to 30) => INT_SUM(0 to 30)
        );
INT_CARRY(0) <= LOGIC_ZERO;
INT_SUM(31) <= LOGIC_ZERO;
D:DBLCADDER_32_32
        port map
        (
                OPA(0 to 31) => INT_SUM(0 to 31),
                OPB(0 to 31) => INT_CARRY(0 to 31),
                CIN => LOGIC_ZERO,
                PHI => PHI,
                SUM(0 to 31) => RESULT(0 to 31)
        );
end MULTIPLIER;
```

Squarebit

```
library ieee;
use ieee.std_logic_1164.all;



entity SQUARE_BIT_16 is
port
(
                OPA: in std_logic_vector(0 to 15);
                SUMMAND: out std_logic_vector(0 to 136)
);
end SQUARE_BIT_16;

architecture SQUARE_BIT of SQUARE_BIT_16 is
begin
-- Gates in square matrix column 0
SUMMAND(0) <= OPA(0);
-- End square matrix column 0
-- Gates in square matrix column 1
-- End square matrix column 1
-- Gates in square matrix column 2
SUMMAND(1) <= OPA(0) and OPA(1);
SUMMAND(2) <= OPA(1);
-- End square matrix column 2
-- Gates in square matrix column 3
SUMMAND(3) <= OPA(0) and OPA(2);
-- End square matrix column 3
-- Gates in square matrix column 4
SUMMAND(4) <= OPA(0) and OPA(3);
SUMMAND(5) <= OPA(1) and OPA(2);
SUMMAND(6) <= OPA(2);
-- End square matrix column 4
-- Gates in square matrix column 5
SUMMAND(7) <= OPA(0) and OPA(4);
SUMMAND(8) <= OPA(1) and OPA(3);
-- End square matrix column 5
-- Gates in square matrix column 6
SUMMAND(9) <= OPA(0) and OPA(5);
SUMMAND(10) <= OPA(1) and OPA(4);
```

```
SUMMAND(11) <= OPA(2) and OPA(3);
SUMMAND(12) <= OPA(3);
-- End square matrix column 6
-- Gates in square matrix column 7
SUMMAND(13) <= OPA(0) and OPA(6);
SUMMAND(14) <= OPA(1) and OPA(5);
SUMMAND(15) <= OPA(2) and OPA(4);
-- End square matrix column 7
-- Gates in square matrix column 8
SUMMAND(16) <= OPA(0) and OPA(7);
SUMMAND(17) <= OPA(1) and OPA(6);
SUMMAND(18) <= OPA(2) and OPA(5);
SUMMAND(19) <= OPA(3) and OPA(4);
SUMMAND(20) <= OPA(4);
-- End square matrix column 8
-- Gates in square matrix column 9
SUMMAND(21) <= OPA(0) and OPA(8);
SUMMAND(22) <= OPA(1) and OPA(7);
SUMMAND(23) <= OPA(2) and OPA(6);
SUMMAND(24) <= OPA(3) and OPA(5);
-- End square matrix column 9
-- Gates in square matrix column 10
SUMMAND(25) <= OPA(0) and OPA(9);
SUMMAND(26) <= OPA(1) and OPA(8);
SUMMAND(27) <= OPA(2) and OPA(7);
SUMMAND(28) <= OPA(3) and OPA(6);
SUMMAND(29) <= OPA(4) and OPA(5);
SUMMAND(30) <= OPA(5);
-- End square matrix column 10
-- Gates in square matrix column 11
SUMMAND(31) <= OPA(0) and OPA(10);
SUMMAND(32) <= OPA(1) and OPA(9);
SUMMAND(33) <= OPA(2) and OPA(8);
SUMMAND(34) <= OPA(3) and OPA(7);
SUMMAND(35) <= OPA(4) and OPA(6);
-- End square matrix column 11
-- Gates in square matrix column 12
SUMMAND(36) <= OPA(0) and OPA(11);
SUMMAND(37) <= OPA(1) and OPA(10);
SUMMAND(38) <= OPA(2) and OPA(9);
SUMMAND(39) <= OPA(3) and OPA(8);
```

```
SUMMAND(40) <= OPA(4) and OPA(7);
SUMMAND(41) <= OPA(5) and OPA(6);
SUMMAND(42) <= OPA(6);
-- End square matrix column 12
-- Gates in square matrix column 13
SUMMAND(43) <= OPA(0) and OPA(12);
SUMMAND(44) <= OPA(1) and OPA(11);
SUMMAND(45) <= OPA(2) and OPA(10);
SUMMAND(46) <= OPA(3) and OPA(9);
SUMMAND(47) <= OPA(4) and OPA(8);
SUMMAND(48) <= OPA(5) and OPA(7);
-- End square matrix column 13
-- Gates in square matrix column 14
SUMMAND(49) <= OPA(0) and OPA(13);
SUMMAND(50) <= OPA(1) and OPA(12);
SUMMAND(51) <= OPA(2) and OPA(11);
SUMMAND(52) <= OPA(3) and OPA(10);
SUMMAND(53) <= OPA(4) and OPA(9);
SUMMAND(54) <= OPA(5) and OPA(8);
SUMMAND(55) <= OPA(6) and OPA(7);
SUMMAND(56) <= OPA(7);
-- End square matrix column 14
-- Gates in square matrix column 15
SUMMAND(57) <= OPA(0) and OPA(14);
SUMMAND(58) <= OPA(1) and OPA(13);
SUMMAND(59) <= OPA(2) and OPA(12);
SUMMAND(60) <= OPA(3) and OPA(11);
SUMMAND(61) <= OPA(4) and OPA(10);
SUMMAND(62) <= OPA(5) and OPA(9);
SUMMAND(63) <= OPA(6) and OPA(8);
SUMMAND(64) <= OPA(15);
-- End square matrix column 15
-- Gates in square matrix column 16
SUMMAND(65) <= (OPA(0)) and OPA(15);
SUMMAND(66) <= OPA(1) and OPA(14);
SUMMAND(67) <= OPA(2) and OPA(13);
SUMMAND(68) <= OPA(3) and OPA(12);
SUMMAND(69) <= OPA(4) and OPA(11);
SUMMAND(70) <= OPA(5) and OPA(10);
SUMMAND(71) <= OPA(6) and OPA(9);
SUMMAND(72) <= OPA(7) and OPA(8);
```

```
SUMMAND(73) <= OPA(8);
-- End square matrix column 16
-- Gates in square matrix column 17
SUMMAND(74) <= (OPA(1)) and OPA(15);
SUMMAND(75) <= OPA(2) and OPA(14);
SUMMAND(76) <= OPA(3) and OPA(13);
SUMMAND(77) <= OPA(4) and OPA(12);
SUMMAND(78) <= OPA(5) and OPA(11);
SUMMAND(79) <= OPA(6) and OPA(10);
SUMMAND(80) <= OPA(7) and OPA(9);
-- End square matrix column 17
-- Gates in square matrix column 18
SUMMAND(81) <= (OPA(2)) and OPA(15);
SUMMAND(82) <= OPA(3) and OPA(14);
SUMMAND(83) <= OPA(4) and OPA(13);
SUMMAND(84) <= OPA(5) and OPA(12);
SUMMAND(85) <= OPA(6) and OPA(11);
SUMMAND(86) <= OPA(7) and OPA(10);
SUMMAND(87) <= OPA(8) and OPA(9);
SUMMAND(88) <= OPA(9);
-- End square matrix column 18
-- Gates in square matrix column 19
SUMMAND(89) <= (OPA(3)) and OPA(15);
SUMMAND(90) <= OPA(4) and OPA(14);
SUMMAND(91) <= OPA(5) and OPA(13);
SUMMAND(92) <= OPA(6) and OPA(12);
SUMMAND(93) <= OPA(7) and OPA(11);
SUMMAND(94) <= OPA(8) and OPA(10);
-- End square matrix column 19
-- Gates in square matrix column 20
SUMMAND(95) <= (OPA(4)) and OPA(15);
SUMMAND(96) <= OPA(5) and OPA(14);
SUMMAND(97) <= OPA(6) and OPA(13);
SUMMAND(98) <= OPA(7) and OPA(12);
SUMMAND(99) <= OPA(8) and OPA(11);
SUMMAND(100) <= OPA(9) and OPA(10);
SUMMAND(101) <= OPA(10);
-- End square matrix column 20
-- Gates in square matrix column 21
SUMMAND(102) <= (OPA(5)) and OPA(15);
SUMMAND(103) <= OPA(6) and OPA(14);
```

