IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Leibniz
Universität
Hannover

Master's Thesis

Evaluation and Optimization of a Complex Filter System Using Different Application-Specific Instruction-Set Processor Configurations

Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Mikroelektronische Systeme
Fachgebiet Architekturen und Systeme

Jonas Rinke

April 16, 2021

IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Leibniz
Universität
Hannover

Master's Thesis

Evaluation and Optimization of a Complex Filter System Using Different Application-Specific Instruction-Set Processor Configurations

Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Mikroelektronische Systeme
Fachgebiet Architekturen und Systeme

presented by

Jonas Rinke

Matriculation Number 3216830
born on: December 28th, 1996 in: Hanover

First Examiner: Prof. Dr.-Ing. Holger Blume
Second Examiner: Prof. Dr.-Ing. Bernhard Wicht
Supervisor: M. Sc. Jens Karrenbauer

Hannover, April 16, 2021

IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Leibniz
Universität
Hannover

Hannover, 16. October 2020

Master's Thesis

for Jonas Rinke

Evaluation and Optimization of a Complex Filter System using different Application-Specific Instructions-Set Processor Configurations

Due to digital progress and Industry 4.0, production machines are becoming larger and more complex. The devices are more interconnected and equipped with a higher number of sensors to monitor the newly added system functions. Due to moving and metallic elements in such machines, neither large cable harnesses nor wireless connections can transport the sensor data. Therefore, data pre-processing and sensor fusion directly at the source are usually required. However, since such machines' space and energy-consumption are limited, the architecture for signal-processing must be as small and energy-efficient as possible. Furthermore, these systems must guarantee a real-time capability to react in time to safety-relevant disturbances.

At the department "Architectures and Systems" of the Institute of Microelectronic Systems, dedicated and programmable architectures with special requirements for real-time capability and power consumption are designed, implemented, and tested. A suitable architecture that meets these requirements is an application-specific instruction-set processor (ASIP). A unique feature of this architecture is that it can be optimized for specific application classes. Therefore, it can be extended by special hardware units, increasing performance or reducing energy consumption. The analysis is not limited to the hardware, but also the signal processing algorithms and filter systems for these applications are explored and optimized for the target architecture. A real-time capable, space-saving, and energy-efficient complex filter system for sensor data processing can be researched by combining these methods.

In the context of this work, Mr. Rinke will analyze a reference filter system for computing an order analysis implemented on an ARM Cortex M4. He will evaluate the System's theoretical background to identify hotspots and parts of the algorithm that can be improved. To compare the benefits of the different ASIP Configurations, Mr. Rinke will first do a baseline implementation on a Cadence Tensilica Xtensa LX7, with varying configurations of the hardware. He will continue to adapt these implementations to the unique features of the Cadence Tensilica HiFi 3, HiFi mini, and Fusion G3 by using architecture-specific intrinsic. Furthermore, to save area, the processor configurations should not contain a floating-point unit, and therefore, floating-point operations of the reference must be replaced with adequate fixed-point function. In the end, the different features and benefits of the configurations should be evaluated.

Proper documentation has to be ensured. The submitted copies and the results of the work remain the property of the institute.

Prof. Dr.-Ing. Holger Blume

Prof. Dr.-Ing Holger Blume
blume@ims.uni-hannover.de

Appelstraße 4 | 30167 Hannover
fon 0511 762-19640 | fax 0511 762-19601
www.ims.uni-hannover.de | info@ims.uni-hannover.de

Erklärung

Ich versichere hiermit, dass ich die vorstehende Arbeit selbständig angefertigt, keine anderen als die angegebenen Hilfsmittel benutzt und sowohl wörtliche, als auch sinngemäßentlehnte Stellen als solche kenntlich gemacht habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Hannover, den April 16, 2021

Jonas Rinke

Jonas Rinke

Contents

Acronym Index VIII

Symbol Index IX

List of Figures X

List of Tables XII

1 Introduction 1

2 Fundamentals 2
- 2.1 Order Analysis 2
- 2.2 Application-Specific Instruction-Set Processors 3
- 2.3 Hardware Parallelism 4
  - 2.3.1 Very Long Instruction Words 4
  - 2.3.2 Single-Instruction, Multiple-Data 5
- 2.4 Fixed Point 5
  - 2.4.1 Arithmetic with Fixed Point Numbers 7

3 Structure of the Filter System 12
- 3.1 Overview 12
  - 3.1.1 X- and Y-Values 12
  - 3.1.2 Signal Flow 12
- 3.2 Band-pass Filter 14
- 3.3 Envelope Detector 14
  - 3.3.1 Analytic Signals and the Hilbert Transform 15
  - 3.3.2 The Discrete Case 18
- 3.4 Decimation Filter 21
  - 3.4.1 Aliasing 21
  - 3.4.2 Anti-aliasing Filter 22
  - 3.4.3 Decimating the X-Values 23
- 3.5 Interpolation Filter 23
  - 3.5.1 Fractional Delay Filters 25
  - 3.5.2 Algorithm 26
- 3.6 Fast Fourier Transform 28
  - 3.6.1 Butterflies and Bit Reversals 28
- 3.7 Filter System Configuration 30

4 Implementation 34
- 4.1 Differences from the Reference Implementation 34
  - 4.1.1 Skipping the Decimation Filter 34

4.1.2 Restricting the Range of the X-Values 34
4.2 Fixed Point Implementation 35
4.2.1 Motivation 35
4.2.2 Data Types 36
4.2.3 Division 36
4.2.4 Hypot Function 41
4.3 ASIP Configurations 42
4.3.1 Base Architecture 42
4.3.2 First Coprocessor 44
4.3.3 Second Coprocessor 45
4.3.4 Third Coprocessor 46

5 Evaluation and Optimization **48**
5.1 Design Space Exploration 48
5.2 Evaluation Setup 48
5.2.1 Test Data Sets 49
5.3 LX7 50
5.4 HiFi 3 53
5.4.1 Optimization Process 53
5.4.2 Final Evaluation of the HiFi 3 Implementation 54
5.5 HiFi mini 57
5.6 Fusion G3 60
5.6.1 Optimization Process 61
5.6.2 Final Evaluation of the Fusion G3 Implementation 61
5.7 Hardware Properties 63
5.8 Discussion 66
5.9 Future Work 67

6 Conclusion **68**

Bibliography **69**

Acronym Index

ALU	Arithmetic/Logic Unit
AM	Amplitude Modulation
ASIP	Application-Specific Instruction-Set Processor
CORDIC	Coordinate Rotation Digital Computer
DFT	Discrete Fourier Transform
DSP	Digital Signal Processing
DTFT	Discrete Time Fourier Transform
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPU	Floating Point Unit
FU	Functional Unit
ISA	Instruction Set Architecture
MAC	Multiply-Accumulate
NOP	No-Operation Instruction
NSA	Normalize Shift Amount
SIMD	Single Instruction, Multiple Data
VLIW	Very Long Instruction Word

Symbol Index

$\mathscr{F}\{\cdot\}$	Fourier transform
$\mathscr{H}\{\cdot\}$	Hilbert transform
w_N^k	Fourier transform coefficient, shorthand for $e^{-j\frac{2\pi k}{N}}$

List of Figures

2.1 Comparison between a frequency spectrum (left) and an order spectrum (right) of a simulated ramp-up and coast-down. 3
2.2 VLIW-Architecture of the HiFi3, showcasing the different Functional Units of each issue slot [1]. 5
2.3 Binary number in integer format and Q3.13 fixed point format 6
2.4 Multiplication of $2.7423 \cdot 1.23 = 3.3731$ in Q3.13 format 8
2.5 Division of $2.7423/1.23 = 2.2296$ in Q3.13 format 9
2.6 Example of the CORDIC algorithm in vectoring mode to compute the magnitude of $(0.7, 0.5)$ 11

3.1 Overview of the filter system. 13
3.2 Example for Amplitude Modulation 15
3.3 The real and imaginary parts of an analytic signal. 18
3.4 Hilbert coefficients from sampling the continuous-time Hilbert transform 19
3.5 Hilbert coefficients from the inverse DTFT 20
3.6 Example signal and spectrum 21
3.7 Example of aliasing. 22
3.8 Example of an anti-aliasing filter 23
3.9 Decimation with an anti-aliasing filter. 24
3.10 Function of the interpolation filter 25
3.11 Examples for fractional delay filters 26
3.12 Butterflies of different sizes [2] 30
3.13 Example of an envelope detector with too short of a Hilbert filter (left) and a sufficiently long Hilbert filter (right) 31

4.1 Three iterations of the Newton-Raphson method to compute $1/0.625$ 38
4.2 Overview of the LX7 architecture [3] 43
4.3 HiFi3 DSP Coprocessor [1] 44
4.4 HiFi mini DSP Coprocessor [4] 46
4.5 Fusion G3 DSP Coprocessor [5] 47

5.1 Number of samples at the input of each component 49
5.2 Average cycles per sample for the LX7 baseline on test setups A and B 50
5.3 Average cycles per sample for the LX7+MUL16 on test setups A and B 51
5.4 Average cycles per sample for the LX7+MUL32 on test setups A and B 51
5.5 Relative error for the LX7 configurations 52
5.6 Cycle count improvements of the HiFi 3 implementation over time 54
5.7 Average cycles per sample for the HiFi 3 on test setups A and B 55
5.8 Cycles per component for the HiFi 3 and LX7+MUL16 56
5.9 Output error for the HiFi 3 57

5.10 Average cycles per sample for the HiFi mini on test setups A and B 58
5.11 Cycles per component for the HiFi mini and LX7+MUL16 59
5.12 Output error for the HiFi mini 60
5.13 Cycle count improvements of the Fusion G3 implementation over time 60
5.14 Average cycles per sample for the Fusion G3 on test setups A and B 61
5.15 Cycles per component for the Fusion G3 and LX7+MUL16 62
5.16 Output error for the Fusion G3 63
5.17 Comparison of hardware properties 64

List of Tables

2.1 The value 0.2 in different number formats 6

3.1 Shortcuts for multiplying a complex number $a+jb$ with specific twiddle factors. 29
3.2 Re-ordered indices for radix-2 4 and 8 FFTs 31

4.1 Example calls to `normalize_shift_amount(x)` 39

5.1 Maximum clock frequency supported by the processor configurations 65
5.2 Minimum clock frequency required for real-time processing 65

1 Introduction

Monitoring the condition of rotating machines can help in predicting failures or showing wear of certain parts of the machine. Information about the condition of machine parts can aid in maintenance or prevent a break-down of the machine. Break-down through faults and wear in complex or critical machinery can have outcomes ranging from economic losses through down-time of the machine, to serious accidents [6]. Due to the increasing size and complexity of rotating machines, maintenance also becomes more difficult. One approach which can convey a lot of important information about the machine's condition is to analyse the vibration produced by the machine. For example, faults or cracks on parts like rotor shafts or rolling element bearings produce characteristic frequencies in the vibration signal that can be detected. The technique for vibration analysis used in this thesis is part of predictive maintenance and makes use of an accelerometer signal to relate the vibration signal to the machine's rotational speed. This method is called *order tracking* or *order analysis* [7].

To facilitate an on-line analysis of the vibration signal, fast signal processing is required. On the other hand, the analysis should have a low power consumption and cost. One way to address these two potentially conflicting requirements is to use Application-Specific Instruction-Set Processors. By tailoring a processor to a specific application, low hardware and power costs can be reached while retaining the flexibility to change and improve the underlying algorithm [8]. This thesis presents a design space exploration of different processor architectures and evaluates each of them according to their performance and efficiency when performing order analysis.

The design space exploration is carried out through an implementation of a filter system for order analysis. This implementation is a port of an existing code base to six processor configurations based on the Cadence® Tensilica® Xtensa® Instruction Set Architecture. The configurations include both variations of a base processor core as well as extensions of the base architecture through DSP coprocessors. The coprocessors feature hardware parallelism through both Very Long Instruction Words and Single Instruction, Multiple Data instructions. Each of the configurations is evaluated in terms of execution speed and hardware cost.

The following chapter explains the fundamental concepts and techniques which are used in subsequent chapters. Chapter 3 introduces the reference filter system, including it's individual components, functions and mathematical background. Chapter 4 proposes a new implementation of the filter system on six Application-Specific Instruction-Set Processor configuration, discusses implementation details and decisions, and gives an overview of the hardware features. Chapter 5 presents an evaluation and analysis of the proposed filter system implementation and it's performance on each of the processor configurations. Finally, the thesis is concluded in chapter 6.

2 Fundamentals

This chapter aims to introduce the fundamentals concepts which are used in the later chapters. First, an overview of two primary topics of this thesis is given, which are order analysis and Application-Specific Instruction-Set Processors. Next, two techniques for hardware parallelism are introduced, which will be used by the implementation. Finally, an introduction to fixed point numbers is given, which includes both representation and arithmetic.

2.1 Order Analysis

In order to monitor rotating machinery, an analysis of the vibrations produced by the machine can often reveal faults or wear on parts that would have otherwise stayed undetected. Defective parts tend to produce characteristic frequencies in the overall vibration signal, that a proper analysis can pick out. A technique that is often used for this analysis is Order Tracking or Order Analysis.

The vibration signal from a machine (e.g. a gearbox) is generally not stationary but varies with the rotational speeds that are present. Simply transforming the raw vibration signal into frequency domain will introduce errors due to the fact that the frequencies might not stay constant during a single frame of a Fourier Transform. These errors are called *smearing*. To prevent frequency smearing, the vibration frequencies have to be viewed proportional to the rotational speed of the machine. Figure 2.1 shows the effects of smearing, where the frequency spectrum has a slightly fuzzy look to it while the order spectrum is more crisp. The figure also shows that the frequency spectrum depicts the raw absolute frequencies of the signal as the base frequency sweeps up and down, while the order spectrum transforms them proportionally to the base frequency. The y-axis of the order spectrum shows the *orders*, which are multiples of the base frequency (e.g. rotational speed).

The algorithm used in figure 2.1 is called *Computed Order Tracking*. This method works by resampling a vibration signal at varying rates to get a signal that is independent from the rotational speed of the machine. The rates and points at which the vibration signal needs to be sampled at can be computed e.g. from a tachometer signal. The spectrum is then calculated from this resampled signal, instead of the original one. This spectrum then forms the basis for further and more application-specific analysis. An advantage of Computed Order Tracking is that it shows a full spectrum of orders of the vibration signal. The downside of this is that the orders of the signal can be difficult to separate from each other.

