

Examensarbete

**Direct Digital Frequency Synthesis in
Field-Programmable Gate Arrays**

Examensarbete utfört i Elektroniska System
vid Tekniska högskolan i Linköping
av

Petter Källström

LiTH-ISY-EX--10/4403--SE

Linköping 2010



Linköpings universitet
TEKNISKA HÖGSKOLAN

Direct Digital Frequency Synthesis in Field-Programmable Gate Arrays

Examensarbete utfört i Elektroniska System
vid Tekniska högskolan i Linköping
av


Petter Källström

LiTH-ISY-EX--10/4403--SE

Handledare: **Oscar Gustafsson**
ISY, Linköpings universitet

Examinator: **Oscar Gustafsson**
ISY, Linköpings universitet

Linköping, 19 April, 2010

 Avdelning, Institution Division, Department Electronic Systems Department of Electrical Engineering Linköpings universitet SE-581 83 Linköping, Sweden		Datum Date 2010-04-19
Språk Language <input type="checkbox"/> Svenska/Swedish <input checked="" type="checkbox"/> Engelska/English <input type="checkbox"/> _____	Rapporttyp Report category <input type="checkbox"/> Licentiatavhandling <input checked="" type="checkbox"/> Examensarbete <input type="checkbox"/> C-uppsats <input type="checkbox"/> D-uppsats <input type="checkbox"/> Övrig rapport <input type="checkbox"/> _____	ISBN _____ ISRN LiTH-ISY-EX--10/4403--SE Serietitel och serienummer ISSN Title of series, numbering _____
URL för elektronisk version http://www.es.isy.liu.se http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-56550		
Titel Title Digital Frekvenssyntes för FPGAer Direct Digital Frequency Synthesis in Field-Programmable Gate Arrays Författare Petter Källström Author		
Sammanfattning Abstract This thesis is about creation of a Matlab program that suggests and automatically generates a Phase to Sine Amplitude Converter (PSAC) in the hardware language VHDL, suitable for Direct Digital Frequency Synthesis (DDFS). Main hardware target is Field Programmable Gate Arrays (FPGAs). Focus in this report is how an FPGA works, different methods for sine amplitude generation and their signal qualities vs the hardware resources they use.		
Nyckelord Keywords PSAC, DDFS, FPGA, Matlab, Frequency Synthesis		

Abstract

This thesis is about creation of a Matlab program that suggests and automatically generates a Phase to Sine Amplitude Converter (PSAC) in the hardware language VHDL, suitable for Direct Digital Frequency Synthesis (DDFS). Main hardware target is Field Programmable Gate Arrays (FPGAs).

Focus in this report is how an FPGA works, different methods for sine amplitude generation and their signal qualities vs the hardware resources they use.

Sammanfattning

Detta exjobb handlar om att skapa ett Matlab-program som föreslår och implementerar en sinusgenerator i hårdvaruspråket VHDL, avsedd för digital frekvenssyntes (DDFS). Ämnad hårdvara för implementeringen är en fältprogrammerbar grindmatris (FPGA).

Fokus i denna rapport ligger på hur en FPGA är uppbyggd, olika metoder för sinusgenerering och vilka kvaliteter på sinusvågen de ger och vilka resurser i hårdvaran de använder.

Acknowledgments

I would like to thank my supervisor and examiner Oscar Gustafsson, and Daniel Källming for a good opponent. I would also want to thank Kent Palmkvist and a few others for technical support and advices during the thesis. The entire ES corridor, Mikael Karlsson, Emanuel Eliasson, Kaveh Azizi and the Bertramm group have also been very supportive and have kept me in a good mood - thank you all.

Contents

1	Introduction	1
1.1	Background	1
1.1.1	DDFS	1
1.2	Purpose	2
1.2.1	Quality vs Resource Problem	2
1.3	This Document	2
1.4	Project Approach and Overview	2
1.4.1	Implementation Language	2
1.4.2	Finding Existing Methods	2
1.4.3	Target Architectures	3
1.4.4	Modeling and Analysis	3
1.4.5	VHDL Implementation	3
1.4.6	Suggester	3
1.5	Limitations	3
1.6	Notations and Abbreviations	3
2	Methods	5
2.1	Symmetry Using Range Divider	5
2.2	ROM/LUT	6
2.3	Decomposition	7
2.3.1	Polynomial Interpolation Alternative	7
2.3.2	Phase Truncation Alternative	9
2.3.3	Hutchinson's Approach	9
2.3.4	Sunderland's Approach	10
2.3.5	Nicolas' Approach	11
2.3.6	Curticăpean's Approach	11
2.4	CORDIC	11
2.4.1	Janiszewskis Hybrid	12
2.5	Sine Compression	12
2.5.1	Very Coarse Approximations	12
2.6	Complex Rotation	14
3	Target Architectures	15
3.1	Common FPGA Architecture	15
3.2	Altera	15
3.3	Xilinx	16
4	Modeling	19
4.1	Quality Units	19
4.2	Frequency Control Word Effects	20
4.3	Rounding Noise Analysis	20
4.3.1	Methods	20
4.4	Algorithm Verification	21
4.4.1	ROM/Polynomial	21
4.4.2	Other Decomposition Solutions	24
4.4.3	CORDIC	24
4.4.4	Sine Compression	24
4.4.5	Method Codes	24

4.5	Truncation Noise Analysis	25
4.5.1	Polynomial	26
4.5.2	Other Decomposition Solutions	27
4.5.3	Sine Compression	28
4.6	Conclusion	28
5	Implementations	31
5.1	ROM	31
5.1.1	The Function create_rom	31
5.1.2	The VHDL Implementation	31
5.2	SURD Implementation	32
5.3	Polynomials	32
5.3.1	The Function psac_polynomial_rom	32
5.3.2	The Function psac_polynomial	32
5.3.3	The Function create_polynomial	33
5.3.4	The VHDL Implementation	33
5.4	Sunderland's	33
5.4.1	The Function psac_sunderland_rom	34
5.4.2	The Function psac_sunderland	34
5.4.3	The Function create_sunderland	34
5.4.4	The VHDL Implementation	34
5.5	Test Bench	35
5.5.1	The Function create_testbench	35
5.5.2	The VHDL Solution	35
5.6	Automatic Generation/Verification	35
5.6.1	The Function test_psac	35
6	Suggester	37
6.1	The Properties	37
6.2	Cost Model	38
6.3	Algorithm	38
7	Result	39
8	Conclusions And Possible Improvements	41
8.1	Conclusions	41
8.2	Suggested Improvements	41
	Bibliography	43
A	What is...?	45
B	Quality and Resource Tables	47
B.1	ROM/Polynomial	47
B.2	Other Decompositions	51
B.2.1	The F and Method Groupings	51
B.2.2	The Quality Groupings	51
B.3	Sine Compression	53
C	Polynomial VHDL Example	58

Chapter 1

Introduction

This chapter will discuss the project and this report, and introduce some terms that can be good to know in the thesis.

1.1 Background

Digital electronics become more and more widely used compared to analogue electronics. Many tasks that have earlier been implemented with analogue circuits are today more suitable - one way or another - to be replaced by digital technology.

One area that grows fast is for instance wireless communication, where information is sent as radio waves. This requires that a sine wave is generated that can carry the information. This sine is comparably simple to generate in analogue electronics, and rather complicated to calculate for digital circuits, why the analogue method still is in use. The analogue method has two drawbacks; it is hard to control the frequency exactly, and the generated signal may be hard to manipulate.

You can use a microprocessor to calculate the sine, but this report will focus on the ASIC¹ implementation - that is how to program logics that calculate the sine, rather than how to program the instructions that is executed by the microprocessor in order to calculate the sine.

The main target for the ASIC is to use an FPGA - Field Programmable Gate Array, that is an electrical chip with such programmable logic. This will be further described in chapter 3, “Target Architecture”.

1.1.1 DDFS

The term DDFS stands for Direct Digital Frequency Synthesis, and in this context means a way to produce a sine wave with a given frequency. The method also use a clock signal that defines the time between one calculation and the next. The clock signals typically flips between '0' and '1' at a frequency of f_{clk} , for instance 300 MHz.

The DDFS usually contains a counter that counts from 0 to something big, and then restarts, and counts up with a number – the Frequency’s Control Word (FCW). The content of the counter is treated as a phase (angle, here called x_N), and is then sent to a Phase to Sine Amplitude Converter (PSAC) that calculates a sine value, y , for the phase. See figure 1.1 for an illustration.

This sine value will then over time have the shape of a sine wave with the exact frequency that was given to the DDFS.

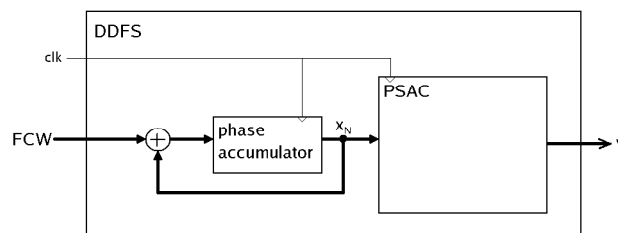


Figure 1.1. Illustrations of the basic DDFS structure

¹Application Specific Integrated Circuit, a method of customizing the hardware to a specific need

1.2 Purpose

This thesis aims to develop a method for automatic generation of the PSACs for FPGAs, for different PSAC methods, including a way of suggesting a suitable PSAC method for different types of FPGAs and for different requirements. The purpose of this is to simplify the creation of the PSACs for FPGA developers.

1.2.1 Quality vs Resource Problem

One of the main problems with a manual implementation is that the user may have different requirements on the signal. It can for instance be suitable to lower the quality on the signal in order to save some resources. Problems like this can be time wasting to solve manually, and this project includes a solution for that (see chapter 6, “Suggester”).

1.3 This Document

This documents is mainly intended for those with a basic knowledge in digital technology and a basic knowledge in math. For those who are not familiar with all terms and expressions, appendix A, “What is...?” contains a list of abbreviations/concepts and a short explanation.

1.4 Project Approach and Overview

Before the project was started there were some things to decide. First of all which language the project should be implemented in, but also how to split up the project in sub tasks.

This section introduce each of those sub tasks, which more or less corresponds to the chapters in this report.

1.4.1 Implementation Language

The first thing to do was to decide in which programming language the system should be built. There were three main alternatives: Matlab from The Mathwork Inc, Microsoft Excel and any high level language, like C++ or Java.

Language	Benefit	Drawbacks
Matlab	Very good support for mathematic analysis, convenient file I/O, widely used for similar tasks.	Everyone doesn't have Matlab.
Excel	Easy to create good graphical user interface, and to store much data.	Hard to do an FFT in Visual Basic.
C++	Widely known.	Not very suitable for this kind of calculations

Table 1.1. Benefits and drawbacks for different implementation languages

The choice fell on Matlab, mostly because of the mathematical intensity of the program.

1.4.2 Finding Existing Methods

The first thing to do was to elaborate what else had already been done on the topic. The most interesting and suitable methods were then chosen for further analysis.

In this task it is good to have read up on the FPGA architectures, to easier know which methods are interesting, and which one can be omitted at once. However, when studying the FPGAs it is good to know the methods, to know what to look for. Therefore this task is put before the FPGA architecture study.

Many methods have one or more parameters, the most obvious one is the decomposition of the phase bits, which can be done in $N+1$ ways (if the phase is N bits wide). In this report, and in the Matlab implementation, the word *configuration* is used to describe a specific method and its different parameters. More about this in chapter 2, “Methods”.

1.4.3 Target Architectures

The main target is FPGAs, and the *suggest* functionality² should have a good knowledge about the different FPGA types and architectures, why a study of the most common FPGA types is required.

1.4.4 Modeling and Analysis

The chosen methods are verified and analyzed in Matlab. The worst methods are discarded from the project.

In order to analyze the resources needed for the methods, this task should be performed after the FPGA architecture study.

1.4.5 VHDL Implementation

The main purpose with the project is to create a PSAC. In practice that means that one or more files with VHDL-code³ are generated.

There are several other hardware describing languages as well, but the requirements specify a VHDL generator.

The Matlab implementation should be able to generate VHDL implementations for the given methods, in all possible configurations, plus testbenches that verify the functionality.

This task must of course be performed after the method modeling. This task is covered in chapter 5.

1.4.6 Suggester

One of the more tricky parts of the project is to write an algorithm that checks for the best implementation according to given preferences/limitations and a given FPGA architecture.

This task must also be performed after the method modeling. In order to get a better model of the resources that are used, this task is placed after the VHDL implementation.

1.5 Limitations

The main limitation of this master thesis is the time of 800 hour, including time to present and defend the work and the time to hold the place of an opponent. Within this time limit the thesis aims to be as good as possible, according to some prioritation.

The main effects are a reduced subset of algorithms/methods, a limited extension of configurations within the methods, and a restriction to only test the code for Matlab in Unix, in difference to Windows and/or Octave⁴. It also affects the suggester function, both in efficiency and in complexity. More about the limitations in the respective chapters.

1.6 Notations and Abbreviations

This is only a short list of the most important abbreviations. See Appendix A - "What is...?" for a complete list of abbreviations, notations and descriptions.

dB - decibel, a logarithmic scale for comparing relative difference.

dBc - dB relative to the carrier, measured as the difference in power.

FCW - Frequency Control Word

LUT - Look Up Table, or function generator.

ModelSim - A program from Mentor Graphics aimed to compile and simulate for instance VHDL code.

PSAC - Phase to Sine Amplitude Converter, a sine function.

SFDR - Spurious Free Dynamic Range, a way to measure signal purity.

SINAD - Signal to Noise And Distortion ratio, a way to measure signal purity.

²Suggests a suitable configuration - see 1.4.6

³the VHDL code describes how the FPGA should be programmed to realize the intended behaviors

⁴Octave is an open source variant of Matlab

SNR - Signal to Noise Ratio, a way to measure signal purity.

SURD - Symmetry Using Range Divider. A phrase within this thesis for a way to reduce the input range to the PSAC.

VHDL - A Hardware Description Language, describes how the information flows and is treated in the FPGA.

Chapter 2

Methods

There are of course a number of ways to compute a sine wave.

The authors Langlois *et al.* has summarized a large number of DDFS techniques[1], which is the main source of methods in this thesis.

Some notations:

N - Number of bits in phase. The two MSBs are not used more than in the SURD.

C - Coarse, some of the most significant bits from the phase (except the two used by the SURD).

F - Fine, some of the least significant bits from the phase.

D - Some bits between C and F in Sunderland's method. D is just a suitable letter between C and F .

W - Number of bits in amplitude. The MSB is not used more than in the SURD.

K - Number of coefficients in polynomial solutions.

SURD - Symmetry Using Range Divider. A method that uses the symmetry of the sine to reduce the phase with two bits.

x - The first 90° of the phase.

$x_{N,Q,C,D,F}$ - The phase containing the N , C , D resp. F bits. x_Q is the two bits used by SURD.

Those notations will be further described in their respective sections below.

2.1 Symmetry Using Range Divider

The Symmetry Using Range Divider (SURD) decreases input phase from range $0-360^\circ$ to either $0-90^\circ$, for algorithms returning only sine, or $0-45^\circ$ for quadrature algorithms, returning both sine and cosine, by using the symmetry in the sine wave. This method is used as a wrapper function around other algorithms, minimizing the input range to the functions and compensating the output result from the function. All discussions and illustrations here will assume the $0-90^\circ$ version, for simplicity reasons. See figure 2.1 for the phase decomposition, or figure 2.2 for the phase and amplitude effects. Equation 2.1 illustrate the math behind the SURD.

$$\sin(90^\circ \cdot x_N) = \sin(90^\circ \cdot (x_Q + x)) = \begin{cases} \sin(90^\circ \cdot x), & x_Q = 0 (0^\circ) \\ \sin(90^\circ \cdot (1 - x)), & x_Q = 1 (90^\circ) \\ -\sin(90^\circ \cdot x), & x_Q = 2 (180^\circ) \\ -\sin(90^\circ \cdot (1 - x)), & x_Q = 3 (270^\circ) \end{cases} \quad (2.1)$$

The SURD is divided into the two parts *PreSURD* and *PostSURD*, where *PreSURD* handles the phases phase inversion $(1 - x)$, and *PostSURD* inverts the result, if needed.

The values x_N are in equation 2.1 meant to be a number 0 to 4 ($0 \leq x_N < 4$), which implies that x_Q (that are the 2 MSBs of x_N) are an integer with value 0, 1, 2 or 3. x is then the fraction part, $0 \leq x < 1$.

Note that x_N is just a series of ones and zeros, and that this representation uses a decimal point between the x_Q and x fields. The sine approximation function may set the decimal point somewhere else for a suitable representation.

Because of the great benefits with SURD (see table 2.1), it is used in all implementations.

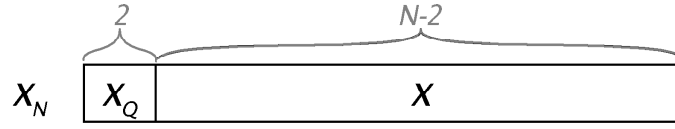


Figure 2.1. SURD decomposition

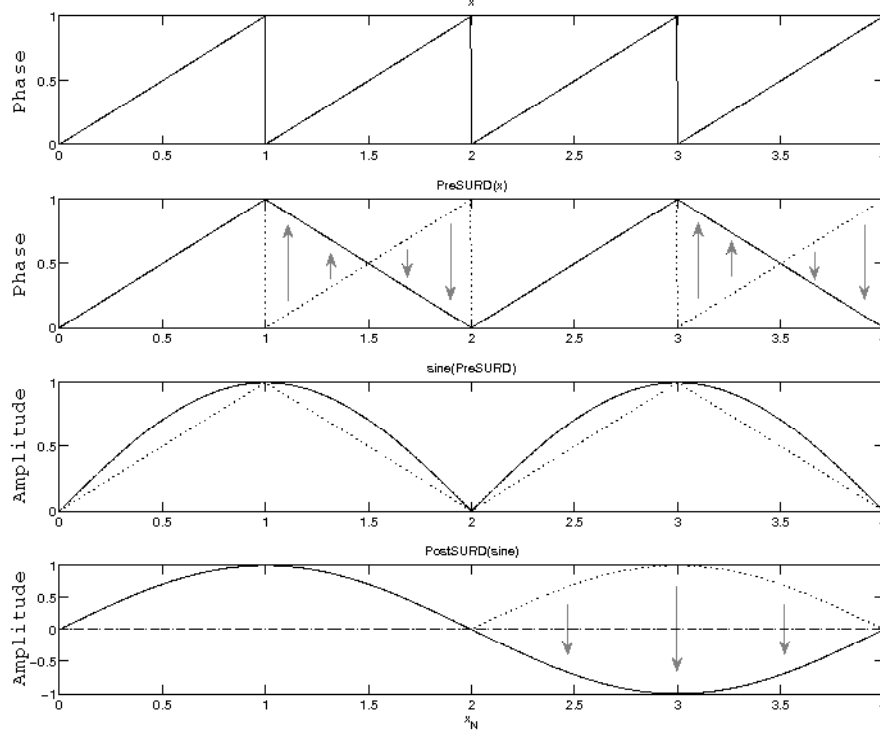


Figure 2.2. The SURD signal effects.

Benefits	The other algorithms can be designed with a very much smaller memories.
Drawbacks	In those cases there is a ROM dedicated for this task and it is big enough to fit the entire 0–360°, than that may be slightly faster.
Other properties	This method does not affect the precision of the result, but will increase the t_{CO} and t_{SU} with the time of one adder resp. one inverter, and will use slightly more logic resources. Depending on which other algorithm that is used it will however save loads of ROMs and/or FAs.

Table 2.1. Some properties for SURD

2.2 ROM/LUT

The ROM/LUT method uses a big look up table, or better known as ROM, to store the entire function. See table 2.2.

$$\sin(x) = R[x], \quad (2.2)$$

where R is the ROM and x is used as the address.

This method has high priority according to the requirements. Langlois[1] mention it.

Benefits	Works fast, very simple.
Drawbacks	Grows exponentially with input width.
Other properties	Exact result (as exact as possible with actual word width). Suitable for simulator and FPGAs with big ROMs and/or high performance requirements.

Table 2.2. Some properties for ROM implementation

2.3 Decomposition

Another method than the ROM solution may be “decomposition solutions” (or “bipartite solutions”), discussed as follows.

Split the $N - 2$ input bits into C (Coarse) MSB and F (Fine) LSB, where $N = 2 + C + F$ (the 2 MSBs are reserved for the SURD). Let $x = x_C + x_F$ be the values stored by the C and F bits, as illustrated in figure 2.3. Or, in the Sunderland’s approach case (see section 2.3.4 below), let N be $2 + C + D + F$ and $x = x_C + x_D + x_F$ in a similar way. See figure 2.3 and 2.4.

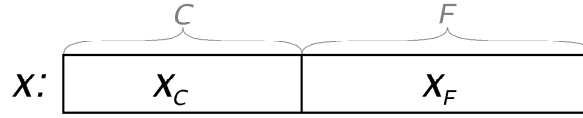


Figure 2.3. Decomposition of quartile phase into two fields

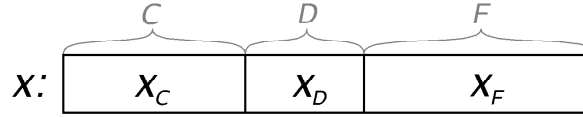


Figure 2.4. Decomposition of quartile phase into three fields

For example, if $N = 11$, $C = 4$, $F = 5$ and $x_N = 11000011111$, then $x_Q = 11$, $x_C = 0000$ and $x_F = 11111$.

2.3.1 Polynomial Interpolation Alternative

Split the phase into 2^C parts, and calculate each part as a polynomial, according to figure 2.5. Because you will use all coefficients for a certain polynomial in the same time (\pm a few clock cycles), you can read out all the coefficients for that polynomial in the same time. Therefore, you can use a ROM with 2^C lines and store the K coefficients side by side in it, one polynomial per ROM line. This way you only need one ROM (which may however require several ROMs if the total required number of bits does not fit into one ROM).

When using K coefficients you get a $(K - 1)$:th grade polynomials. The “x” in the polynomials are the F fraction bits, x_F , according to equations 2.3.

$$\begin{aligned}
 \sin(x_C + x_F) &\approx R_1[x_C], & K &= 1 \\
 \sin(x_C + x_F) &\approx R_1[x_C] \cdot x_F + R_2[x_C], & K &= 2 \\
 \sin(x_C + x_F) &\approx (R_1[x_C] \cdot x_F + R_2[x_C]) \cdot x_F + R_3[x_C], & K &= 3 \\
 \sin(x_C + x_F) &\approx (\dots(R_1[x_C] \cdot x_F + R_2[x_C])\dots + R_{K-1}[x_C]) \cdot x_F + R_K[x_C], & K &\geq 4,
 \end{aligned} \tag{2.3}$$

where $R_k[x_C]$ is coefficient k for polynomial x_C .

The coefficients on a line is used for a $(K - 1)$:th grade polynomial applied on the F truncated bits. This allows C to be small if K is fairly big without losing too much quality. In table 2.3 the quality terms *SINAD* and *SFDR* are used, those will be described in section 4.1, *Modelling - Quality Units*.