```
SUMMAND(104) <= OPA(7) and OPA(13);
SUMMAND(105) <= OPA(8) and OPA(12);
SUMMAND(106) <= OPA(9) and OPA(11);
-- End square matrix column 21
-- Gates in square matrix column 22
SUMMAND(107) <= (OPA(6)) and OPA(15);
SUMMAND(108) <= OPA(7) and OPA(14);
SUMMAND(109) <= OPA(8) and OPA(13);
SUMMAND(110) <= OPA(9) and OPA(12);
SUMMAND(111) <= OPA(10) and OPA(11);
SUMMAND(112) <= OPA(11);
-- End square matrix column 22
-- Gates in square matrix column 23
SUMMAND(113) <= (OPA(7)) and OPA(15);
SUMMAND(114) <= OPA(8) and OPA(14);
SUMMAND(115) <= OPA(9) and OPA(13);
SUMMAND(116) <= OPA(10) and OPA(12);
-- End square matrix column 23
-- Gates in square matrix column 24
SUMMAND(117) <= (OPA(8)) and OPA(15);
SUMMAND(118) <= OPA(9) and OPA(14);
SUMMAND(119) <= OPA(10) and OPA(13);
SUMMAND(120) <= OPA(11) and OPA(12);
SUMMAND(121) <= OPA(12);
-- End square matrix column 24
-- Gates in square matrix column 25
SUMMAND(122) <= (OPA(9)) and OPA(15);
SUMMAND(123) <= OPA(10) and OPA(14);
SUMMAND(124) <= OPA(11) and OPA(13);
-- End square matrix column 25
-- Gates in square matrix column 26
SUMMAND(125) <= (OPA(10)) and OPA(15);
SUMMAND(126) <= OPA(11) and OPA(14);
SUMMAND(127) <= OPA(12) and OPA(13);
SUMMAND(128) <= OPA(13);
-- End square matrix column 26
-- Gates in square matrix column 27
SUMMAND(129) <= (OPA(11)) and OPA(15);
SUMMAND(130) <= OPA(12) and OPA(14);
-- End square matrix column 27
-- Gates in square matrix column 28
```

```
SUMMAND(131) <= (OPA(12)) and OPA(15);
SUMMAND(132) <= OPA(13) and OPA(14);
SUMMAND(133) <= OPA(14);
-- End square matrix column 28
-- Gates in square matrix column 29
SUMMAND(134) <= (OPA(13)) and OPA(15);
-- End square matrix column 29
-- Gates in square matrix column 30
SUMMAND(135) <= (OPA(14)) and OPA(15);
SUMMAND(136) <= OPA(15);
-- End square matrix column 30
-- Gates in square matrix column 31
-- End square matrix column 31
end SQUARE_BIT;
```

Wallace Tree

```
library ieee;
use ieee.std_logic_1164.all;


entity FULL_ADDER is
port
(
        DATA_A, DATA_B, DATA_C: in std_logic;
        SAVE, CARRY: out std_logic
);
end FULL_ADDER;

library ieee;
use ieee.std_logic_1164.all;
entity HALF_ADDER is
port
(
        DATA_A, DATA_B: in std_logic;
        SAVE, CARRY: out std_logic
);
end HALF_ADDER;

library ieee;
use ieee.std_logic_1164.all;
entity WALLACE_16_16 is
port
(
        SUMMAND: in std_logic_vector(0 to 136);
        CARRY: out std_logic_vector(0 to 30);
        SUM: out std_logic_vector(0 to 30)
);
end WALLACE_16_16;



architecture FULL_ADDER of FULL_ADDER is

        signal TMP: std_logic;
```

```
begin

        TMP <= DATA_A xor DATA_B;
        SAVE <= TMP xor DATA_C;
        CARRY <= not((not (TMP and DATA_C)) and
          (not (DATA_A and DATA_B)));
end FULL_ADDER;


architecture HALF_ADDER of HALF_ADDER is

begin
        SAVE <= DATA_A xor DATA_B;
        CARRY <= DATA_A and DATA_B;
end HALF_ADDER;


--
-- Wallace tree architecture
--

architecture WALLACE of WALLACE_16_16 is

-- Components used in the netlist

component FULL_ADDER
port
(
        DATA_A, DATA_B, DATA_C: in std_logic;
        SAVE, CARRY: out std_logic
);
end component;
component HALF_ADDER
port
(
        DATA_A, DATA_B: in std_logic;
        SAVE, CARRY: out std_logic
);
end component;

-- Signals used inside the wallace trees
```

```
        signal INT_CARRY: std_logic_vector(0 to 73);
        signal INT_SUM: std_logic_vector(0 to 110);

begin -- netlist

-- Begin WT-branch 1
---- Begin NO stage
SUM(0) <= SUMMAND(0); -- At Level 1
CARRY(0) <= '0';
---- End NO stage
-- End WT-branch 1

-- Begin WT-branch 2
-- An empty column!
SUM(1) <= '0';
CARRY(1) <= '0';
-- End WT-branch 2

-- Begin WT-branch 3
---- Begin HA stage
HA_0:HALF_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(1), DATA_B => SUMMAND(2),
                SAVE => SUM(2), CARRY => CARRY(2)
        );
---- End HA stage
-- End WT-branch 3

-- Begin WT-branch 4
---- Begin NO stage
SUM(3) <= SUMMAND(3); -- At Level 1
CARRY(3) <= '0';
---- End NO stage
-- End WT-branch 4

-- Begin WT-branch 5
---- Begin FA stage
FA_0:FULL_ADDER -- At Level 1
        port map
```

```
        (
                DATA_A => SUMMAND(4), DATA_B => SUMMAND(5),
 DATA_C => SUMMAND(6),
                SAVE => SUM(4), CARRY => CARRY(4)
        );
---- End FA stage
-- End WT-branch 5

-- Begin WT-branch 6
---- Begin HA stage
HA_1:HALF_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(7), DATA_B => SUMMAND(8),
                SAVE => SUM(5), CARRY => CARRY(5)
        );
---- End HA stage
-- End WT-branch 6

-- Begin WT-branch 7
---- Begin FA stage
FA_1:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(9), DATA_B => SUMMAND(10),
 DATA_C => SUMMAND(11),
                SAVE => INT_SUM(0), CARRY => INT_CARRY(0)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(1) <= SUMMAND(12); -- At Level 1
---- End NO stage
---- Begin HA stage
HA_2:HALF_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(0), DATA_B => INT_SUM(1),
                SAVE => SUM(6), CARRY => CARRY(6)
        );
---- End HA stage
-- End WT-branch 7
```

```
-- Begin WT-branch 8
---- Begin FA stage
FA_2:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(13), DATA_B => SUMMAND(14),
 DATA_C => SUMMAND(15),
                SAVE => INT_SUM(2), CARRY => INT_CARRY(1)
        );
---- End FA stage
---- Begin HA stage
HA_3:HALF_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(2), DATA_B => INT_CARRY(0),
                SAVE => SUM(7), CARRY => CARRY(7)
        );
---- End HA stage
-- End WT-branch 8

-- Begin WT-branch 9
---- Begin FA stage
FA_3:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(16), DATA_B => SUMMAND(17),
 DATA_C => SUMMAND(18),
                SAVE => INT_SUM(3), CARRY => INT_CARRY(2)
        );
---- End FA stage
---- Begin HA stage
HA_4:HALF_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(19), DATA_B => SUMMAND(20),
                SAVE => INT_SUM(4), CARRY => INT_CARRY(3)
        );
---- End HA stage
---- Begin FA stage
FA_4:FULL_ADDER -- At Level 2
```

```
        port map
        (
                DATA_A => INT_SUM(3), DATA_B => INT_SUM(4),
 DATA_C => INT_CARRY(1),
                SAVE => SUM(8), CARRY => CARRY(8)
        );
---- End FA stage
-- End WT-branch 9

-- Begin WT-branch 10
---- Begin FA stage
FA_5:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(21), DATA_B => SUMMAND(22),
 DATA_C => SUMMAND(23),
                SAVE => INT_SUM(5), CARRY => INT_CARRY(4)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(6) <= SUMMAND(24); -- At Level 1
---- End NO stage
---- Begin FA stage
FA_6:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(5), DATA_B => INT_SUM(6),
 DATA_C => INT_CARRY(2),
                SAVE => INT_SUM(7), CARRY => INT_CARRY(5)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(8) <= INT_CARRY(3); -- At Level 2
---- End NO stage
---- Begin HA stage
HA_5:HALF_ADDER -- At Level 3
        port map
        (
                DATA_A => INT_SUM(7), DATA_B => INT_SUM(8),
                SAVE => SUM(9), CARRY => CARRY(9)
        );
```