An alternative method for order tracking is the *Vold-Kalman Filter*. While Computed Order Tracking transforms a vibration signal into an order spectrum showing multiple order at once, the Vold-Kalman filter extracts only a single order at a time, but operates entirely in time-domain. This difference in operation means that some issues with Computed Order Tracking are circumvented. For example, when the base frequency changes quickly, the resolution of the order spectrum will degrade. And overall, the range of an order spectrum is limited by the frame size of the

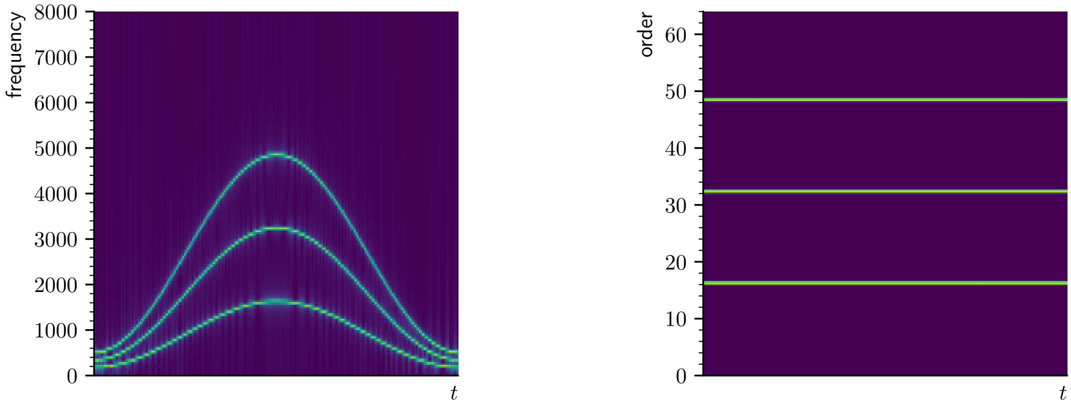


Figure 2.1: Comparison between a frequency spectrum (left) and an order spectrum (right) of a simulated ramp-up and coast-down.

underlying Fourier Transform, whereby higher orders require larger frame sizes. Because the Vold-Kalman filter works in time-domain only, these issues do not apply to it. Furthermore, because the Vold-Kalman Filter returns individual orders, there is no need for separating orders in the spectrum[7, 9].

2.2 Application-Specific Instruction-Set Processors

Knowing in advance which kinds of applications are to be ran on a given system opens a unique opportunity for optimization. By knowing which operations are commonly used in a given code base, it is possible to design custom hardware that is targeted to run these operations with high performance.

When designing a custom architecture for a specific application, a designer has to choose how closely the hardware should match that application. One possibility is to create a completely custom chip which directly implements the targeted application in hardware (Application-Specific Integrated Circuit), which yields very high performance at the cost of very low flexibility. An alternative is to design a customized processor which can execute a wider variety of programs, but is targeted to run a specific kind of application with high performance. The advantage of this approach is that it is easier to apply fixes and updates to the application without having to design and manufacture a new IC. Such a processor is called an Application-Specific Instruction-Set Processor (ASIP).

One way to further simplify the design process is to not design a completely new architecture from scratch, but to instead licence an existing templated ASIP design and customize that. An example for this is the Cadence® Tensilica® Xtensa® LX7, which is the IP used in this thesis [3]. Common options include adding instructions, configuring the bit widths number of registers and the width of the load/store-unit. For example, a DSP application often benefits from having hardware support for multiplication or Multiply-Accumulate operations. Furthermore, instructions and Functional Units, which are not needed for an application can be excluded to save area and have higher power efficiency. Finally, a designer can create completely custom functional units, which are targeted for specific, performance-critical parts of the application.

The downside of using an ASIP is that it has to be specially manufactured, which incurs significantly higher costs than simply using an off-the-shelf architecture. Whether it is worth to invest in manufacturing a custom processor is generally determined by running the target application on a software simulator of the ASIP and collecting profiling information on how well the architecture performs [10].

2.3 Hardware Parallelism

This section focuses on two forms of hardware-parallelism which will be used by the architectures in chapter 4 to increase the throughput of the filter system. The first technique (Very Long Instruction Words) exploits parallelism in the instruction stream, while the second technique (Single Instruction, Multiple Data) focuses on parallelism on the data level.

2.3.1 Very Long Instruction Words

One possible approach to increase the performance of an architecture is to not limit it to a single sequential stream of instructions but to issue multiple instructions in parallel. This leads to the idea of Very Long Instruction Word (VLIW) architectures, which describes groups of multiple instructions packed into a single instruction word. These instructions are grouped ahead-of-time by the compiler, are read at once by the processor and are executed in parallel.

In hardware, VLIW can be realized by adding the capability to run multiple Functional Units (FUs) in parallel. This can and often does include duplication of certain FUs, to allow for greater flexibility in the scheduling of instructions. A VLIW architecture generally provides a number of *issue slots*, which represent a group of such FUs. Each issue slot takes a single instructions from a very long instruction word and executes it concurrently with the other slots. Issue slots can have differing capabilities, depending on which kinds of FUs are attached. For example, if an architecture has a data bus of limited width, there is little use in having a load/store unit in multiple slots when the bus can not handle multiple concurrent memory accesses. In this case only one issue slot will have the capability to access memory.

A main bottleneck of VLIW architectures is the register file, as it needs to be able to handle concurrent access from all issue slots. Adding additional read/write ports to a register file incurs hardware costs proportional to the square of the number of ports [11]. An alternative is to separate the register file and functional units into *clusters*, which are multiple smaller register files where each only connects to a subset of the FUs. This saves ports on the register files, but also adds additional overhead through the now necessary communication between register files.

Another challenge on VLIW architectures is that the instructions are scheduled statically, i.e. at compile time, which means that it is the task of the compiler to group instructions together that should be executed in parallel. In order to do that, the compiler has to be able to identify code parts that can be safely executed in parallel and resolve any dependencies between instructions. For example, if an instruction uses the result of the instruction directly preceding it, these two instructions have a *Read-After-Write dependency* and can not be run in parallel.

Finally, if the compiler cannot find any viable parallelism in the code, it inserts No-Operation Instructions (NOPs) into the unused slots. However, for highly sequential code this becomes

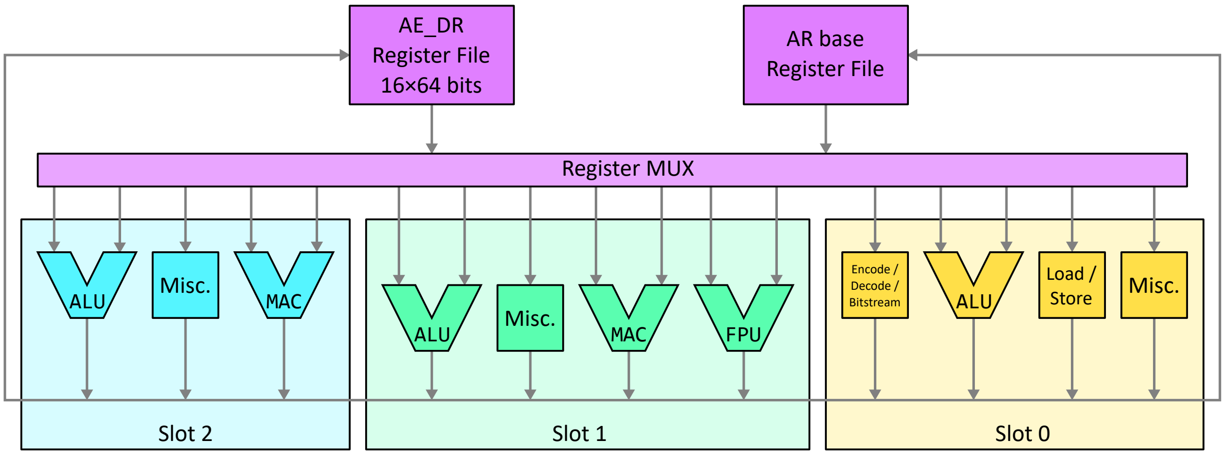


Figure 2.2: VLIW-Architecture of the HiFi3, showcasing the different Functional Units of each issue slot [1].

problematic. Because VLIW hardware generally requires an instruction for each issue slot, a high number of NOPs can increase the size of the compiled program significantly. A consequence of this is that a larger instruction memory might be needed and instruction caches will have more cache misses. There are different approaches to dealing with this problem, generally involving some encoding scheme to assign parts of the instruction word to issue slots [11].

2.3.2 Single-Instruction, Multiple-Data

When the same computation has to be performed on many data items, it is possible to parallelize the computation using Single Instruction, Multiple Data (SIMD) hardware. As the name implies, SIMD means to run the same set of instructions on multiple data items in parallel. This kind of parallelism is useful e.g. to speed up calculations on arrays, where the same operations are performed on each element independently. By using SIMD, multiple elements can be processed simultaneously.

There are multiple ways to utilize SIMD in code. A compiler can sometimes identify opportunities for vectorization in the code and generate the appropriate SIMD instructions automatically. However, higher performance is often reached when the programmer explicitly adds SIMD operations to the code. Compilers for SIMD-architectures generally provide specialized data types for vector registers, as well as intrinsic functions for SIMD instructions. The downside of this approach is that it often requires the programmer to program in an assembly-like fashion, resulting in more complicated code. Using SIMD intrinsics also limits the code to a specific family of instruction set architectures, meaning it can no longer be compiled for other architectures without modifications [12].

2.4 Fixed Point

In computing, two different representation formats for numbers have established themselves, which are integer and floating point. The integer format is used to hold only integer values, while the floating point format is used for fractional values. Most programming languages provide data types

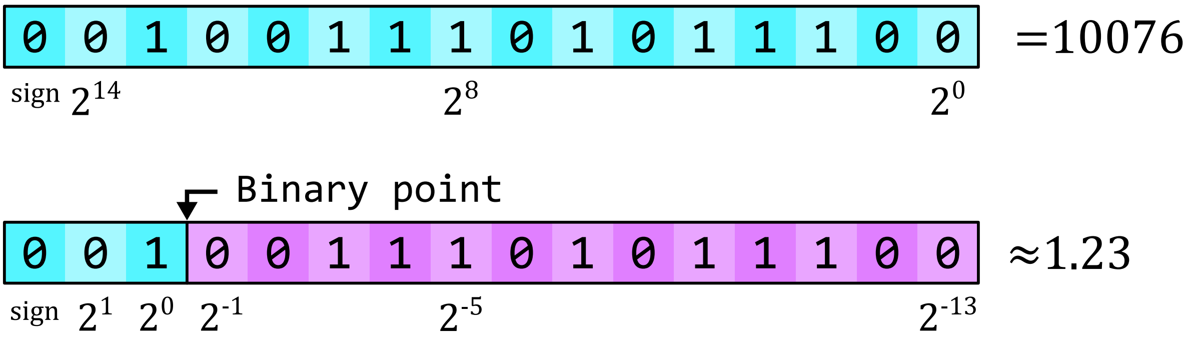


Figure 2.3: Binary number in integer format and Q3.13 fixed point format

Format	Bit representation	Actual value
Q16.16	`0000000000000000.0011001100110011`	$0.2-3.052\cdot 10^{-6}$
Q9.23	`000000000.00110011001100110011010`	$0.2+4.768\cdot 10^{-8}$
Q7.25	`0000000.0011001100110011001100110`	$0.2-1.192\cdot 10^{-8}$
Q5.27	`00000.001100110011001100110011010`	$0.2+2.980\cdot 10^{-9}$
Q3.29	`000.00110011001100110011001100110`	$0.2-7.451\cdot 10^{-10}$
Q1.31	`0.0011001100110011001100110011010`	$0.2+1.863\cdot 10^{-10}$
32-bit float	`0 01111100 10011001100110011001101`	$0.2+2.980\cdot 10^{-9}$

Table 2.1: The value 0.2 in different number formats

for these formats and modern general-purpose CPUs have hardware support for them, making computations in these formats very fast.

However, on embedded platforms and especially digital signal processors, a third format is frequently used which is the fixed point format. This format can hold fractional values, but still use the hardware for the integer format for it's arithmetic. By placing a decimal point after some number of bits, one can represent fractional numbers using the integer format. As can be seen in figure 2.3, this has the effect of shifting the values for each bit to the left and introducing fractional values on the right. Since fixed point numbers are inherently binary numbers, this decimal point is referred to as the *binary point*.

The position of the binary point determines the *fixed point format*. A format is classified by the number of *integer* bits, i.e. the number of bits to the left of the binary point, and the number of *fractional* bits, i.e. the number of bits to the right of the binary point. Since the placement of the binary point does not change the total number of bits, there is a trade-off between the precision of the fractional values and the range of possible values that can be represented. Larger numbers of fractional bits increase the precision of the values, but also reduce the range of values that can be represented through the format [13].

The notation for fixed point formats varies throughout the literature. This thesis uses the notation "Q$I.F$" to denote a fixed point format with I integer bits and F fractional bits. The number of integer bits does include a possible sign bit. Other notation schemes might assume the sign bit to always be present and therefore do not explicitly count it.

Table 2.1 shows how the accuracy of the fixed point number increases as more fractional bits are added. The value 0.2 has a binary expansion that is non-terminating, so it can never be represented exactly in a fixed point format. It should also be noted that some fixed point formats can achieve greater precision than floating point formats, depending on the range of values that is represented.

2.4.1 Arithmetic with Fixed Point Numbers

Up until this point, the difference between integer and fixed point formats was purely a matter of interpretation. The more practical differences become apparent when performing arithmetic on fixed point numbers.