Worth mentioning is that a high K requires complex calculations, which will cause a high latency and/or a huge restriction of the clock frequency.

The last coefficient in the ROM (R_K) will always be $W - 1$ bits wide, because it must store the “offset” position for that polynomial, which will be within the range $[0, 2^{W-1})$. The previous coefficients will shrink with roughly C bits per coefficient.

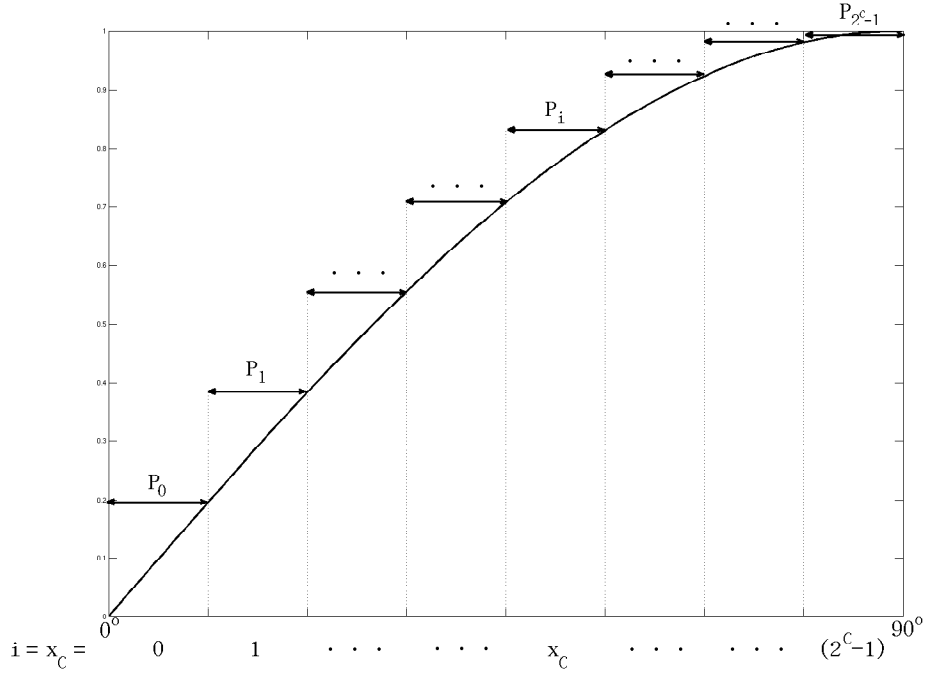


Figure 2.5. Polynomial decomposition illustration

K	C	Benefit	Drawbacks
Big	Big	Exact value (except noise in W's LSBs).	Much memory and many (rather small) multipliers needed.
Big	Small	Few ROMs are needed.	Many and big multipliers are required. Low SFDR.
Small	Big	Few and small mults are required.	Much memory is required. Low SINAD.
Small	Small	Resources effective, easy to calculate.	Rough approximation. Very low SFDR and SINAD.

Table 2.3. Some properties for polynomial solution

One of the big implementation problems with polynomials is to create the coefficients, and which approach to use. In table 2.4 some methods are mentioned, which will be analyzed further in chapter 4.

Least Square	Minimize average error \Rightarrow maximize SFDR (in general)
Chebyshev	A kind of interpolation where the interpolation points have been chosen to minimize the maximum error \Rightarrow maximize ENOB \Rightarrow maximize SINAD (in general).
Interpolation	Like Chebyshev interpolation, but “stretch out” the points so there are one point in each edge of the ranges.
TaylorLeft	Taylor series from a point in the left edge of the F interval.
TaylorMid	Taylor series from a point in the middle of the F interval.
Truncation	Just pick the left most value in each range. Requires that $K=1$.
ROM	The ROM solution (section 2.2) is a truncation special case where $F=0$.

Table 2.4. Some methods for polynomial coefficients

An explanation to the Chebyshev points: They are placed within each polynomial ranges in the same way as the values $\cos(\frac{90^\circ + i \cdot 180^\circ}{K})$, $i = 0..K - 1$ are placed in the range $(-1, 1)$.

2.3.2 Phase Truncation Alternative

Truncate the input to the C MSB in order to save some ROM. This is a special case of polynomial interpolation where $K = 1$.

This method can be seen either as a way of reducing the ROM size as mentioned (by reducing the phase *bus* to the memory), or it can be seen as taking a ROM solution and increase the phase *accumulator* width with F bits in order to increase the frequency precision, but without giving the extra bits to the ROM – see table 2.5.

$$\sin(x_C + x_F) \approx \sin(x_C) = R[x_C] \quad (2.4)$$

View	Truncated phase bus	Increased phase accumulator
Benefits	Smaller ROM.	Higher frequency resolution.
Drawbacks	Much more noise.	More noise.

Table 2.5. Some properties for phase truncation

2.3.3 Hutchinson’s Approach

Hutchinson *et al.*[2] suggested a trigonometric approximation. This approach[1] uses the approximation $\sin(x) = \sin(x_C + x_F) = \sin(x_C) \cos(x_F) + \sin(x_F) \cos(x_C) \approx \sin(x_C) + \sin(x_F) \cos(x_C)$ which is without multiplication, since $\sin(x_F) \cos(x_C)$ is precalculated and stored in a separate ROM - a ROM that will be as high as the pure ROM solution, but around $W - F$ bits wide instead of $W - 1$ bits. In figure 2.6 the 3 graphs illustrate a solution where C is 5, 3 and 1. The *einf* and *e2* values are the maximum and average errors for the values, the terms will be better described in section 4.1. See also table 2.6.

$$\sin(x) \approx R_1[x_C] + R_2[x] \quad (2.5)$$

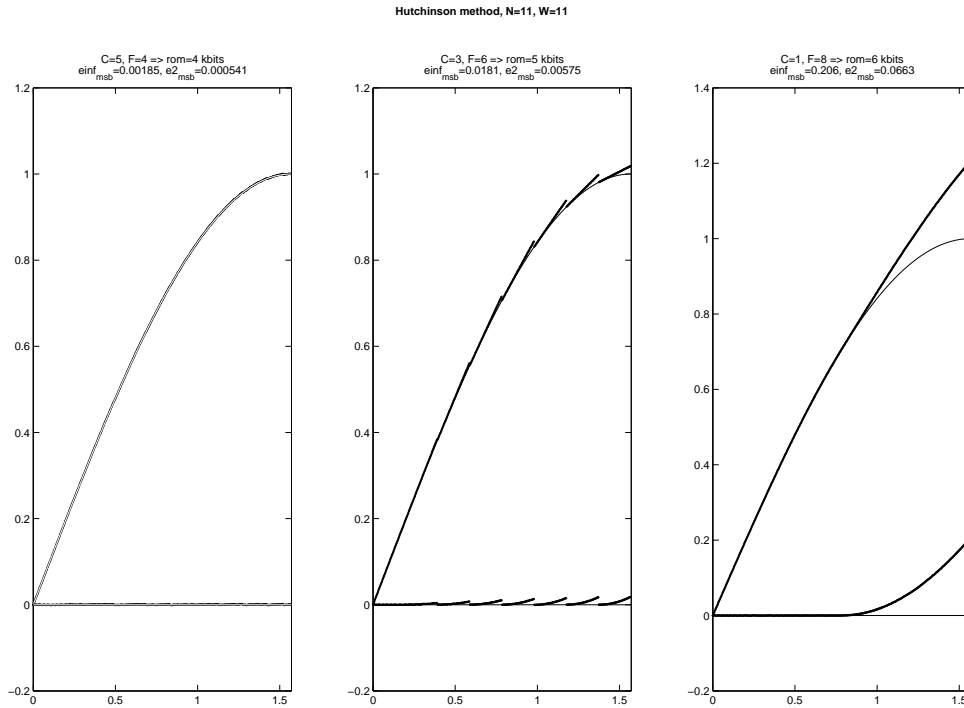


Figure 2.6. Hutchinsons implementation examples

The method can be improved by setting the content of ROM 2 to a “correction” to ROM 1. $R_2[x] = \sin(x) - R_1[x_C]$, rather than Hutchinsons original assignment $R_2[x] = \sin(x_F) \cos(x_C)$. This improvement does not affect the implementation cost of the algorithm, but removes the algorithmic error. This variant is in this thesis called *Hutchinson’s 2*, and it will be further analyzed in chapter 4, *Modelling*, page 19.

Benefit	Very simple and quick.
Drawbacks	Requires very much memory, and is rather inexact.

Table 2.6. Some properties for Hutchinsons approach

2.3.4 Sunderland's Approach

An extended version of Hutchinson's method was suggested by Sunderland *et al.*[3], where they divide the $N - 2$ input bits into 3 fields:

$$\begin{aligned}
 \sin(x) &= \sin(x_C + x_D + x_F) \\
 &= \sin(x_C + x_D) \cdot \cos(x_F) + \cos(x_C + x_D) \cdot \sin(x_F) \\
 &= \sin(x_C + x_D) \cdot \cos(x_F) + \cos(x_C) \cdot \cos(x_D) \cdot \sin(x_F) - \sin(x_C) \cdot \sin(x_D) \cdot \sin(x_F) \\
 &\approx \sin(x_C + x_D) + \cos(x_C) \cdot \sin(x_F)
 \end{aligned} \tag{2.6}$$

and so

$$\begin{aligned}
 \sin(x) &= R_1[x_{C+D}] + R_2[x_{C+F}] \\
 R_1[x_{C+D}] &= \sin(x_C + x_D) \\
 R_2[x_{C+F}] &= \cos(x_C) \cdot \sin(x_F)
 \end{aligned} \tag{2.7}$$

where x_{C+D} is the x_C and x_D bits concatenated together, and corresponding for the x_{C+F} bits.

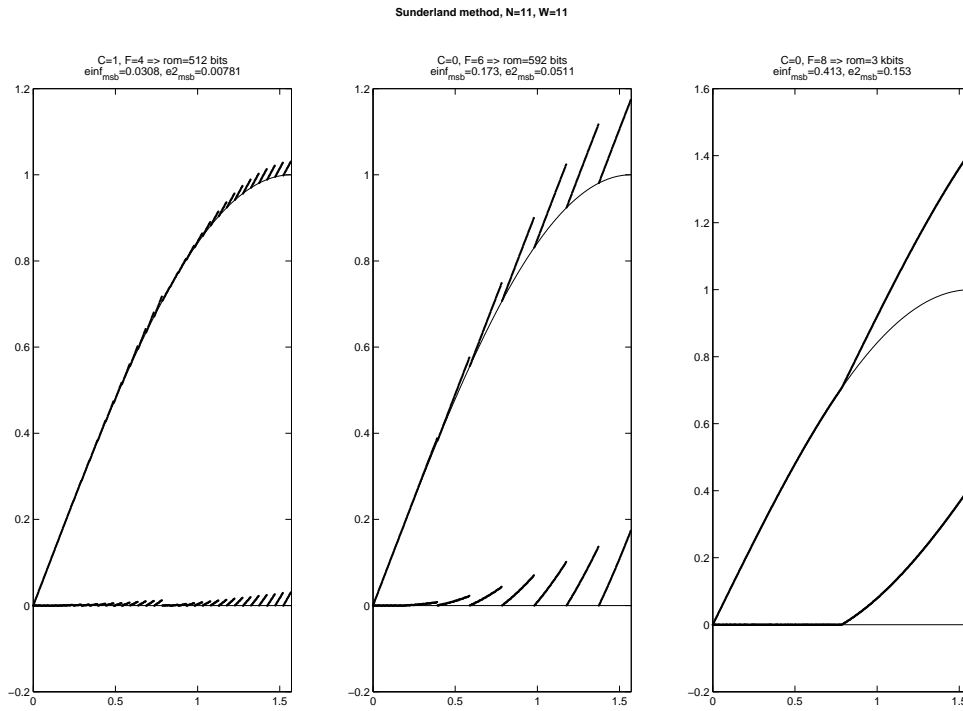


Figure 2.7. Sunderlands implementation examples

Benefit	Very simple to implement in VHDL.
Drawbacks	Rather bad precision.
Other properties	Due to the triple decomposition of the phase, this method have very many configurations when N is big.

Table 2.7. Some properties for Sunderlands method

2.3.5 Nicolas' Approach

Nicholas *et al.* [4] suggested that the ROM contents in Sunderland's approach should be changed and optimized according to some goal (e.g. high SFDR). Due to time limitation this optimization will not be investigated.

2.3.6 Curticăpean's Approach

Curticăpean *et al.* [5] suggested an improvement to Hutchinson's approach, that stores $\sin(x_F)$ and $\cos(x_C)$ in one ROM each and multiplies them together. See table 2.8 and figure 2.8.

$$\sin(x) \approx R_1[x_C] + R_2[x_C] \cdot R_3[x_F] \quad (2.8)$$

where R_1 and R_2 can be stored side by side in one Rom (because both are addressed with x_C).

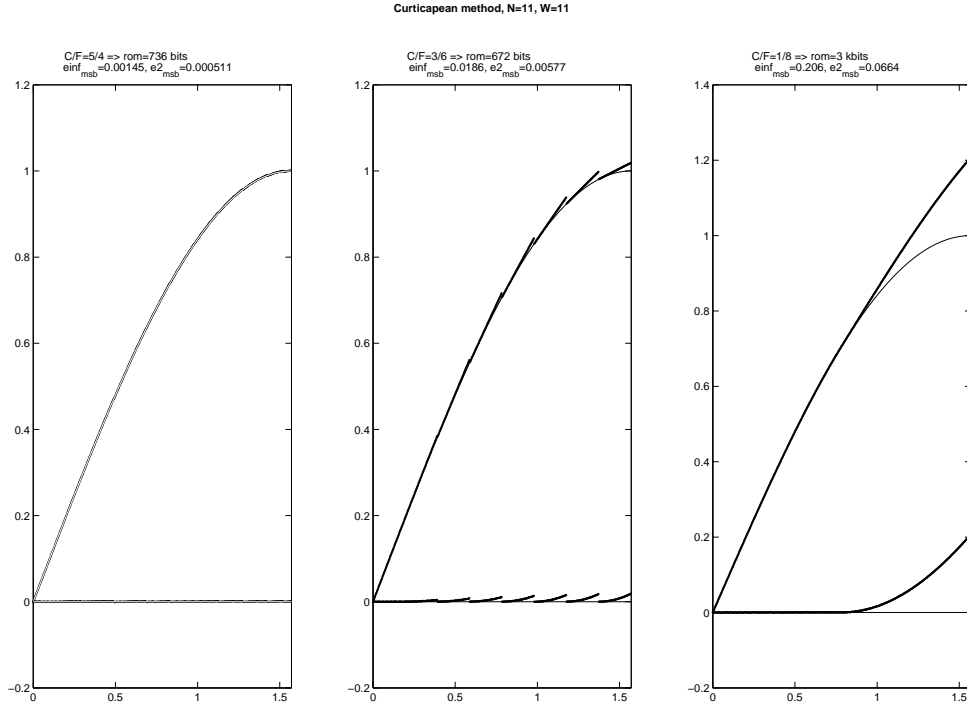


Figure 2.8. Curticăpeans implementation examples

Benefit	Saves some ROM.
Drawbacks	Cost one multiplication.

Table 2.8. Some properties for Curticăpeans method, relative Hutchinson's

2.4 CORDIC

The CORDIC algorithm is a very resource efficient and “exact” method for sine calculation, it requires no multiplication and very little ROM. It does, however, require $N - 2$ comparison with corresponding additions/subtractions, that must be executed after each others, which gives either a slow clock or a very large latency.

CORDIC stands for **C**oordinate **R**otation **D**igital **C**omputer, and is a set of algorithms based on the idea to process the input with smaller and smaller steps toward the zero, and at the same time process the output with some corresponding operations. In each step there is a binary decision, like “increase or decrease” which affect the following operations on the (modified) input and output values. After a predefined number of steps, the output is ready. The CORDIC sine method is carefully described on for instance Wikipedia[6].

This method has medium priority according to specification.

Benefit	No multiplication, very limited ROM.
Drawbacks	Takes long time.

Table 2.9. Some properties for CORDIC

2.4.1 Janiszewskis Hybrid

Look up the first C bits in a LUT, and feed that to step $K..W$ of the CORDIC. Due to the drawback in table 2.10, this method will not be implemented.

Benefit	Faster than CORDIC, smaller than LUT
Drawback	Still not as fast as for instance polynomial solution.

Table 2.10. Some properties for Janiszewskis hybrid

2.5 Sine Compression

Calculate a rough estimation to the sine somehow, and include a ROM that contains the errors. This ROM will be as high as the pure ROM solution, but much thinner.

As estimation it is convenient to use any of the methods discussed in Decomposition above, or a “very coarse approximation” (see below). Some drawbacks and benefits are shown in table 2.11.

Benefit	You get an “exact” solution (errors $\leq \frac{1}{2}LSB$)
Drawbacks	The extra ROM needs to be 2^{N-2} rows high.

Table 2.11. Some properties for sine compression

This compressed ROM may then and now be called a “correction ROM” to the approximating function.

2.5.1 Very Coarse Approximations

Some extremely simple approximations is worth mentioning. All approximating $\sin(90^\circ \cdot x)$, where $0 \leq x < 1$.

The cost of using those approximation is the cost of the approximations themselves plus one adder, where you add the approximated value to the compressed ROM output. The cost of one register for storing the approximation may be needed. See figure 2.9 for illustrations.

Due to time limitations, those methods will not be implemented.

Identity Approximation

The simplest one.

$$\sin(x) \approx x. \quad (2.9)$$

This can be shown to save 2 bits of ROM width, which holds even if x is truncated to 5 bits.

Langlois Approximation

Named after Langlois and Al-Khalili[7].

$$\sin(x) \approx \begin{cases} \frac{3}{2}x, & 0 \leq x < \frac{5}{16} \\ \frac{5}{32} + x, & \frac{5}{16} \leq x < \frac{3}{4} \\ \frac{1+x}{2}, & \frac{3}{4} \leq x < 1 \end{cases} \quad (2.10)$$

This will save 4 bits of ROM width. All segments in this solution uses at most one adder and no other logic.

Sodogar Approximation

Named after Sodogar and Lahiji[8].

$$\sin(x) \approx x(2 - x), \quad 0 \leq x < 2. \quad (2.11)$$

The subtraction is done using a simple inverse of all bits in x since $0 \leq x < 2$ causes $2 - x$ to be equal $-x$, and the negation of x can be approximated to bitwise inverse in this case. The formula is valid for $0 \leq x < 2$, but the interesting part is $0 \leq x < 1$ due to the SURD.

The error here is $< \frac{1}{16}$, which means we save 4 bits of ROM width.

LUT Approximation

As will be described in chapter 3, “Target”, the FPGAs are to a high degree built up of very small memories, so called LUTs. These have typically 4 or 6 address bits, say L input bits, and just one output bit. If you feed the L MSBs of x_C to some LUTs, the LUT outputs can act as an approximation, and will save $L - 1$ bits from the ROM, assuming at least L LUTs are used. Except the Identity matrix, that do not require any LUTs at all, the other “very coarse approximation” methods will require at least as many LUTs as precision uses for the approximation. Because of that, this method will not be more expensive than for instance Langlois approximation.

If you take only the $L - 1$ MSB of x_C , you can use the output from the compressed ROM as the last bit, and combine it with the adder functionality (if the used FPGA architecture allows it). In this way the compression may be around $L - 2$ bits to the same cost as the identity approximation. The exact value will however not be investigated due to time limitations.

In the illustration, L is set to 3 for illustration reasons, and the worst error is ≈ 0.2 , just below position $x = 0.125$. This saves 2 bits of ROM width. The figure does not illustrate effects from a limited number of LUTs used.

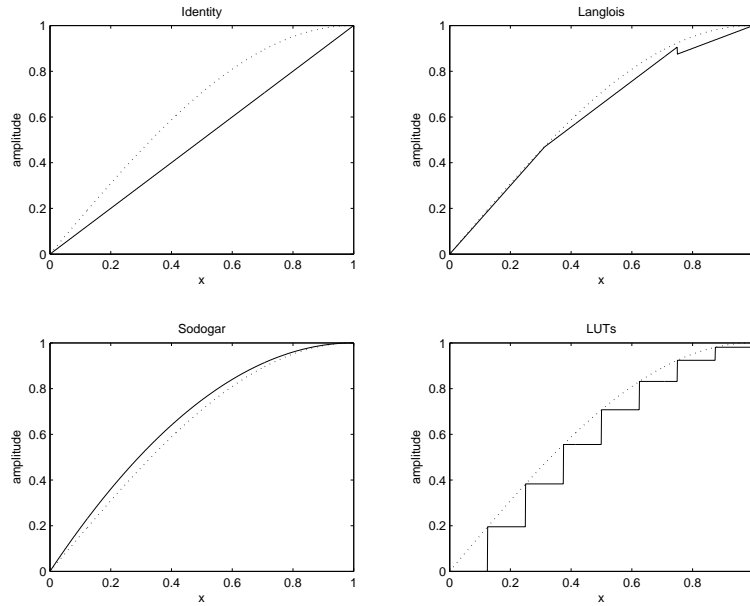


Figure 2.9. Some “Very Coarse Approximations”

Usability

The four approximations above can be used not only to compress the pure ROM solutions; they can also be used to compress any coefficients that store a sine table, e.g. the last field in the polynomial solutions, or the first ROM in Sunderlands method.

2.6 Complex Rotation

Input the frequency rather than the angle. Have a complex vector v , that is multiplied with the complex constant $e^{angle \cdot i}$ in each step.

Benefit	Pretty simple, low power, since the sine and cosine is to be calculated once per frequency change rather than once per sample.
Drawbacks	Low precision.

Table 2.12. Some properties for complex rotation

This method has very low priority, according to thesis specification.

In applications where the power is critical and resources are cheap this is interesting. If you use a PSAC to calculate the complex value $e^{freq \cdot i}$ every time the frequency is changed, and then and now between that do a correction of the actual vector v , the method can save some power because the complex multiplication in some cases may be more energy efficient than the PSAC calculation.

Chapter 3

Target Architectures

The main implementation target device is FPGAs, Field Programmable Gate Arrays. The FPGA is a chip that is configurable to behave in almost any way the user want it.

There are two main competing FPGA vendors, Altera and Xilinx, with some FPGA families and generations each. There are other vendors as well, but this thesis will only cover those two.

To simplify the handling of the different families/generations of FPGAs in this thesis, they have been assigned abbreviations, or codes. See tables 3.2 and 3.4 for the codes.

3.1 Common FPGA Architecture

The FPGA is normally used for digital signal processing, “glue logic”, and other types of digital tasks. Therefore it is both generalized and specialized in the same time. Typical FPGA components are:

Logic: FPGAs are mainly built up by many small *LUTs* (look-up tables), *FAs* (Full Adders), and *DFFs* (D-flip-flops), and small *muxes*. Many LUTs can also work as memories. Xilinx group the logic into slices/CLBs, Altera group it into LE/ALM/LABs. See figure 3.1 for a very simplified example of how the structure can be organized. A LUT with e.g. 6 inputs (6-LUT) can implement any combinatorial function of those inputs.