```
---- End HA stage
-- End WT-branch 10

-- Begin WT-branch 11
---- Begin FA stage
FA_7:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(25), DATA_B => SUMMAND(26),
 DATA_C => SUMMAND(27),
                SAVE => INT_SUM(9), CARRY => INT_CARRY(6)
        );
---- End FA stage
---- Begin FA stage
FA_8:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(28), DATA_B => SUMMAND(29),
 DATA_C => SUMMAND(30),
                SAVE => INT_SUM(10), CARRY => INT_CARRY(7)
        );
---- End FA stage
---- Begin FA stage
FA_9:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(9), DATA_B => INT_SUM(10),
 DATA_C => INT_CARRY(4),
                SAVE => INT_SUM(11), CARRY => INT_CARRY(8)
        );
---- End FA stage
---- Begin HA stage
HA_6:HALF_ADDER -- At Level 3
        port map
        (
                DATA_A => INT_SUM(11), DATA_B => INT_CARRY(5),
                SAVE => SUM(10), CARRY => CARRY(10)
        );
---- End HA stage
-- End WT-branch 11
```

```
-- Begin WT-branch 12
---- Begin FA stage
FA_10:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(31), DATA_B => SUMMAND(32),
 DATA_C => SUMMAND(33),
                SAVE => INT_SUM(12), CARRY => INT_CARRY(9)
        );
---- End FA stage
---- Begin HA stage
HA_7:HALF_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(34), DATA_B => SUMMAND(35),
                SAVE => INT_SUM(13), CARRY => INT_CARRY(10)
        );
---- End HA stage
---- Begin FA stage
FA_11:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(12), DATA_B => INT_SUM(13),
 DATA_C => INT_CARRY(6),
                SAVE => INT_SUM(14), CARRY => INT_CARRY(11)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(15) <= INT_CARRY(7); -- At Level 2
---- End NO stage
---- Begin FA stage
FA_12:FULL_ADDER -- At Level 3
        port map
        (
                DATA_A => INT_SUM(14), DATA_B => INT_SUM(15),
 DATA_C => INT_CARRY(8),
                SAVE => SUM(11), CARRY => CARRY(11)
        );
---- End FA stage
-- End WT-branch 12
```

```
-- Begin WT-branch 13
---- Begin FA stage
FA_13:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(36), DATA_B => SUMMAND(37),
 DATA_C => SUMMAND(38),
                SAVE => INT_SUM(16), CARRY => INT_CARRY(12)
        );
---- End FA stage
---- Begin FA stage
FA_14:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(39), DATA_B => SUMMAND(40),
 DATA_C => SUMMAND(41),
                SAVE => INT_SUM(17), CARRY => INT_CARRY(13)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(18) <= SUMMAND(42); -- At Level 1
---- End NO stage
---- Begin FA stage
FA_15:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(16), DATA_B => INT_SUM(17),
 DATA_C => INT_SUM(18),
                SAVE => INT_SUM(19), CARRY => INT_CARRY(14)
        );
---- End FA stage
---- Begin HA stage
HA_8:HALF_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_CARRY(9), DATA_B => INT_CARRY(10),
                SAVE => INT_SUM(20), CARRY => INT_CARRY(15)
        );
---- End HA stage
---- Begin FA stage
FA_16:FULL_ADDER -- At Level 3
```

```
        port map
        (
                DATA_A => INT_SUM(19), DATA_B => INT_SUM(20),
 DATA_C => INT_CARRY(11),
                SAVE => SUM(12), CARRY => CARRY(12)
        );
---- End FA stage
-- End WT-branch 13

-- Begin WT-branch 14
---- Begin FA stage
FA_17:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(43), DATA_B => SUMMAND(44),
 DATA_C => SUMMAND(45),
                SAVE => INT_SUM(21), CARRY => INT_CARRY(16)
        );
---- End FA stage
---- Begin FA stage
FA_18:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(46), DATA_B => SUMMAND(47),
 DATA_C => SUMMAND(48),
                SAVE => INT_SUM(22), CARRY => INT_CARRY(17)
        );
---- End FA stage
---- Begin FA stage
FA_19:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(21), DATA_B => INT_SUM(22),
 DATA_C => INT_CARRY(12),
                SAVE => INT_SUM(23), CARRY => INT_CARRY(18)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(24) <= INT_CARRY(13); -- At Level 2
---- End NO stage
---- Begin FA stage
```

```
FA_20:FULL_ADDER -- At Level 3
        port map
        (
                DATA_A => INT_SUM(23), DATA_B => INT_SUM(24),
 DATA_C => INT_CARRY(14),
                SAVE => INT_SUM(25), CARRY => INT_CARRY(19)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(26) <= INT_CARRY(15); -- At Level 3
---- End NO stage
---- Begin HA stage
HA_9:HALF_ADDER -- At Level 4
        port map
        (
                DATA_A => INT_SUM(25), DATA_B => INT_SUM(26),
                SAVE => SUM(13), CARRY => CARRY(13)
        );
---- End HA stage
-- End WT-branch 14

-- Begin WT-branch 15
---- Begin FA stage
FA_21:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(49), DATA_B => SUMMAND(50),
 DATA_C => SUMMAND(51),
                SAVE => INT_SUM(27), CARRY => INT_CARRY(20)
        );
---- End FA stage
---- Begin FA stage
FA_22:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(52), DATA_B => SUMMAND(53),
 DATA_C => SUMMAND(54),
                SAVE => INT_SUM(28), CARRY => INT_CARRY(21)
        );
---- End FA stage
---- Begin NO stage
```

```
INT_SUM(29) <= SUMMAND(55); -- At Level 1
---- End NO stage
---- Begin NO stage
INT_SUM(30) <= SUMMAND(56); -- At Level 1
---- End NO stage
---- Begin FA stage
FA_23:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(27), DATA_B => INT_SUM(28),
 DATA_C => INT_SUM(29),
                SAVE => INT_SUM(31), CARRY => INT_CARRY(22)
        );
---- End FA stage
---- Begin FA stage
FA_24:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(30), DATA_B => INT_CARRY(16),
 DATA_C => INT_CARRY(17),
                SAVE => INT_SUM(32), CARRY => INT_CARRY(23)
        );
---- End FA stage
---- Begin FA stage
FA_25:FULL_ADDER -- At Level 3
        port map
        (
                DATA_A => INT_SUM(31), DATA_B => INT_SUM(32),
 DATA_C => INT_CARRY(18),
                SAVE => INT_SUM(33), CARRY => INT_CARRY(24)
        );
---- End FA stage
---- Begin HA stage
HA_10:HALF_ADDER -- At Level 4
        port map
        (
                DATA_A => INT_SUM(33), DATA_B => INT_CARRY(19),
                SAVE => SUM(14), CARRY => CARRY(14)
        );
---- End HA stage
-- End WT-branch 15
```

```
-- Begin WT-branch 16
---- Begin FA stage
FA_26:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(57), DATA_B => SUMMAND(58),
 DATA_C => SUMMAND(59),
                SAVE => INT_SUM(34), CARRY => INT_CARRY(25)
        );
---- End FA stage
---- Begin FA stage
FA_27:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(60), DATA_B => SUMMAND(61),
 DATA_C => SUMMAND(62),
                SAVE => INT_SUM(35), CARRY => INT_CARRY(26)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(36) <= SUMMAND(63); -- At Level 1
---- End NO stage
---- Begin FA stage
FA_28:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(34), DATA_B => INT_SUM(35),
 DATA_C => INT_SUM(36),
                SAVE => INT_SUM(37), CARRY => INT_CARRY(27)
        );
---- End FA stage
---- Begin HA stage
HA_11:HALF_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_CARRY(20), DATA_B => INT_CARRY(21),
                SAVE => INT_SUM(38), CARRY => INT_CARRY(28)
        );
---- End HA stage
---- Begin FA stage
```

```
FA_29:FULL_ADDER -- At Level 3
        port map
        (
                DATA_A => INT_SUM(37), DATA_B => INT_SUM(38),
 DATA_C => INT_CARRY(22),
                SAVE => INT_SUM(39), CARRY => INT_CARRY(29)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(40) <= INT_CARRY(23); -- At Level 3
---- End NO stage
---- Begin FA stage
FA_30:FULL_ADDER -- At Level 4
        port map
        (
                DATA_A => INT_SUM(39), DATA_B => INT_SUM(40),
 DATA_C => INT_CARRY(24),
                SAVE => SUM(15), CARRY => CARRY(15)
        );
---- End FA stage
-- End WT-branch 16