Addition and Subtraction

Since integer hardware should be used for fixed point arithmetic, the hardware will interpret the fixed point numbers as integers. This change in interpretation means that when the programmer sees a (possibly fractional) fixed point value u, the hardware will see the same value as $u \cdot 2^F$, where F is the number of fractional bits in the fixed point format. Therefore addition and subtraction are performed as follows, assuming two fixed point values u and v in format $QI.F$:

$$u \cdot 2^F \pm v \cdot 2^F = s \cdot 2^F \tag{2.1}$$

As can be seen, both the operands and the result are scaled by the factor 2^F, meaning that the result has the same fixed point format as the two operands. Therefore, no additional modifications or corrections are necessary to obtain the correct result from an addition. In fact, addition and subtraction of fixed point numbers works exactly as it does for integers. However, because using a fixed point format means trading integer bits for fractional bits, overflows will happen at lower values. Therefore, one of the challenges with using fixed point number is figuring out which range of values a variable can have in a program. Some processors also support *saturating arithmetic*, where an overflow will instead result in the minimum/maximum fixed point value [14].

Multiplication

The first operation that differs from integer arithmetic is multiplication. Again, integer hardware is used for multiplication, so the hardware will perform the following calculation:

$$(u \cdot 2^F) \cdot (v \cdot 2^F) = p \cdot 2^{2F} \tag{2.2}$$

Equation 2.2 shows that when multiplying two $QI.F$ numbers, the result will have the format $Q2I.2f$. This highlights two potential problems that need to be addressed: First, the number of bits required to store the number doubled during the calculation. This is a general property of integer multiplication and not specific to fixed point. Some possible ways to address this issue are to store the result of a multiplication in a register with a greater bit width or to restrict the bit widths of the operands to ensure that the result will fit.

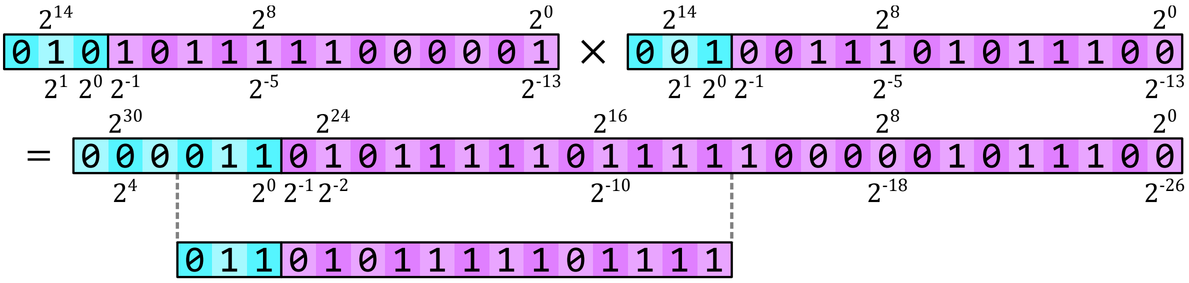


Figure 2.4: Multiplication of $2.7423 \cdot 1.23 = 3.3731$ in Q3.13 format

The second problem with fixed point multiplication is that the format of the result is different from the format of the operands. That means that 1) the bits need to be shifted right by F and 2) I bits on the left need to be truncated. Figure 2.4 shows this process for a 16×16 bit multiplication with fixed point format Q3.13. The truncation can also be realized by storing the result in a register of lower bit width [14]. Depending on the hardware, an opportunity for optimization arises here: since addition does not change the fixed point format, the programmer might choose to postpone the correction shift, for example when the results of many multiplications are added together. This optimization is useful e.g. for calculating convolutions.

Division

Another operation that results in a different fixed point format from the operands is division. Again using integer hardware, the following calculation is performed:

$$\left\lfloor \frac{u \cdot 2^F}{v \cdot 2^F} \right\rfloor = \lfloor q \rfloor \tag{2.3}$$

As can be seen from equation 2.3, conversely to multiplication, division reduces the number of bits in the result, because the factor 2^F cancels out. This essentially produces a number in a fixed point format with $F = 0$ fractional bits, which is impractical. To solve this problem, the numerator needs to be shifted left by F bits prior to the division in order for the result to have correct format. Equation 2.3 thus becomes the following:

$$\left\lfloor \frac{u \cdot 2^F \cdot 2^F}{v \cdot 2^F} \right\rfloor = \lfloor q \cdot 2^F \rfloor \tag{2.4}$$

Figure 2.5 shows the division process for two 16-bit numbers in Q3.13 format. Similar to multiplication, a register large enough to hold the shifted numerator is required for division [14].

CORDIC algorithm

For more complex functions, the Coordinate Rotation Digital Computer (CORDIC) algorithm is used. This algorithm can be configured to compute various common mathematical functions like square roots or trigonometrical functions. One of the most prominent uses of CORDIC was by

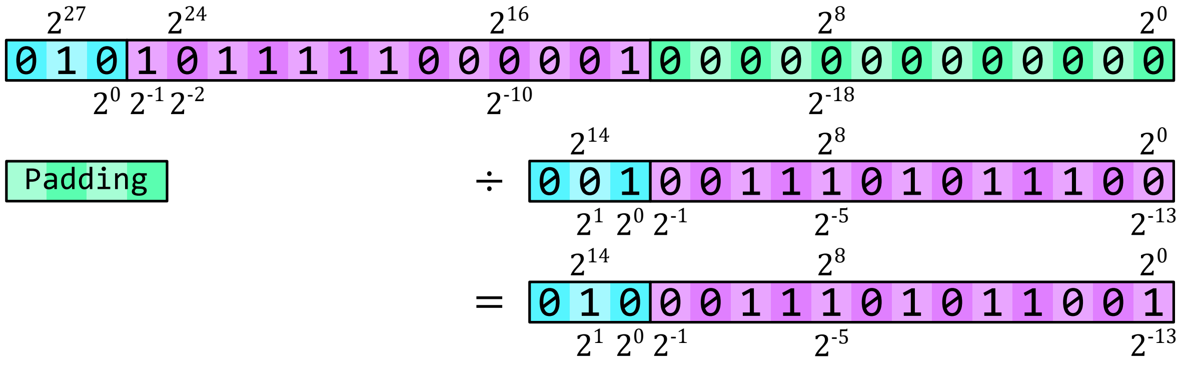


Figure 2.5: Division of $2.7423/1.23 = 2.2296$ in Q3.13 format

the Intel® 8087 math coprocessor, which largely contributed to the implementation of floating point arithmetic on modern desktop processors [15].

The CORDIC algorithm keeps track of a vector in 2D-space (x,y) as well as an accumulator z. Each iteration of the algorithm, vector (x,y) is rotated around the origin until a certain condition is met. The accumulator z records the angle that the vector has been rotated by so far. Therefore, the following equations form the basis for the CORDIC algorithm:

$$\begin{aligned} \begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} &= \begin{pmatrix} \cos\delta_n & -\sin\delta_n \\ \sin\delta_n & \cos\delta_n \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix} \\ z_{n+1} &= z_n + \delta_n \end{aligned} \tag{2.5}$$

The rotation matrix poses a problem because by this point there is no method to compute the sin and cos of a number (which is what the CORDIC algorithm is supposed to do). Therefore another method is needed to compute the rotation matrix which does not involve trigonometrical functions. Using the following identity

$$\begin{aligned} & (\cos\theta)^2 + (\sin\theta)^2 &=& \; 1 \\ \Leftrightarrow\; & (\cos\theta)^2 + \frac{(\cos\theta)^2}{(\tan\theta)^2} &=& \; 1 \\ \Leftrightarrow\; & (\cos\theta)^2 &=& \; \frac{1}{1+(\tan)^2} \\ \Leftrightarrow\; & \cos\theta &=& \; \frac{1}{\sqrt{1+(\tan\theta)^2}} \end{aligned} \tag{2.6}$$

the rotation matrix changes to

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \frac{1}{\sqrt{1+(\tan\delta_n)^2}} \begin{pmatrix} 1 & -\tan\delta_n \\ \tan\delta_n & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix} \tag{2.7}$$

This modification might seem counter-intuitive at first, since the sin and cos functions are replaced by tan, which is also not easily computed. However, by choosing particular values for the δ_n, it is possible reduce equation 2.7 to additions and bit shifts. In order to realize the multiplication with $\tan\delta_n$ with a bit shift, the result of $\tan\delta_n$ has to be a power of 2. That means that the values for δ_n are:

$$\delta_n = \arctan(\sigma_n \cdot 2^{-n}) = \sigma_n \cdot \arctan(2^{-n}) \tag{2.8}$$

The parameter σ_n determines the direction of rotation. It is set for each iteration individually to either $+1$ (counter-clockwise) or -1 (clockwise), depending on the mode of operation and the values of y_n or z_n. Because both tan and arctan are odd functions ($f(-x) = -f(x)$), this parameter can be moved outside of the functions. The values for $\arctan 2^{-n}$ can then be computed ahead-of-time and stored in a lookup-table. Substituting equation 2.8 into equation 2.7 results in the tan and arctan cancelling out:

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \frac{1}{\sqrt{1+(\sigma_n \cdot 2^{-n})^2}} \begin{pmatrix} 1 & -\sigma_n \cdot 2^{-n} \\ \sigma_n \cdot 2^{-n} & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix} \tag{2.9}$$

This new matrix multiplication can be done entirely with additions and shifts. Without the factor $\frac{1}{\sqrt{1+(\sigma_n \cdot 2^{-n})^2}}$, the iteration equations for the CORDIC algorithm are

$$x'_{n+1} = x'_n - \sigma_n 2^{-n} y'_n \tag{2.10}$$
$$y'_{n+1} = y'_n + \sigma_n 2^{-n} x'_n \tag{2.11}$$
$$z_{n+1} = z_n + \sigma_n \arctan 2^{-n} \tag{2.12}$$

The parameter σ_n is determined by the operating mode of the algorithm. The CORDIC algorithm defines two modes: In *rotation mode*, σ_n is used to move the accumulator z_n towards 0. Therefore, σ_n is set to -1 if $z_n > 0$ and to $+1$ if $z_n < 0$. In *vectoring mode*, the parameter y_n is moved to 0, so σ_n is -1 if $y_n > 0$ and $+1$ if $y_n < 0$.

Because equations 2.10 and 2.11 do not include the factor $\frac{1}{\sqrt{1+(\sigma_n \cdot 2^{-n})^2}}$, the parameters x'_n and y'_n are scaled by the inverse of this factor each iteration. To arrive at the final values for x_n and y_n, the parameters have to be corrected by the accumulated scale factor after N iterations:

$$K_N = \prod_{n=0}^{N} \frac{1}{\sqrt{1+(\sigma_n \cdot 2^{-n})^2}} = \prod_{n=0}^{N} \frac{1}{\sqrt{1+2^{-2n}}} \tag{2.13}$$

The parameter σ_n can be omitted from the factor because of the squaring step. When the number of iterations is fixed beforehand, this correction factor can be precalculated similar to the values for arctan. For large values of N, this correction factor tends towards $K_\infty \approx 0.6072529$. The final step of the CORDIC algorithm is therefore to scale the parameters x'_N and y'_N by the correction factor:

$$x_N = K_N \cdot x'_N \tag{2.14}$$
$$y_N = K_N \cdot y'_N \tag{2.15}$$

The algorithm so far can rotate a vector (x, y) either by a given angle or until the y component is equal to 0. The next step is to use this algorithm to synthesize different mathematical functions. For example, if the vector $(1, 0)$ is rotated by a given angle φ, the y component of the result will be the sine of φ and the x component will be the cosine. Therefore, by setting the parameters $x_0 = 1$, $y_0 = 0$ and $z_0 = \varphi$, as well as putting the algorithm into rotation mode, this creates a method to approximate the sine and cosine functions.

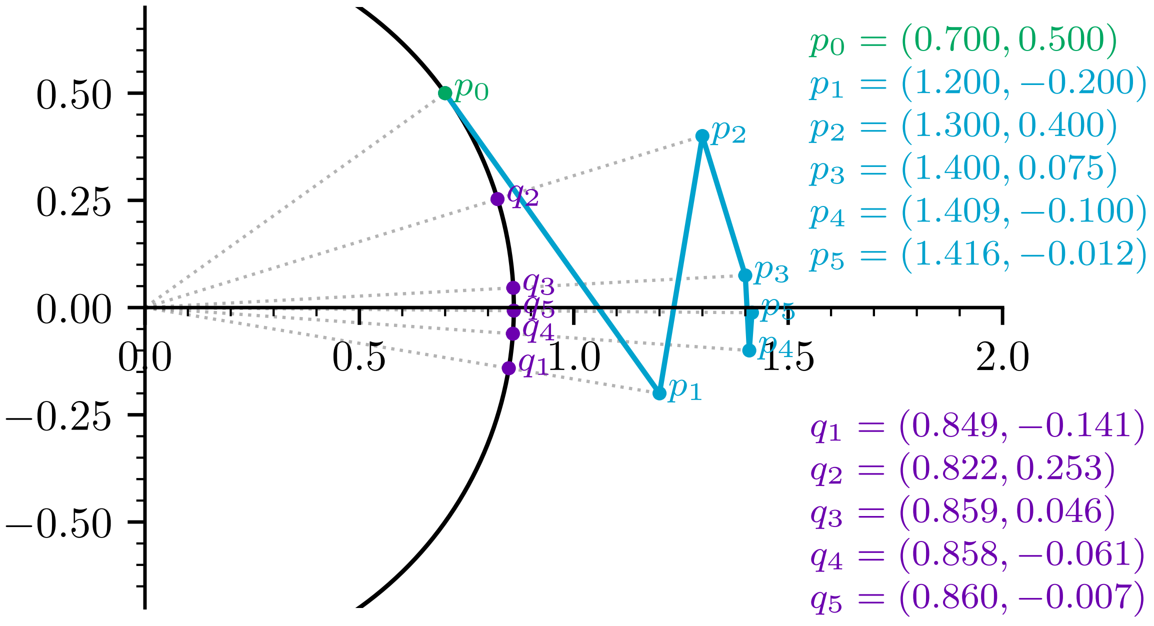


Figure 2.6: Example of the CORDIC algorithm in vectoring mode to compute the magnitude of $(0.7, 0.5)$

Another function which will be important for the implementation is calculating the magnitude of a vector. For this, the x_0 and y_0 parameters are set to the given vector and the algorithm is put into vectoring mode. In this mode, the vector is rotated to be as parallel as possible to the x-axis, so the magnitude can be read from the x-component of the result [16]. Figure 2.6 shows this process for the vector $(0.7, 0.5)$. After five iterations, the x-component of q_5 contains the approximation $|p_0| \approx 0.860$. The vectors p_n contain the values for x'_n and y'_n, while the vectors q_n contain the scaled values x_n and y_n.