Memories: There use to be synchronous memory blocks, often configurable as RAM, ROM, shift register or FIFO buffer, and in several different heights and widths. For example the size 4 kBits = 2^{12} bits can be shaped as 8 bits wide and $512=2^9$ rows high, or 32 bits wide and $128=2^7$ rows high. The address widths are 9 and 7 bits in those cases, respectively.

Memories can have single port (SP) or dual port (DP) features, meaning you can access the memory content from one and two sides, respectively. This thesis will only use ROMs with SP configurations, and hence the rest of the memory configurations are not listed here. For futher improvements the DP may be interesting, as noted below.

In most cases there are optional bits reserved for parities, usually 1 parity bit for each byte.

The LUTs are very small ROMs. Many of the LABs/CLBs can configure the LUTs as e.g. RAM, but in this thesis that is extraneous. The ROM function is implicit, and omitted in the ROM list.

Multipliers: Most FPGAs are equipped with dedicated binary multipliers. Altera uses 18×18 bits signed or unsigned, Xilinx uses 25×18 signed in their latest FPGAs.

FPGAs typically contains a lots of other features as well, but nothing interesting in this thesis.

3.2 Altera

Altera has a number of different FPGA families[9]. Common for all is:

Logic: The logic is based on LABs (Logic Array Blocks), consisting of either 8-10 ALMs (Adaptive Logic Module) or 10-16 LE (Logical Element).

- The **LE** has typically a 4-LUT, a FA and a DFF. In some devices the entire FA is implemented into the LUT.

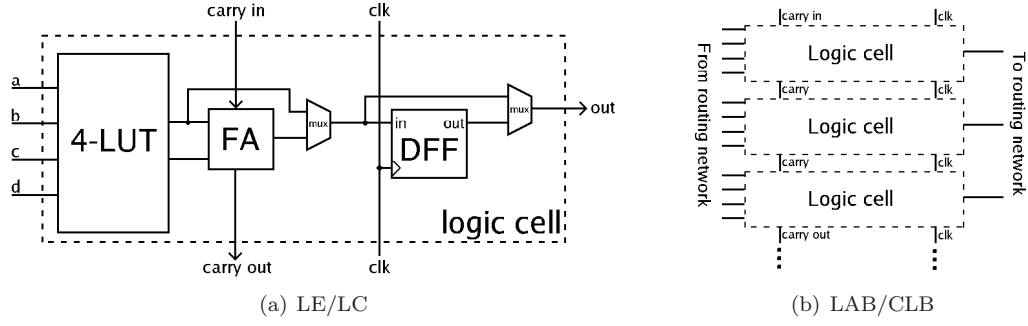


Figure 3.1. Simplified example of a general FPGA logic architecture

- The **ALM** has typically four 4-LUTs, two FAs and two DFF, but can be configured as two 5-LUTs with two DFF, or one 6-LUT with a DFF.

ROMs: There are a few different ROMs. See table 3.1.

Multipliers: They have 18x18-multipliers, most of them configurable as $2 \cdot (9 \times 9)$ or $1 \cdot (18 \times 18)$. Both signed and unsigned (and combined) values are accepted.

Name	Size	Modes	Address widths
M512	512 bits + 1 parity/byte	SP	5 – 9 (18–1 bits data wide)
M4K	4 kBits + 1 parity/byte	SP/DP	7 – 12 (36–1 bits data wide)
M9K	8 kBits + 1 parity/byte	SP/DP	8 – 13 (36–1 bits data wide)
M144K	128 kBits + 1 parity/byte	SP/DP	11 – 14 (72–8 bits data wide)

Table 3.1. ROMs in Altera’s FPGAs

A summary of the Altera FPGAs and their relevant resources can be seen in table 3.2.

Device	Code	ROM	LABs	Multipliers
Cyclone 1 [10]	cy1	M4K	10 LEs	no
Cyclone 2 [11]	cy2	M4K	16 LEs	18x18
Cyclone 3 [12], Cyclone 4 [13]	cy3, cy4	M9K	16 LEs	18x18
Arria 1 [14]	ar1	M512, M4K	8 ALMs	18x18
Arria 2 [15]	ar2	M9K	10 ALMs	18x18
Stratix 1 [16], Stratix 1 GX [17]	sx1	M512, M4K	10 LEs	18x18
Stratix 2 [18] Stratix 2 GX [19]	sx2	M512, M4K	8 ALMs	18x18
Stratix 3 [20], Stratix 4 [21]	sx3, sx4	M9K, M144K	10 ALMs	18x18

Table 3.2. Alteras FPGA Families

3.3 Xilinx

Xilinx has two different families with different generations. Common for all:

Logic: The logic is based on *CLBs* (Configurable Logic Blocks), which have two or four *Slices*. Two notations within this thesis:

- The **slice₂** has $2 \cdot (4\text{-LUT} + \text{FA} + \text{DFF})$, but can act as a $5\text{-LUT} + 1 \text{ DFF}$. [22]
- The **slice₄** has $4 \cdot (6\text{-LUT} + \text{FA} + 2 \text{ DFFs})$, but can act as an $8\text{-LUT} + 1 \text{ DFF}$. [23]

Xilinx often implements the FAs into the LUTs, except the carry logic, to optimize speed and complexity.

Memories: There are two types of memories: Block SelectRAM = 4-32 kBits, in this thesis abbreviated ‘bsRam’, or Slices configured as 16 or 64 bits. Many bsRams do not have any preload functionality and will not be listed as ROM. The slices ROM functionality is simple LUT usage, which are implicit and will not be listed in table 3.3.

Multipliers: Xilinx' multipliers are typically 18x18 bits signed. Unlike Alteras FPGAs you cannot configure them to be unsigned. To do an unsigned multiplication, which this thesis deals with, you must sacrifice the sign bit, so that typically 17x17 bits are used.

Name	Size	Modes	Address Widths
bsRam16	16 kbits + 1 parity/byte	SP/DP	10–14 (18–1 bits data width)
bsRam32	32 kbits + 1 parity/byte	SP/DP	10–15 (36–1 bits data width)

Table 3.3. ROMs in Xilinx' FPGAs

The resource types for the different generations are summarized in table 3.4.

Device	Code	ROM	CLBs	Multipliers
Spartan-3 [24]	sp3	bsRam16	4 slices ₂	18x18 signed
Spartan-6 [25]	sp6	bsRam16 [26]	2 slices ₄	18x18 signed [27]
Virtex [28, 29, 30]	vx1	no	2 slices ₂	8x8 or 16x16
Virtex-II [31, 32]	vx2	no	4 slices ₂	18x18
Virtex-4 [33, 34]	vx4	bsRam16	4 slices ₂	18x18 signed
Virtex-5 [35], Virtex-6 [36]	vx5,vx6	bsRam32 [37]	2 slices ₄	25x18 signed [38]

Table 3.4. Xilinx' FPGA Families

Chapter 4

Modeling

As decided in the introduction chapter, section 1.4.1, the main language will be Matlab. Therefore all models will be built in Matlab.

The models have two purposes: Verify (and understand) the algorithm, and to analyze the signal quality. The quality term refers to how good the signal is compared to how much noise it contains.

In the digital environment in an FPGA there are two types of noises: Truncation and rounding noise. Truncation noise origins from approximation errors¹, e.g. introduced error when doing piecewise linear approximation. Rounding noise is introduced because of finite word length in operations and result, e.g. 11 bits precision in result, or several mult/adds where each operation introduce a rounding error.

4.1 Quality Units

The quality of a signal uses to be measured in SFDR² and SNR³. The SFDR measures how much “louder” the carrier is than the loudest noise tone. The SNR compares the carrier to the sum of the noise, after the harmonics to the signal has been removed, this is meant to measure the noise that does not belong to the carrier.

Because the base frequency is $\frac{f_{clk}}{2^N}$ (where f_{clk} is the clock frequency), and all occurring tones, carrier as well as noise, have frequencies $FCW \cdot \frac{f_{clk}}{2^N}$, all tones are harmonics to the base frequency. Therefore there is no ‘SNR noise’. However, the rounding noise is “white” over these discreet frequencies, and can be seen as not belonging to the carrier signal. Therefore the SNR will in this thesis be counted as only the Carrier-to-rounding-noise error.

To count the harmonics as noise, you use the measurement $SINAD$ ⁴, which counts everything that is not the carrier as noise.

The SFDR compares the carrier and the loudest noise tone. That tone is usually generated by a truncation error. The truncation errors is only limited by the approximating algorithm, and they occurs often “in groups” - that is, if the approximation has an error in one point, it is likely to have similar errors in the closest points.

Other quality methods are the ENOB⁵, average error and maximum error, according to the following definitions:

$$\begin{aligned} ENOB &= \frac{SINAD-1.76}{6.02} \\ e2 &= \sqrt{\text{mean}(e_i^2)}, \quad e_i = \text{error in point no } i. \\ einf &= \max(|e_i|) \quad i = 0, 1, \dots, 2^N - 1 \end{aligned} \tag{4.1}$$

$e2$ is the average error (or more accurately the RMS⁶ error), and $einf$ is the maximum error. The notations are derived from the mathematical terms e_2 and e_∞ , where $e_\alpha = (\text{mean}(e_i^\alpha))^{1/\alpha}$

¹error = difference between calculated and exact sine value

²Spurious Free Dynamic Range

³Signal to Noise Ratio

⁴Signal to noise and distortion ratio

⁵Effective Number Of Bits

⁶Root Mean Square

4.2 Frequency Control Word Effects

The DDS's generated signal frequency is controlled through the Frequency Control Word (FCW), according to equation 4.2.

$$f(FCW) = f_{clk} \cdot \frac{FCW}{2^N}, \quad 0 \leq FCW < \frac{2^N}{2} \quad (4.2)$$

Because of frequency mirror effects the harmonic tones will be mirrored back again when FCW is big, and due to number theoretical effects, it will never be added upon any other frequency as long as FCW is odd. Therefore the amplitude of the noise is not affected by (odd) FCW, and neither are the amplitude of the carrier signal.

In figure 4.1 you can see how the qualities are constant when the FCW is odd, but how they are affected when FCW is even. One reason for the even FCW behavior is that half of the phases are used twice, and the rest is not used at all. Some rounding errors will therefore not affect the result. Which phases that are used or not depends on which phase you “start” on, with many possible quality values for a given signal at a given (even) FCW.

Because of this a normal approach is to measure only one odd FCW, and the analysis of a signal is therefore drastically eased. The analysis in this thesis will use that technique.

enf and $e2$ are dependent of the output values for each phase, independent of the order, and are thus FCW independent (as long as FCW is odd).

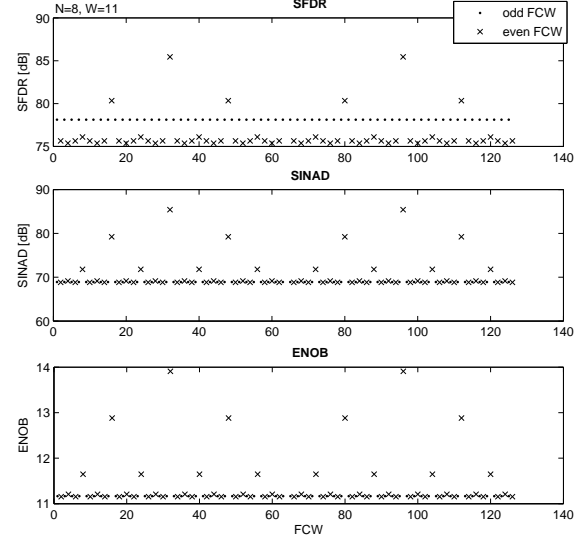


Figure 4.1. FCW effects on the SFDR, SINAD and ENOB

4.3 Rounding Noise Analysis

In this chapter rounding noise from operations will be discussed under the noise sections belonging to each method, and this section will only deal with method independent noise.

The rounding noise can be seen as white if W is fairly big, typically at least $N - 2$, and is therefore not sensitive to FCW. If W is less than $N - 2$, then some harmonics can occur, but those will never be bigger than the base tone divided by 2^W . In cases of decomposition, the limit may be $C - 2$ rather than $N - 2$, or some other limit.

4.3.1 Methods

Three rounding methods are covered, illustrated by an example where $W = 11$, which gives an integer range between -1023 and 1023 for the result after rounding. E.g. $y(\text{phase}) \approx 1023 \cdot \sin_0(\text{phase})$, where y is the output value from the DDS, phase is an angle (x_Q in the DDS), and \sin_0 is an approximation algorithm (the PSAC).

Method 1 describes the maximum amplitude solution.

$$y_0(\text{phase}) = \text{round}(1023.5 \cdot \sin_0(\text{phase})) \quad (4.3)$$

where $\text{round}(\dots)$ round toward closest integer.

Method 2 gives possibly less rounding errors when the phase is very close to $\pm 90^\circ$, but over all very similar noise analyzing results as Method 1.

$$y_0(\text{phase}) = \text{round}(1023 \cdot \sin_0(\text{phase})) \quad (4.4)$$

Method AWGN is a method to add some White Gaussian Noise with amplitude $\frac{1}{2} \cdot \text{LSB}$.

$$y_0(\text{phase}) = \text{round}(1023 \cdot \sin_0(\text{phase}) + \text{rnd} - 0.5) \quad (4.5)$$

where rnd is a (new) random value between 0 and 1 for each phase.

An example can illustrate the direct effect of the AWGN: If the amplitude y_0 for a phase should have been 511.75 before rounding, then it may be 511, but with 75 % probability it will be 512.

The effect on the signal, compared to method 2, is that the **SFDR** is increased because the probability rounding tends to counteract the continuous variations in the rounding errors that can occur when $W < N - 2$.

The **SINAD** is decreased, because we add some noise. **ENOB** and **einf** is 0.5 bits worse, because it can be rounded away up to 1 LSB. **e2** is increased with around 41 % (that means multiplied with $\approx \sqrt{2}$).

Due to all the negative aspects of *method AWGN*, and the notice that the only profit of it is not likely to happen often, that method is discarded. And because of this, the exact effect from decomposition will not be investigated.

The difference between method 1 and 2 is so small so only method 1 will be used (a bigger amplitude is always preferred).

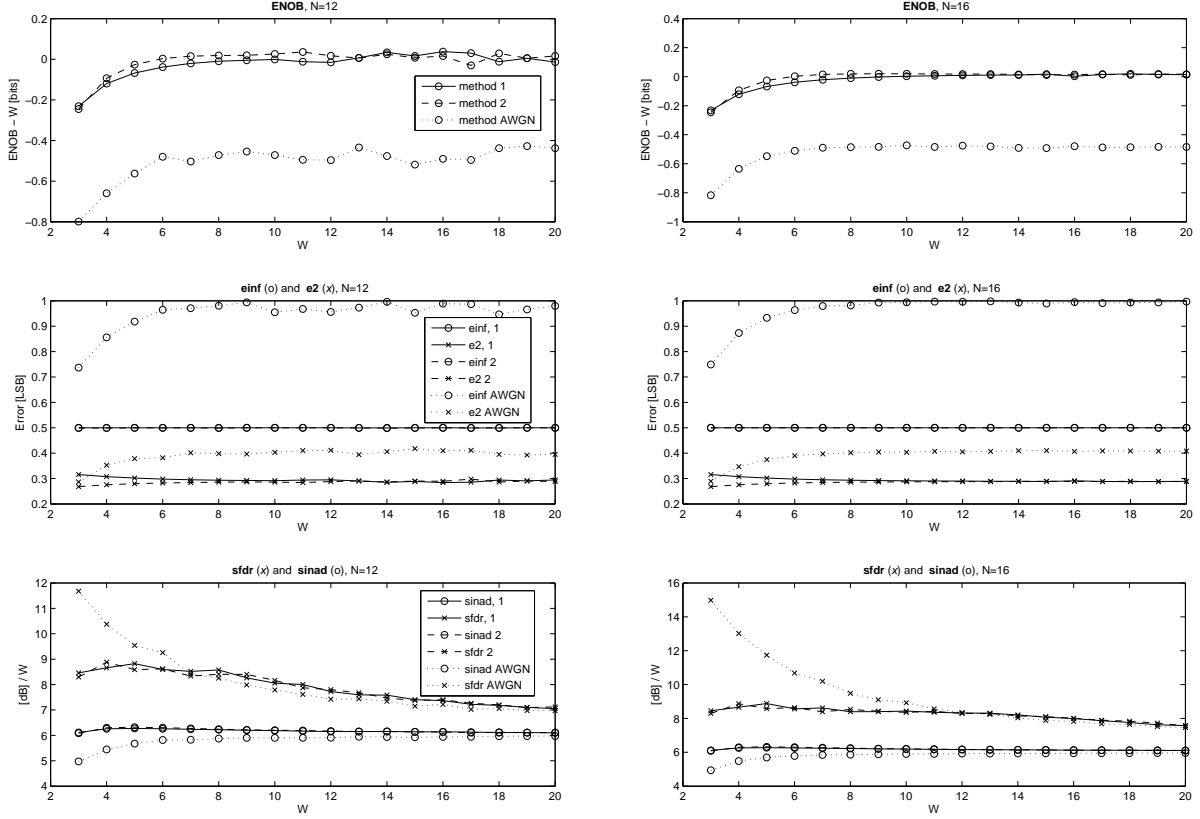


Figure 4.2. The ROM signal quality for the different rounding methods

Figure 4.2 illustrates the methods for $N=12$ and 16. You can see how *Method AWGN* (dotted lines) boosts the SFDR (cross marker in lower graphs) when W is small, but make all other quality units worse.

4.4 Algorithm Verification

This section discusses how the algorithms from chapter 2 works, and briefly what signal qualities that can be expected from the different configurations.

As defined chapter 2, the $phase = 90^\circ \cdot (x_Q + x)$, where $x_Q = 0, 1, 2$ or 3, and $0 \leq x < 1$. Due to the SURD⁷ implementation, this chapter will only illustrate the first quadrant, which means $x_Q = 0$.

For decomposition methods, that split the $N - 2$ bits in x into $C + F$ or $C + D + F$ bits, the (virtual) decimal point in x will be moved from the left to the right of x_C , making x_C to an integer, $0 \leq x_C < 2^C$, indexing all sub ranges in the first quartile, and x_F (or $x_D + x_F$) to the new fractional part.

4.4.1 ROM/Polynomial

In this chapter $x = x_C + x_F$, where x_C is the integer part $0 \dots (2^C - 1)$, and x_F is the fractional part $0 \leq x_F < 1$ with F bits precision, which will be used as the 'x' in the polynomials. Remember that there will be one

⁷Symmetry Using Range Divider

polynomial for each x_C , whose coefficients are stored in one (or several) ROMs/LUTs addressed with the x_C .

The number of coefficients to the polynomials (per range) is denoted K , which gives $K = \text{polynomial order} + 1$. The implementation uses Horner's scheme in order to save some multiplication, it calculates e.g. the polynomial $P(x_F) = a_0 + a_1x_F + a_2x_F^2 + a_3x_F^3$ as $((a_3x_F + a_2)x_F + a_1)x_F + a_0$, if $K = 4$.

Coefficient Evaluation Methods

There are a number of different methods for calculating the content of the ROM storing the polynomial coefficients, which affects the qualities of the result.

This thesis will investigate five methods, each of them is well known or inherits from a well known approximation method.

The graphs in figure 4.3 (on page 23) shows only the first quadrant, because the rest is just mirrored versions of this. For illustration reasons C is set to 1 in all examples, which gives 2 polynomials in the shown quadrant. Furthermore $N=10$, which gives $x_F = 7$ bits, or a resolution of $2^{-7} = 1/128$, or 128 points per polynomial. The average and maximum errors are mentioned in each graph, as well as the number of ROM bits used to store the actual solution. This is further discussed in the analysis section.

- The **TaylorLeft** approach approximates the sine with a Taylor polynomial around $\sin(x_C)$ in the range $[x_C, x_C + 1)$. This gives good approximation for small x_F , but bad when x_F is close to 1. In the analysis section there will be shown that this method is the worst method in all categories.
- The **TaylorMid** approach works like the TaylorLeft, but approximates the sine around $x_C + 0.5$, which increases the quality significantly compared to TaylorLeft.
- The **Chebyshev** approach equals the sine in K equality points distributed according to the Chebyshev zeros, to minimize the maximum error. The Chebyshev zeros in the range $(-1, 1)$ are the solution to $0 = \cos(K \cdot \arccos(x))$, which gives $x = \cos(\frac{\pi}{2K} \cdot \{1, 3, \dots, 2K - 1\})$. Because $0 \leq x < 1$, rather than -1 to 1, the Chebyshev points are just rescaled from $(-1, 1)$ to $(0, 1)$.

An exception is made for $K = 1$ (when the sine is nothing but a constant within each range), which should have given a Chebyshev point in $x_F = 0.5$, but instead takes the mean of the max and min sine values in the range. This way the approximation still minimize the maximum error for $K=1$.

- The **Interpolation** approach works like the Chebyshev, but the equality points are “stretched out” so the left- and rightmost points are placed in $x_F = 0$ and $x_F = 1 - 2^{-F}$ to ensure the result will be continuous between the integer ranges. When $K = 1$ the $x_F = 0.5$ point is used as equality point.
- The **LeastSquare** approach is a least square polynomial approximation, that minimizes the sum of the squares of the errors. When the Chebyshev approach tries to minimize the maximum error *inf*, this one minimizes the RMS error⁸ *e2*.

In figure 4.3 the different methods are illustrated. Some clarification about the graphs: The three plots for each method differs in the K value only, which is 1, 2 and 3, from left to right. The thin line is the real sine. The upper thick line of dots is the approximated sine, and the lower thick line of dots is the error = the approximated minus the real sine.

The two Taylor methods are mostly mathematical approximations that are “good” when x_F is “close to” 0 resp. 0.5. x_F is however uniformly distributed over $[0, 1)$, why those methods (especially TaylorLeft) are supposed to be worse in all quality measurements than the other three, who are better optimized for the entire range $[0, 1)$.

For these five methods there are however a security scaling, that is not plotted in figure 4.3. This scales down the coefficients if the sine exceeds 1 in any point, which saves the final hardware implementation the need of an overflow detection.

Some comments about different variants:

- The pure **ROM/LUT** solution is any of the methods with $K = 1$ and $F = 0$.
- The **truncation** solution is the TaylorLeft with $K = 1$ and $F > 0$ (see fig 4.3a, left graph).
- The **linear interpolation** is the Interpolation with $K = 2$ and $F > 0$ (see fig 4.3d, middle graph).