-- Begin WT-branch 17
---- Begin FA stage
FA_31:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(64), DATA_B => SUMMAND(65),
 DATA_C => SUMMAND(66),
                SAVE => INT_SUM(41), CARRY => INT_CARRY(30)
        );
---- End FA stage
---- Begin FA stage
FA_32:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(67), DATA_B => SUMMAND(68),
 DATA_C => SUMMAND(69),
                SAVE => INT_SUM(42), CARRY => INT_CARRY(31)
        );
---- End FA stage
```

```
---- Begin FA stage
FA_33:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(70), DATA_B => SUMMAND(71),
 DATA_C => SUMMAND(72),
                SAVE => INT_SUM(43), CARRY => INT_CARRY(32)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(44) <= SUMMAND(73); -- At Level 1
---- End NO stage
---- Begin FA stage
FA_34:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(41), DATA_B => INT_SUM(42),
 DATA_C => INT_SUM(43),
                SAVE => INT_SUM(45), CARRY => INT_CARRY(33)
        );
---- End FA stage
---- Begin FA stage
FA_35:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(44), DATA_B => INT_CARRY(25),
 DATA_C => INT_CARRY(26),
                SAVE => INT_SUM(46), CARRY => INT_CARRY(34)
        );
---- End FA stage
---- Begin FA stage
FA_36:FULL_ADDER -- At Level 3
        port map
        (
                DATA_A => INT_SUM(45), DATA_B => INT_SUM(46),
 DATA_C => INT_CARRY(27),
                SAVE => INT_SUM(47), CARRY => INT_CARRY(35)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(48) <= INT_CARRY(28); -- At Level 3
```

```
---- End NO stage
---- Begin FA stage
FA_37:FULL_ADDER -- At Level 4
        port map
        (
                DATA_A => INT_SUM(47), DATA_B => INT_SUM(48), DATA_C =>
                SAVE => SUM(16), CARRY => CARRY(16)
        );
---- End FA stage
-- End WT-branch 17

-- Begin WT-branch 18
---- Begin FA stage
FA_38:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(74), DATA_B => SUMMAND(75),
 DATA_C => SUMMAND(76),
                SAVE => INT_SUM(49), CARRY => INT_CARRY(36)
        );
---- End FA stage
---- Begin FA stage
FA_39:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(77), DATA_B => SUMMAND(78),
 DATA_C => SUMMAND(79),
                SAVE => INT_SUM(50), CARRY => INT_CARRY(37)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(51) <= SUMMAND(80); -- At Level 1
---- End NO stage
---- Begin FA stage
FA_40:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(49), DATA_B => INT_SUM(50), DATA_C =>
                SAVE => INT_SUM(52), CARRY => INT_CARRY(38)
        );
---- End FA stage
```

```
---- Begin FA stage
FA_41:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_CARRY(30), DATA_B => INT_CARRY(31),
 DATA_C => INT_CARRY(32),
                SAVE => INT_SUM(53), CARRY => INT_CARRY(39)
        );
---- End FA stage
---- Begin FA stage
FA_42:FULL_ADDER -- At Level 3
        port map
        (
                DATA_A => INT_SUM(52), DATA_B => INT_SUM(53),
 DATA_C => INT_CARRY(33),
                SAVE => INT_SUM(54), CARRY => INT_CARRY(40)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(55) <= INT_CARRY(34); -- At Level 3
---- End NO stage
---- Begin FA stage
FA_43:FULL_ADDER -- At Level 4
        port map
        (
                DATA_A => INT_SUM(54), DATA_B => INT_SUM(55),
 DATA_C => INT_CARRY(35),
                SAVE => SUM(17), CARRY => CARRY(17)
        );
---- End FA stage
-- End WT-branch 18

-- Begin WT-branch 19
---- Begin FA stage
FA_44:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(81), DATA_B => SUMMAND(82),
 DATA_C => SUMMAND(83),
                SAVE => INT_SUM(56), CARRY => INT_CARRY(41)
        );
```

```
---- End FA stage
---- Begin FA stage
FA_45:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(84), DATA_B => SUMMAND(85),
 DATA_C => SUMMAND(86),
                SAVE => INT_SUM(57), CARRY => INT_CARRY(42)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(58) <= SUMMAND(87); -- At Level 1
---- End NO stage
---- Begin NO stage
INT_SUM(59) <= SUMMAND(88); -- At Level 1
---- End NO stage
---- Begin FA stage
FA_46:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(56), DATA_B => INT_SUM(57),
 DATA_C => INT_SUM(58),
                SAVE => INT_SUM(60), CARRY => INT_CARRY(43)
        );
---- End FA stage
---- Begin FA stage
FA_47:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(59), DATA_B => INT_CARRY(36),
 DATA_C => INT_CARRY(37),
                SAVE => INT_SUM(61), CARRY => INT_CARRY(44)
        );
---- End FA stage
---- Begin FA stage
FA_48:FULL_ADDER -- At Level 3
        port map
        (
                DATA_A => INT_SUM(60), DATA_B => INT_SUM(61),
 DATA_C => INT_CARRY(38),
                SAVE => INT_SUM(62), CARRY => INT_CARRY(45)
```

```
        );
---- End FA stage
---- Begin NO stage
INT_SUM(63) <= INT_CARRY(39); -- At Level 3
---- End NO stage
---- Begin FA stage
FA_49:FULL_ADDER -- At Level 4
        port map
        (
                DATA_A => INT_SUM(62), DATA_B => INT_SUM(63),
 DATA_C => INT_CARRY(40),
                SAVE => SUM(18), CARRY => CARRY(18)
        );
---- End FA stage
-- End WT-branch 19

-- Begin WT-branch 20
---- Begin FA stage
FA_50:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(89), DATA_B => SUMMAND(90),
 DATA_C => SUMMAND(91),
                SAVE => INT_SUM(64), CARRY => INT_CARRY(46)
        );
---- End FA stage
---- Begin FA stage
FA_51:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(92), DATA_B => SUMMAND(93),
 DATA_C => SUMMAND(94),
                SAVE => INT_SUM(65), CARRY => INT_CARRY(47)
        );
---- End FA stage
---- Begin FA stage
FA_52:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(64), DATA_B => INT_SUM(65),
 DATA_C => INT_CARRY(41),
```

```
                SAVE => INT_SUM(66), CARRY => INT_CARRY(48)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(67) <= INT_CARRY(42); -- At Level 2
---- End NO stage
---- Begin FA stage
FA_53:FULL_ADDER -- At Level 3
        port map
        (
                DATA_A => INT_SUM(66), DATA_B => INT_SUM(67),
 DATA_C => INT_CARRY(43),
                SAVE => INT_SUM(68), CARRY => INT_CARRY(49)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(69) <= INT_CARRY(44); -- At Level 3
---- End NO stage
---- Begin FA stage
FA_54:FULL_ADDER -- At Level 4
        port map
        (
                DATA_A => INT_SUM(68), DATA_B => INT_SUM(69),
 DATA_C => INT_CARRY(45),
                SAVE => SUM(19), CARRY => CARRY(19)
        );
---- End FA stage
-- End WT-branch 20

-- Begin WT-branch 21
---- Begin FA stage
FA_55:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(95), DATA_B => SUMMAND(96),
 DATA_C => SUMMAND(97),
                SAVE => INT_SUM(70), CARRY => INT_CARRY(50)
        );
---- End FA stage
---- Begin FA stage
FA_56:FULL_ADDER -- At Level 1
```

```
        port map
        (
                DATA_A => SUMMAND(98), DATA_B => SUMMAND(99),
 DATA_C => SUMMAND(100),
                SAVE => INT_SUM(71), CARRY => INT_CARRY(51)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(72) <= SUMMAND(101); -- At Level 1
---- End NO stage
---- Begin FA stage
FA_57:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(70), DATA_B => INT_SUM(71),
 DATA_C => INT_SUM(72),
                SAVE => INT_SUM(73), CARRY => INT_CARRY(52)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(74) <= INT_CARRY(46); -- At Level 2
---- End NO stage
---- Begin NO stage
INT_SUM(75) <= INT_CARRY(47); -- At Level 2
---- End NO stage
---- Begin FA stage
FA_58:FULL_ADDER -- At Level 3
        port map
        (
                DATA_A => INT_SUM(73), DATA_B => INT_SUM(74),
 DATA_C => INT_SUM(75),
                SAVE => INT_SUM(76), CARRY => INT_CARRY(53)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(77) <= INT_CARRY(48); -- At Level 3
---- End NO stage
---- Begin FA stage
FA_59:FULL_ADDER -- At Level 4
        port map
        (
```