3 Structure of the Filter System

In this chapter, the filter system of which the implementation is subject to optimization will be described. The first sections aim to give a general overview of the components of the system, while the later sections describe each component in more detail, including the mathematical background.

3.1 Overview

The filter system consists of several components which together form a filter chain that performs *Computed Order Tracking*. The individual components are a band-pass filter, an envelope detector, two decimation filters, an interpolation filter and a Fourier Transform. In the following, an overview of the connections between these parts is given.

3.1.1 X- and Y-Values

Throughout the following sections, the concept of *x- and y-values* will be used extensively to differentiate between the two sequences of inputs that the filter system receives. The x-values are either a timestamp or an angular displacement associated with a y-value. Whether the x-values are timestamps or angles determines whether the output of the filter system is a frequency spectrum or an order spectrum. The y-values are the signal from one of the three axes of an accelerometer, which is selected beforehand. These two sequences are called x- and y-values because of their use in the interpolation filter (section 3.5).

With the exception of the interpolation filter, most of the stages of the filter system only deal with the y-values. The x-values are simply passed along in a way to ensure that they stay aligned with their corresponding y-values. In figure 3.1, the x-values are passed through the components on the left-hand side while the y-values are passed through the ones on the right.

3.1.2 Signal Flow

Figure 3.1 shows a high-level overview of the filter system, along with the configuration parameters of the individual components.

The system starts by selecting either the timestamps or positions for the x-values, depending on whether a Fourier transform or order analysis should be performed. Next, the y-values are filtered through a band-pass filter (Section 3.2), which selects the frequency band which is of interest.

After the band-pass filter, an envelope detector can optionally be added to the filter chain (Section 3.3). For certain applications, like detecting faults in rolling element bearings, order analysis on

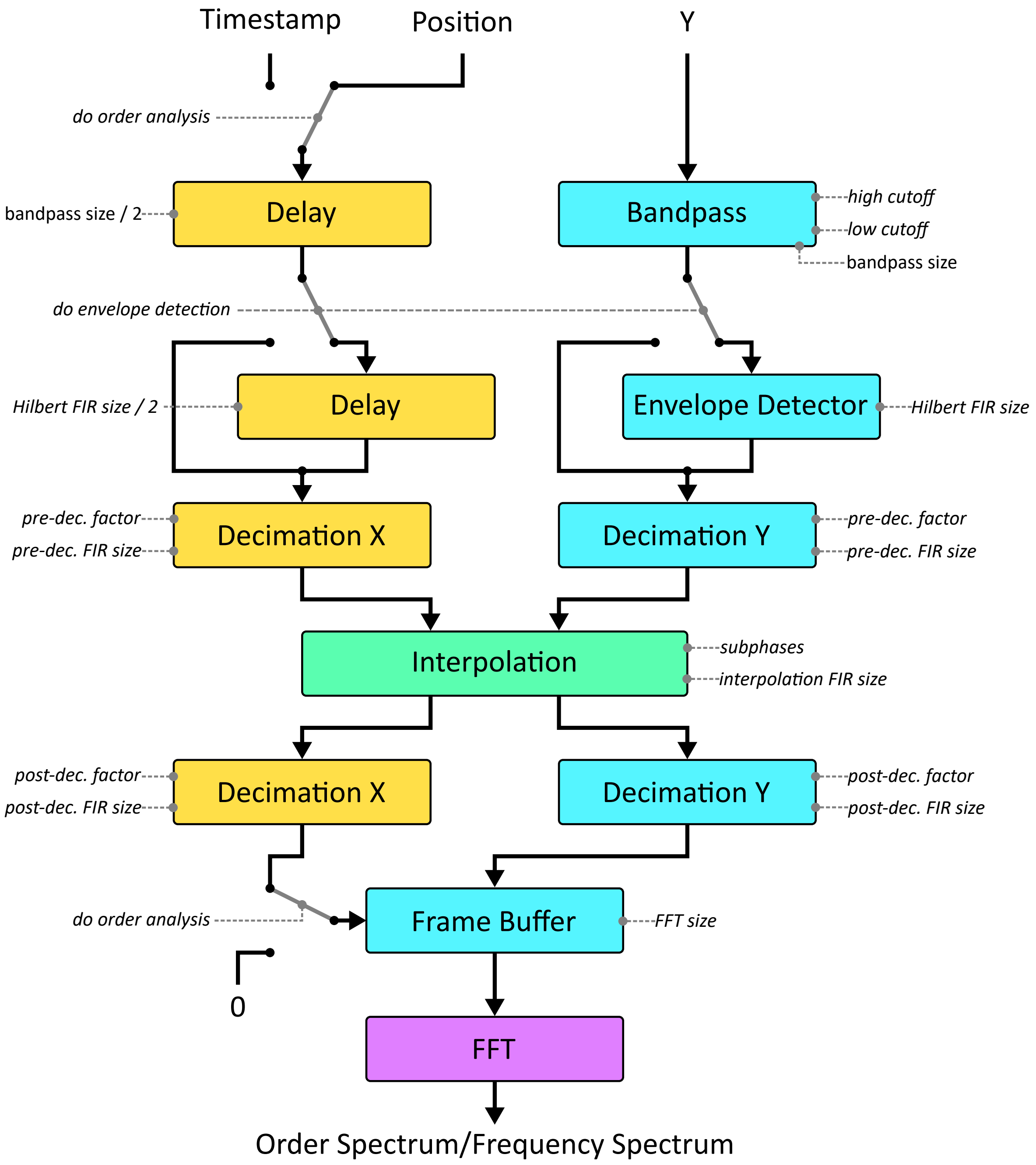


Figure 3.1: Overview of the filter system.

the signal's envelope is performed, while other applications require the raw signal. Therefore the envelope detector can be bypassed with a switch.

In the next stage, the sampling rate of the signals is lowered. This is done because the following components are computationally expensive and a lower sampling rate results in faster execution. The down-sampling is performed by a decimation filter (Section 3.4). The factor to decimate by is determined from the requested resolution of the output spectrum.

The next component is the interpolation filter (Section 3.5). It has the function of interpreting the two input sequences as a single signal with a non-constant sampling rate and transforming it into a constant-rate signal. It is this transformation which differentiates the output of the filter system from a simple Fourier transform. Because this interpolation process oversamples to prevent aliasing, another decimation filter is added to lower it to the requested sampling rate.

The final two components perform the transformation into frequency space. The first of the two components is a buffer which has the function to collect incoming values until the amount of data is large enough to perform the FFT. If order analysis is performed, this buffer includes a special trigger function, which only starts collecting data after the x-values have reached a specific angle. When the buffer is full, it is forwarded to the FFT (Section 3.6).

3.2 Band-pass Filter

The first stage of the filter system is a band-pass filter, which is used to select the frequency band of interest. The filter is created from by taking the difference between two FIR low-pass filters.

$$h_n = W(n)(LP(f_{high})_n - LP(f_{low})_n) \tag{3.1}$$

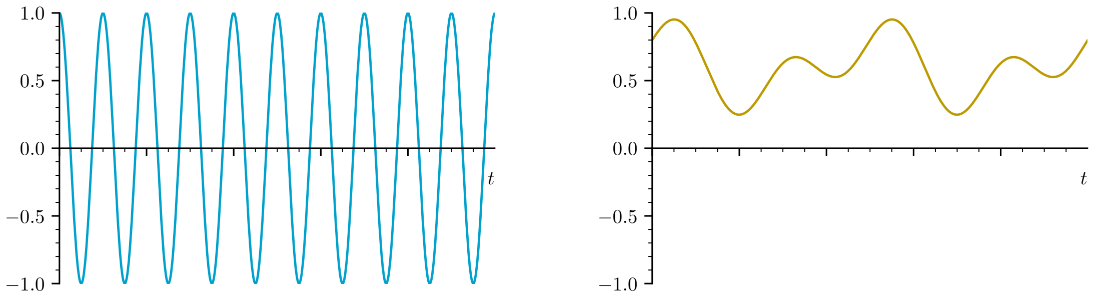
where

- $LP(f)$ are the coefficients of a low-pass filter with cutoff frequency f
- $W(n)$ is a window function. The filter system uses a Blackman-Harris-Window.
- h_n are the coefficients of the band-pass filter.

The low-pass filter is implemented using a *sinc* function. Because the coefficients will be stored in an array, all indices need to be positive. This means that the coefficients need to be shifted so that all values of n are greater or equal to 0. As a consequence of this, the filtered signal will be delayed by $(N_{bp} - 1)/2$, where N_{bp} is the length of the filter. Since only the y-values are passed through the band-pass, the x-values have to be delayed as well to compensate. This pattern of the y-values being filtered and the x-values being delayed is also common in the other components which use some form of FIR-filter.

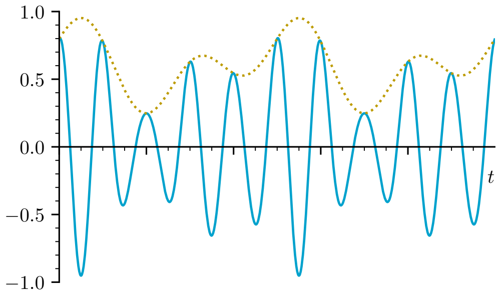
3.3 Envelope Detector

Since different parts of a machine can fail in different ways, some of them (e.g. rolling element bearings) require a special variant of order analysis called *envelope order analysis*. This method is similar to conventional order analysis, but is instead performed on the signal's envelope instead of



(a) Carrier signal $c(t)$

(b) Envelope signal $m(t)$



(c) AM signal $s(t)$

Figure 3.2: Example for Amplitude Modulation

the signal itself. Therefore, in order to support both order analysis and envelope order analysis, the signal path includes an optional envelope detector to extract the amplitude envelope of the y-values.

In order to explain how the envelope detector works, the following signal is considered [17]:

$$s(t) = m(t)c(t) \tag{3.2}$$

where (cf. figure 3.2)

- $c(t) = \cos(\omega_c t + \varphi_c)$ is a carrier signal, where ω_c is the frequency and φ_c is the phase offset.
- $m(t)$ is an arbitrary envelope signal.
- $s(t)$ is the result of the Amplitude Modulation (AM) of the envelope $m(t)$ onto the carrier signal $c(t)$.

The goal of the envelope detector is to retrieve the envelope $m(t)$ from the AM signal $s(t)$. While there are several different methods to achieve this, the filter system uses an algorithm based on the analytic representation of the signal $s(t)$, which is explained in the following.

3.3.1 Analytic Signals and the Hilbert Transform

An analytic signal is a signal which has no negative frequency components, i.e. the left half of it's spectrum is all zeroes. Therefore, to turn a signal into an analytic signal, the negative frequencies need to be removed. One possible way to achieve this is to add an imaginary component equal to the original signal, but where all frequencies are phase-shifted by one quarter of their period. In case of a basic cosine function, this corresponds to a sine function with the same frequency [18]:

$$\begin{aligned} &\cos(\omega_0 t) + j\cos(\omega_0 t - \pi/4) \\ =&\cos(\omega_0 t) + j\sin(\omega_0 t) \\ =&e^{j\omega_0 t} \end{aligned} \tag{3.3}$$

The last step is based on Euler's Formula:

$$e^{jx} = \cos(x) + j\sin(x) \tag{3.4}$$

Equation 3.3 shows that this modification collapses the cosine function into a single positive frequency component but it also turned the real-valued cosine into a complex-valued function in the process. This is a trade-off that has to be considered when working with an analytic signal.

To see how an analytic signal is created for any arbitrary function, the previous equation is transformed into frequency domain:

$$\begin{aligned} &\mathscr{F}\{\cos(\omega t) + j\sin(\omega t)\} \\ =&\mathscr{F}\{\cos(\omega t)\} + j\mathscr{F}\{\sin(\omega t)\} \\ =&\frac{\delta(\omega-\omega_0)+\delta(\omega+\omega_0)}{2} + j\frac{\delta(\omega-\omega_0)-\delta(\omega+\omega_0)}{2j} \end{aligned} \tag{3.5}$$

To see more clearly what is happening here, some of the terms are rearranged:

$$\begin{aligned} &\frac{\delta(\omega-\omega_0)+\delta(\omega-\omega_0)}{2} + \frac{\delta(\omega+\omega_0)-\delta(\omega+\omega_0)}{2} \\ =&\delta(\omega-\omega_0) \end{aligned} \tag{3.6}$$

As can be seen, the positive frequency component $\delta(\omega-\omega_0)$ is added to itself, resulting in a doubling while the negative frequency component $\delta(\omega+\omega_0)$ is subtracted from itself, resulting in a cancellation.

In order to generalize this formula for arbitrary signals, the final step of equation 3.5 is separated into it's individual components $\frac{1}{2}\delta(\omega-\omega_0)$ and $\frac{1}{2}\delta(\omega+\omega_0)$. These components can be matched to parts of the spectrum of the original cosine function: The positive half of the spectrum corresponds to $\frac{1}{2}\delta(\omega-\omega_0)$ and the negative half to $\frac{1}{2}\delta(\omega+\omega_0)$, each of which appear twice in the equation.

For the next step, an arbitrary signal $f(t)$ with spectrum $F(\omega)$ is considered. Let the positive half of the spectrum be $F_+(\omega)$ and the negative half $F_-(\omega)$. Now the final step of 3.5 can be used again but this time $F_+(\omega)$ and $F_-(\omega)$ are substituted in place of the positive and negative halves of the cosine spectrum:

$$(F_+(\omega)+F_-(\omega)) + j\left(\frac{F_+(\omega)-F_-(\omega)}{j}\right) \tag{3.7}$$

The contents of the left set of parentheses correspond to the original spectrum, while the contents of the right set is the expression we need to add in order to create the analytic representation. From the right-hand expression, the following operator can be derived:

$$\frac{F_+(\omega) - F_-(\omega)}{j} = F(\omega) \cdot H(\omega) = \mathscr{F}\{\mathscr{H}\{f\}\}(\omega) \tag{3.8}$$

where

$$H(\omega) = \begin{cases} -j & \text{for } \omega > 0 \\ 0 & \text{for } \omega = 0 \\ j & \text{for } \omega < 0 \end{cases} \tag{3.9}$$

This operator $\mathscr{H}\{f\}$ is known as the *Hilbert transform* and corresponds to a multiplication with $H(\omega)$ in frequency domain. It's function is to shift the phase of every frequency component by one quarter of it's period, similar to equation 3.3.