⁸root-mean-square error = square root of the mean of the |errors|²

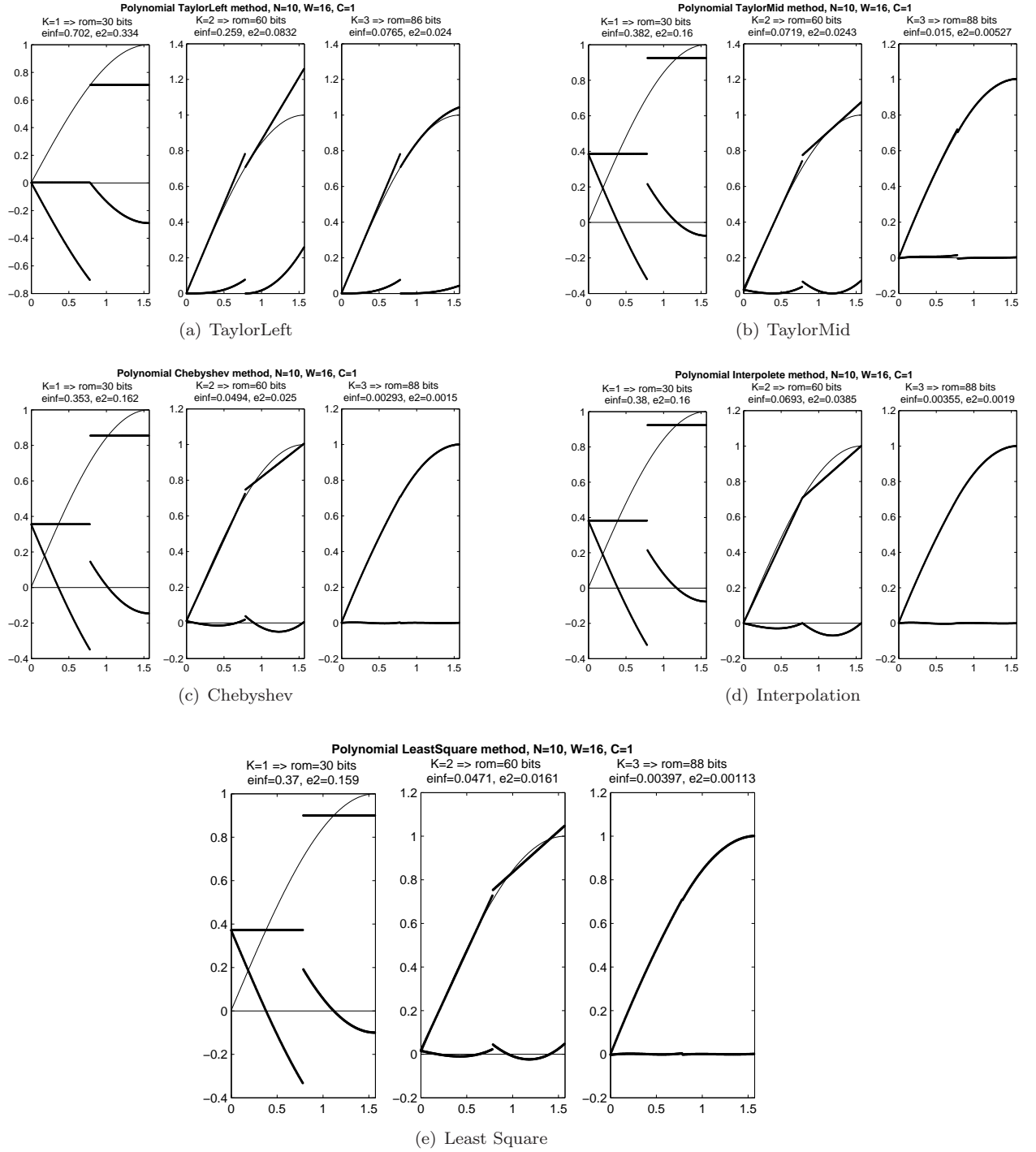


Figure 4.3. Illustrations of the different polynomial coefficients methods

4.4.2 Other Decomposition Solutions

After polynomial solution there are three mentioned methods left from the decomposition classification: Hutchinson's, Sunderland's and Curticăpean's. The Hutchinson's approach can be modified by changing the value in one of the ROMs, which gives a fourth approach.

Hutchinson's approach: $\sin(x) \approx \sin(x_C) + \sin(x_F) \cdot \cos(x_C)$, used without multiplication, since $\sin(x_F) \cdot \cos(x_C)$ is precalculated and stored in a ROM (same height but thinner than storing entire $\sin(x)$). This approximation requires that $2C > W + 1$ to "hide" the truncation errors in the rounding noise. See figure 2.6 (pg 9) for an illustration.

The alternative to Hutchinson's approach, in this project called "**Hutchinson's 2**", is to store $\sin(x) - \sin(x_C)$ rather than $\sin(x_F) \cdot \cos(x_C)$, which will give no truncation error at all, and in general no higher resource usage than Hutchinson's original approach, why the original will be discarded. This approach will be further investigated under the subject "Sine compression".

Sunderland's approach: $\sin(x) \approx \sin(x_C + x_D) + \cos(x_C) \cdot \sin(x_F)$. Store $\sin(x_C + x_D)$ and $\cos(x_C) \cdot \sin(x_F)$ in one ROM each, and add together. This reduces the size of the ROM quite markedly when comparing with Hutchinson's. The approximation requires that $2C + D > W$ to hide the truncation errors. Illustrated in figure 2.7.

Curticăpean's approach: $\sin(x) \approx \sin(x_C) + \sin(x_F) \cdot \cos(x_C)$, used with multiplication, since $\sin(x_F)$ and $\cos(x_C)$ are stored in one ROM each. This method gives the same truncation but slightly more rounding noise than Hutchinson's, but on the other hand reduces the ROM size markedly. This also needs $2C > W + 1$ to hide the truncation error. Illustrated in figure 2.8.

However; if $3C > W + 2$ then $\sin(x_F) = x_F$ (but scaled due to the adjusted phase unit), why this method turns to polynomial. Therefore, assume $3C \leq W + 2$ and $2C \leq W + 1$ in the analysis.

4.4.3 CORDIC

The CORDIC algorithm is a classical way to calculate the sine.

Advantage: Uses no multiplication. Only as many ROM words are needed as there are bits in the output. It is quadrature (generates both sine and cosine).

Disadvantage: Requires very much logic/low clock frequency, very low throughput, or very high latency, depending on how it is pipelined. Of course you can have a trade off between those.

Due to time limitation and low priority, this method will not be implemented.

4.4.4 Sine Compression

This method is very general. It can be combined with any of the methods mentioned earlier. It stores all truncation errors (but negated), and those are added to the approximated signal in the PSAC. This gives the effect that all truncation errors disappear, and only rounding noise is left. The cost is a large ROM with high $= 2^{N-2}$ rows, and as wide as it takes to store the truncation errors. Therefore there is only a need to look at the maximum error factor (*eof*) at the used method. For the polynomial case this means that the Chebyshev method will be used in most cases. Experiments show that Chebyshev and interpolation polynomials and the Sunderland's approach are the best methods.

The choice of approximation method and its parameters is called the *configuration* of the sine compression in this thesis.

Table 4.1 shows the resources needed for some configurations. The last field of the configuration is the method code (see table 4.3). The columns *ROM (appr)*, *ROM (corr)* and *ROM (tot)* are the ROM sizes used by the approximating method, the Correction ROM, and the sum of them respectively. The column *mults* shows the number of multipliers that is used.

When letting the sine compression function choose a configuration from given N and W only (without specifying the rest of the configuration), it minimizes the number of ROM bits used for methods using zero to three multipliers. See table 4.2 for some examples. Polynomial configurations will normally use two ROMs while Sunderland's will use three.

See the Appendix B for more tables showing resource usages.

4.4.5 Method Codes

The final product will use a number of codes for the methods and their sub groups, as shown in table 4.3.

For instance *cpi* is a polynomial method with the interpolation coefficients approach, and a correction ROM on that.

Configuration	ROM (appr)	ROM (corr)	ROM (tot)	mults
N=10, W=11, F=1, K=1, pls	1.25 kbits	1 kbits	2.25 kbits	0
N=10, W=11, F=2, K=1, pls	640 bits	1.25 kbits	1.88 kbits	0
N=10, W=11, F=4, K=1, pls	160 bits	1.75 kbits	1.91 kbits	0
N=10, W=11, F=2, K=2, pls	960 bits	512 bits	1.44 kbits	1
N=10, W=11, F=4, K=2, pls	272 bits	768 bits	1.02 kbits	1
N=10, W=11, F=2, K=3, pls	960 bits	768 bits	1.69 kbits	1
N=10, W=11, F=4, K=3, pls	320 bits	768 bits	1.06 kbits	2
N= 9, W=10, F=4, K=2, pc	128 bits	384 bits	512 bits	1
N= 9, W=12, F=4, K=2, pc	160 bits	512 bits	672 bits	1
N=11, W=10, F=4, K=2, pc	448 bits	1.5 kbits	1.94 kbits	1
N=11, W=12, F=4, K=2, pc	576 bits	1 kbits	1.56 kbits	1
N=11, W=10, F=6, K=2, pc	128 bits	1.5 kbits	1.62 kbits	1
N=11, W=12, F=6, K=2, pc	160 bits	2 kbits	2.16 kbits	1
N=10, W=11, F=3, C=4, s	1.06 kbits	1 kbits	2.06 kbits	0
N=10, W=11, F=3, C=3, s	704 bits	1.25 kbits	1.94 kbits	0
N=10, W=11, F=4, C=3, s	1.03 kbits	1.25 kbits	2.28 kbits	0

Table 4.1. Some examples of `sine_compression` resources

N	W	0 mults	1 mults	2 mults	3 mults
10	10	$F=2, C=3, s$ ROMs = 1.44 kbits	$F=4, K=2, pc$ ROMs = 752 bits	$F=6, K=3, pc$ ROMs = 604 bits	$F=7, K=4, pc$ ROMs = 574 bits
10	16	$F=3, K=1, pc$ ROMs = 3.22 kbits	$F=3, K=2, pc$ ROMs = 1.81 kbits	$F=5, K=3, pc$ ROMs = 1.05 kbits	$F=5, K=4, pc$ ROMs = 864 bits
10	20	$F=4, K=1, pc$ ROMs = 4.3 kbits	$F=4, K=2, pc$ ROMs = 3.05 kbits	$F=4, K=3, pc$ ROMs = 1.48 kbits	$F=5, K=4, pls$ ROMs = 992 bits
16	10	$F=6, C=4, s$ ROMs = 36.2 kbits	$F=10, K=2, pc$ ROMs = 32.2 kbits	$F=12, K=3, pls$ ROMs = 32.1 kbits	$F=14, K=4, pc$ ROMs = 48 kbits
16	16	$F=4, C=5, s$ ROMs = 66 kbits	$F=5, K=2, pc$ ROMs = 43 kbits	$F=11, K=3, pc$ ROMs = 48.3 kbits	$F=12, K=4, pc$ ROMs = 48.2 kbits
16	20	$F=3, C=8, s$ ROMs = 104 kbits	$F=6, K=2, pc$ ROMs = 55.8 kbits	$F=9, K=3, pc$ ROMs = 49.4 kbits	$F=11, K=4, pc$ ROMs = 48.5 kbits
20	10	$F=9, K=1, pc$ ROMs = 517 kbits	$F=14, K=2, pc$ ROMs = 512 kbits	$F=16, K=3, pls$ ROMs = 512 kbits	$F=18, K=4, pc$ ROMs = 768 kbits
20	16	$F=6, C=6, s$ ROMs = 588 kbits	$F=9, K=2, pc$ ROMs = 523 kbits	$F=15, K=3, pc$ ROMs = 768 kbits	$F=16, K=4, pc$ ROMs = 768 kbits
20	20	$F=5, C=9, s$ ROMs = 776 kbits	$F=7, K=2, pc$ ROMs = 568 kbits	$F=13, K=3, pc$ ROMs = 769 kbits	$F=15, K=4, pc$ ROMs = 768 kbits

Table 4.2. Sine compression optimized for some N and W

Code	Method	VHDL
p	Any kind of polynomial	
ptl	Polynomial with Taylor Left coefficients	
ptm	Polynomial with Taylor Mid coefficients	
pc	Polynomial with Chebyshev coefficients	✓
pi	Polynomial with Interpolation coefficients	✓
pls	Polynomial with Least Square coefficients	✓
s	Sunderland	✓
c#	Sine Compression. # = any of the methods above	✓

Note: The methods marked with a “✓” in the column “VHDL” are implemented as VHDL generators.

Table 4.3. The method codes for modelled methods

All methods mentioned in chapter 2, *Methods*, are not listed here. Only those that has been modelled are listed.

4.5 Truncation Noise Analysis

Most of the truncation (algorithmic) errors will appear as an overtone to the base frequency, due to the systematic pattern in the error – if a point is too low then its neighbors are probably also too low, which typically gives

a harmonic. However, errors that occur in only one or perhaps two neighboring phases will not be identified as a real overtone, because it will occur only once (or twice) every 2^N times, why those will be thought of as white noise.

4.5.1 Polynomial

Figure 4.4 illustrates the quality measurements when modeling the methods for $C = 4$ and 10, and with rather big F and W to filter out all but the truncation effects. This gives a huge error for small K , and a big improvement when K increases. Note the rather strange scales in the figures, especially the error graphs.

Some test where different C , F and K has been tested has resulted in the conclusion as follows (See appendix B for tables with results).

One limitation in all coefficient assignments is the double floating point precision used by Matlab, causing an upper limit of about 50 bits. This affects the *LeastSquare* method most and is clearly visible in figure 4.4(b), the ' \triangleright ' marker where $K = 4$.

- The **TaylorLeft** approach ('+' marker in the figures) gives of course very bad qualities. According to the tests the TaylorLeft has the worst qualities of the five methods. In figure 4.4 this is clearly visible, low ENOB/SFDR/SINAD and high error. This method is therefore discarded.
- The **TaylorMid** approach ('x' marker in the figures) are (as expected) better than TaylorLeft, but not as good as the three later methods (at least for $K > 2$). This method is also discarded.
- The **Chebyshev** approach (' \triangleleft ' marker in the figures) have a rather low maximum error (continuous line to the right in fig 4.4a and b). The Chebyshev approach to minimize the max error works as best when C and/or K is big.
- The **Interpolation** approach (' \diamond ' marker in the figures) works like the Chebyshev, but distributes the equality points different.
- The **LeastSquare** approach (' \triangleright ' marker in the figures) minimizes the $e2$ factor.

Some examples of frequency spectrum's derived from truncation errors are shown in figure 4.5. The graphs shows the configuration 'method=pi, C=4', for $N = 14$, W is set to "big" to remove the rounding errors. K is set to 1, 2 and 4 in the graphs. FCW is set to 1489 in this case. All three plots have frequency components around -350 dB and below, but those are derived from rounding errors rather than truncation errors, and have therefore been removed in the graphs. The main noise frequency is harmonics number $\{1, 2, 3, \dots\} \cdot 2^{2+C} \pm 1 = \{63, 65, 127, 129, 191, 193, \dots\}$ to the main tone, with decreasing amplitude.

Figure 4.6 show the SFDR, SINAD and ROM result for different C values, $N = 16$, $W = 20$, $K = \{1, 2, 3\}$, and Chebyshev coefficients. You can clearly see the W effects on the SFDR. The upper graph to the right shows number of kBits ROM used, and the lower graph to the right the same thing, but rescaled y-axis.

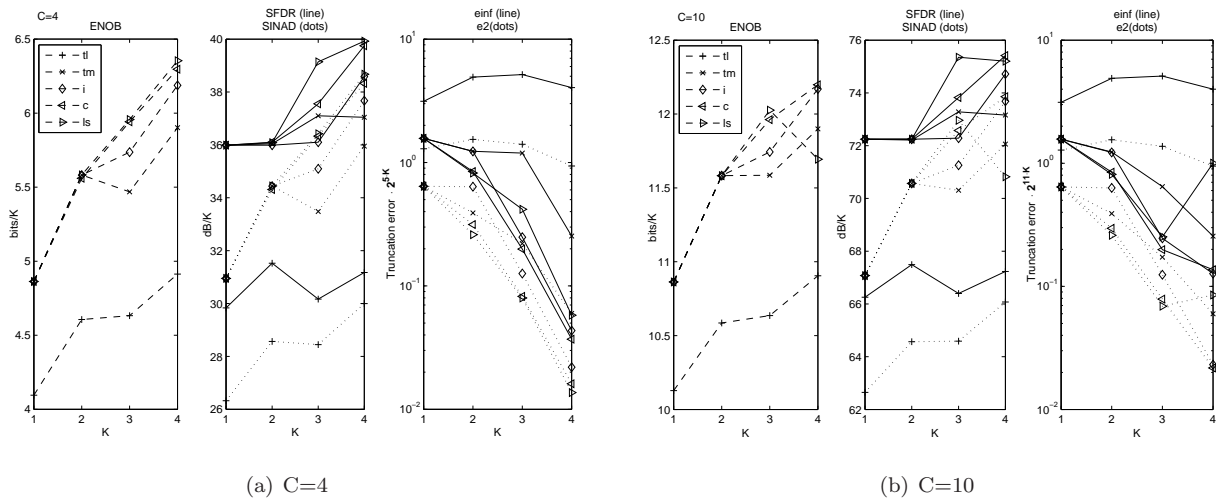


Figure 4.4. Example of different polynomial qualities

See Appendix B, Analysis, for tables with values.

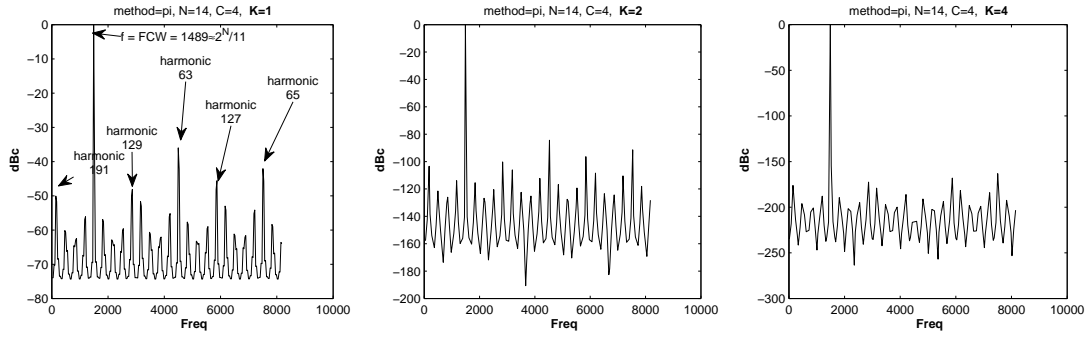


Figure 4.5. Examples of truncation derived frequency spectrum's for polynomial interpolation, where $K=1, 2$ and 4

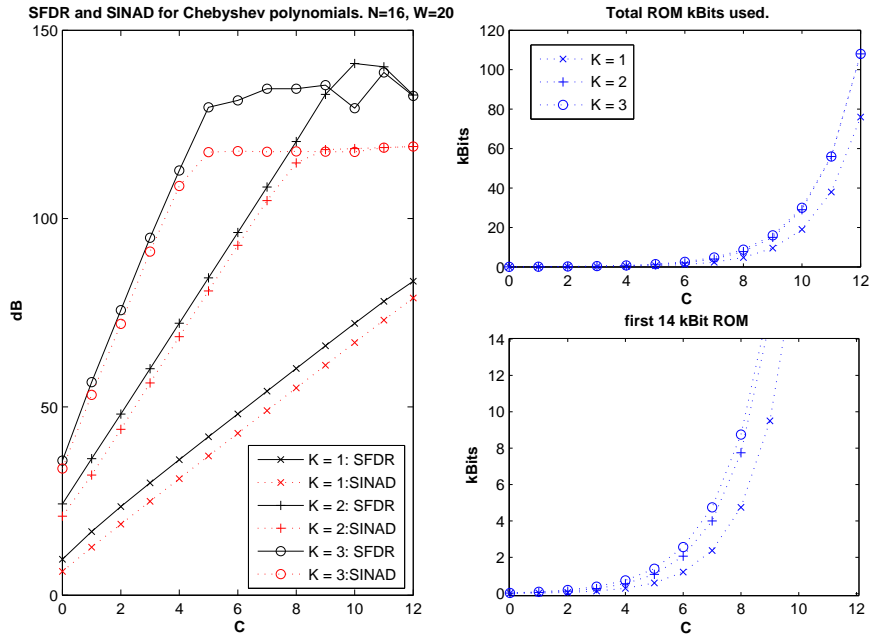


Figure 4.6. SFDR/SINAD and ROM usage for Chebyshev polynomials with different C

4.5.2 Other Decomposition Solutions

The four solutions Hutchinson's, Hutchinson's 2, Sunderland's and Curticăpean's differs in results.

- The **Hutchinson's** has no advantages at all over Hutchinson's 2, and is therefore discarded.
- The **Hutchinson's 2** is in fact a sine compression method, because Hutchinson's first ROM is the TaylorLeft polynomial with $K=1$, and the second ROM is a correction to that. The analysis of the sine compression shows that the Taylor Left is not efficient as approximation method, why Hutchinson's 2 are also discarded.
- The **Curticăpean's** is a TaylorLeft polynomial if $3C \geq W + 2$, why this analysis will assume that $3C \leq W + 1$ (then the truncation errors are not hidden in the rounding noise). Within this range the method may be slightly better than the TaylorLeft polynomial. However, according to equations 4.6 and 4.7 (where dy denotes the errors), the TaylorMid should always have around 4 times better (smaller) maximum truncation error than Curticăpean.

Curticăpean:

$$\begin{aligned}
 y_1 &= \sin(x_C) + \cos(x_C) \cdot \sin(x_F). \\
 |dy_1| &= \sin(x_C) \cdot (1 - \cos(x_F)) \approx \sin(x_C) \cdot \frac{x_F^2}{2} \leq \frac{x_F^2}{2}; 0 \leq x_F < \frac{\pi/2}{2^C} \\
 |dy_1| &\lesssim \frac{\pi^2}{8 \cdot 2^{2 \cdot C}}
 \end{aligned} \tag{4.6}$$

Polynomial/TaylorMid:

$$\begin{aligned}
 y_2 &= \sin(x_C) + x_F \cdot \cos(x_C) \\
 |dy_2| &\approx x_F^2 \frac{\sin(x_C)}{2} \leq \frac{x_F^2}{2}; -\frac{\pi/2}{2 \cdot 2^C} \leq x_F < \frac{\pi/2}{2 \cdot 2^C} \\
 |dy_2| &\lesssim 0.5 \frac{\pi^2/4}{2^{2C}} = \frac{\pi^2}{32 \cdot 2^{2C}}
 \end{aligned} \tag{4.7}$$

A similar calculation can be done to show the factor 4 for average truncation error. The ENOB, SINAD and SFDR have their sources in the errors, why they should follow and be better for the TaylorMid. Therefore Curticăpean's method is discarded.

- The **Sunderland's** method is quite good for not using any multiplication. Due to the extra parameter ($D = N - 2 - D - F$) the method is flexible, and if F and C are set right there is a good balance between the number of ROM bits and the quality.

In figure 4.7 the SFDR, SINAD and ROM usage are plotted when $N = 16$, $W = 27$ and C is swept from 0 to 12. Three different alternatives for D and F are shown. If $D = 0$ the Sunderland turns out to be the discarded Hutchinson's approach, why $D = 1$ is one of the alternatives. If $F = 0$, we get a pure ROM solution, why $F = 1$ is another alternative. The third alternative is the "middle", where $D = F = \frac{N-2-C}{2}$, or $D = F + 1$ when $N - C - 2$ is odd. $W = 27$ is the least output width that will not affect the SINAD. A lower W will save a lot of memory, but reduce the signal quality in cases where C is close to $N-2$. Notable is how the $D = F$ alternative saves ROM compared to the other alternatives.