```
                DATA_A => INT_SUM(76), DATA_B => INT_SUM(77),
 DATA_C => INT_CARRY(49),
                SAVE => SUM(20), CARRY => CARRY(20)
        );
---- End FA stage
-- End WT-branch 21

-- Begin WT-branch 22
---- Begin FA stage
FA_60:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(102), DATA_B => SUMMAND(103),
 DATA_C => SUMMAND(104),
                SAVE => INT_SUM(78), CARRY => INT_CARRY(54)
        );
---- End FA stage
---- Begin HA stage
HA_12:HALF_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(105), DATA_B => SUMMAND(106),
                SAVE => INT_SUM(79), CARRY => INT_CARRY(55)
        );
---- End HA stage
---- Begin FA stage
FA_61:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(78), DATA_B => INT_SUM(79),
 DATA_C => INT_CARRY(50),
                SAVE => INT_SUM(80), CARRY => INT_CARRY(56)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(81) <= INT_CARRY(51); -- At Level 2
---- End NO stage
---- Begin FA stage
FA_62:FULL_ADDER -- At Level 3
        port map
        (
```

```
                DATA_A => INT_SUM(80), DATA_B => INT_SUM(81),
 DATA_C => INT_CARRY(52),
                SAVE => INT_SUM(82), CARRY => INT_CARRY(57)
        );
---- End FA stage
---- Begin HA stage
HA_13:HALF_ADDER -- At Level 4
        port map
        (
                DATA_A => INT_SUM(82), DATA_B => INT_CARRY(53),
                SAVE => SUM(21), CARRY => CARRY(21)
        );
---- End HA stage
-- End WT-branch 22

-- Begin WT-branch 23
---- Begin FA stage
FA_63:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(107), DATA_B => SUMMAND(108),
 DATA_C => SUMMAND(109),
                SAVE => INT_SUM(83), CARRY => INT_CARRY(58)
        );
---- End FA stage
---- Begin FA stage
FA_64:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(110), DATA_B => SUMMAND(111),
 DATA_C => SUMMAND(112),
                SAVE => INT_SUM(84), CARRY => INT_CARRY(59)
        );
---- End FA stage
---- Begin FA stage
FA_65:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(83), DATA_B => INT_SUM(84),
 DATA_C => INT_CARRY(54),
                SAVE => INT_SUM(85), CARRY => INT_CARRY(60)
```

```
        );
---- End FA stage
---- Begin NO stage
INT_SUM(86) <= INT_CARRY(55); -- At Level 2
---- End NO stage
---- Begin FA stage
FA_66:FULL_ADDER -- At Level 3
        port map
        (
                DATA_A => INT_SUM(85), DATA_B => INT_SUM(86),
 DATA_C => INT_CARRY(56),
                SAVE => INT_SUM(87), CARRY => INT_CARRY(61)
        );
---- End FA stage
---- Begin HA stage
HA_14:HALF_ADDER -- At Level 4
        port map
        (
                DATA_A => INT_SUM(87), DATA_B => INT_CARRY(57),
                SAVE => SUM(22), CARRY => CARRY(22)
        );
---- End HA stage
-- End WT-branch 23

-- Begin WT-branch 24
---- Begin FA stage
FA_67:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(113), DATA_B => SUMMAND(114),
 DATA_C => SUMMAND(115),
                SAVE => INT_SUM(88), CARRY => INT_CARRY(62)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(89) <= SUMMAND(116); -- At Level 1
---- End NO stage
---- Begin FA stage
FA_68:FULL_ADDER -- At Level 2
        port map
        (
```

```
                DATA_A => INT_SUM(88), DATA_B => INT_SUM(89),
 DATA_C => INT_CARRY(58),
                SAVE => INT_SUM(90), CARRY => INT_CARRY(63)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(91) <= INT_CARRY(59); -- At Level 2
---- End NO stage
---- Begin FA stage
FA_69:FULL_ADDER -- At Level 3
        port map
        (
                DATA_A => INT_SUM(90), DATA_B => INT_SUM(91),
 DATA_C => INT_CARRY(60),
                SAVE => INT_SUM(92), CARRY => INT_CARRY(64)
        );
---- End FA stage
---- Begin HA stage
HA_15:HALF_ADDER -- At Level 4
        port map
        (
                DATA_A => INT_SUM(92), DATA_B => INT_CARRY(61),
                SAVE => SUM(23), CARRY => CARRY(23)
        );
---- End HA stage
-- End WT-branch 24

-- Begin WT-branch 25
---- Begin FA stage
FA_70:FULL_ADDER -- At Level 1
        port map
        (
                DATA_A => SUMMAND(117), DATA_B => SUMMAND(118),
 DATA_C => SUMMAND(119),
                SAVE => INT_SUM(93), CARRY => INT_CARRY(65)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(94) <= SUMMAND(120); -- At Level 1
---- End NO stage
---- Begin NO stage
```

```
INT_SUM(95) <= SUMMAND(121); -- At Level 1
---- End NO stage
---- Begin FA stage
FA_71:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => INT_SUM(93), DATA_B => INT_SUM(94),
 DATA_C => INT_SUM(95),
                SAVE => INT_SUM(96), CARRY => INT_CARRY(66)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(97) <= INT_CARRY(62); -- At Level 2
---- End NO stage
---- Begin FA stage
FA_72:FULL_ADDER -- At Level 3
        port map
        (
                DATA_A => INT_SUM(96), DATA_B => INT_SUM(97),
 DATA_C => INT_CARRY(63),
                SAVE => INT_SUM(98), CARRY => INT_CARRY(67)
        );
---- End FA stage
---- Begin HA stage
HA_16:HALF_ADDER -- At Level 4
        port map
        (
                DATA_A => INT_SUM(98), DATA_B => INT_CARRY(64),
                SAVE => SUM(24), CARRY => CARRY(24)
        );
---- End HA stage
-- End WT-branch 25

-- Begin WT-branch 26
---- Begin FA stage
FA_73:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => SUMMAND(122), DATA_B => SUMMAND(123),
 DATA_C => SUMMAND(124),
                SAVE => INT_SUM(99), CARRY => INT_CARRY(68)
```

```
        );
---- End FA stage
---- Begin NO stage
INT_SUM(100) <= INT_CARRY(65); -- At Level 2
---- End NO stage
---- Begin FA stage
FA_74:FULL_ADDER -- At Level 3
        port map
        (
                DATA_A => INT_SUM(99), DATA_B => INT_SUM(100),
 DATA_C => INT_CARRY(66),
                SAVE => INT_SUM(101), CARRY => INT_CARRY(69)
        );
---- End FA stage
---- Begin HA stage
HA_17:HALF_ADDER -- At Level 4
        port map
        (
                DATA_A => INT_SUM(101), DATA_B => INT_CARRY(67),
                SAVE => SUM(25), CARRY => CARRY(25)
        );
---- End HA stage
-- End WT-branch 26

-- Begin WT-branch 27
---- Begin FA stage
FA_75:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => SUMMAND(125), DATA_B => SUMMAND(126),
 DATA_C => SUMMAND(127),
                SAVE => INT_SUM(102), CARRY => INT_CARRY(70)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(103) <= SUMMAND(128); -- At Level 2
---- End NO stage
---- Begin FA stage
FA_76:FULL_ADDER -- At Level 3
        port map
        (
```

```
                DATA_A => INT_SUM(102), DATA_B => INT_SUM(103),
 DATA_C => INT_CARRY(68),
                SAVE => INT_SUM(104), CARRY => INT_CARRY(71)
        );
---- End FA stage
---- Begin HA stage
HA_18:HALF_ADDER -- At Level 4
        port map
        (
                DATA_A => INT_SUM(104), DATA_B => INT_CARRY(69),
                SAVE => SUM(26), CARRY => CARRY(26)
        );
---- End HA stage
-- End WT-branch 27

-- Begin WT-branch 28
---- Begin NO stage
INT_SUM(105) <= SUMMAND(129); -- At Level 2
---- End NO stage
---- Begin NO stage
INT_SUM(106) <= SUMMAND(130); -- At Level 2
---- End NO stage
---- Begin FA stage
FA_77:FULL_ADDER -- At Level 3
        port map
        (
                DATA_A => INT_SUM(105), DATA_B => INT_SUM(106)
, DATA_C => INT_CARRY(70),
                SAVE => INT_SUM(107), CARRY => INT_CARRY(72)
        );
---- End FA stage
---- Begin HA stage
HA_19:HALF_ADDER -- At Level 4
        port map
        (
                DATA_A => INT_SUM(107), DATA_B => INT_CARRY(71)
,                SAVE => SUM(27), CARRY => CARRY(27)
        );
---- End HA stage
-- End WT-branch 28
```