As a final step to create an analytic signal, equation 3.7 is transformed back into time domain [18]:

$$\begin{aligned} &\mathscr{F}^{-1}\left\{(F_+(\omega) + F_-(\omega)) + j\left(\frac{F_+(\omega) - F_-(\omega)}{j}\right)\right\} \\ =&\mathscr{F}^{-1}\{F(\omega) + j\mathscr{F}\{\mathscr{H}\{f\}\}(\omega)\} \\ =&f(t) + j\mathscr{H}\{f\}(t) \end{aligned} \tag{3.10}$$

As can be seen, the analytic representation has the original signal as the real part and it's Hilbert transform as the complex part. It will later be shown that the Hilbert transform can actually be computed using an FIR filter, which makes it possible to create an analytic signal without ever going through frequency domain.

Now all that is still needed is to tie the analytic signal in with envelope detection. The initial goal was to extract the envelope signal $m(t)$ from equation 3.2. The problem faced is that the parameters ω_c and φ_c of the carrier signal $c(t)$ are in most cases, including this one, assumed to be unknown. Furthermore, these parameters might also change over time so just dividing the AM signal by the carrier is generally not an option. It can be assumed however, that the carrier signal is some form of cosine function.

Converting the AM signal into an analytic signal yields the following:

$$\begin{aligned} s_a(t) &= m(t)\cos(\omega_c t + \varphi_c) + j\mathscr{H}\{m(t)\cos(\omega_c t + \varphi_c)\} \\ &= m(t)\cos(\omega_c t + \varphi_c) + jm(t)\sin(\omega_c t + \varphi_c) \\ &= m(t)e^{j\omega_c + \varphi_c} \end{aligned} \tag{3.11}$$

The first step made use of the property that the Hilbert transform shifts the phases of the frequency components by a quarter period, which means that the cosine function becomes a sine function. Following that, Euler's formula (3.4) was applied.

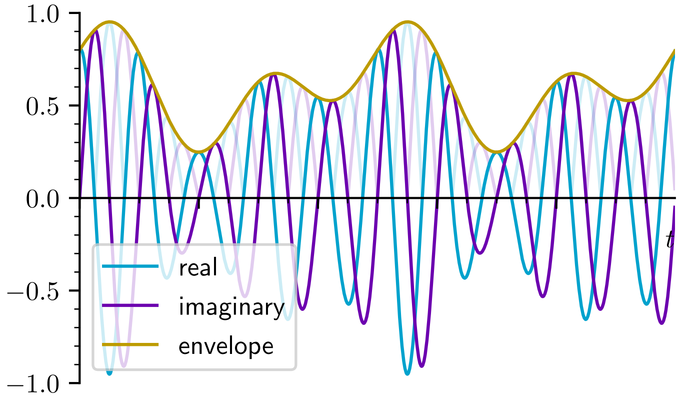


Figure 3.3: The real and imaginary parts of an analytic signal.

It can now be seen that the final step required for extracting the envelope signal is taking the absolute value (cf. figure 3.3):

$$m(t) = |m(t)e^{j\omega_c + \varphi_c}| = |s_a(t)| \tag{3.12}$$

3.3.2 The Discrete Case

Up until now all signals have been continuous, however a practical implementation requires discrete (sampled) signals. For this reason, the Hilbert transform has to be adapted to work in a discrete time environment. Since the Hilbert transform can be performed by a multiplication in frequency domain, it is reasonable to assume that it can also be implemented as a convolution in time domain. In that case, the aim should be for a form like this:

$$\mathscr{H}\{x\}_n = x_n * h_n \tag{3.13}$$

As a first idea to get the values for the h_n, it might be possible to calculate them similarly to the band-pass filter, by simply sampling the continuous-time representation of the Hilbert transform, which is

$$h(t) = \frac{1}{\pi t} \tag{3.14}$$

However, there is problem with this approach: equation 3.14 is undefined for $t = 0$. When performing a continuous convolution, this problem can be circumvented by using the *Cauchy principle value* for the convolution integral, a method which is not possible in discrete time [19]. One possible way to resolve this issue for discrete convolutions is to offset the location of the samples by 0.5 creating a sequence for t like $(..., -1.5, -0.5, 0.5, 1.5, ...)$. This yields the coefficients shown in figure 3.4.

Unfortunately, this approach creates another problem, which is related to the analytic signal. Since the analytic signal is built from both the original signal and the Hilbert transform of it, this

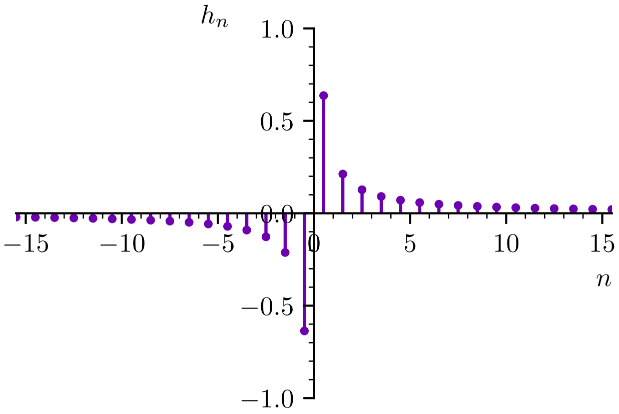


Figure 3.4: Hilbert coefficients from sampling the continuous-time Hilbert transform

means that these two signal needs to be aligned in time with each other. Offsetting the sample locations by 0.5 introduces a half-sample delay into the filter which needs to be accounted for in the original signal as well, which in turn requires interpolation.

However, there is another way to calculate the Hilbert filter coefficients, which does not introduce a fractional delay. This approach makes use of the frequency domain representation of the Hilbert transform. The spectrum of the Hilbert transform shown in equation 3.9 is continuous, so in order to make use of it in equation 3.13, a continuous spectrum for x_n is needed as well. For that the Discrete Time Fourier Transform (DTFT) can be used, which transforms a discrete function in time domain into a continuous function in frequency domain. The DTFT is defined as

$$DTFT\{f\}(\omega) = \sum_{-\infty}^{\infty} x_n e^{-j\omega n} = F(\omega) \tag{3.15}$$

and the inverse DTFT as

$$DTFT^{-1}\{F\}(\omega) = \int_{-\pi}^{\pi} F(\omega) e^{j\omega n} = f_n \tag{3.16}$$

Using the DTFT, equation 3.13 becomes [20]

$$\begin{array}{rrcl} & DTFT\{\mathscr{H}\{x\}_n\}(\omega) & = & DTFT\{x_n\}(\omega) \cdot H(\omega) \\ \Leftrightarrow & \mathscr{H}\{x\}_n & = & x_n * DTFT^{-1}\{H(\omega)\} \end{array} \tag{3.17}$$

It can now be seen that inverse DTFT of $H(\omega)$ can be used as coefficients for a discrete Hilbert transform. Substituting the definition for the inverse DTFT from equation 3.16 yields:

$$\begin{aligned} h_n &= DTFT^{-1}\{H(\omega)\} \\ &= \frac{1}{2\pi} \int_{-\pi}^{\pi} H(\omega) e^{j\omega n} d\omega \end{aligned} \tag{3.18}$$

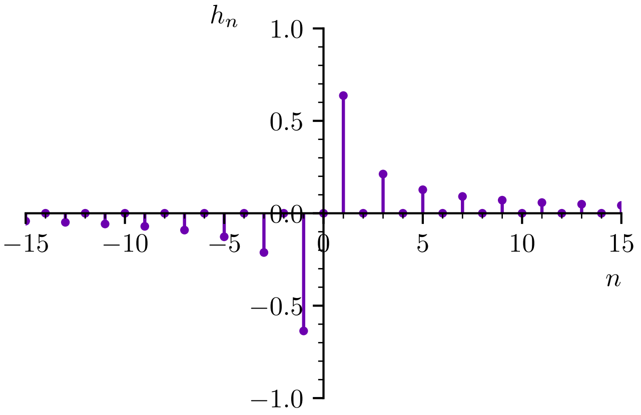


Figure 3.5: Hilbert coefficients from the inverse DTFT

As a first step, the multiple cases of equation 3.9 are resolved, which splits the integral into three:

$$h_n = \frac{1}{2\pi}\left(\int_{-\pi}^{0} je^{j\omega n}d\omega + \int_{0}^{0} 0\cdot e^{j\omega n}d\omega - \int_{0}^{\pi} je^{j\omega n}d\omega\right) \tag{3.19}$$

The middle integral is equal to zero, leaving two integrals that resolve to:

$$\begin{aligned} h_n &= \frac{1}{2\pi}\left(j\frac{1}{jn}e^{j\omega n}\Big|_{-\pi}^{0} - j\frac{1}{jn}e^{j\omega n}\Big|_{0}^{\pi}\right) \\ &= \frac{1}{2\pi}\left(\frac{1}{n}e^{j0n} - \frac{1}{n}e^{-j\pi n} - \frac{1}{n}e^{j\pi n} + \frac{1}{n}e^{j0n}\right) \end{aligned} \tag{3.20}$$

And after a few more simplifications, the equation becomes:

$$\begin{aligned} h_n &= \frac{1}{\pi}\left(\frac{1}{n} - \frac{1}{n}(-1)^n\right) \\ &= \begin{cases} 0 & \text{for } n \text{ even} \\ \frac{2}{\pi n} & \text{for } n \text{ odd} \end{cases} \end{aligned} \tag{3.21}$$

These coefficients are shown in figure 3.5. This definition for the Hilbert filter coefficients does not introduce a fractional delay. In addition, every other coefficient is equal to zero, which can make computing the discrete Hilbert transform using this method about twice as efficient as using the sampled time-domain Hilbert transform. Similar to the band-pass filter, the Hilbert filter also needs to be limited to a finite interval for n, as well as shifted to include an equal amount of coefficients for negative and positive values of n.

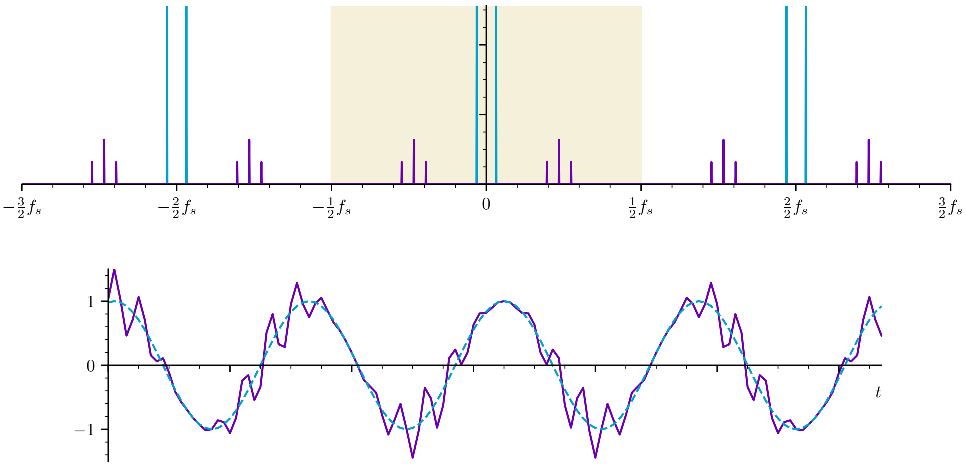


Figure 3.6: Example signal and spectrum

3.4 Decimation Filter

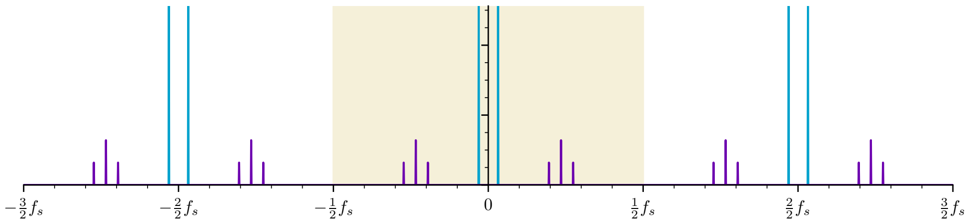
To reduce the computational load on the later components, a decimation filter is used to lower the sampling rate of the signals. In general, a decimation filter can be used to reduce the sampling rate of a signal by any positive integer factor. The process of applying the decimation filter will be explained with the following example (based on [21]).

Considering the example signal and spectrum in 3.6, the signal consists of a cosine at a low frequency as well as three more cosines at higher frequencies. Since a sampled signal inherently has a periodic spectrum, the continuous spectrum (the shaded area) is repeated infinitely across frequency space.

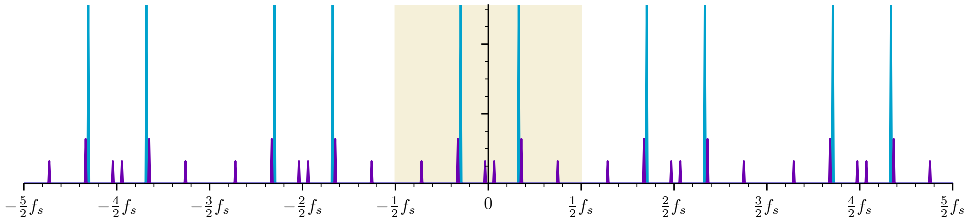
3.4.1 Aliasing

A first approach to decimate by a factor d would be to simply take every dth sample from the signal. However, this will yield incorrect results, because of the *Nyquist-Shannon sampling theorem*. This theorem states that a signal which contains no frequencies higher than $f_s/2$ is uniquely defined by samples spaced $1/f_s$ apart [22]. Following from this, a signal with a sampling rate of f_s will never contain frequency components higher than $f_s/2$.