In figure 4.8 some examples of frequency spectrum derived from truncation errors are shown. The graphs shows the configuration $N = 10$, method=Sunderland's, the D and F are set according to the ROM saving alternative in figure 4.7 ($D = F$). Just like figure 4.5, W is set to "big" to separate and then remove the rounding errors.

4.5.3 Sine Compression

According to statistical testings, the mostly used methods are Chebyshev and Sunderland's, and a few times also Least Square.

See Appendix B, Analysis, for tables showing more results from Sine Compression.

4.6 Conclusion

The following methods are generated:

1. Polynomial, K=1..4
 - (a) Interpolation coefficients.
 - (b) Least Square coefficients.
 - (c) Chebyshev coefficients.
2. Sunderland's method.
3. Sine compression – this will however be implemented as an add-on to the other methods.

The rest of the methods are discarded.

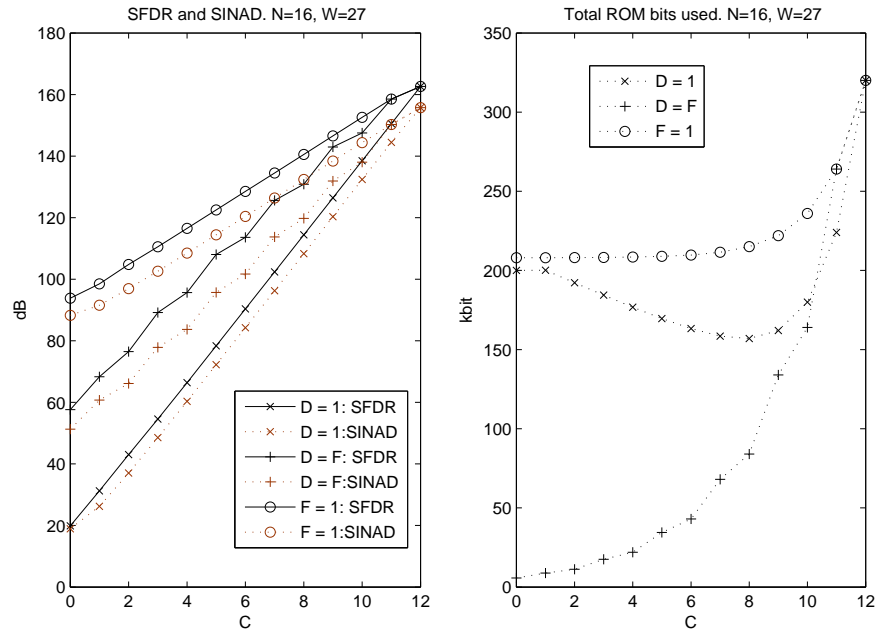


Figure 4.7. SFDR/SINAD and ROM usage for Sunderland's with different C, D and F distributions

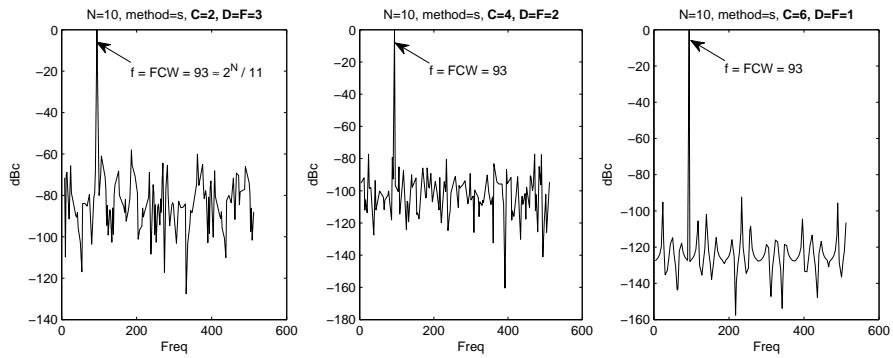


Figure 4.8. Example of truncation derived frequency spectrum for Sunderland's, where C=2, 4 and 6

Chapter 5

Implementations

This chapter will describe the structure of the VHDL modules that are generated. It will also briefly describe the matlab files that generate them.

5.1 ROM

An essential part of the PSAC implementations is the ROMs. In the VHDL solutions those are implemented as arrays of constants, leaving to the compiler to select which ROM(s) to use.

5.1.1 The Function `create_rom`

To simplify the generation of ROMs to the VHDL, there is a matlab file for this, called `create_rom`.

Here are some key properties of `create_rom`:

- It generates VHDL code for a ROM.
- The result can be put to screen, to an own file or to an existing file pointer.
- The module can split output into several fields (used by the polynomial).
- The module can have synchronous or asynchronous output.
- The module can use `std_logic_vector` or `unsigned` as interface vector types.
- It can insert attributes into the entity.

5.1.2 The VHDL Implementation

The main structure of the VHDL file is a bit matrix with ROM data, where the rows are indexed by the address. The result is optionally synchronized in a process, and if many fields are used, the data is also split into those.

Timing Problem

One possible problem with this implementation is that many of the ROMs in the FPGAs have synchronous address inputs. This gives a rather small t_{SU} ¹, which is good, but it also gives a long t_{CO} ², which may lower the maximum clock frequency.

In general, the ROMs will be placed in dedicated ROM blocks, but if they are implemented as LUTs (typically when there are just a few address bits), then the synchronization will be placed at the output rather than the input. This way there will be a bigger t_{SU} , and a very low t_{CO} .

¹ t_{SU} = SetUp time, minimum required time with “stable inputs before clock flank”

² t_{CO} = Guaranteed maximum time from “clock flank to stable output”

5.2 SURD Implementation

The SURD (Symmetry Using Range Divider) reduces the input range from $[0, 360^\circ)$ to $[0, 90^\circ)$. In the implementation this is done in two steps: PreSURD and PostSURD. The PreSURD modifies the x input, so the resulting range will be $[0, 90^\circ)$ or $[180, 270^\circ)$ (first and third quadrant). These ranges are treated exactly equally, and the psac algorithm calculates the first quadrant. The PostSURD handles the difference between $[0, 90^\circ)$ and $[180, 270^\circ)$.

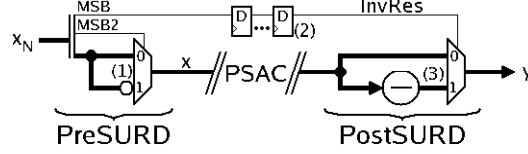


Figure 5.1. An RTL schematic of SURD

Figure 5.1 illustrate the SURD function. The “MSB2” signal in the schematic is the second most significant bit of x_N . The mux (1) inverts the phase (if necessary) and the mux (3) inverts the result if necessary. Compare with figure 2.2 on page 6. The DFFs (2) delays the signal a number of clock cycles – 1 for Sunderland’s and K for polynomial – compare to the RTL schematic examples for each method.

The PreSURD will increase the PSACs t_{SU} with roughly the time for one LUT operation. This may be a problem if the ROMs or other connected components have big t_{SU} or if the setup time is critical.

The PostSURD will increase the t_{CO} with roughly the time for one $W - 1$ bits addition. This may be a problem for Sunderland’s, that already have a big t_{CO} . Polynomials have registered outputs (if no correction ROM is used) and are therefore less critical.

5.3 Polynomials

The polynomial solution uses - as mentioned earlier - Horner’s scheme for calculations, so that $y_i = ((d_i x_F + c_i) x_F + b_i) x_F + a_i$ when $K = 4$ and $i = x_C$, for the calculations. This will minimize the number of multiplications used.

Due to trigonometric effects, a_i and b_i are ≥ 0 , while c_i and d_i are ≤ 0 . The VHDL implementation uses however the unsigned data type in the calculations. To solve this the c and d coefficients are negated (if $K \geq 3$), resulting in the following formula:

$$\begin{aligned} y_i &= ((d_i x_F + c_i) x_F \cdot (-1) + b_i) x_F + a_i \\ &= a_i + x_F (b_i - x_F (c_i + x_F d_i)) \end{aligned} \quad (5.1)$$

The first line shows the implemented order of calculation, the second line shows a simplified variant.

If $K = 1$ then coefficient a_i is used and the rest are zero. If $K = 2$ then b_i is also used, and so on.

5.3.1 The Function psac_polynomial_rom

The coefficients a_i , b_i and so on are created in the function `psac_polynomial_rom`, according to the different polynomial configurations - the main parameters are N , W , F , K and *method*. This function is used by the `psac_polynomial` and `create_polynomial` functions.

Beside the coefficient vector, this function calculates how many bits wide the coefficient fields should be, how big the partial sums are, and also detect and correct possible overflows (where the result will not fit into the W bits). In order to calculate this, the resulting sine approximation must be calculated for the first quadrant. That sine is returned as a byproduct.

5.3.2 The Function psac_polynomial

The modelling of the polynomial is done in the function `psac_polynomial`, which allows the user to select the different configurations. Due to the sine approximation byproduct from the `psac_polynomial_rom`, this method does nothing but calculate the SURD.

This function returns, in addition to the approximated sine vector, a list of the sizes of the multipliers and ROMs that are used.

5.3.3 The Function create_polynomial

The VHDL implementation is created in the function `create_polynomial`. This takes for instance a psac configuration and some output settings arguments, and write the VHDL code.

One feature is to choose whether to express small multiplications as multiplications or as shift-add operations. If the later is chosen, the multiplication operator will be overridden/redefined so it compares the arguments size with a given limit, and either use the built in multiplier or the shift-add structure. This feature is usually not necessary, because the compilers may do this themselves when needed.

The `create_polynomial` function can also add a correction ROM to the solution, to implement the sine compression.

The ROM(s) that are used are put as a private module in the same VHDL file as the PSAC.

This function returns the expected result and which latency that was used (with high C and low W, the need of high grade is reduced, and extra grades only results in coefficients = 0).

5.3.4 The VHDL Implementation

The solution calculates one sample per clock cycle, and is pipelined so each multiplication has one pipeline stage each, according to figure 5.2, which illustrates the solution when $K = 3$. The ROM is synchronous, why that will add another pipeline stage. This way, the polynomial solution has a latency of K clock cycles. Because all coefficients are stored on the same "rows" in the ROM, all those are fetched at the same time. The ROM values must be shifted through the pipeline stages until they are used.

In the case a correction ROM is used, the correction is added at the end, just before the PostSURD operation, without using any extra pipeline stage (gray path in figure 5.2).

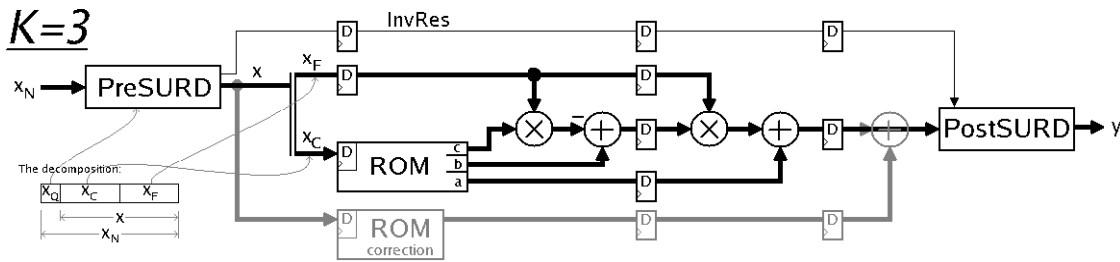


Figure 5.2. Example of an RTL schematic for a polynomial where $K=3$

Timings

As mentioned in the ROM section above, the ROMs will in most cases have synchronous inputs, and therefore not synchronous outputs (as plotted in fig. 5.2). This will be a big drawback for the polynomial timings, because the data is asynchronously fed to the multipliers that delay it even more. Finally the multiplier output is added to another coefficient before it is synchronized again. In total this gives very long delays, which will result in a big limitation of the maximum clock frequency.

Because all PSACs inputs are fed directly to the input to registers (except the PreSURD and the possibly non-asynchronous input to the ROMs), the t_{SU} is not more critical than discussed earlier.

The t_{CO} is dependent on PostSURD and on an eventual correction addition, and should not be worse than one addition.

5.4 Sunderland's

Sunderland's solution is very simple from an implementation point of view. It is easy to calculate the ROM contents, as long as you use the original Sunderland method. Nicholas *et. al.*[4] suggested other and harder ways of compute them, that gives better signal quality. The ROM contents are all positive, which is very convenient in the VHDL implementation.

As mentioned in earlier chapters, the Sunderland method splits the input into three parts (in addition to the initial x_Q for the SURD), that is x_C , x_D and x_F with widths C, D and F respectively. In difference to the polynomial solution, where x_C was the integer part and x_F the fraction part of the angle, all three input parts are treated as integers in Sunderland.

Two more variables are introduced; x_{CD} and x_{CF} , which is the bit field x_C concatenated with x_D and x_F respectively. These two variables are used to index the ROMs, to get the values res_{CD} and res_{CF} , which are added together.

5.4.1 The Function `psac_sunderland_rom`

The values res_{CD} and res_{CF} are created in the function `psac_sunderland_rom`, according to the parameters N , W , C and F (the D field is calculated from N , C and F). This method is used by the `psac_sunderland` and `create_sunderland` functions.

Beside the coefficient vector, this function calculates how many bits wide the ROMs should be, and also detects and corrects possible overflows (where the result will not fit into the W resulting bits). In order to calculate this, the resulting sine approximation must be calculated for the first quadrant. Therefore this result is returned as a byproduct.

5.4.2 The Function `psac_sunderland`

The modelling of the Sunderland method is done in the function `psac_sunderland`, which allows the user to select the different configurations. Due to the sine approximation byproduct from the `psac_sunderland_rom`, this method does nothing but calculate the SURD.

This function returns, in addition to the approximated signal, a list of the ROMs sizes that are used.

5.4.3 The Function `create_sunderland`

The VHDL implementation is created in the function `create_sunderland`. This takes for instance a `psac` configuration and some output settings arguments, and writes the VHDL code.

The `create_sunderland` function can also add a correction ROM to the solution, to implement the sine compression.

The ROMs that are used are put as private modules in the same VHDL file as the PSAC.

5.4.4 The VHDL Implementation

The Sunderland's has only one clock cycle latency (the registers are located in the synchronous ROMs). In difference to the polynomial implementation, this method uses two ROMs for the approximation, named `romCD` and `romCF`. If there is a correction ROM that is used in the same way. The ROM values are added together and fed to the `postSURD` part.

Figure 5.3 shows the Sunderland schematic, and just like in the polynomial figure, the ROM has registered address inputs. A correction ROM is added in gray in the figure.

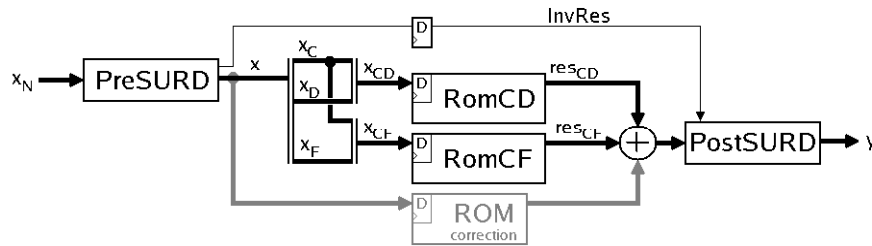


Figure 5.3. Sunderland RTL schematic

Timings

Because there are only one pipeline stage in Sunderland, there are no register-to-register path, and thus no strict upper bound on the frequency. The inputs from the PreSURD is fed to the ROMs (which will most probably be ROM blocks rather than LUTs), why the t_{SU} is rather short.

The output from the ROMs are delayed by the ROMs, then the two ROM outputs are added together, plus the possible correction value, and the result is finally fed to the PostSURD. This gives a very long t_{CO} , roughly t_{CO} for the ROMs plus two adder delays.

5.5 Test Bench

In order to test the solutions there is a test bench generator. The main feature is that the testbench creates a Matlab function file with the test result, or a function file that reads the data from a data file, that is produced as well. The data output file (within the function file or as its own data file) has no correction for the skew in the latency. The function file does however adjust for this, and returns the adjusted simulation result. The function file has also the ability to plot the result and the noise in the signal.

5.5.1 The Function `create_testbench`

The function `create_testbench` creates the testbench in the desired way. The user gives the N and W parameters, the latency and the entity name of the psac. The user can also choose output destination (folder/file/file pointer). Other possible settings are incrementation step (the FCW) and how the matlab function file/data file should be generated.

5.5.2 The VHDL Solution

The target for the testbench VHDL implementation is a simulator, why there is no problem with timings and so on.

The built in clock generator in the test bench runs the simulation clock in 500 MHz³. An automated test system (see section `test_psac` below) will run the simulation for 1 (simulated) second, which is far too much (as long as $N \leq 28$). Because of this, the test bench stops the clock generator when it is done.

The VHDL process that handles the phase accumulator (x_N), also reads the result and write the phase and result to the result file.

5.6 Automatic Generation/Verification

In order to simplify the process, there is a create-simulate-verify-analyze function, that takes a psac configuration and some other settings, and generates/verifies the psac.

5.6.1 The Function `test_psac`

This function is named `test_psac`, and can be seen as the main method for producing the PSACs.

The following actions are typically taken when running `test_psac`:

1. Generate the **psac**.
2. Generate the **testbench**.
3. Generate the simulation **do file** (it is like an instruction file for the simulator).
4. Create a simulation **work Library**, if that does not already exist.
5. Start **ModelSim**, and tell it to execute the **do file**:
 - Compile the psac and testbench.
 - Simulate the testbench for one second (this will generate a matlab and a result file).
 - Exit ModelSim.
6. Execute the produced matlab file, that **reads** the result data file, corrects the latency skew and plots the result.
7. **Verify** the result by comparing it to the expected values. Add some signals to the plot.

Most of the parts in the list above can be deactivated or controlled from the arguments.

`test_psac` can also produce a status to a given file pointer, which is suitable when running scripted tests.

`test_psac` returns a matrix with the tested phases and their results from the simulation. It also returns the carrier.

The output from `test_psac` is (may vary if some parts are disabled):

³The time in the simulation differs from the real time

- psac.vhdl, the psac.
- test_psac.vhdl, the testbench.
- run_psac.do, the ModelSim do-file.
- test_psac_res.m, the resulting function file.
- test_psac_res.txt, the test result data file.
- work, the VHDL library folder

Note: The name “psac” in this list can be changed to anything else.

A part from the test_psac documentation describes it's arguments:

```
[vector,carrier] = test_psac(N, W, config, ename, path, flag1, flag2, ...)
[vector,carrier] = test_psac(N, config, ename, path, flag1, flag2, ...)
[vector,carrier] = test_psac(config, ename, path, flag1, flag2, ...)
Generate and test VHDL code for a psac polynomial implementation of the PSAC
N = optional phase bit width (0:2^N = 0:360 degree).
W = optional output bit width (signed)
config = psac config and its parameters: 'method=?? F=?? C=?? ...'
* method: any of
* * 'ptl' - Using Taylor coefficients from the left of the range
* * 'ptm' - Using Taylor coefficients from the middle of the range
* * 'pi' - Interpolation
* * 'pc' - Minimize maximal error with Chebyshev
* * 'pls' - Least Square Method
* * 's' - Sunderlands
* * 'c*' - Sine compression with *=any of above methods.
* F: number of fine bits = 0..N-2.
* C: number of coarse bits = 0..N-2.
* K: number of coefficients for polynomial solution = grade + 1.
* W: Will overwrite the argument W, if both are given
* N: Will overwrite the argument N, if both are given
ename = psac entity name.
path = optional path where solution will be placed as string.
flags = optional flags as strings:
* 'leavePsac' - do not generate a new Psac
* 'leaveTB' - do not generate a new testbench
* 'leavePsacTB' - neither generate a new Psac nor testbench
* 'leaveSim' - Do not simulate or produce a do-file
* 'leaveRes' - Do not try to run the result file
* 'quiet' - Do not write status
* 'noPlot' - Do not plot anything
* 'TBinc=<N>' - increase phase in testbench with <N> rather than 1 each cycle.
* 'logfp=<fp>' - <fp> is a file pointer to a status log file.
* 'logmsg=<msg>' - <msg> is a message to put into the status log file
* 'minMultW=<N>' - all multipliers with LESS THAN <N> bits in one of the arguments
                    will be replaced by additions. Standard = 1 = disabled

Return values:
* vector(:, 1) = testbenchs input to PSAC.
* vector(:, 2) = output from the PSAC.
* carrier = main tone, all other frequencies ignored.
```


Chapter 6

Suggester

This thesis has two main tasks. The first task is to generate PSAC implementations according to the users specification. The second task is to suggest a suitable specification, according to the users preferences and the selected FPGA. The second task is done in the function `psac_suggest`.

The function takes some preferences from user, and suggests one or more implementation configurations in a format that `test_psac` reads.

`test_psac` takes as argument:

- `N` - The N parameter as an integer.
- `W` - The W parameter as either an integer, a vector of possible values, or a limit/cost.
- Some other properties that describe e.g. required SFDR, device or anything else.

6.1 The Properties

The properties are given as a property name followed by its value as the next argument. The property names are case insensitive. See table 6.1 for a list of available properties.

Example: “`psac_suggest(16, 10, 'sfdr', 100)`” requires a psac with 16 bits phase, (at least) 10 bits output and SFDR ≥ 100 dB. This will probably result in a solution with $W \geq 13$, because it is not possible to get 100 dB SFDR with 10 bits W (see figure 4.2 on page 21).

Name	Value	Units	Default	Meaning	Notes	Description
'Device'	dev			Optimize solution for Xilinx or Alteras FPGA structure. <i>dev</i> = device family codes, e.g. “cy2”, see tables 3.2 and 3.4.		
'SFDR'	lim/w	dBc	0/1	$SFDR \geq \lim$ $Cost = -w \cdot SFDR$	1	Set a limit/weight on the SFDR.
'SINAD'	lim/w	dBc	0/1	$SINAD \geq \lim$ $Cost = -w \cdot SINAD$	1	Set a limit/weight on the SINAD.
'ENOB'	lim/w	#bits	3/1	$ENOB \geq \lim$ $Cost = -w \cdot ENOB$	1	Set a limit/weight on the ENOB.
'e2'	lim/w	LSB	/1	$\log_2(e2) \leq \lim$ $Cost = w \cdot \log_2(e2)$	1	Set a limit/weight on the RMS-error.
'einf'	lim/w	LSB	/1	$\log_2(einf) \leq \lim$ $Cost = w \cdot \log_2(einf)$	1, 2	Set a limit/weight on the max-error. Unit = LSB.
'me2', 'meinf'		MSB		Same as e2 and einf, but errors are measured in MSB.		
'ROMs'	lim/w	#ROMs	/1	$\#ROMs \leq \lim$ $Cost = w \cdot \#ROMs$	1	Set a limit/weight on the number of ROMs to be used.
'RomUnit'	x	kbits	1		3	Set the ROM block size to x kbits.
'Mults'	lim/w	Mults	/1	$\#Mults \leq \lim$ $Cost = w \cdot \#Mults$	1	Set a limit/weight on the mults to be used. Unit = number of multipliers.
'Multsize'	width x height	bits	18x18			Set the multipliers width and height in number of bits.
'Lat'	lim/w	clock cycles	/0	$Latency \leq \lim$ $Cost = w \cdot Latency$	1	Set a limit/weight on the latency in the PSAC.
'ListSize'	'disp/tot'		0/256	The size of the list to display/store. The more to store (tot), the likelier to find real optimum but the longer time it takes. If $disp > 0$ the <i>disp</i> best entries in the list is written to the screen.		