```
-- Begin WT-branch 29
---- Begin FA stage
FA_78:FULL_ADDER -- At Level 2
        port map
        (
                DATA_A => SUMMAND(131), DATA_B => SUMMAND(132)
, DATA_C => SUMMAND(133),
                SAVE => INT_SUM(108), CARRY => INT_CARRY(73)
        );
---- End FA stage
---- Begin NO stage
INT_SUM(109) <= INT_SUM(108); -- At Level 3
---- End NO stage
---- Begin HA stage
HA_20:HALF_ADDER -- At Level 4
        port map
        (
                DATA_A => INT_SUM(109), DATA_B => INT_CARRY(72),
                SAVE => SUM(28), CARRY => CARRY(28)
        );
---- End HA stage
-- End WT-branch 29

-- Begin WT-branch 30
---- Begin NO stage
INT_SUM(110) <= SUMMAND(134); -- At Level 2
---- End NO stage
---- Begin HA stage
HA_21:HALF_ADDER -- At Level 3
        port map
        (
                DATA_A => INT_SUM(110), DATA_B => INT_CARRY(73),
                SAVE => SUM(29), CARRY => CARRY(29)
        );
---- End HA stage
-- End WT-branch 30

-- Begin WT-branch 31
---- Begin HA stage
HA_22:HALF_ADDER -- At Level 2
        port map
```

```
        (
                DATA_A => SUMMAND(135), DATA_B => SUMMAND(136),
                SAVE => SUM(30), CARRY => CARRY(30)
        );
---- End HA stage
-- End WT-branch 31

end WALLACE;
```

DBLC adder

```
library ieee;
use ieee.std_logic_1164.all;


entity INVBLOCK is
port
(
        GIN,PHI:in std_logic;
        GOUT:out std_logic
);
end INVBLOCK;

library ieee;
use ieee.std_logic_1164.all;
entity XXOR1 is
port
(
        A,B,GIN,PHI:in std_logic;
        SUM:out std_logic
);
end XXOR1;

library ieee;
use ieee.std_logic_1164.all;
entity XXOR2 is
port
(
        A,B,GIN,PHI:in std_logic;
        SUM:out std_logic
);
end XXOR2;

library ieee;
use ieee.std_logic_1164.all;
entity BLOCK0 is
port
(
        A,B,PHI:in std_logic;
        POUT,GOUT:out std_logic
```

```
);
end BLOCK0;

library ieee;
use ieee.std_logic_1164.all;
entity BLOCK1 is
port
(
        PIN1,PIN2,GIN1,GIN2,PHI:in std_logic;
        POUT,GOUT:out std_logic
);
end BLOCK1;

library ieee;
use ieee.std_logic_1164.all;

entity BLOCK2 is
port
(
        PIN1,PIN2,GIN1,GIN2,PHI:in std_logic;
        POUT,GOUT:out std_logic
);
end BLOCK2;

library ieee;
use ieee.std_logic_1164.all;
entity BLOCK1A is
port
(
        PIN2,GIN1,GIN2,PHI:in std_logic;
        GOUT:out std_logic
);
end BLOCK1A;

library ieee;
use ieee.std_logic_1164.all;

entity BLOCK2A is
port
(
        PIN2,GIN1,GIN2,PHI:in std_logic;
```

```
        GOUT:out std_logic
);
end BLOCK2A;

library ieee;
use ieee.std_logic_1164.all;
entity PRESTAGE_32 is
port
(
        A: in std_logic_vector(0 to 31);
        B: in std_logic_vector(0 to 31);
        CIN: in std_logic;
        PHI: in std_logic;
        POUT: out std_logic_vector(0 to 31);
        GOUT: out std_logic_vector(0 to 32)
);
end PRESTAGE_32;

library ieee;
use ieee.std_logic_1164.all;
entity DBLC_0_32 is
port
(
        PIN: in std_logic_vector(0 to 31);
        GIN: in std_logic_vector(0 to 32);
        PHI: in std_logic;
        POUT: out std_logic_vector(0 to 30);
        GOUT: out std_logic_vector(0 to 32)
);
end DBLC_0_32;

library ieee;
use ieee.std_logic_1164.all;
entity DBLC_1_32 is
port
(
        PIN: in std_logic_vector(0 to 30);
        GIN: in std_logic_vector(0 to 32);
        PHI: in std_logic;
        POUT: out std_logic_vector(0 to 28);
        GOUT: out std_logic_vector(0 to 32)
```

```
);
end DBLC_1_32;

library ieee;
use ieee.std_logic_1164.all;
entity DBLC_2_32 is
port
(
        PIN: in std_logic_vector(0 to 28);
        GIN: in std_logic_vector(0 to 32);
        PHI: in std_logic;
        POUT: out std_logic_vector(0 to 24);
        GOUT: out std_logic_vector(0 to 32)
);
end DBLC_2_32;

library ieee;
use ieee.std_logic_1164.all;
entity DBLC_3_32 is
port
(
        PIN: in std_logic_vector(0 to 24);
        GIN: in std_logic_vector(0 to 32);
        PHI: in std_logic;
        POUT: out std_logic_vector(0 to 16);
        GOUT: out std_logic_vector(0 to 32)
);
end DBLC_3_32;

library ieee;
use ieee.std_logic_1164.all;
entity DBLC_4_32 is
port
(
        PIN: in std_logic_vector(0 to 16);
        GIN: in std_logic_vector(0 to 32);
        PHI: in std_logic;
        POUT: out std_logic_vector(0 to 0);
        GOUT: out std_logic_vector(0 to 32)
);
end DBLC_4_32;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity XORSTAGE_32 is
port
(
        A: in std_logic_vector(0 to 31);
        B: in std_logic_vector(0 to 31);
        PBIT, PHI: in std_logic;
        CARRY: in std_logic_vector(0 to 32);
        SUM: out std_logic_vector(0 to 31);
        COUT: out std_logic
);
end XORSTAGE_32;

library ieee;
use ieee.std_logic_1164.all;
entity DBLCTREE_32 is
port
(
        PIN:in std_logic_vector(0 to 31);
        GIN:in std_logic_vector(0 to 32);
        PHI:in std_logic;
        GOUT:out std_logic_vector(0 to 32);
        POUT:out std_logic_vector(0 to 0)
);
end DBLCTREE_32;

library ieee;
use ieee.std_logic_1164.all;
entity DBLCADDER_32_32 is
port
(
        OPA:in std_logic_vector(0 to 31);
        OPB:in std_logic_vector(0 to 31);
        CIN:in std_logic;
        PHI:in std_logic;
        SUM:out std_logic_vector(0 to 31)
);
end DBLCADDER_32_32;
```

```
-- Architectures for the DBLC-tree




architecture INVBLOCK_regular of INVBLOCK is
begin
        GOUT <= not GIN;
end INVBLOCK_regular;

library ieee;
use ieee.std_logic_1164.all;
architecture BLOCK1_regular of BLOCK1 is
begin
        POUT <= not(PIN1 or PIN2);
        GOUT <= not(GIN2 and (PIN2 or GIN1));
end BLOCK1_regular;

library ieee;
use ieee.std_logic_1164.all;
architecture BLOCK2_regular of BLOCK2 is
begin
        POUT <= not(PIN1 and PIN2);
        GOUT <= not(GIN2 or (PIN2 and GIN1));
end BLOCK2_regular;

library ieee;
use ieee.std_logic_1164.all;
architecture BLOCK1A_regular of BLOCK1A is
begin
        GOUT <= not(GIN2 and (PIN2 or GIN1));
end BLOCK1A_regular;

library ieee;
use ieee.std_logic_1164.all;
architecture BLOCK2A_regular of BLOCK2A is
begin
```