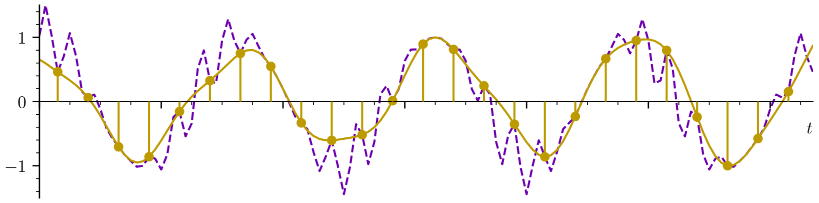
In this example, the decimation factor will be 5. After removing all but every fifth sample from the signal, the sampling rate has been reduced by $1/5$ and is no longer high enough to contain the three high frequency cosines. As can be seen in figure 3.7, the spectrums have moved closer together after the decimation, which caused frequencies from the higher-order spectrums to overlap with the center spectrum. Since the transform back into time-domain uses an inverse DFT and the DFT considers all frequencies from the shaded area of figure 3.7, parts of the higher order spectrums are caught in the transformation. The consequence of this is that the time-domain signal is not the expected low-frequency cosine, but instead also contains other unwanted frequencies from the higher-order spectrums. This effect is known as *aliasing* [21].



(a) Original spectrum



(b) Spectrum after decimation



(c) Signal after decimation

Figure 3.7: Example of aliasing.

3.4.2 Anti-aliasing Filter

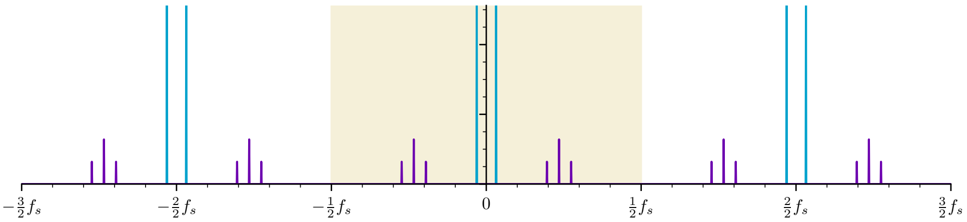
To avoid aliasing through frequencies from the higher order spectrums, the signal needs to be passed through a low-pass filter first. The task of this filter is to remove all frequency components which the sampling rate of the decimated signal would not be able to resolve. Decimation by a factor of d reduces the sampling rate of the signal by a factor of d. Similarly, the highest possible frequency component, which is $\frac{f_s}{2}$, becomes $\frac{f_s}{2d}$ after decimation. Therefore, if the coefficients of the anti-aliasing filter are defined like this:

$$l_n = W(n) \cdot 2f_c \cdot \mathrm{sinc}(2f_c n) \tag{3.22}$$

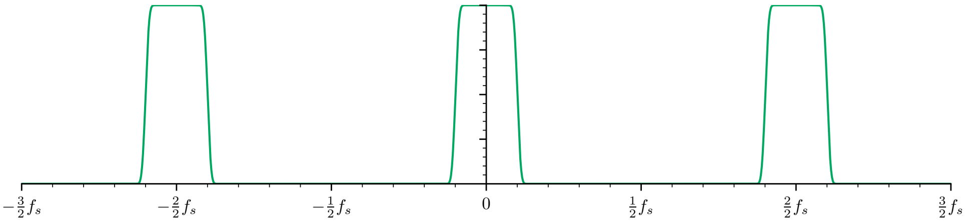
where

- l_n are the filter coefficients.
- $W(n)$ is a windowing function, in case of the reference filter system, a Blackman-Harris-Window.
- f_c is the cutoff frequency.

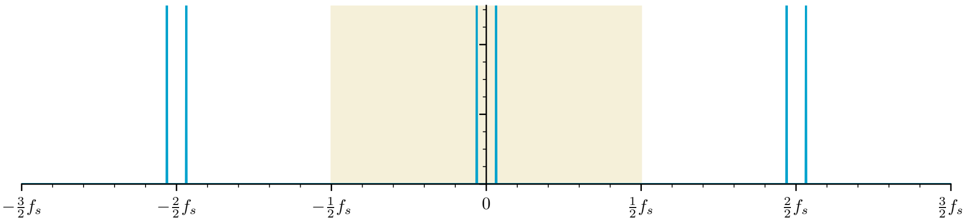
then the cutoff frequency f_c must not be greater than $\frac{1}{2d}$. Note that this value is independent from the sampling rate of the signal, since decimation only performs a *relative* modification to the sampling rate. An example of applying an anti-aliasing filter is shown in figure 3.8.



(a) Original spectrum



(b) Spectrum of the anti-aliasing filter



(c) Spectrum filtered with an anti-aliasing filter

Figure 3.8: Example of an anti-aliasing filter

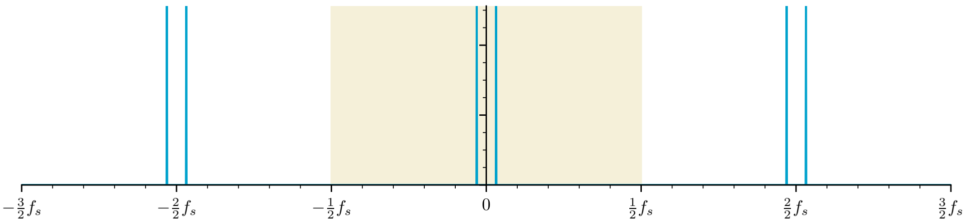
As can be seen in figure 3.9, after applying an anti-aliasing filter only the frequencies that can be resolved by the new sampling rate remain. Therefore, the effects of aliasing are greatly reduced.

3.4.3 Decimating the X-Values

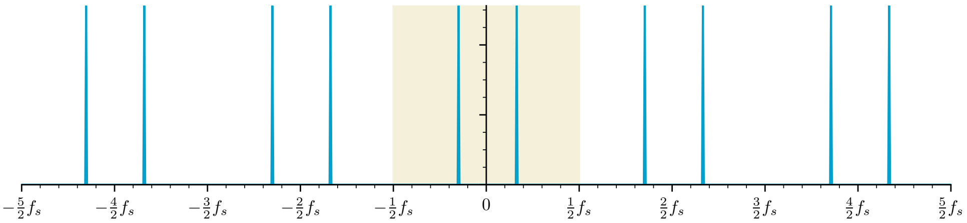
The filter system contains two different types of decimation filters, one for the y-values and one for the x-values. The filter for the y-values is implemented as described in the previous section. The filter for the x-values is similar, but does not contain an anti-aliasing filter. The reason for omitting the anti-aliasing filter for the x-values is, that it is simply not required in this case. While the y-values generally are a fast changing signal, the x-values represent either timestamps or angular displacement (depending on whether the user chose to perform an FFT or order analysis), both of which vary at a much slower rate. Therefore, the effects of aliasing are negligible for the x-values.

3.5 Interpolation Filter

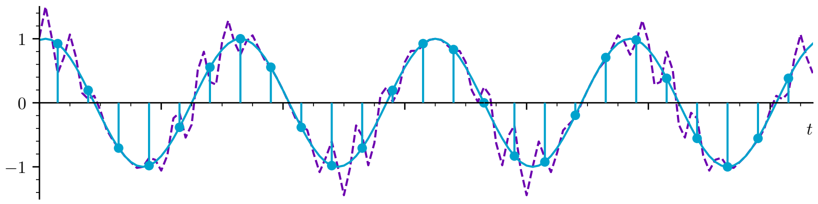
The interpolation filter is one of the most important parts of the filter system, since it is this component which makes the final output an order spectrum. The y-values that are pushed into



(a) Filtered spectrum



(b) Filtered spectrum after decimation



(c) Decimated signal after applying an anti-aliasing filter

Figure 3.9: Decimation with an anti-aliasing filter.

the system do not have a constant rate, instead their intended position in the signal is given by their corresponding x-values. The function of the interpolation filter is to transform this sequence into a constant rate signal, as shown in figure 3.10.

The implementation actually uses a slightly different but still equivalent formulation: Given a sequence of y-values at a constant rate, the goal is to map this sequence onto a non-uniform grid, where the distance between subsequent points is calculated from the corresponding x-values. In either case, values need to be calculated that lie in-between two samples. This poses a problem, since the input signal is in a discrete time-domain representation, where the position of a sample is defined by it's index in an array and the concept of "in-between indices" does not exist.

Fortunately, this problem can be solved by the sampling theorem: What the theorem states is that a signal which only contains frequencies less than $f_s/2$ is uniquely defined by samples spaced $1/f_s$ apart [22]. However, the theorem does not state that sampling always has to start at a multiple of $1/f_s$, which means that it is valid to add a constant offset to the location of the samples, creating a sequence like:

$$t_n = n \cdot \frac{1}{f_s} + c \tag{3.23}$$

If this sequence is used to sample a continuous signal, the samples will be spaced $1/f_s$ apart, fulfilling the requirement of the sampling theorem. In addition, since the sampling theorem

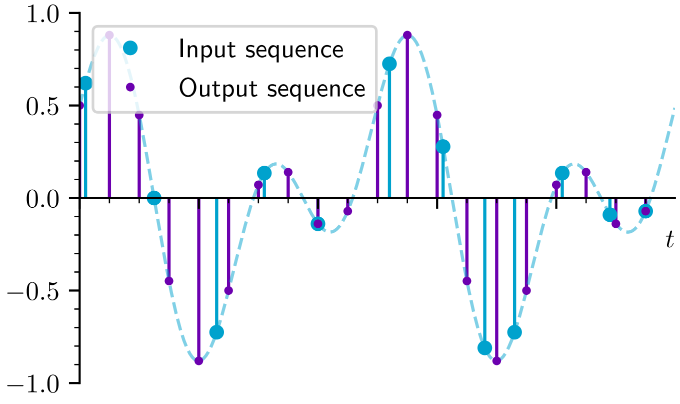


Figure 3.10: Function of the interpolation filter

guarantees uniqueness, the samples will always define the same signal, irrespective of the offset c chosen.

The key insight that will be used in the interpolation filter is that it is possible to change the offset c by only using the given samples [23]. In other words, while it is not possible to index a discrete signal at a fractional index, it is possible to shift the samples by a fractional amount, so that the desired location falls onto an integer index.

3.5.1 Fractional Delay Filters

In order to realize a fractional delay, a signal which can take such a delay is needed. For this reason the discrete signal needs to be made continuous first. This can be achieved by the following formula [22, 23]:

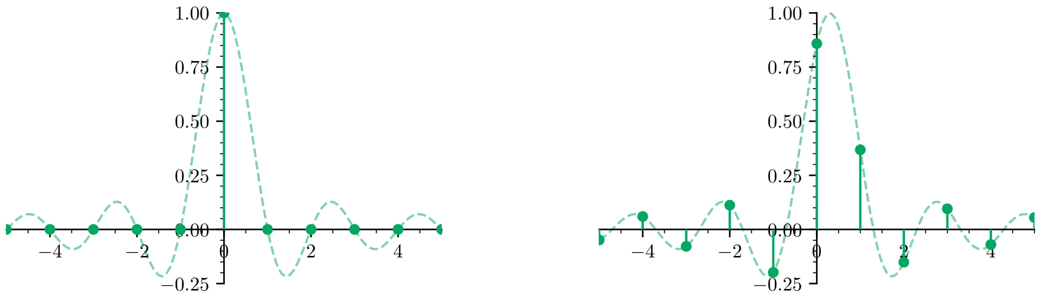
$$\begin{aligned} s(t) &= y_k * \operatorname{sinc}(f_s t) \\ s(t) &= \sum_{k=-\infty}^{\infty} y_k \cdot \operatorname{sinc}(f_s(t-k)) \end{aligned} \tag{3.24}$$

Now that the signal is continuous, it is possible to add any offset c to it:

$$s(t+c) = \sum_{k=-\infty}^{\infty} y_k \cdot \operatorname{sinc}(f_s(t+c-k)) \tag{3.25}$$

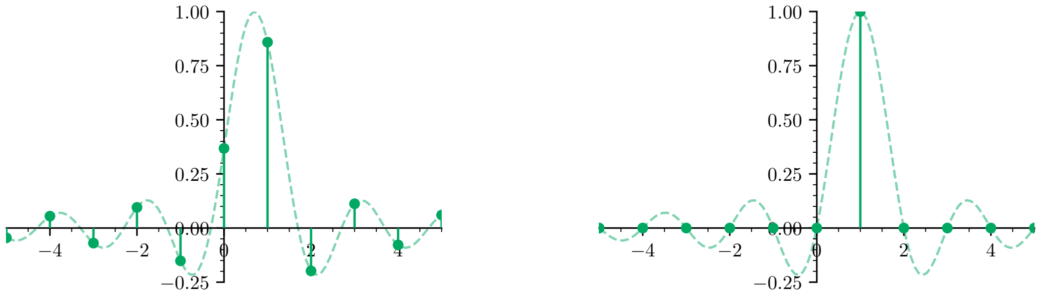
Now all that is needed is to sample the resulting signal again, to obtain a new discrete signal:

$$\begin{aligned} s\left(\frac{n}{f_s}+c\right) &= \sum_{k=-\infty}^{\infty} y_k \cdot \operatorname{sinc}\left(f_s\left(\frac{n}{f_s}+c-k\right)\right) \\ &= y_k * h_n \\ &= z_n \end{aligned} \tag{3.26}$$



(a) 0.0 sample delay

(b) 0.3 sample delay



(c) 0.7 sample delay

(d) 1.0 sample delay

Figure 3.11: Examples for fractional delay filters

As can be seen, it is possible to delay a discrete signal by any fractional amount by using a convolution with the following coefficients:

$$h_n = \operatorname{sinc}\left(f_s\left(\frac{n}{f_s}+c\right)\right) \tag{3.27}$$

Examples of filters for different delays are shown in figure 3.11. It is especially notable that for integer delays, the filter coefficients become a delta impulse, since the zero-crossings of the sinc function line up with the sample locations.

3.5.2 Algorithm

The method that the filter system uses for interpolation is given in algorithm 1. In addition to the input sequences x and y, where N denotes the length of these sequences, it makes use of the following parameters:

- D Length of the fractional delay filters. Ideal fractional delay filters are infinitely long, so the number of coefficients needs to be limited.
- T_{out} Distance between points in the output grid. This parameter determines the sampling rate of the output sequence.
- z_m Sequence of interpolated values. This is the output of the algorithm.