Table 6.1. Available properties for the `psac_suggest` function

Notations from the column *notes*:

- 1: "lim/w" means an optional limit followed by an optional weight, according to any of these four syntaxes: {lim, 'lim', '/w', 'lim/w'}, where lim and w are replaced by values.
- 2: Hint: Set `einf = 0.5` to get 'exact' output quality.
- 3: The ROM unit size can have one of the three syntaxes: {*x*, '*x*', '*p**x*'}, where the "p" adds a parity bit per byte. E.g. '4' defines 4096 bits large ROMs, while 'p4' defines $4096 + 512 = 4608$ bits.

6.2 Cost Model

To decide how good or bad an implementation is, all quality and resource metrics have their own costs. In order to compare quality and resource factors there is a need for a common cost unit. By natural reasons it should be a resource measurement unit. A natural choice is the LAB (for Altera) or CLB (for Xilinx), but those are quite different, and it is better to have one common unit for both vendors. Some more or less common measurement units are discussed in table 6.2.

Unit	Notes	Prospectives	Drawbacks
LAB/CLB	Renamed to something else, e.g. "Block".	This is a very common unit.	Differs very much between the FPGAs.
LUTs	Lookup tables.	Also a common measurement unit.	There are many different LUT solutions, and sometimes several LUTs in different size per DFF.
FAs	Full adder, or corresponding logic.	All FPGAs have Full adders or corresponding. Easy to decide how many FAs that are used by the algorithms.	Full adders are not a very common measurement unit for FPGA resources.
DFFs	Use to occur one per FAs.	Same as FAs.	Same as FAs.

Table 6.2. Different cost units and their perspectives and drawbacks

Because of the drawbacks with the first two units, they are discarded. The alternatives "FAs" and "DFFs" are rather equal, and both represent a cell with one or more LUTs, a FA (or corresponding) and a DFF. This structure is found in all FPGAs, and very convenient. The name FA is used as a unit.

6.3 Algorithm

The method in `psac_suggest` is a kind of approximated brute force. There exists an approximation function, `estimator`, that quickly gives an estimation to the different qualities and resources for a given configuration. This function is called for all possible configurations with the required *N*. The exact steps are as follows:

1. Run a sine compression estimation for different *W*, to find the smallest possible *W* that can fulfill the signal quality requirement.
2. Run an estimation for each possible configuration with the required *N* and the found *W* (also run for some few bits bigger *W*s). Store the 256 best estimations in a list (this number can be changed by the 'ListSize' property).
3. Run a real calculation for each configuration in the saved list.
4. Print the *disp* best solutions, if *disp* was set > 0 by the 'ListSize' property.
5. Return the best configuration as a configuration string (in the format that `test_psac` want it).

This method is far from optimized, due to time limitations, and it is rather slow for the default list size 256. The algorithm is very sensitive to big *N*. The estimations part grows quadratic with *N*, due to the two parameters in the Sunderland method. The real calculation part grows exponentially, due to the number of elements in the result.

Chapter 7

Result

This chapter will discuss the result of the used algorithms, by comparing the SFDR with different hardware costs. The target is an unspecified FPGA with 1 kBits big ROMs and 18x18 unsigned multipliers.

Figure 7.1 shows some relations between SFDR and ROMs (1 kBit without parity). In those graphs the `psac_suggest` function has been run with different requirements on least SFDR, and tried to minimize the ROM usage. Used methods are Sunderland's and polynomials without correction ROM. N is set to 10 in (a), and 16 in (b) and (c). W and other parameters are optimized by `psac_suggest`. (c) is the same as (b), but showing only the first 20 ROMs. The dotted lines are Sunderland's. The markers 'x', '*', '+', and 'o' stands for $K=1, 2, 3$ and 4 respectively. Where the polynomials are missing for a certain K , one or more coefficients in the polynomials has been zero, and the grade has been decreased.

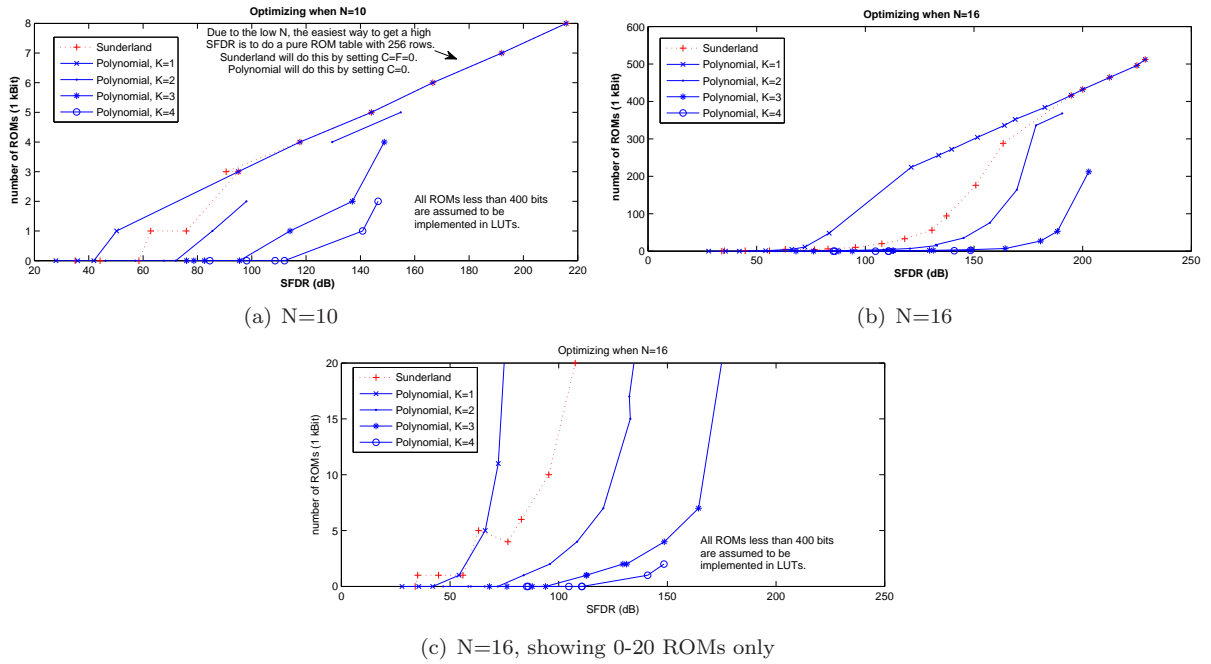
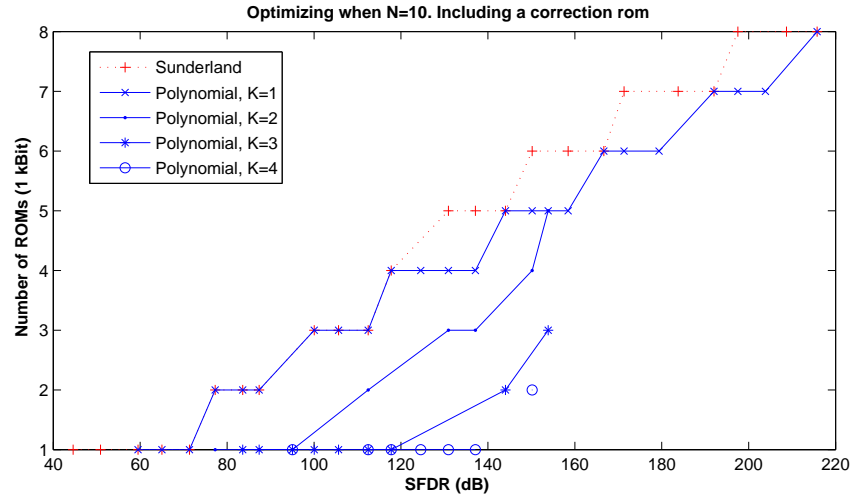


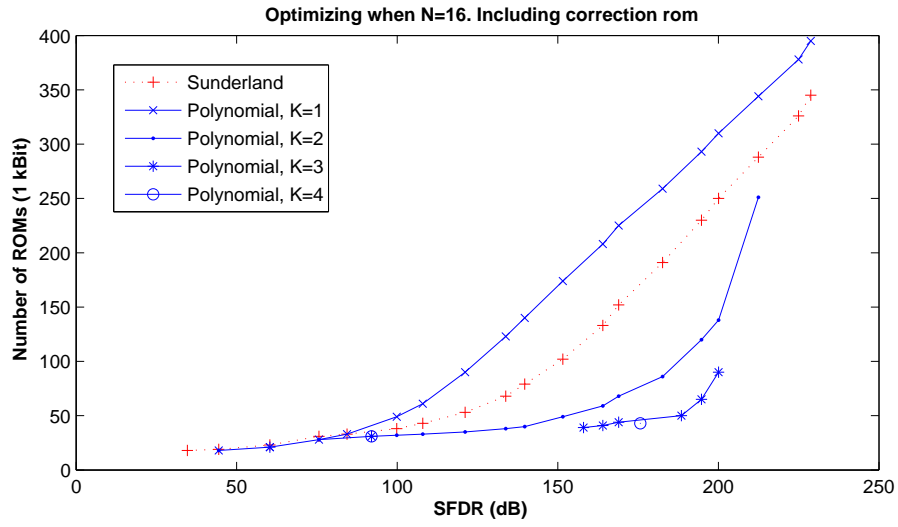
Figure 7.1. SFDR vs ROMs without correction ROM

Figure 7.2 shows the same thing as figure 7.1(a) and (b), but where the methods are extended with a correction ROM, resulting in sine compression methods.

The effect of the C parameter to the quality are discussed in section 4.5 on page 25, and illustrated in figure 4.6 (pg 27) and figure 4.7 (pg 29).



(a) N=10



(b) N=16

Figure 7.2. SFDR vs ROMs for sine compression

Chapter 8

Conclusions And Possible Improvements

There are few conclusions and many possible improvements left after this thesis.

8.1 Conclusions

The project has been running in a number of phases. Different conclusions has been made during the different phases.

- During the **method** phase the main conclusion was that many people have invented many different algorithms, and that many of them can be expressed as different special cases of the polynomial implementation.
- During the **target** phase, the main task was to find out what resources are available for the different implementations. One non-surprising conclusion was that FPGAs are well suited to implement a digital DDFS.
- In the **modelling** phase, some conclusions about different algorithms where made.

AWGN: May be an interesting rounding method, and rather easy to implement, but doubles the number of possible configurations, which add complexity to the program.

Polynomial: The polynomial solution is an efficient method that offers a spectrum of possible configurations.

Taylor: The TaylorLeft and TaylorMid coefficients assignments to the polynomial are not very efficient in this application.

Hutchinson's: This psac algorithm is not worth implementing. Its optimized variant, *Hutchinson 2*, is a zero-grade polynomial with a correction ROM.

Curticăpean's: This method uses more resources than a good polynomial.

Sunderland's: This method is superior to the polynomial in some cases.

- The **VHDL implementation** and **Suggester** construction phases were mainly just construction phases, without any bigger conclusions.

One generally important conclusion is that this project requires far more than one 800 hours thesis to be really good.

8.2 Suggested Improvements

There are many possible improvements that can be done, some examples:

- The `psac_suggesters` optimization algorithm.
- Implement methods for quadrature algorithms (returning both sine and cosine).
- Better cost model for the suggester.

- Support for the Very Coarse Approximations (see section 2.5.1, page 12) to the existing PSACs.
- Frequency handling in the suggerster.
- Find a better coefficient assigning method for Sunderlands, according to e.g. Nicholas *et al.* See section 2.3.5.
- Implement support for different pipeline levels (especially a register level after ROMs).
- Implement support for DP ROMs (Dual Port, see section 3.1), to minimize the amount of pipelined data in the polynomials.
- Adjust all coefficients to the 'middle' points in the polynomials, and calculate using $x_{F2} = [-0.5, 0.5) = x_F - 0.5$, this should lower the rounding noise for higher grade polynomials. It should also give Xilinx' multipliers one more bit to work with (they operate with e.g. 18 bits signed or 17 bits unsigned).
- Support for bigger N , during which Matlab will never store the entire sine vector in any specific moment. During the development there was huge memory problems when $N \geq 24$
- Model/implement more methods. e.g. a new coefficient assignment method to the Curticăpean algorithm could make that one better.

Bibliography

- [1] J.M.P. Langlois and D. Al-Khalili. Phase to sinusoid amplitude conversion techniques for direct digital frequency synthesis. *IEE Proc.-Circuits Devices Syst.*, 151(6), December 2004. URL: <http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=1387797&k2dockey=1387797@ieejrns>.
- [2] B.H. Hutchinson Jr. Contemporary frequency synthesis techniques. In J. (Ed.) Gorski-Pcpicl, editor, *Frequency synthesis: techniques and applications*, pages 25–45. IEEE Press, 1975.
- [3] D.A. Sunderland, D.A. Strauch, S.S. Wharfield, H.T. Peterson, and Cole C.R. CMOS/SOS frequency synthesizer LSI circuit for spread spectrum communications. *IEEE J. Solid-State Circuits*, 19, 1984. pp. 497-505.
- [4] H.T. Nicholas, Samuelli H., and B Kim. The optimization of direct digital frequency synthesizer performance in the presence of finite word length effects. *Annual Frequency Control Symposium*, 1988. pp. 357-363.
- [5] F. Curticăpean, K.I. Palomäki, and J Niittylahti. The optimization of direct digital frequency synthesiser with high memory compression ratio. *Electron. Lett*, 2001.
- [6] Wikipedia’s article about cordic from 25 mars 2010. <http://en.wikipedia.org/w/index.php?title=CORDIC&oldid=351988079>.
- [7] J.M.P. Langlois and D. Al-Khalili. ROM size reduction with low processing cost for direct digital frequency synthesis. In *Proc. IEEE Pacific Rim Conference on Communication, Computers and Signal Processing*, August 2001. URL: <http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=1387797&k2dockey=1387797@ieejrns>.
- [8] A.M. Sodagar and G.R. Lahiji. Mapping from phase to sine-amplitude in direct digital frequency synthesizers using parabolic approximation. *IEEE Transaction on Circuits and Systems-II, Analog Digit Signal Process.*, 47, 2000.
- [9] Alteras webbsite. <http://www.altera.com/products/devices/dev-index.jsp>.
- [10] *Cyclone Architecture*. http://www.altera.com/literature/hb/cyc/cyc_c51002.pdf.
- [11] *Cyclone II: Architecture*. http://www.altera.com/literature/hb/cyc2/cyc2_cii51002.pdf.
- [12] *Cyclone III: Device Core*. http://www.altera.com/literature/hb/cyc3/cyc3_ciii5v1_01.pdf.
- [13] *Cyclone IV: Device Core*. <http://www.altera.com/literature/hb/cyclone-iv/cyiv-5v1-01.pdf>.
- [14] *Arria GX Architecture*. http://www.altera.com/literature/hb/agx/agx_51002.pdf.
- [15] *Arria II GX: Device Core*. http://www.altera.com/literature/hb/arria-ii-gx/aiigx_5v1_01.pdf.
- [16] *Stratix Architecture*. http://www.altera.com/literature/hb/stx/ch_2_vol_1.pdf.
- [17] *Stratix GX Architecture*. http://www.altera.com/literature/hb/sgx/sgx_sgx51004.pdf.
- [18] *Stratix II Architecture*. http://www.altera.com/literature/hb/stx2/stx2_sii51002.pdf.
- [19] *Stratix II GX Architecture*. http://www.altera.com/literature/hb/stx2gx/stxiigx_sii51003.pdf.
- [20] *Stratix III Device Core*. http://www.altera.com/literature/hb/stx3/stx3_siii5v1_01.pdf.
- [21] *Stratix IV Device Core*. http://www.altera.com/literature/hb/stratix-iv/stx4_5v1_01.pdf.
- [22] *Spartan-6 CLB User Guide*. http://www.xilinx.com/support/documentation/user_guides/ug384.pdf.

- [23] *Virtex-6 CLB User Guide*. http://www.xilinx.com/support/documentation/user_guides/ug364.pdf.
- [24] *Spartan-3 User Guide*. http://www.xilinx.com/support/documentation/user_guides/ug331.pdf.
- [25] *Spartan-6 Overview*. http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf.
- [26] *Spartan-6 BlockRAM User Guide*. http://www.xilinx.com/support/documentation/user_guides/ug383.pdf.
- [27] *Spartan-6 DSP Slice User Guide*. http://www.xilinx.com/support/documentation/user_guides/ug389.pdf.
- [28] *Virtex Data Sheet*. http://www.xilinx.com/support/documentation/data_sheets/ds003.pdf.
- [29] *Virtex-E Data Sheet*. http://www.xilinx.com/support/documentation/data_sheets/ds022.pdf.
- [30] *Virtex-E ExtMem Data Sheet*. http://www.xilinx.com/support/documentation/data_sheets/ds025.pdf.
- [31] *Virtex-II Data Sheet*. http://www.xilinx.com/support/documentation/data_sheets/ds031.pdf.
- [32] *Virtex-II Pro Data Sheet*. http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf.
- [33] *Virtex-4 User Guide*. http://www.xilinx.com/support/documentation/user_guides/ug070.pdf.
- [34] *Virtex-4 Overview*. http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf.
- [35] *Virtex-5 User Guide*. http://www.xilinx.com/support/documentation/user_guides/ug190.pdf.
- [36] *Virtex-6 Overview*. http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf.
- [37] *Virtex-6 Memory User Guide*. http://www.xilinx.com/support/documentation/user_guides/ug363.pdf.
- [38] *Virtex-6 DSP Slice User Guide*. http://www.xilinx.com/support/documentation/user_guides/ug369.pdf.

Appendix A

What is...?

An appendix with descriptions of some of the used concepts.

ALM – Adaptive Logic Module. Structural unit in Alteras’ FPGAs. Contains one or more LUT, DFF and some other logics.

Altera® – FPGA vendor. Have the three main FPGA families Cyclone, Arria and Stratix. Main competitor to Xilinx.

AWGN – Adding White Gaussian Noise, a White Gaussian Noise have the same power in all frequencies.

Bit – The smallest piece of information in digital systems. May be '1' or '0'.

Carry – A time critical signal in an adder. Compare with adding a digit to the value 99995. If the digit is ≥ 5 then the entire chain of digits will change, the carry is then the “memory digit” that goes through all the digits.

Carrier – The main tone of a signal. In this thesis it is the intended sine wave without any approximation or rounding errors.

CLB – Configurable Logic Block, a structural unit in Xilinx’ FPGAs. May contain one or several *Slices*.

Combinatorial – A combinatorial digital function calculate a result from it’s current inputs. It is independent of earlier inputs.

CPLD – Complex Programmable Logic Device.

dB – decibel, a logarithmic scale for comparing relative difference in power between signals.

dBc – dB relative to the carrier. For example a harmonic may have a $\text{dBc} = -20$, why it have a power of $10^{-20/10}$ relative the base tone.

DDFS – Direct Digital Frequency Synthesizer. A logical module that takes a frequency (in any unit) as argument, and produce a pure sine wave with that frequency. A DDFS contains nothing but a phase counter and a PSAC.

DFF – D-type FlipFlop, a small unit that delays a digital signal one clock cycle.

e2 – $\|error\|_2$. The average (RMS) error of a signal.

einf – $\|error\|_\infty$. The maximum error of a signal.

ENOB – Effective Number Of Bits - approximately “number of correct bits” for a sine wave.

Error – The difference between the exact desired signal and its rounded and approximated value.

FA – Full Adder, a basic digital component.

FCW – Frequency Control Word, a number that is added to the phase accumulator in each clock cycle.

FPGA – Field Programmable Gate Array. An electronic chip that contains a lot of digital logic. The logic is programmable to a very high degree, why the FPGA can be programmed to behave in a very complex way, as long as the required behavior is digital. One type of use can be to code a signal from pure digital into a sine modulated signal. In that case an important component would be the DDFS.

Harmonic – A tone is harmonic to another if its frequency is an integer multiple of the other tone's frequency. The tone 440 Hz have the harmonics 2×440 , 3×440 , 4×440 , ... Hz.

LAB – Logic Array Block, a structural unit in Alteras' FPGAs. May contain one or several *ALMs* or *LEs*.

LC – Logic Cell. Structural unit in Xilinx' FPGAs. Contains typically a LUT, a DFF and some carry logic.

LE – Logic Element. Structural unit in Alteras' FPGAs. Contains typically a LUT, a DFF and some carry logic.

LSB – Least Significant Bit, the rightmost digit in a binary number (compare the '4' in 1024).

LUT – Look Up Table, or function generator. Takes a few signals and produces a required result as a function of those.

Matlab® – A program from The Mathsoft Inc. for mathematic calculations.

Mirror effect - The mirror effect is an effect when you sample a signal into finitely many values. If the signal frequency goes higher than half of the sample frequency, the sampled signals frequency will "bounce" in the $(\text{sample frequency})/2$.

ModelSim® – A program from Mentor Graphics aimed to compile and simulate for instance VHDL code.

MSB – Most significant bit (or bits), the leftmost digit in a binary number (compare the 1 in 1024).

Multiplier – A component in an FPGA that performs a multiplication.

PSAC – Phase to Sine Amplitude Converter. A logical module that takes a phase and responds with its corresponding sine amplitude.

Quadrant – A quarter of a rotation. The four quadrants represents $0-90^\circ$, $90-180^\circ$, $180-270^\circ$ and $270-360^\circ$ respectively.

RMS – Root Mean Square. The RMS of a set of values is the square root of the average of the squares of the values.

ROM – Read Only Memory, use to occur in a rather large amount in FPGAs, usually 512 bits to 8 kbits large.

RTL – Register Transfer Level, a low abstraction level view of a digital system/module/function.

SFDR – Spurious Free Dynamic Range, a way to measure signal purity.

SINAD – Signal to Noise And Distortion ratio, a way to measure signal purity.

Slice – Structural unit in Xilinx' FPGAs. Contains one or more LC (logic cell), and usually some other logic.

SNDR – Signal to Noise and Distortion Ratio, see SINAD.

SNR – Signal to Noise Ratio, a way to measure signal purity.

SURD – Symmetry Using Range Divider or Division. A notation in this thesis for a way to reduce the input range to the PSAC.

VHDL – VHSIC Hardware Description Language (VHSIC: Very High Speed Integrated Circuit). A language to describe how to program e.g. an FPGA.

Xilinx® – FPGA vendor. Have the two main FPGA families Spartan and Virtex. Main competitor to Altera.

Appendix B

Quality and Resource Tables

Tables from analysis. None of these aims to test all possible solutions, but to spread a number of tests within the range of possible solutions.

B.1 ROM/Polynomial

These tables means to compare the polynomial methods for some different K s and course/fine ratios.

For all tables are both accumulators and results 16 bits wide.