```
        GOUT <= not(GIN2 or (PIN2 and GIN1));
end BLOCK2A_regular;

library ieee;
use ieee.std_logic_1164.all;
architecture XXOR_regular of XXOR1 is
begin
        SUM <= (not (A xor B)) xor GIN;
end XXOR_regular;

library ieee;
use ieee.std_logic_1164.all;
architecture XXOR_true of XXOR2 is
begin
        SUM <= (A xor B) xor GIN;
end XXOR_true;

library ieee;
use ieee.std_logic_1164.all;
architecture BLOCK0_regular of BLOCK0 is
begin
        POUT <= not(A or B);
        GOUT <= not(A and B);
end BLOCK0_regular;

library ieee;
use ieee.std_logic_1164.all;
architecture PRESTAGE of PRESTAGE_32 is
component BLOCK0
port
(
        A,B,PHI: in std_logic;
        POUT,GOUT: out std_logic
);
end component;
component INVBLOCK
port
(
        GIN,PHI:in std_logic;
        GOUT:out std_logic
);
```

```
end component;

begin -- PRESTAGE
U1:for I in 0 to 31 generate
        U11: BLOCK0 port map(A(I),B(I),PHI,POUT(I),GOUT(I+1));
end generate U1;
U2: INVBLOCK port map(CIN,PHI,GOUT(0));
end PRESTAGE;
-- The DBLC-tree: Level 0

library ieee;
use ieee.std_logic_1164.all;
architecture DBLC_0 of DBLC_0_32 is
component INVBLOCK
port
(
        GIN,PHI:in std_logic;
        GOUT:out std_logic
);
end component;

component BLOCK1
port
(
        PIN1,PIN2,GIN1,GIN2,PHI:in std_logic;
        POUT,GOUT:out std_logic
);
end component;

component BLOCK1A
port
(
        PIN2,GIN1,GIN2,PHI:in std_logic;
        GOUT:out std_logic
);
end component;

begin -- Architecture DBLC_0
U1: for I in 0 to 0 generate
        U11: INVBLOCK port map(GIN(I),PHI,GOUT(I));
end generate U1;
```

```
U2: for I in 1 to 1 generate
        U21: BLOCK1A port map(PIN(I-1),GIN(I-1),GIN(I),PHI,
GOUT(I));
end generate U2;
U3: for I in 2 to 32 generate
        U31: BLOCK1 port map(PIN(I-2),PIN(I-1),GIN(I-1),
GIN(I),PHI,POUT(I-2),GOUT(I));
end generate U3;
end DBLC_0;

-- The DBLC-tree: Level 1

library ieee;
use ieee.std_logic_1164.all;
architecture DBLC_1 of DBLC_1_32 is
component INVBLOCK
port
(
        GIN,PHI:in std_logic;
        GOUT:out std_logic
);
end component;

component BLOCK2
port
(
        PIN1,PIN2,GIN1,GIN2,PHI:in std_logic;
        POUT,GOUT:out std_logic
);
end component;

component BLOCK2A
port
(
        PIN2,GIN1,GIN2,PHI:in std_logic;
        GOUT:out std_logic
);
end component;

begin -- Architecture DBLC_1
U1: for I in 0 to 1 generate
```

```
        U11: INVBLOCK port map(GIN(I),PHI,GOUT(I));
end generate U1;
U2: for I in 2 to 3 generate
        U21: BLOCK2A port map(PIN(I-2),GIN(I-2),GIN(I),PHI,
GOUT(I));
end generate U2;
U3: for I in 4 to 32 generate
        U31: BLOCK2 port map(PIN(I-4),PIN(I-2),GIN(I-2),
GIN(I),PHI,POUT(I-4),GOUT(I));
end generate U3;
end DBLC_1;

-- The DBLC-tree: Level 2

library ieee;
use ieee.std_logic_1164.all;
architecture DBLC_2 of DBLC_2_32 is
component INVBLOCK
port
(
        GIN,PHI:in std_logic;
        GOUT:out std_logic
);
end component;

component BLOCK1
port
(
        PIN1,PIN2,GIN1,GIN2,PHI:in std_logic;
        POUT,GOUT:out std_logic
);
end component;

component BLOCK1A
port
(
        PIN2,GIN1,GIN2,PHI:in std_logic;
        GOUT:out std_logic
);
end component;
```

```
begin -- Architecture DBLC_2
U1: for I in 0 to 3 generate
        U11: INVBLOCK port map(GIN(I),PHI,GOUT(I));
end generate U1;
U2: for I in 4 to 7 generate
        U21: BLOCK1A port map(PIN(I-4),GIN(I-4),GIN(I),PHI,
GOUT(I));
end generate U2;
U3: for I in 8 to 32 generate
        U31: BLOCK1 port map(PIN(I-8),PIN(I-4),GIN(I-4),
GIN(I),PHI,POUT(I-8),GOUT(I));
end generate U3;
end DBLC_2;

-- The DBLC-tree: Level 3

library ieee;
use ieee.std_logic_1164.all;
architecture DBLC_3 of DBLC_3_32 is
component INVBLOCK
port
(
        GIN,PHI:in std_logic;
        GOUT:out std_logic
);
end component;

component BLOCK2
port
(
        PIN1,PIN2,GIN1,GIN2,PHI:in std_logic;
        POUT,GOUT:out std_logic
);
end component;

component BLOCK2A
port
(
        PIN2,GIN1,GIN2,PHI:in std_logic;
        GOUT:out std_logic
);
```

```
end component;

begin -- Architecture DBLC_3
U1: for I in 0 to 7 generate
        U11: INVBLOCK port map(GIN(I),PHI,GOUT(I));
end generate U1;
U2: for I in 8 to 15 generate
        U21: BLOCK2A port map(PIN(I-8),GIN(I-8),GIN(I),PHI,
GOUT(I));
end generate U2;
U3: for I in 16 to 32 generate
        U31: BLOCK2 port map(PIN(I-16),PIN(I-8),GIN(I-8),
GIN(I),PHI,POUT(I-16),GOUT(I));
end generate U3;
end DBLC_3;

-- The DBLC-tree: Level 4

library ieee;
use ieee.std_logic_1164.all;
architecture DBLC_4 of DBLC_4_32 is
component BLOCK1
port
(
        PIN1,PIN2,GIN1,GIN2,PHI:in std_logic;
        POUT,GOUT:out std_logic
);
end component;

component BLOCK1A
port
(
        PIN2,GIN1,GIN2,PHI:in std_logic;
        GOUT:out std_logic
);
end component;

begin -- Architecture DBLC_4
        GOUT(0 to 15) <= GIN(0 to 15);
U2: for I in 16 to 31 generate
        U21: BLOCK1A port map(PIN(I-16),GIN(I-16),GIN(I),PHI,
```

```
GOUT(I));
end generate U2;
U3: for I in 32 to 32 generate
        U31: BLOCK1 port map(PIN(I-32),PIN(I-16),GIN(I-16),
GIN(I),PHI,POUT(I-32),GOUT(I));
end generate U3;
end DBLC_4;

library ieee;
use ieee.std_logic_1164.all;
architecture XORSTAGE of XORSTAGE_32 is
component XXOR1
port
(
        A,B,GIN,PHI:in std_logic;
        SUM:out std_logic
);
end component;
component XXOR2
port
(
        A,B,GIN,PHI:in std_logic;
        SUM:out std_logic
);
end component;
component BLOCK2A
port
(
        PIN2,GIN1,GIN2,PHI:in std_logic;
        GOUT:out std_logic
);
end component;

begin -- XORSTAGE
U2:for I in 0 to 15 generate
        U22: XXOR1 port map(A(I),B(I),CARRY(I),PHI,SUM(I));
end generate U2;
U3:for I in 16 to 31 generate
        U33: XXOR2 port map(A(I),B(I),CARRY(I),PHI,SUM(I));
end generate U3;
U1: BLOCK2A port map(PBIT,CARRY(0),CARRY(32),PHI,COUT);
```

```
end XORSTAGE;