Algorithm 1 Realization of the interpolation filter

```
 1: procedure INTERPOLATE($x, y, N$)
 2:     delay $x$ by $(D-1)/2$ samples
 3:     $m \leftarrow 0$
 4:     for $n \leftarrow 1, N-2$ do
 5:         $x_{start} \leftarrow (x_{n-1} + x_n)/2$
 6:         $x_{end} \leftarrow (x_{n+1} + x_n)/2$
 7:         while $m \cdot T_{out} < x_{end}$ do
 8:             $c \leftarrow (m \cdot T_{out} - x_{start})/(x_{end} - x_{start})$
 9:             $\vec{h} \leftarrow$ Fractional delay filter with delay closest to $c$
10:             $\vec{y} \leftarrow (y_n, y_{n-1}, \ldots, y_{n-D+1})$
11:             $z_m \leftarrow \vec{h} \cdot \vec{y}^T$
12:             $m \leftarrow m + 1$
13:         end while
14:     end for
15: end procedure
```

In order to speed up the interpolation, the filter system does not calculate the coefficients for fractional delay filters for every possible delay, but instead limits the delays to a finite set of values. The amount of possible delay values determines how accurate the result is, where a higher number of delays will yield a more accurate result. The coefficients for each possible delay are calculated ahead of time, which means that there is a trade-off between the quality of the output and the memory footprint of the algorithm.

The algorithm starts off by applying a delay to the sequence of x-values (line 2). The reason for this is that the fractional delay filters, in addition to a fractional delay, also incur a delay of $(D-1)/2$ samples. This is due to the fact that when calculating the coefficients using equation 3.27, an equal number of positive and negative values for n is needed. In practice, negative indices are not possible, so the coefficients need to be shifted into positive values. This is the same procedure that was also used for the other FIR filters.

Following that, for each x-value the algorithm considers the interval between the midpoint of the current and previous value and the midpoint between the current and next value (lines 5,6). Then each point on the output grid that falls into that interval is calculated. To compute an output point, the offset c needs to calculated first, which is the relative position of the current output index m in the interval. This position is then mapped onto the range $(0,1)$ to get the offset for the current output point (line 8).

Finally, to calculate the output, a set from the previously calculated coefficients is selected which has a fractional delay closest to the offset c (line 9). A single value from the convolution of the delay filter and the current part of the y-value sequence is performed, to get an output value (lines 9-11). This is repeated until the position in the output grid is no longer inside the interval $[x_{start}, x_{end})$. The position in the output grid corresponds to the current index of the output multiplied by the period of the distance between two points T_{out} in the output grid.

3.6 Fast Fourier Transform

The final step of transforming the interpolated signal into a spectrum is done by a Fast Fourier Transform (FFT).

The simplest form of the Fast Fourier Transform is derived by the following procedure, known as the *Cooley-Tukey Algorithm* [24]: Starting with the Discrete Fourier Transform (DFT)

$$\mathscr{F}\{x\}(k) = \sum_{n=0}^{N-1} x_n w_N^{nk} \tag{3.28}$$

where $w_N = e^{-j\frac{2\pi}{N}}$, the sum is split into even and odd indices. This allows rewriting the formula in terms of two Fourier transformations with each of the two being half the length of the original transformation.

$$\begin{aligned}\mathscr{F}\{x\}(k) &= \sum_{n=0}^{N/2-1} x_{2n} w_N^{2nk} + \sum_{n=0}^{N/2-1} x_{2n+1} w_N^{(2n+1)k} \\ &= \mathscr{F}\{x_{even}\}(k) + w_N^k \mathscr{F}\{x_{odd}\}(k)\end{aligned} \tag{3.29}$$

The same steps are now applied to the two new transforms, which then yield four Fourier transforms in total, each being a quarter of the length of the original. This pattern continues until there are only sums of length two left, which are calculated directly.

The key insight here is that this algorithm is not limited to only splitting the sum into two. Instead of using the even $(2n)$ and odd $(2n+1)$ indices, it is similarly possible to split the sum into three using the index sequences $3n$, $3n+1$ and $3n+2$. And in general, a Fourier transform can be split any number of times by using the indices $rn, rn+1, \ldots, rn+(r-1)$, where r is called the *radix*. The implementation used a radix-8 FFT which looks like this:

$$\begin{aligned}\mathscr{F}\{x\}(k) &= \sum_{n=0}^{N/8-1} x_{8n} w_N^{8nk} + \sum_{n=0}^{N/8-1} x_{8n+1} w_N^{(8n+1)k} + \ldots + \sum_{n=0}^{N/8-1} x_{8n+7} w_N^{(8n+7)k} \\ &= \mathscr{F}\{x_{8m}\}(k) + w_N^k \mathscr{F}\{x_{8m+1}\}(k) + \ldots + w_N^{7k} \mathscr{F}\{x_{8m+7}\}(k)\end{aligned} \tag{3.30}$$

One thing to note is that, in order to apply a radix-r FFT, the length N of the transform needs to be divisible by r. If this is not the case however, one can still use a different radix because the radix does not need to be the same for all transforms. For example, a Fourier transform of length $N = 1024$ can be realized by recursively applying a radix-8 FFT three times and finishing with a radix-2 FFT (because $1024 = 8 \cdot 8 \cdot 8 \cdot 2$).

3.6.1 Butterflies and Bit Reversals

While equations 3.29 and 3.30 already increased the performance of the Fourier transform, there is another improvement that is used in practice. By making use of some properties of the exponential

Twiddle factor	Value	Result
w_N^0	1	$a+jb$
$w_N^{N/8}$	$c-jc$	$c(a+b)+jc(-a+b)$
$w_N^{N/4}$	$-j$	$-b+ja$
$w_N^{3N/8}$	$-c-jc$	$c(-a+b)+jc(-a-b)$
$w_N^{N/2}$	-1	$-a-jb$
$w_N^{5N/8}$	$-c+jc$	$c(-a-b)+jc(a-b)$
$w_N^{3N/4}$	j	$b-ja$
$w_N^{7N/8}$	$c+jc$	$c(a-b)+jc(a+b)$

Table 3.1: Shortcuts for multiplying a complex number $a+jb$ with specific twiddle factors.

function, it is possible to calculate multiple values of the Fourier transform together while using less calculations than what would have been needed if one were to calculate the values separately. Specifically, identities such as

$$w_N^{N/2} = -1 \quad \Leftrightarrow \quad w_N^k = -w_N^{k+N/2} \tag{3.31}$$

can be used to simplify or save complex multiplications. This equation shows that instead of calculating both w_N^k and $w_N^{k+N/2}$ from scratch, only one of the two values needs to be calculated, while the other one can be obtained simply by negation. Applying this equation to the FFT yields the following (The smaller FFTs are periodic and have a period length of $N/2$)

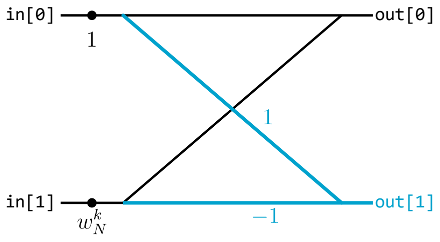
$$\begin{aligned} \mathscr{F}\{x\}(k) &= \mathscr{F}\{x_{even}\}(k) + w_N^k \mathscr{F}\{x_{even}\}(k) \\ \mathscr{F}\{x\}(k+N/2) &= \mathscr{F}\{x_{even}\}(k) - w_N^k \mathscr{F}\{x_{even}\}(k) \end{aligned} \tag{3.32}$$

Similar shortcuts also exist for radix-4 and radix-8, which can speed up the complex multiplications. Furthermore, while two output values are calculated simultaneously for the radix-2 case, four values are calculated for radix-4 and eight values for radix-8. A list of identities similar to equation 3.31 for radix-8 is given in table 3.1. The constant c used in the table is equal to $1/\sqrt{2}$.

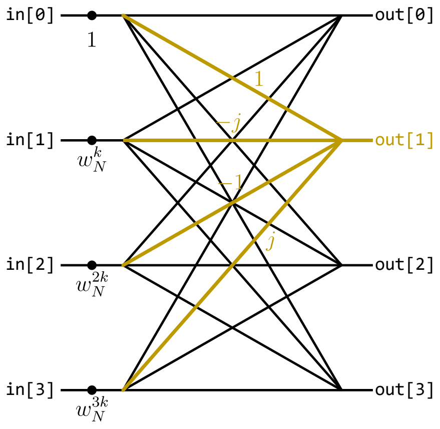
Making use of such shortcuts leads to a form of the FFT that can be used both for visualization and for implementation, which is the *Butterfly-diagram*. In a butterfly diagram each connection between an input and an output represents one of these identities. Butterfly diagrams for radix values 2, 4 and 8 are shown in figure 3.12. It can be seen that the complexity of butterfly diagrams increases exponentially with the radix.

There is a downside to using butterfly diagrams for the FFT: Depending on how the formula for the FFT was derived, either the input values or the output values need to be reordered. The algorithm discussed in this section is the *Decimation in Time* variant, which requires that the input values are reordered before they are passed into the butterflies. There is another variant called *Decimation in Frequency* which keeps the order of the input values, but instead changes the order of the output [21].

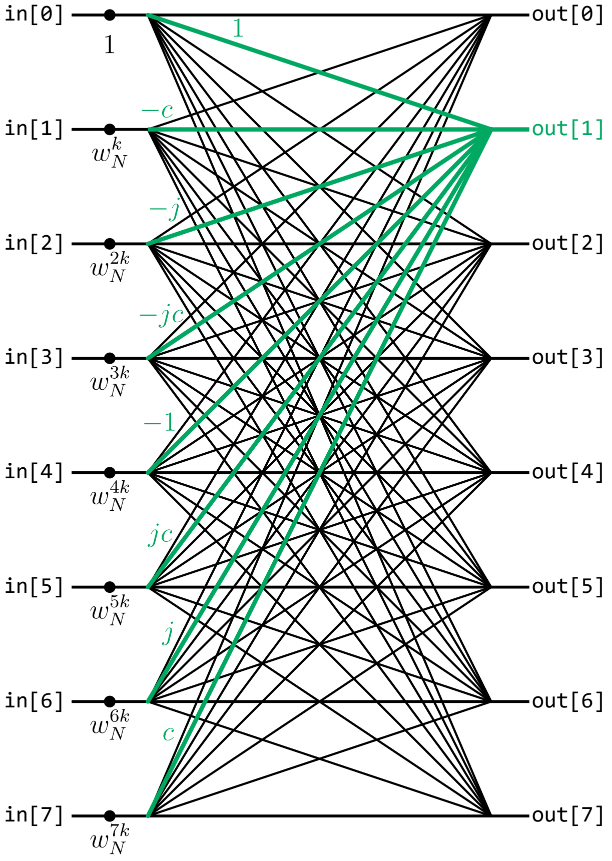
For the radix-2 case of equation 3.29, the reordering is done by reversing the bits of the indices. This can again be generalized for higher radix values: For a radix value r the reordering is performed



(a) Radix-2 butterfly



(b) Radix-4 butterfly



(c) Radix-8 butterfly

Figure 3.12: Butterflies of different sizes [2]

by writing out each index in base r and then reversing the digits. If r is a power of two, this can alternatively be represented by taking the binary representation of the index and looking at groups of bits. The order of the groups is reversed, but the order of bits within a groups remains the same. If the bits cannot be grouped equally, like in the radix-8 case, groups can also be smaller, which indicates that a smaller radix values needs to be used at some point in the computation. An example for this reordering is shown in table 3.2, where the input indices are shown on the left and the reordered indices on the right [2].

3.7 Filter System Configuration

The individual components of the filter system each provide some parameters for configuration. These parameters are either set to or calculated from a set of user-supplied values. In the following, the computation of each of the parameters of the filter system is explained.

The band-pass filter has three parameters that can be tuned: the upper and lower cutoff frequencies f_{high} and f_{low}, as well as the length N_{bp} of the FIR filter. The filter length N_{bp} is set to a constant value at compile time. The cutoff frequencies are directly provided by the user, with no further calculations.

Index	Radix-2			Radix-4			Radix-8		
0	0 0 0 0	0 0 0 0	0	00 00	00 00	0	000 0	0 000	0
1	0 0 0 1	1 0 0 0	8	00 01	01 00	4	000 1	1 000	8
2	0 0 1 0	0 1 0 0	4	00 10	10 00	8	001 0	0 001	1
3	0 0 1 1	1 1 0 0	12	00 11	11 00	12	001 1	1 001	9
4	0 1 0 0	0 0 1 0	2	01 00	00 01	1	010 0	0 010	2
5	0 1 0 1	1 0 1 0	10	01 01	01 01	5	010 1	1 010	10
6	0 1 1 0	0 1 1 0	6	01 10	10 01	9	011 0	0 011	3
7	0 1 1 1	1 1 1 0	14	01 11	11 01	13	011 1	1 011	11
8	1 0 0 0	0 0 0 1	1	10 00	00 10	2	100 0	0 100	4
9	1 0 0 1	1 0 0 1	9	10 01	01 10	6	100 1	1 100	12
10	1 0 1 0	0 1 0 1	5	10 10	10 10	10	101 0	0 101	5
11	1 0 1 1	1 1 0 1	13	10 11	11 10	14	101 1	1 101	13
12	1 1 0 0	0 0 1 1	3	11 00	00 11	3	110 0	0 110	6
13	1 1 0 1	1 0 1 1	11	11 01	01 11	7	110 1	1 110	14
14	1 1 1 0	0 1 1 1	7	11 10	10 11	11	111 0	0 111	7
15	1 1 1 1	1 1 1 1	15	11 11	11 11	15	111 1	1 111	15

Table 3.2: Re-ordered indices for radix-2 4 and 8 FFTs

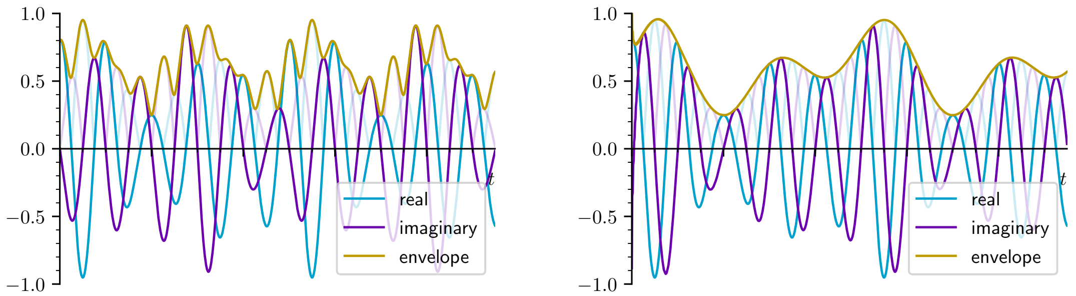


Figure 3.13: Example of an envelope detector with too short of a Hilbert filter (left) and a sufficiently long Hilbert filter (right)

The envelope detector has one parameter which is the length of the underlying Hilbert FIR filter. This parameter must be set according to the range of frequencies that are expected at the input of the envelope detector. If the frequencies are too high, the carrier signal and the envelope will "melt together" and can not longer be easily separated. If the frequencies is lower, the Hilbert FIR has to be longer for the envelope detector to correctly extract the envelope. Figure 3.13 shows an example of the case when the Hilbert filter is too short.