Average error: e2

C=1, F=13, W=16

K =	1	2	3	4	5
TaylorLeft	0.337	0.103	0.0226	0.00301	0.000615
TaylorMid	0.16	0.0358	0.00314	0.000237	2.29e-05
Interpole	0.16	0.0393	0.00195	8.38e-05	5.22e-06
Chebyshev	0.163	0.0373	0.00137	6.23e-05	7.96e-06
LeastSqr	0.159	0.0363	0.00142	5.55e-05	1.57e-05

C=6, F=8, W=16

K =	1	2	3	4	5
TaylorLeft	0.01	0.000166	1.62e-05	1.62e-05	1.62e-05
TaylorMid	0.00501	3.61e-05	1.66e-05	1.62e-05	1.62e-05
Interpole	0.00501	3.72e-05	1.67e-05	1.62e-05	1.62e-05
Chebyshev	0.00501	4.44e-05	1.67e-05	1.62e-05	1.62e-05
LeastSqr	0.00501	3.42e-05	1.67e-05	1.62e-05	1.62e-05

C=11, F=3, W=16

K =	1	2	3	4	5
TaylorLeft	0.000283	9.79e-06	9.79e-06	9.79e-06	9.79e-06
TaylorMid	0.000159	9.81e-06	9.79e-06	9.79e-06	9.79e-06
Interpole	0.000156	9.8e-06	9.79e-06	9.79e-06	9.79e-06
Chebyshev	0.000159	9.81e-06	9.79e-06	9.79e-06	9.79e-06
LeastSqr	0.000156	9.81e-06	9.79e-06	9.79e-06	9.79e-06

Max error: einf

C=1, F=13, W=16

K =	1	2	3	4	5
TaylorLeft	0.707	0.161	0.0449	0.0127	0.00244
TaylorMid	0.383	0.0642	0.00961	0.00096	9.52e-05
Interpole	0.383	0.0704	0.00364	0.000159	1.53e-05
Chebyshev	0.354	0.0671	0.00266	0.000127	1.52e-05
LeastSqr	0.373	0.0653	0.00375	0.000228	3.42e-05

C=6, F=8, W=16

K =	1	2	3	4	5
TaylorLeft	0.0244	0.000327	4.56e-05	4.56e-05	4.56e-05
TaylorMid	0.0123	8.35e-05	4.93e-05	4.56e-05	4.56e-05
Interpole	0.0122	8.58e-05	4.93e-05	4.56e-05	4.56e-05
Chebyshev	0.0123	0.000102	4.93e-05	4.56e-05	4.56e-05
LeastSqr	0.0122	7.69e-05	4.93e-05	4.56e-05	4.56e-05

C=11, F=3, W=16

K =	1	2	3	4	5
TaylorLeft	0.000686	2.79e-05	2.79e-05	2.79e-05	2.79e-05
TaylorMid	0.000398	2.79e-05	2.79e-05	2.79e-05	2.79e-05
Interpole	0.000351	2.79e-05	2.79e-05	2.79e-05	2.79e-05
Chebyshev	0.000398	2.79e-05	2.79e-05	2.79e-05	2.79e-05
LeastSqr	0.000351	2.79e-05	2.79e-05	2.79e-05	2.79e-05

Effective Number of Bits: ENOB

C=1, F=13, W=16

K =	1	2	3	4	5
TaylorLeft	0.771	3.63	5.04	7.97	10.2
TaylorMid	1.84	5.17	7.78	11.6	14.8
Interpole	1.84	5.06	8.24	12.8	16.8
Chebyshev	1.84	5.16	8.89	13.2	17.3
LeastSqr	1.84	5.18	9.09	13.4	16.3

C=6, F=8, W=16

K =	1	2	3	4	5
TaylorLeft	6.12	13.2	15.7	15.7	15.7
TaylorMid	6.86	15	15.7	15.7	15.7
Interpole	6.86	14.9	15.7	15.7	15.7
Chebyshev	6.86	14.9	15.7	15.7	15.7
LeastSqr	6.86	15	15.7	15.7	15.7

C=11, F=3, W=16

K =	1	2	3	4	5
TaylorLeft	11.2	15.9	15.9	15.9	15.9
TaylorMid	11.9	15.9	15.9	15.9	15.9
Interpole	11.9	15.9	15.9	15.9	15.9
Chebyshev	11.9	15.9	15.9	15.9	15.9
LeastSqr	11.9	15.9	15.9	15.9	15.9

Signal to Noise and Distorsion Ratio: SINAD

C=1, F=13, W=16

K =	1	2	3	4	5
TaylorLeft	6.31	23.5	32	49.7	63
TaylorMid	12.8	32.8	48.5	71.4	90.9
Interpole	12.8	32.1	51.3	78.6	103
Chebyshev	12.8	32.7	55.2	81.3	106
LeastSqr	12.8	32.8	56.4	82.4	99.9

C=6, F=8, W=16

K =	1	2	3	4	5
TaylorLeft	38.5	81	96.1	96.1	96.1
TaylorMid	43	91.8	96	96.1	96.1
Interpole	43	91.6	96	96.1	96.1
Chebyshev	43	91.3	96	96.1	96.1
LeastSqr	43	91.9	96	96.1	96.1

C=11, F=3, W=16

K =	1	2	3	4	5
TaylorLeft	69.4	97.2	97.2	97.2	97.2
TaylorMid	73	97.2	97.2	97.2	97.2
Interpole	73.2	97.2	97.2	97.2	97.2
Chebyshev	73	97.2	97.2	97.2	97.2
LeastSqr	73.2	97.2	97.2	97.2	97.2

Spurious Free Dynamic Range: SFDR

C=1, F=13, W=16

K =	1	2	3	4	5
TaylorLeft	9.54	27.3	35.3	54.6	66.3
TaylorMid	16.9	35.3	55.4	75.2	95.9
Interpole	16.9	33.8	53.8	81.6	106
Chebyshev	16.9	35.8	58.8	85.8	110
LeastSqr	16.9	35.8	61.7	87.2	103

C=6, F=8, W=16

K =	1	2	3	4	5
TaylorLeft	42.1	87	108	108	108
TaylorMid	48.1	96.1	108	108	108
Interpole	48.1	96.1	109	108	108
Chebyshev	48.1	96.1	109	108	108
LeastSqr	48.1	96.1	109	108	108

C=11, F=3, W=16

K =	1	2	3	4	5
TaylorLeft	73.4	117	117	117	117
TaylorMid	76.7	117	117	117	117
Interpole	76.7	117	117	117	117
Chebyshev	76.7	117	117	117	117
LeastSqr	76.7	117	117	117	117

Polynomial resources

C=1, F=13, W=16

K =	1	2	3	4	5
TaylorLeft	R: 15	R: [15 15], $\Sigma = 30$ M: 15	R: [13 15 15], $\Sigma = 43$ M: [13 15]	R: [12 13 15 15], $\Sigma = 55$ M: [12 14 15]	R: [9 12 13 15 15], $\Sigma = 64$ M: [9 12 14 15]
TaylorMid	R: 15	R: [15 15], $\Sigma = 30$ M: 15	R: [14 15 16], $\Sigma = 45$ M: [14 15]	R: [12 13 15 16], $\Sigma = 56$ M: [12 14 15]	R: [9 12 14 15 15], $\Sigma = 65$ M: [9 12 15 15]
Interpole	R: 15	R: [15 15], $\Sigma = 30$ M: 15	R: [14 15 15], $\Sigma = 44$ M: [14 15]	R: [12 13 15 15], $\Sigma = 55$ M: [12 14 15]	R: [9 12 14 15 15], $\Sigma = 65$ M: [9 12 15 15]
Chebyshev	R: 15	R: [15 15], $\Sigma = 30$ M: 15	R: [14 15 16], $\Sigma = 45$ M: [14 15]	R: [12 13 15 16], $\Sigma = 56$ M: [12 14 15]	R: [9 12 14 15 15], $\Sigma = 65$ M: [9 12 15 15]
LeastSqr	R: 15	R: [15 15], $\Sigma = 30$ M: 15	R: [14 15 16], $\Sigma = 45$ M: [14 15]	R: [12 13 15 16], $\Sigma = 56$ M: [12 14 15]	R: [9 12 14 15 15], $\Sigma = 65$ M: [9 12 15 15]

Some notes: Rom: $2 \times \dots$, Mult: $13 \times \dots$

C=6, F=8, W=16

K =	1	2	3-5
TaylorLeft	R: 15	R: [10 15], $\Sigma = 25$ M: 10	R: [4 10 15], $\Sigma = 29$ M: [4 10]
TaylorMid	R: 15	R: [10 15], $\Sigma = 25$ M: 10	R: [4 10 15], $\Sigma = 29$ M: [4 10]
Interpole	R: 15	R: [10 15], $\Sigma = 25$ M: 10	R: [4 10 15], $\Sigma = 29$ M: [4 10]
Chebyshev	R: 15	R: [10 15], $\Sigma = 25$ M: 10	R: [4 10 15], $\Sigma = 29$ M: [4 10]
LeastSqr	R: 15	R: [10 15], $\Sigma = 25$ M: 10	R: [4 10 15], $\Sigma = 29$ M: [4 10]

Some notes: Rom: $64 \times \dots$, Mult: $8 \times \dots$

C=11, F=3, W=16

K =	1	2-5
TaylorLeft	R: 15	R: [5 15], $\Sigma = 20$ M: 5
TaylorMid	R: 15	R: [5 15], $\Sigma = 20$ M: 5
Interpole	R: 15	R: [5 15], $\Sigma = 20$ M: 5
Chebyshev	R: 15	R: [5 15], $\Sigma = 20$ M: 5
LeastSqr	R: 15	R: [5 15], $\Sigma = 20$ M: 5

Some notes: Rom: $2k \times \dots$, Mult: $3 \times \dots$

B.2 Other Decompositions

The Hutchinson's, Sunderland's and Curticăpean's methods are handled here.

For simplicity reasons, $N = W = 16$ bits in all these cases.

The data is presented in two ways; first grouped by F and method, and then by quality types.

B.2.1 The F and Method Groupings

These groupings are suitable for analyzing each method.

F=3

Hutchinson's		Hutchinson's 2		Curticăpean's	
C =	11	C =	11	C =	11
e2	1.19e-05	e2	1.19e-05	e2	1.04e-05
einf	3e-05	einf	3.04e-05	einf	2.82e-05
ENOB	15.6	ENOB	15.6	ENOB	15.8
Sinad	95.5	Sinad	95.5	Sinad	96.6
SFDR	116	SFDR	116	SFDR	106

Sunderland's

C =	0	1	2	3	4	5	6	7	8	9	10,11
e2	1.93e-4	1.19e-4	6.25e-5	3.26e-5	1.94e-5	1.4e-5	1.25e-5	1.21e-5	1.19e-5	1.19e-5	1.19e-5
einf	6.81e-4	4.98e-4	2.53e-4	1.32e-4	7.57e-5	5.14e-5	4.18e-5	3.47e-5	3.2e-5	3.12e-5	3e-5
ENOB	12.3	12.8	13.7	14.6	15.2	15.5	15.5	15.6	15.6	15.6	15.6
Sinad	75.4	78.9	84.4	89.4	93	94.7	95.3	95.4	95.5	95.5	95.5
SFDR	83	87.6	92	98.4	102	108	114	115	116	116	116

F=8

Hutchinson's		Hutchinson's 2		Curticăpean's	
C =	6	C =	6	C =	6
e2	9.64e-05	e2	1.26e-05	e2	9.62e-05
einf	0.000321	einf	3.03e-05	einf	0.00031
ENOB	13.2	ENOB	15.5	ENOB	13.2
Sinad	80.9	Sinad	95.1	Sinad	81
SFDR	87.1	SFDR	109	SFDR	87.1

Sunderland's

C =	0	1	2	3	4	5	6
e2	0.0067	0.00412	0.00216	0.00109	0.000532	0.000249	9.64e-05
einf	0.0242	0.017	0.00908	0.00447	0.00212	0.000919	0.000321
ENOB	7.25	7.82	8.71	9.67	10.7	11.7	13.2
Sinad	45.3	48.7	54.1	59.9	65.8	72.2	80.9
SFDR	51.6	56.2	64.3	71	75.3	78.3	87.1

F=13

Hutchinson's		Hutchinson's 2		Curticăpean's	
C =	1	C =	1	C =	1
e2	0.0665	e2	9.17e-06	e2	0.0665
einf	0.207	einf	1.9e-05	einf	0.207
ENOB	3.77	ENOB	16	ENOB	3.77
Sinad	24.4	Sinad	98	Sinad	24.4
SFDR	26.1	SFDR	119	SFDR	26.1

Sunderland's

C =	0	1
e2	0.153	0.0665
einf	0.414	0.207
ENOB	2.87	3.77
Sinad	18.9	24.4
SFDR	19.9	26.1

B.2.2 The Quality Groupings

These groupings are suitable when comparing the different methods.

e2

F =	0	2	4	6	8	10	12	14
Hutchinson's	8.79e-06	1.13e-05	1.21e-05	1.41e-05	9.64e-05	0.00147	0.0209	8.79e-06
Hutchinson's 2	8.79e-06	1.13e-05	1.21e-05	1.23e-05	1.26e-05	1.32e-05	1.15e-05	8.79e-06
Curticapean's	8.79e-06	1.03e-05	1.04e-05	1.3e-05	9.62e-05	0.00147	0.0209	1.39e-05
Sunderland's, C=2	8.79e-06	7.97e-05	0.000403	0.00167	0.0067	0.0264	0.0964	8.79e-06
Sunderland's, C=4	8.79e-06	3.06e-05	0.000131	0.000541	0.00216	0.00822	0.0209	
Sunderland's, C=6	8.79e-06	1.33e-05	3.58e-05	0.000139	0.000532	0.00147		
Sunderland's, C=8	8.79e-06	1.14e-05	1.49e-05	3.64e-05	9.64e-05			
Sunderland's, C=10	8.79e-06	1.13e-05	1.23e-05	1.41e-05				
Sunderland's, C=12	8.79e-06	1.13e-05	1.21e-05					
Sunderland's, C=14	8.79e-06	1.13e-05						
Sunderland's, C=16	8.79e-06							

einf

F =	0	2	4	6	8	10	12	14
Hutchinson's	1.53e-05	3e-05	3.06e-05	4.49e-05	0.000321	0.00481	0.0703	1.53e-05
Hutchinson's 2	1.53e-05	3e-05	3.04e-05	3.02e-05	3.03e-05	3.02e-05	2.81e-05	1.53e-05
Curticapean's	1.53e-05	2.82e-05	2.85e-05	3.5e-05	0.00031	0.00479	0.0703	3.05e-05
Sunderland's, C=2	1.53e-05	0.000287	0.00144	0.00603	0.0242	0.0931	0.306	1.53e-05
Sunderland's, C=4	1.53e-05	0.000134	0.000553	0.0023	0.00908	0.0327	0.0703	
Sunderland's, C=6	1.53e-05	5.21e-05	0.000157	0.000567	0.00212	0.00481		
Sunderland's, C=8	1.53e-05	3.5e-05	5.65e-05	0.000156	0.000321			
Sunderland's, C=10	1.53e-05	3.09e-05	3.62e-05	4.49e-05				
Sunderland's, C=12	1.53e-05	3e-05	3.06e-05					
Sunderland's, C=14	1.53e-05	3e-05						
Sunderland's, C=16	1.53e-05							

ENOB

F =	0	2	4	6	8	10	12	14
Hutchinson's	16	15.7	15.6	15.5	13.2	9.24	5.47	16
Hutchinson's 2	16	15.7	15.6	15.5	15.5	15.4	15.6	16
Curticapean's	16	15.9	15.8	15.7	13.2	9.24	5.47	16
Sunderland's, C=2	16	13.4	11.3	9.24	7.25	5.3	3.52	16
Sunderland's, C=4	16	14.6	12.7	10.7	8.71	6.76	5.47	
Sunderland's, C=6	16	15.5	14.5	12.6	10.7	9.24		
Sunderland's, C=8	16	15.6	15.4	14.5	13.2			
Sunderland's, C=10	16	15.7	15.5	15.5				
Sunderland's, C=12	16	15.7	15.6					
Sunderland's, C=14	16	15.7						
Sunderland's, C=16	16							

SINAD

F =	0	2	4	6	8	10	12	14
Hutchinson's	98.1	95.9	95.3	94.8	80.9	57.3	34.6	98.1
Hutchinson's 2	98.1	95.9	95.3	95.2	95.1	94.6	95.8	98.1
Curticapean's	98.1	97.1	96.6	95.9	81	57.3	34.6	98.1
Sunderland's, C=2	98.1	82.4	69.4	57.3	45.3	33.6	22.9	98.1
Sunderland's, C=4	98.1	89.3	78.3	66.2	54.1	42.4	34.6	
Sunderland's, C=6	98.1	95.1	88.9	77.7	65.8	57.3		
Sunderland's, C=8	98.1	95.9	94.4	88.7	80.9			
Sunderland's, C=10	98.1	95.9	95.3	94.8				
Sunderland's, C=12	98.1	95.9	95.3					
Sunderland's, C=14	98.1	95.9						
Sunderland's, C=16	98.1							

SFDR

F =	0	2	4	6	8	10	12	14
Hutchinson's	128	114	118	112	87.1	63.5	40.4	128
Hutchinson's 2	128	114	118	113	109	104	102	128
Curticapean's	128	109	107	105	87.1	63.5	40.4	128
Sunderland's, C=2	128	88.4	76.4	63.9	51.6	39.1	26.2	128
Sunderland's, C=4	128	97.4	89.1	76.6	64.3	51.7	40.4	
Sunderland's, C=6	128	111	101	89.7	75.3	63.5		
Sunderland's, C=8	128	112	112	99.4	87.1			
Sunderland's, C=10	128	114	117	112				
Sunderland's, C=12	128	114	118					
Sunderland's, C=14	128	114						
Sunderland's, C=16	128							

Resources

F =	0	2	4	6	8	10	12	14
Hutchinson's	R: 16kx15	R: 4kx15 + 16kx4	R: 1kx15 + 16kx6	R: 256x15 + 16kx8	R: 64x15 + 16kx10	R: 16x15 + 16kx12	R: 4x15 + 16kx14	R: 16kx15
Hutchinson's 2	R: 16kx15	R: 4kx15 + 16kx4	R: 1kx15 + 16kx6	R: 256x15 + 16kx8	R: 64x15 + 16kx10	R: 16x15 + 16kx12	R: 4x15 + 16kx14	R: 16kx15
Curticapean's	R: 16kx31 M: 0x16	R: 4kx31 + 4x4 M: 4x16	R: 1kx31 + 16x6 M: 6x16	R: 256x31 + 64x8 M: 8x16	R: 64x31 + 256x10 M: 10x16	R: 16x31 + 1kx12 M: 12x16	R: 4x31 + 4kx14 M: 14x16	R: 16kx15 + 1x16 M: 15x16
Sunderland's, C=2	R: 16kx15	R: 4kx15 + 4x4	R: 1kx15 + 16x6	R: 256x15 + 64x8	R: 64x15 + 256x10	R: 16x15 + 1kx12	R: 4x15 + 4kx14	R: 16kx15
Sunderland's, C=4	R: 16kx15	R: 4kx15 + 16x4	R: 1kx15 + 64x6	R: 256x15 + 256x8	R: 64x15 + 1kx10	R: 16x15 + 4kx12	R: 4x15 + 16kx14	
Sunderland's, C=6	R: 16kx15	R: 4kx15 + 64x4	R: 1kx15 + 256x6	R: 256x15 + 1kx8	R: 64x15 + 4kx10	R: 16x15 + 16kx12		
Sunderland's, C=8	R: 16kx15	R: 4kx15 + 256x4	R: 1kx15 + 1kx6	R: 256x15 + 4kx8	R: 64x15 + 16kx10			
Sunderland's, C=10	R: 16kx15	R: 4kx15 + 1kx4	R: 1kx15 + 4kx6	R: 256x15 + 16kx8				
Sunderland's, C=12	R: 16kx15	R: 4kx15 + 4kx4	R: 1kx15 + 16kx6					
Sunderland's, C=14	R: 16kx15	R: 4kx15 + 16kx4						
Sunderland's, C=16	R: 16kx15							

B.3 Sine Compression

The sine compression function can take an entire configuration and create a sine compression, or if you leave parts of the configuration it will generate the cheapest solution (with respect to number of ROM bits). The Hutchinson's method is worse than the Hutchinson's 2, and Hutchinson's 2 is already a sine compression (with *ptl* as method), why those two are not listed. The $m=...$ codes are the method codes as defined in table 4.3 on page 25, with the exception that c is the Curticapean's method. The three tables are:

1. **Full control:** Testing the resources for some given inputs. No optimization. The entire configuration is handled to the PSAC creator. Only a very few values are tested for each method in the table.
2. **Method given:** Testing the best result for every combination of $N=\{8,12,16,20,24\}$, $W=\{10,16,20,32\}$ and all algorithms (all coefficient assignment methods are tested for $N=20$, $W=20$). Other configurations are optimized by the PSAC creator. There are one table for each tested N , 5 tables in total.
3. **Full optimization:** All combinations of $N=\{8,12,16,20,24\}$ and $W=\{10,16,20,32\}$ are tested, entire configuration is optimized.