-- The DBLC-tree: All levels encapsulated

library ieee;
use ieee.std_logic_1164.all;
architecture DBLCTREE of DBLCTREE_32 is
component DBLC_0_32
port
(
        PIN: in std_logic_vector(0 to 31);
        GIN: in std_logic_vector(0 to 32);
        PHI: in std_logic;
        POUT: out std_logic_vector(0 to 30);
        GOUT: out std_logic_vector(0 to 32)
);
end component;

component DBLC_1_32
port
(
        PIN: in std_logic_vector(0 to 30);
        GIN: in std_logic_vector(0 to 32);
        PHI: in std_logic;
        POUT: out std_logic_vector(0 to 28);
        GOUT: out std_logic_vector(0 to 32)
);
end component;

component DBLC_2_32
port
(
        PIN: in std_logic_vector(0 to 28);
        GIN: in std_logic_vector(0 to 32);
        PHI: in std_logic;
        POUT: out std_logic_vector(0 to 24);
        GOUT: out std_logic_vector(0 to 32)
);
end component;

component DBLC_3_32
```

```
port
(
        PIN: in std_logic_vector(0 to 24);
        GIN: in std_logic_vector(0 to 32);
        PHI: in std_logic;
        POUT: out std_logic_vector(0 to 16);
        GOUT: out std_logic_vector(0 to 32)
);
end component;

component DBLC_4_32
port
(
        PIN: in std_logic_vector(0 to 16);
        GIN: in std_logic_vector(0 to 32);
        PHI: in std_logic;
        POUT: out std_logic_vector(0 to 0);
        GOUT: out std_logic_vector(0 to 32)
);
end component;

signal INTPROP_0: std_logic_vector(0 to 30);
signal INTGEN_0: std_logic_vector(0 to 32);
signal INTPROP_1: std_logic_vector(0 to 28);
signal INTGEN_1: std_logic_vector(0 to 32);
signal INTPROP_2: std_logic_vector(0 to 24);
signal INTGEN_2: std_logic_vector(0 to 32);
signal INTPROP_3: std_logic_vector(0 to 16);
signal INTGEN_3: std_logic_vector(0 to 32);
begin -- Architecture DBLCTREE
U_0: DBLC_0_32 port map(PIN=>PIN,GIN=>GIN,PHI=>PHI,
POUT=>INTPROP_0,GOUT=>INTGEN_0);
U_1: DBLC_1_32 port
map(PIN=>INTPROP_0,GIN=>INTGEN_0,PHI=>PHI,POUT=>INTPROP_1,
GOUT=>INTGEN_1);
U_2: DBLC_2_32 port
map(PIN=>INTPROP_1,GIN=>INTGEN_1,PHI=>PHI,POUT=>INTPROP_2,
GOUT=>INTGEN_2);
U_3: DBLC_3_32 port
map(PIN=>INTPROP_2,GIN=>INTGEN_2,PHI=>PHI,POUT=>INTPROP_3,
GOUT=>INTGEN_3);
```

```
U_4: DBLC_4_32 port map(PIN=>INTPROP_3,GIN=>INTGEN_3,
PHI=>PHI,POUT=>POUT,GOUT=>GOUT);
end DBLCTREE;

library ieee;
use ieee.std_logic_1164.all;
architecture DBLCADDER of DBLCADDER_32_32 is
component PRESTAGE_32
port
(
        A: in std_logic_vector(0 to 31);
        B: in std_logic_vector(0 to 31);
        CIN: in std_logic;
        PHI: in std_logic;
        POUT: out std_logic_vector(0 to 31);
        GOUT: out std_logic_vector(0 to 32)
);
end component;

component DBLCTREE_32
port
(
        PIN:in std_logic_vector(0 to 31);
        GIN:in std_logic_vector(0 to 32);
        PHI:in std_logic;
        GOUT:out std_logic_vector(0 to 32);
        POUT:out std_logic_vector(0 to 0)
);
end component;

component XORSTAGE_32
port
(
        A: in std_logic_vector(0 to 31);
        B: in std_logic_vector(0 to 31);
        PBIT: in std_logic;
        PHI: in std_logic;
        CARRY: in std_logic_vector(0 to 32);
        SUM: out std_logic_vector(0 to 31);
        COUT: out std_logic
);
```

```
end component;

signal INTPROP: std_logic_vector(0 to 31);
signal INTGEN: std_logic_vector(0 to 32);
signal PBIT:std_logic_vector(0 to 0);
signal CARRY: std_logic_vector(0 to 32);

begin -- Architecture DBLCADDER

U1: PRESTAGE_32 port map(OPA,OPB,CIN,PHI,INTPROP,INTGEN);
U2: DBLCTREE_32 port map(INTPROP,INTGEN,PHI,CARRY,PBIT);
U3: XORSTAGE_32 port map(OPA(0 to 31),OPB(0 to 31),PBIT(0),
PHI,CARRY(0 to 32),SUM,open);


end DBLCADDER;
```

Testbench

```
Library IEEE;
use IEEE.Std_Logic_1164.all;
use Std.textio.all;
use IEEE.numeric_std.all;


Entity TEST_SQ is

end TEST_SQ;

architecture TESTBENCH of TEST_SQ is


  Component SQ
    port
    (
            X: in std_logic_vector(15 downto 0);
         CLK: in std_logic;
         P: out std_logic_vector(31 downto 0)
    );

  end Component;


  signal A: std_logic_vector(15 downto 0);
  signal Q: std_logic_vector(31 downto 0);
  signal LOGIC_ZERO: Bit;
  signal CLK: std_logic :='0';



 begin


 --Instantierer "Unit Under Test"

 UUT : SQ
 port map
 (
```

```
   X          =>  A,
   P          =>  Q,
   CLK         => CLK


 );

  -- Handle input and output files


read_in : process

    file in_vec: text open read_mode is "M:\New Folder\input.txt";
    file out_vec: text open write_mode is "out.txt";
 variable ILine,OLine : line;
 variable A_in  :  natural;
 variable Q_out :  bit_vector (31 downto 0);

begin
 STIMULI :


  while not endfile(in_vec) loop
  readline(in_vec, ILine);
          read(ILine, A_in);
          A <= std_logic_vector(to_unsigned(A_in,16));


wait for 100 ns;
      Q_out:=to_bitvector(Q);

   write(OLine, Q_out);
      writeline(out_vec,OLine) ;

 end loop;
 file_close(in_vec);
 file_close(out_vec);
 end process;
 end TESTBENCH;
```

Bibliography

[1] Mats Høvin, Trond Sæther, Dag T. Wisland, and Tor Sverre Lande. A narrow band delta-sigma frequency-to-digital converter. *IEEE international symposium on circuits and systems*, 1997.

[2] Mohammed Ali Saber. Lc vco tuning with parallel mos-varactor in fdsm application. Master's thesis, University of Oslo, 2006.

[3] *http://www.wikipedia.com*.

[4] M. Høvin, A. Olsen, T. S. Lande, and C. Toumazou. Delta-sigma modulators using frequency modulated intermediate values. *IEEE JSSC*, 1997.

[5] M.Høvin, A.Olsen, T.S.Lande, and C.Toumasou. Novel second-order delta-sigma modulator/frequeny-to-digital converter. *IEEE electronics letter*, 1995.

[6] J.C.Candy and G.C.Temes. Oversmpling delta-sigma data converters. *IEEE Press,New York*, 1992.

[7] Karianne Øysted. A novel micromechanical digital integrated cmos pressure sensor using delta-sigma noiseshaping. Master's thesis, university of Oslo, 2004.

[8] Ulrik Wismar, Dag Wisland, and Pietro Andreani. Linearity of bulk-controlled inverter ring vco in weak and strong inversion. *Norchip*, 2005.

[9] Yun-Hsueh Chuang, Sheng-Lyang Jang, Jian-Feng Lee, and Shao-Hua Lee. A low voltage 900mhz voltage controlled ring oscillator with wide tuning range. *The 2004 IEEE Asia-Pacific Conference on Circuits and Systems*, 2004.

[10] Jie Long, Jo Yi Foo, and Robert J.WEber. A 2.4 ghz low-power low-phase-noise cmos lc vco. *ISVLSI 04*, 2004.

[11] Bjørn Christian Paulseth. Low-power 5ghz voltage contolled oscillator. Master's thesis, University of Oslo, 2006.

[12] Rudy J.Van De Plassche and Hans J. Schouwenaars. A monolithic 14 bit a/d converter. *IEEE Journal of Solid State Circuits*, 1982.

[13] Umberto gatti, Guiseppe Gazzoli, and Franco Maloberti. Improving the linearity in high-speed-analog-to-digital converters. *IEEE*, 1998.

[14] Jørg Gustran, Falko Fiechtner, and Michael Hoffmann. Vco linearisation by frequency feedback. *Radio Frequency Integrated circuits Symposium*, 1998.

[15] Eduri U. and Maloberti F. Online digital correction of the harmonic distortion in analog-to digital converters. *Electronics, circuits and systems IEEE*, 2001.

[16] *http://members.tripod.com/michaelgellis/mixersin.html*.

[17] Jan arne Lescczynski. Liner voltage controlled oscillator. Master's thesis, University os Oslo, 2006.

[18] *Multiplication in FPGAs.http://www.andraka.com*.

[19] *web.syr.edu/ yzhao07/notes_architectureI/09_AdditionMultiplication.ppt*.

[20] Johnny Pihl and Einar J. Aas. A mutiplier and squarer generator for high performance dsp applications. *39th Midwest symposium on Ciruits and Systems*, 1997.