Full control psac compression resources

Config	rom (appr)	rom (corr)	rom (tot)	used multiplications
N=16, W=16, F=8, K=1, m=ptl	960 bits	176 kbits	177 kbits	0
N=16, W=16, F=8, K=2, m=ptl	1.56 kbits	80 kbits	81.6 kbits	1
N=16, W=16, F=8, K=3, m=ptl	1.81 kbits	48 kbits	49.8 kbits	2
N=20, W=16, F=12, K=2, m=ptl	1.56 kbits	1.25 Mbits	1.25 Mbits	1
N=20, W=16, F=8, K=2, m=ptl	21 kbits	512 kbits	533 kbits	1
N=16, W=20, F=8, K=2, m=ptl	2.06 kbits	144 kbits	146 kbits	1
N=16, W=16, F=8, K=2, m=ptm	1.56 kbits	64 kbits	65.6 kbits	1
N=20, W=16, F=12, K=2, m=ptm	1.56 kbits	1 Mbits	1 Mbits	1
N=20, W=16, F=8, K=2, m=ptm	21 kbits	512 kbits	533 kbits	1
N=16, W=20, F=8, K=2, m=ptm	2.06 kbits	112 kbits	114 kbits	1
N=16, W=16, F=8, K=2, m=pc	1.56 kbits	48 kbits	49.6 kbits	1
N=20, W=16, F=12, K=2, m=pc	1.56 kbits	768 kbits	770 kbits	1
N=20, W=16, F=8, K=2, m=pc	21 kbits	512 kbits	533 kbits	1
N=16, W=20, F=8, K=2, m=pc	2.06 kbits	96 kbits	98.1 kbits	1
N=16, W=16, F=8, K=2, m=pi	1.56 kbits	48 kbits	49.6 kbits	1
N=20, W=16, F=12, K=2, m=pi	1.56 kbits	768 kbits	770 kbits	1
N=20, W=16, F=8, K=2, m=pi	21 kbits	512 kbits	533 kbits	1
N=16, W=20, F=8, K=2, m=pi	2.06 kbits	112 kbits	114 kbits	1
N=16, W=16, F=8, K=2, m=pls	1.56 kbits	48 kbits	49.6 kbits	1
N=20, W=16, F=12, K=2, m=pls	1.56 kbits	768 kbits	770 kbits	1
N=20, W=16, F=8, K=2, m=pls	21 kbits	512 kbits	533 kbits	1
N=16, W=20, F=8, K=2, m=pls	2.06 kbits	96 kbits	98.1 kbits	1
N=16, W=16, F=8, C=6, m=c	4.44 kbits	80 kbits	84.4 kbits	1
N=20, W=16, F=12, C=6, m=c	41.9 kbits	1.25 Mbits	1.29 Mbits	1
N=20, W=16, F=8, C=10, m=c	32.5 kbits	768 kbits	801 kbits	1
N=16, W=20, F=8, C=6, m=c	5.94 kbits	144 kbits	150 kbits	1
N=16, W=16, F=6, C=3, m=s	7.75 kbits	112 kbits	120 kbits	0
N=20, W=16, F=6, C=7, m=s	92 kbits	512 kbits	604 kbits	0
N=20, W=16, F=6, C=3, m=s	62 kbits	768 kbits	830 kbits	0
N=20, W=16, F=10, C=3, m=s	67.8 kbits	1.75 Mbits	1.82 Mbits	0
N=16, W=20, F=6, C=3, m=s	10.8 kbits	176 kbits	187 kbits	0

Some notes:

rom (appr): Rom sizes for approximating psac.

rom (corr): Correction rom size

m=c is methods Curticăpean's

Optimized psac compression for N = 8 and some W and methods

N,W,method	0 mults	1 mults	2 mults	3 mults
W=10, m=s	F=2, C=4 roms = 560 bits			
W=10, m=c	F=0, C=10 roms = 1.44 kbits	F=3, C=10 roms = 528 bits		
W=10, m=pls	F=3, K=1 roms = 520 bits	F=3, K=2 roms = 320 bits	F=4, K=3 roms = 220 bits	F=5, K=4 roms = 190 bits
W=16, m=s	F=2, C=3 roms = 1.08 kbits			
W=16, m=c	F=0, C=10 roms = 2.56 kbits	F=3, C=10 roms = 1.03 kbits		
W=16, m=pls	F=3, K=1 roms = 952 bits	F=3, K=2 roms = 736 bits	F=4, K=3 roms = 484 bits	F=4, K=4 roms = 328 bits
W=20, m=s	F=3, C=2 roms = 1.41 kbits			
W=20, m=c	F=0, C=10 roms = 3.31 kbits	F=3, C=10 roms = 1.38 kbits		
W=20, m=pls	F=1, K=2 roms = 1.19 kbits	F=3, K=2 roms = 1.03 kbits	F=3, K=3 roms = 720 bits	F=4, K=4 roms = 520 bits
W=32, m=s	F=3, C=2 roms = 2.34 kbits			
W=32, m=c	F=0, C=10 roms = 5.56 kbits	F=3, C=10 roms = 2.41 kbits		
W=32, m=pls	F=1, K=2 roms = 1.94 kbits	F=5, K=2 roms = 1.87 kbits	F=4, K=3 roms = 1.66 kbits	F=4, K=4 roms = 1.38 kbits

Some notes: **m=c** is methods Curticăpean's

Optimized psac_compression for N = 12 and some W and methods

N/W	0 mults	1 mults	2 mults	3 mults
W=10, m=s	F=3, C=6 roms = 3.5 kbits			
W=10, m=c	F=1, C=10 roms = 11.5 kbits	F=6, C=10 roms = 3.67 kbits		
W=10, m=pls	F=3, K=1 roms = 4.12 kbits	F=6, K=2 roms = 2.23 kbits	F=8, K=3 roms = 2.09 kbits	F=8, K=4 roms = 2.1 kbits
W=16, m=s	F=3, C=6 roms = 10 kbits			
W=16, m=c	F=0, C=10 roms = 37 kbits	F=4, C=10 roms = 8.09 kbits		
W=16, m=pls	F=3, K=1 roms = 10.9 kbits	F=4, K=2 roms = 4.56 kbits	F=7, K=3 roms = 3.3 kbits	F=8, K=4 roms = 3.2 kbits
W=20, m=s	F=3, C=6 roms = 15 kbits			
W=20, m=c	F=0, C=10 roms = 49 kbits	F=4, C=10 roms = 12.7 kbits		
W=20, m=pls	F=4, K=1 roms = 15.2 kbits	F=3, K=2 roms = 9 kbits	F=5, K=3 roms = 4.38 kbits	F=7, K=4 roms = 3.47 kbits
W=32, m=s	F=3, C=3 roms = 29.3 kbits			
W=32, m=c	F=0, C=10 roms = 85 kbits	F=5, C=10 roms = 25.8 kbits		
W=32, m=pls	F=4, K=1 roms = 27.9 kbits	F=4, K=2 roms = 22.6 kbits	F=4, K=3 roms = 13.8 kbits	F=5, K=4 roms = 7 kbits

Some notes: **m=c** is methods Curticăpean's

Optimized psac_compression for N = 16 and some W and methods

N/W	0 mults	1 mults	2 mults	3 mults
W=10, m=s	F=6, C=5 roms = 35.2 kbits			
W=10, m=c	F=4, C=10 roms = 51 kbits	F=8, C=10 roms = 34.2 kbits		
W=10, m=pls	F=5, K=1 roms = 36.5 kbits	F=10, K=2 roms = 32.2 kbits	F=13, K=3 roms = 48.1 kbits	F=14, K=4 roms = 48 kbits
W=16, m=s	F=4, C=7 roms = 66 kbits			
W=16, m=c	F=0, C=10 roms = 544 kbits	F=6, C=10 roms = 56.2 kbits		
W=16, m=pls	F=3, K=1 roms = 110 kbits	F=5, K=2 roms = 43 kbits	F=11, K=3 roms = 48.3 kbits	F=12, K=4 roms = 48.2 kbits
W=20, m=s	F=4, C=8 roms = 125 kbits			
W=20, m=c	F=0, C=10 roms = 720 kbits	F=6, C=10 roms = 107 kbits		
W=20, m=pls	F=4, K=1 roms = 179 kbits	F=5, K=2 roms = 63 kbits	F=9, K=3 roms = 49.4 kbits	F=11, K=4 roms = 48.5 kbits
W=32, m=s	F=4, C=8 roms = 341 kbits			
W=32, m=c	F=0, C=10 roms = 1.27 Mbits	F=6, C=10 roms = 305 kbits		
W=32, m=pls	F=4, K=1 roms = 383 kbits	F=4, K=2 roms = 229 kbits	F=6, K=3 roms = 81.8 kbits	F=8, K=4 roms = 53.6 kbits

Some notes: **m=c** is methods Curticăpean's

Optimized psac_compression for $N = 20$, some W and all methods

N/W	0 mults	1 mults	2 mults	3 mults
W=10, m=s	F=9, C=3 roms = 519 kbits			
W=10, m=c	F=8, C=10 roms = 531 kbits	F=10, C=10 roms = 519 kbits		
W=10, m=pls	F=9, K=1 roms = 517 kbits	F=14, K=2 roms = 512 kbits		
W=16, m=s	F=6, C=7 roms = 580 kbits			
W=16, m=c	F=3, C=10 roms = 1.47 Mbits	F=10, C=10 roms = 784 kbits		
W=16, m=pls	F=3, K=1 roms = 992 kbits	F=9, K=2 roms = 523 kbits	F=14, K=3 roms = 769 kbits	F=14, K=4 roms = 769 kbits
W=20, m=s	F=5, C=9 roms = 948 kbits			
W=20, m=c	F=0, C=10 roms = 10.5 Mbits	F=8, C=10 roms = 810 kbits		
W=20, m=ptl	F=4, K=1 roms = 2.05 Mbits	F=6, K=2 roms = 620 kbits	F=12, K=3 roms = 771 kbits	F=13, K=4 roms = 770 kbits
W=20, m=ptm	F=4, K=1 roms = 1.8 Mbits	F=6, K=2 roms = 620 kbits	F=13, K=3 roms = 769 kbits	F=15, K=4 roms = 768 kbits
W=20, m=pi	F=4, K=1 roms = 1.8 Mbits	F=7, K=2 roms = 568 kbits	F=13, K=3 roms = 769 kbits	F=15, K=4 roms = 768 kbits
W=20, m=pls	F=4, K=1 roms = 1.8 Mbits	F=7, K=2 roms = 568 kbits	F=13, K=3 roms = 769 kbits	F=14, K=4 roms = 769 kbits
W=20, m=pc	F=4, K=1 roms = 1.8 Mbits	F=7, K=2 roms = 568 kbits	F=13, K=3 roms = 769 kbits	F=15, K=4 roms = 768 kbits
W=32, m=s	F=5, C=9 roms = 3.82 Mbits			
W=32, m=c	F=0, C=10 roms = 19.2 Mbits	F=8, C=10 roms = 3.57 Mbits		
W=32, m=pls	F=4, K=1 roms = 4.98 Mbits	F=4, K=2 roms = 1.52 Mbits	F=9, K=3 roms = 802 kbits	F=12, K=4 roms = 774 kbits

Some notes: **m=c** is methods Curticăpean's

Optimized psac_compression for N = 24 and some W and methods

N/W	0 mults	1 mults	2 mults	3 mults
W=10, m=s	F=12, C=2 roms = 8.01 Mbits			
W=10, m=c	F=12, C=10 roms = 8.02 Mbits	F=13, C=10 roms = 8.02 Mbits		
W=10, m=pls	F=13, K=1 roms = 8 Mbits	F=14, K=2 roms = 8 Mbits		
W=16, m=s	F=9, C=6 roms = 8.14 Mbits			
W=16, m=c	F=6, C=10 roms = 9.94 Mbits	F=12, C=10 roms = 8.05 Mbits		
W=16, m=pls	F=7, K=1 roms = 8.47 Mbits	F=13, K=2 roms = 8.01 Mbits		
W=20, m=s	F=8, C=10 roms = 8.67 Mbits			
W=20, m=c	F=3, C=10 roms = 27.5 Mbits	F=12, C=10 roms = 12.1 Mbits		
W=20, m=pls	F=4, K=1 roms = 16.8 Mbits	F=11, K=2 roms = 8.05 Mbits	F=14, K=3 roms = 12 Mbits	
W=32, m=s	F=6, C=10 roms = 42.2 Mbits			
W=32, m=c	F=0, C=10 roms = 292 Mbits	F=10, C=10 roms = 40.3 Mbits		
W=32, m=pls	F=4, K=1 roms = 63.8 Mbits	F=7, K=2 roms = 13.5 Mbits	F=13, K=3 roms = 12 Mbits	F=14, K=4 roms = 12 Mbits

Some notes: m=c is methods Curticăpean's

Optimized psac_compression for some N and W

N/W	0 mults	1 mults	2 mults	3 mults
N= 8, W=10	F=3, K=1, m=pc roms = 520 bits	F=3, K=2, m=pc roms = 320 bits	F=4, K=3, m=pc roms = 220 bits	F=5, K=4, m=pc roms = 190 bits
N= 8, W=16	F=3, K=1, m=pc roms = 952 bits	F=3, K=2, m=pc roms = 736 bits	F=4, K=3, m=pc roms = 484 bits	F=4, K=4, m=pls roms = 328 bits
N= 8, W=20	F=1, K=2, m=pls roms = 1.19 kbits	F=4, K=2, m=pc roms = 1.02 kbits	F=3, K=3, m=pls roms = 720 bits	F=4, K=4, m=pc roms = 520 bits
N= 8, W=32	F=1, K=2, m=pls roms = 1.94 kbits	F=4, K=2, m=pc roms = 1.86 kbits	F=4, K=3, m=pc roms = 1.66 kbits	F=4, K=4, m=pc roms = 1.38 kbits
N=12, W=10	F=3, C=6, m=s roms = 3.5 kbits	F=6, K=2, m=pc roms = 2.23 kbits	F=8, K=3, m=pls roms = 2.09 kbits	F=8, K=4, m=pc roms = 2.1 kbits
N=12, W=16	F=3, C=6, m=s roms = 10 kbits	F=4, K=2, m=pc roms = 4.56 kbits	F=7, K=3, m=pc roms = 3.3 kbits	F=8, K=4, m=pc roms = 3.2 kbits
N=12, W=20	F=3, C=6, m=s roms = 15 kbits	F=3, K=2, m=pc roms = 8 kbits	F=5, K=3, m=pc roms = 4.38 kbits	F=7, K=4, m=pc roms = 3.47 kbits
N=12, W=32	F=4, K=1, m=pc roms = 27.9 kbits	F=4, K=2, m=pc roms = 21.6 kbits	F=4, K=3, m=pc roms = 13.8 kbits	F=5, K=4, m=pc roms = 7 kbits
N=16, W=10	F=6, C=5, m=s roms = 35.2 kbits	F=10, K=2, m=pc roms = 32.2 kbits	F=13, K=3, m=pc roms = 48.1 kbits	F=14, K=4, m=pc roms = 48 kbits
N=16, W=16	F=4, C=7, m=s roms = 66 kbits	F=5, K=2, m=pc roms = 43 kbits	F=11, K=3, m=pc roms = 48.3 kbits	F=12, K=4, m=pc roms = 48.2 kbits
N=16, W=20	F=4, C=8, m=s roms = 125 kbits	F=6, K=2, m=pc roms = 55.8 kbits	F=9, K=3, m=pc roms = 49.4 kbits	F=11, K=4, m=pc roms = 48.5 kbits
N=16, W=32	F=4, C=8, m=s roms = 341 kbits	F=4, K=2, m=pc roms = 213 kbits	F=6, K=3, m=pc roms = 81.8 kbits	F=8, K=4, m=pc roms = 53.6 kbits
N=20, W=10	F=9, K=1, m=pc roms = 517 kbits	F=14, K=2, m=pc roms = 512 kbits	F=16, K=3, m=pc roms = 768 kbits	F=16, K=4, m=pc roms = 768 kbits
N=20, W=16	F=6, C=7, m=s roms = 580 kbits	F=9, K=2, m=pc roms = 523 kbits	F=15, K=3, m=pc roms = 768 kbits	F=16, K=4, m=pc roms = 768 kbits
N=20, W=20	F=5, C=9, m=s roms = 948 kbits	F=7, K=2, m=pc roms = 568 kbits	F=13, K=3, m=pc roms = 769 kbits	F=15, K=4, m=pc roms = 768 kbits
N=20, W=32	F=5, C=9, m=s roms = 3.82 Mbits	F=5, K=2, m=pc roms = 1.39 Mbits	F=9, K=3, m=pc roms = 802 kbits	F=12, K=4, m=pc roms = 774 kbits
N=24, W=10	F=13, K=1, m=pc roms = 8 Mbits	F=16, K=2, m=pc roms = 8 Mbits		
N=24, W=16	F=9, C=6, m=s roms = 8.14 Mbits	F=13, K=2, m=pc roms = 8.01 Mbits	F=16, K=3, m=pc roms = 12 Mbits	
N=24, W=20	F=8, C=10, m=s roms = 8.67 Mbits	F=11, K=2, m=pc roms = 8.05 Mbits	F=16, K=3, m=pc roms = 12 Mbits	
N=24, W=32	F=6, C=10, m=s roms = 42.2 Mbits	F=8, K=2, m=pc roms = 12.8 Mbits	F=13, K=3, m=pc roms = 12 Mbits	F=16, K=4, m=pc roms = 12 Mbits

Appendix C

Polynomial VHDL Example

This is an example of the VHDL polynomial code that is generated. To keep memory size down, the *C* parameter has been set rather small (3).

The following configuration was used: 'N=20, W=16, method=pc, F=15, K=4'.

```
-- A PSAC sine implementation, using 2^3 polynomial of grade 3.
-- Each polynomial has 2^15 points
-- Input width: 20
-- Output width: 16
-- Used ROM: 8 x 44 bits
-- Used Mults: 3
-- File autogenerated from Matlab
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity my_pvac_example is
  port(
    clk: in std_logic;
    x : in UNSIGNED(19 downto 0);
    a : out SIGNED(15 downto 0) := (others=>'0');
  end my_pvac_example;

architecture my_pvac_example_architecture of my_pvac_example is
  constant N : integer := 20;    -- number of phase bits.
  constant W : integer := 16;    -- number of result width.
  constant F : integer := 15;    -- number of fine bits.
  constant C : integer := N-2-F; -- 3 coarse bits.
  constant R1 : integer := 6;    -- ROM1 width
  constant R2 : integer := 10;   -- ROM2 width
  constant R3 : integer := 13;   -- ROM3 width
  constant R4 : integer := 15;   -- ROM4 width
  signal inv_res : std_logic_vector(0 to 4) := (others => '0'); -- if res should be negated.
  signal x2      : UNSIGNED(N-3 downto 0) := (others=>'0'); -- x(N-3 downto 0) xor x(MSB-1)
  signal xC      : UNSIGNED(C-1 downto 0) := (others=>'0'); -- x(N-3 downto F) xor x(MSB-1)
  signal xF      : UNSIGNED(F-1 downto 0) := (others => '0'); -- x(F-1 downto 0) xor x(MSB-1)
  signal ResQ1 : UNSIGNED(W-2 downto 0) := (others => '0'); -- Result for quadrant 1
  -- signals for iteration 1
  signal xF_1 : UNSIGNED(F-1 downto 0) := (others => '0'); -- 14..0
  signal Res_1 : UNSIGNED(R1-1 downto 0) := (others => '0'); -- 5..0
  -- signals for iteration 2
  signal xF_2 : UNSIGNED(F-1 downto 0) := (others => '0'); -- 14..0
  signal data2_1: UNSIGNED(R2-1 downto 0) := (others => '0'); -- 9..0
  signal Res_2 : UNSIGNED(R2-1 downto 0) := (others => '0'); -- 9..0
  -- signals for iteration 3
  signal xF_3 : UNSIGNED(F-1 downto 0) := (others => '0'); -- 14..0
  signal data3_1: UNSIGNED(R3-1 downto 0) := (others => '0'); -- 12..0
  signal data3_2: UNSIGNED(R3-1 downto 0) := (others => '0'); -- 12..0
  signal Res_3 : UNSIGNED(R3-1 downto 0) := (others => '0'); -- 12..0
  -- signals for iteration 4
  signal data4_1: UNSIGNED(R4-1 downto 0) := (others => '0'); -- 14..0
  signal data4_2: UNSIGNED(R4-1 downto 0) := (others => '0'); -- 14..0
  signal data4_3: UNSIGNED(R4-1 downto 0) := (others => '0'); -- 14..0
  signal Res_4 : UNSIGNED(R4-1 downto 0) := (others => '0'); -- 14..0
  -- *K => signals is delayed K clock cycles from it's x input.
  -- dataM = the synchronous ROM output.
  -- Res = (result of iter M=K) = dataM +- xF*(data(M-1) -+ xF*(...)).
```

```

component POLY_ROM is
  port( clk : in std_logic; --synchronous ROM
        addr : in unsigned(C-1 downto 0); -- 2..0
        data1: out unsigned(R1-1 downto 0); -- 5..0
        data2: out unsigned(R2-1 downto 0); -- 9..0
        data3: out unsigned(R3-1 downto 0); -- 12..0
        data4: out unsigned(R4-1 downto 0)); -- 14..0
end component;

-- "+"/"-" for adding/subtracting a carry to/from a vector
function "-"(l : unsigned; r : std_logic) return unsigned is
begin
  return unsigned(ieee.std_logic_unsigned."-(std_logic_vector(l), r));
end "-";
function "+"(l : unsigned; r : std_logic) return unsigned is
begin
  return unsigned(ieee.std_logic_unsigned."+(std_logic_vector(l), r));
end "+";
begin

  -- Initial phase adjustment to quadrant:
  x2 <= x(N-3 downto 0) when x(N-2) = '0' else
    not x(N-3 downto 0);
  xC <= x2(N-3 downto F); -- 17..15
  xF <= x2(F-1 downto 0); -- 14..0
  inv_res(0) <= x(N-1);

  rom : component POLY_ROM
    port map( clk => clk, -- output is delayed one cycle
              addr => xC,
              data1 => Res_1,
              data2 => data2_1,
              data3 => data3_1,
              data4 => data4_1);

  process(clk)
    variable tmp : UNSIGNED(W + F - 1 downto 0); -- 30..0
  begin
    if rising_edge(clk) then
      ----- Pipeline stage 1 -----
      xF_1 <= xF;
      ----- Pipeline stage 2 -----
      xF_2 <= xF_1;
      tmp := (others => '0');
      tmp(R1+F-1 downto 0) := Res_1 * xF_1; -- tmp(20..0)
      Res_2 <= data2_1 + tmp(R2+F-1 downto F) + tmp(F-1);
      data3_2 <= data3_1;
      data4_2 <= data4_1;
      ----- Pipeline stage 3 -----
      xF_3 <= xF_2;
      tmp := (others => '0');
      tmp(R2+F-1 downto 0) := Res_2 * xF_2; -- tmp(24..0)
      Res_3 <= data3_2 - tmp(R3+F-1 downto F) - tmp(F-1);
      data4_3 <= data4_2;
      ----- Pipeline stage 4 -----
      tmp := (others => '0');
      tmp(R3+F-1 downto 0) := Res_3 * xF_3; -- tmp(27..0)
      Res_4 <= data4_3 + tmp(R4+F-1 downto F) + tmp(F-1);

      inv_res(1 to inv_res'high) <= inv_res(0 to inv_res'high-1);
    end if;
  end process;

  ResQ1 <= Res_4;
  -- final result adjustment to quadrant:
  a <= signed('0' & ResQ1) when inv_res(inv_res'high) = '0' else
    -signed('0' & ResQ1);
end my_psac_example_architecture;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity POLY_ROM is

```

```

port (
  clk : in  std_logic; --synchronous ROM
  addr : in  UNSIGNED(2 downto 0);
  data1: out UNSIGNED(5 downto 0) := (others => '0');
  data2: out UNSIGNED(9 downto 0) := (others => '0');
  data3: out UNSIGNED(12 downto 0) := (others => '0');
  data4: out UNSIGNED(14 downto 0) := (others => '0'));
end POLY_ROM;

architecture POLY_ROM_architecture of POLY_ROM is
  type romt is array(0 to 7) of UNSIGNED(43 downto 0);
  signal rom_data : romt;
  signal tmp : UNSIGNED(43 downto 0);
begin
  -- Wait with ROM data until all other is done
  process(clk) begin
    if clk'event and clk = '1' then
      data1 <= tmp(43 downto 38);
      data2 <= tmp(37 downto 28);
      data3 <= tmp(27 downto 15);
      data4 <= tmp(14 downto 0);
    end if;
  end process;

  tmp <= rom_data(to_integer(addr));
  -- Here comes the ROM data
  rom_data <= (b"101001_0000000000_1100100100010_00000000000000",
    b"101000_0001111100_1100010100110_001100011111001",
    b"100100_0011110011_1011100111000_011000011111011",
    b"100000_0101100000_1010011100110_100011100011100",
    b"011010_0111000001_1000111000110_101101010000010",
    b"010011_1000001111_0110111110111_110101001101101",
    b"001100_1001001010_0100110011111_111011001000001",
    b"000100_1001101110_0010011101000_111110110001001");

end POLY_ROM_architecture;

```