To ensure that the envelope detector functions properly, a Hilbert filter length N_H of twice the period of the lowest frequency component is used. Since the Hilbert filter is located after the band-pass filter, the lowest frequency component can be approximated as the lower cutoff frequency of the band-pass. Therefore, the length of the Hilbert FIR is calculated as

$$N_H = 2T_{low} = 2\frac{f_s}{f_{low}} \tag{3.33}$$

where f_s is the sampling rate of the input signal. The value of twice the lower cutoff period has been determined through testing.

The first decimation filter is controlled by two parameters, which are the length of the filter N_{pre} and the decimation factor q_{pre}. The decimation factor is determined by the maximum order O_{max} that should be tracked by the filter system, which is set by the user. This value in turn specifies the maximum sampling rate that is required and therefore the output sampling rate of the first decimation filter.

An order k has a frequency of k times the fundamental frequency of the input signal. In case of this filter system, the fundamental frequency is the rotational speed of the part that is measured. The highest frequency that can theoretically occur in the input signal is therefore the maximum order times the highest possible rotational speed R_{max}, both of which have to be provided by the user. Following the sampling theorem, the required sample rate to resolve this frequency is

$$f_{pre} = 2 \cdot O_{max} \cdot R_{max} \tag{3.34}$$

Equation 3.34 is the minimum sampling rate required to resolve all frequencies which are of interest. The signal can therefore be downsampled to match this rate, leading to a decimation factor of

$$q_{pre} = \left\lfloor \frac{f_s}{f_{pre}} \right\rfloor \tag{3.35}$$

The length of the filter is then calculated from the decimation factor as follows:

$$N_{pre} = 64 q_{pre} - 1 \tag{3.36}$$

Calculating the filter length from the decimation factor gives the filter a near constant overhead, because as more samples are skipped by a higher downsampling factor, the filter length increases proportionally. The -1 is used to make the filter length odd. This is done since the x-values need to stay in sync with the y-values and an even filter length would introduce a fractional delay.

The interpolation filter provides one parameter which is the resolution of it's output grid f_{int}, however a second value is directly involved in the calculation which is the oversampling factor q_{int}. The oversampling factor is calculated by first determining the highest frequency component of the y-values at this point. Since the signal is low-pass filtered twice up until this point (by the band-pass filter and the first decimation filter), the highest frequency can be determined as the higher cutoff frequency:

$$f'_{max} = \min\left\{ \frac{f_s}{2 \cdot q_{pre}}, f_{high} + \varepsilon \right\} \tag{3.37}$$

The sampling rate required to resolve this frequency is

$$f_{max} = 2 f'_{max} \tag{3.38}$$

This frequency is the highest component in the input signal. However, from the perspective of the input signal, the output of the interpolation filter will have a sample rate that varies with time.

For this reason it must be ensured that at no point this output rate is lower than the Nyquist-limit. The output rate at any point is the current rotational speed times the maximum order. Therefore, the lowest output rate occurs when the rotational speed is the lowest, which is at R_{min}:

$$f_{min} = 2 \cdot O_{max} \cdot R_{min} \tag{3.39}$$

Because the interpolation filter does not have an anti-aliasing filter, any sampling rate conversion that would cause aliasing needs to be avoided. For this reason, the interpolation filter is configured to re-sample the input signal at a higher rate than is required for order analysis. Aliasing would occur when the output rate is too low to resolve the highest frequency component of the input signal, which will happen when the rate is lower than f_{max}. Therefore the output rate must be scaled so that it is always higher than f_{max}, leading to an oversampling factor of

$$q_{int} = \left\lceil \frac{f_{max}}{f_{min}} \right\rceil \tag{3.40}$$

With the oversampling factor, the resolution of the output grid can finally be calculated as

$$f_{int} = 2 \cdot O_{max} \cdot q_{int} \tag{3.41}$$

The oversampling factor directly leads to the parameters of the component following the interpolation filter, which is the second decimation filter. Like the first decimation filter, it requires it's length and decimation factor to be set. The decimation factor is simply the oversampling factor from the interpolation filter. For this reason, the second decimation filter can be seen as the anti-aliasing filter that was missing from the interpolation filter. The length of the decimation FIR is calculated similar to equation 3.36 for the first decimation filter:

$$\begin{aligned} q_{post} &= q_{int} \\ N_{post} &= 64 q_{post} - 1 \end{aligned} \tag{3.42}$$

The final component of the filter system, which is the FFT, provides one parameter which is the length of a single frame. This value is calculated from the maximum order and the number of rotations that should be contained in one frame:

$$N_{FFT} = 2 \cdot O_{max} \cdot R \tag{3.43}$$

Both the maximum order O_{max} and the number of rotations R are set by user. They must be set so that the length of a frame N_{FFT} is a power of 2.

4 Implementation

This chapter presents the proposed implementation of the filter system described in chapter 3. First, some of the changes which where made to the filter system are highlighted. Next, the specifics of the implementation of fixed point numbers are introduced. Finally, the ASIP configurations on which the implementation will be evaluated are presented.

4.1 Differences from the Reference Implementation

The reference implementation was designed to run primarily on an ARM® Cortex® M4-based microcontroller. The code was written in C++ using the Embarcadero® C++Builder IDE and therefore made use in parts of the Delphi® run-time library [25]. The new implementation is designed to run on Xtensa LX7-based architectures and was written in C11 using the Cadence® Tensilica® Xtensa® Xplorer™ IDE (abbreviated as Xtensa IDE) [26].

During the port to the LX7, a number of changes have been made to the reference implementation, which are described in the following:

4.1.1 Skipping the Decimation Filter

For some configurations of the filter system, the factor for either of the two decimation filters can become 1. The reference implementation does not have any special behaviour in place for this case. The proposed implementation will conditionally remove decimation filters with a factor of 1 from the filter system. Recalling from section 3.4, a decimation factor of n signifies that only every nth sample should be forwarded to the output. A factor of 1 means that every sample is passed on, effectively reducing the decimation filter to a low-pass filter with a cutoff frequency slightly below the Nyquist-limit. Such a filter does not contribute to the overall output of the system and thus can be skipped.

4.1.2 Restricting the Range of the X-Values

The x-values represent the rotational position at which the current sample was taken. Because they are constantly increasing throughout the running time of the filter system, problems start to arise when the numbers get too large. For a floating point implementation, this means that less bits are used to store the fractional part of the number, leading to a loss of accuracy. For a fixed point implementation, the values can unexpectedly overflow when the integer bits run out.

An increase of 1.0 in the x-values correspond to one full rotation. Therefore, the fractional part represents the current phase of the rotation ranging from 0.0 to 1.0, while the integer part represents the number of rotation that have passed since the start of the filter system. However, the number of rotations in never explicitly needed in the system. Therefore, to address the problems mentioned above, the integer part is completely truncated from the x-values, leaving only the phase of the rotation. For fixed point values, this procedure can be seen as a kind of "controlled overflow", where the values are wrapped at known instances.

The only component that needs to be modified in response to this change is the interpolation filter. The interpolation uses the distance between successive x-values to select a delay. For this reason, the x-values need to be increasing for the calculation to be correct. To achieve this, lines 5 and 6 of algorithm 1 are changed as follows:

Algorithm 2 Modification to the interpolation filter

```
1: $x_{start} \leftarrow (x_{n-1} + x_n)/2$
2: $x_{end} \leftarrow (x_{n+1} + x_n)/2$
3: if $x_{n-1} > x_n$ and $(x_{n-1} - x_n) > 0.5$ then
4:     $x_{start} \leftarrow x_{start} - 0.5$
5: end if
6: if $x_n > x_{n+1}$ and $(x_n - x_{n+1}) > 0.5$ then
7:     $x_{end} \leftarrow x_{end} + 0.5$
8: end if
```

Each of the two branches correspond to one of the locations where the wraparound can occur. Lines 6-8 are ran if the wraparound happens between x_n and x_{n+1}, in which case x_{n+1} should be treated as being 1.0 higher than it actually is. This is because x_{n-1} and x_n are part of the current rotation since they are in increasing order, but x_{n+1} is part of the next rotation. The next rotation would have phase values greater than 1, however, due to the truncation of the integer part, the value for x_{n+1} is instead wrapped back around. Therefore, the formula in line 2 is corrected by adding 0.5, which corresponds to the difference of 1 in x_{n+1}.

Similarly, lines 3-5 take care if the wraparound happens between x_{n-1} and x_n, in which case both x_n and x_{n+1} are part of the next rotation and should be 1 higher. Because the interpolation filter only needs the difference between the x-values and not the absolute values themselves, the same outcome is reached by treating x_{n-1} as being 1 lower. Therefore, the formula in line 1 is corrected by subtracting 0.5 from x_{start}.

This code makes the assumption that successive x-values never change by more than 0.5. In practice, the distances between successive x-values have been observed to be much lower, but the restriction still exists.

4.2 Fixed Point Implementation

This section focuses on specific implementation details found in the implementation of fixed point arithmetic.

4.2.1 Motivation

To reduce hardware costs, Floating Point Units (FPUs) have been omitted from all architectures used in the implementation. This means that there is no hardware support for floating point data types like `float` and `double`. It is still possible to use these types in code, however, when encountering floating point arithmetic, the compiler will polyfill a software implementation of the corresponding hardware function. These functions are often orders of magnitude slower than the hardware versions. For this reason, performance-critical code will not use any floating point operations, but instead resort to fixed point.

4.2.2 Data Types

The proposed implementation makes use of two data types to denote the two different fixed point formats used throughout the code base. These data types are `signal_t` and `coeff_t`, both of which are derived from a hardware-specific integer type. The compiler provides intrinsic data types for each of the architectures, which map to their corresponding register files.

The `signal_t` type is used as a general replacement for C's floating point types and has a format which is set through preprocessor macros at compile time. To facilitate such a variable format, arithmetic on the `signal_t` type is generally not performed with C's build-in operators, but with dedicated functions. These functions use the mentioned preprocessor macros to adjust their correction shift to maintain the fixed point format. This procedure makes it very easy to test the filter system with different fixed point formats, e.g. to evaluate their accuracy. The primary use of the `signal_t` type is to store the samples passed through the filter system's components.

The `coeff_t` type is used exclusively for the coefficients of the various FIR filters. `coeff_t` will always have a fixed point format with exactly one integer bit, e.g. $Q1.31$ or $Q1.23$. The reason for this choice is related to the fact some of the architectures provide dedicated instructions for multiplying two fixed point values in in such a format, but these instructions have the property that they shift the result one bit to the left. This design choice was likely made because a format with only one integer bit (which will be the sign bit) can only hold values in the interval $[-1.0, 1.0)$. It can be easily seen that multiplying two values in this range will always yield a value which is also in the range $[-1.0, 1.0)$ and in turn, no additional integer bits are needed to store the result.

However, there is a side effect to handling multiplication like this: When multiplying a $Q1.31$ number with a number in any other fixed point format, the result will contain a number in that same format in the upper 32 bits. For example when multiplying a $Q1.31$ number and a $Q9.23$ number the result (after the left shift by 1 bit) will have the format $Q9.55$. Because the architectures used can often access individual sets of 32 bits, this can potentially save an otherwise necessary correction shift. Furthermore, the hardware specific DSP libraries often expect inputs to be $Q1.31$ format [27, 28, 29]. For example, an FIR filter implementation will maintain the fixed point format of the input data when it's coefficients have a fixed point format of $Q1.31$.

4.2.3 Division

The implementation of the interpolation filter needs to perform a division to determine which fractional delay filter should be used for each sample. Some of the architectures used support hardware-division, however in all cases using the hardware division would have been impractical. The main reason for not using the hardware division was that the divider does not support a 64-bit numerator which would have been required for a 32-bit fixed point division (cf. section 2.4). Instead the algorithm uses a combination of a lookup-table and Newton-Raphson Division.

Newton-Raphson Division

Newton's method or the *Newton-Raphson method* is an iterative algorithm used to find the roots (zeros) of a function. In particular, it operates by first making an initial estimate for the location of the root and then repeatedly refining that estimate until it is within some margin of the actual root. The refinement is done by placing a tangent at the current estimate and then observing where that tangent line crosses the x-axis. This location is then used as the new value for the estimate and the calculation repeats. The equation for the tangent line $t_i(x)$ is

$$t_i(x) = f'(x_i)(x - x_i) + f(x_i) \tag{4.1}$$

where

- $f(x)$ is the function of which the root should be found
- $f'(x)$ is the first derivative of $f(x)$
- x_i is the current estimate

The zero of the tangent can be easily found by setting $t_i(x)$ and solving for x. This leads to the following formula for the iteration step for the Newton-Raphson method:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{4.2}$$

For the division algorithm, the reciprocal of the denominator d is calculated first using the Newton-Raphson method and is then multiplied by the numerator. The function that is commonly used to calculate the reciprocal is

$$f(x) = \frac{1}{x} - d \tag{4.3}$$

Substituting equation 4.3 into 4.2 yields

$$x_{i+1} = x_i(2 - dx_i) \tag{4.4}$$

Equation 4.4 is used a fixed number of times in the division algorithm to refine the initial guess taken from a lookup table [30]. An example of an application of the Newton-Raphson method for calculating the reciprocal is shown in figure 4.1, by finding the root of $f(x) = 1/x - 0.625$.

Generating the Lookup-Table

Because Newton-Raphson division requires a first estimate as a start value, a lookup-table is generated to provide these values. The table contains reciprocals for the range $[0.5, 1.0)$ in a fixed point format. The values are calculated to minimize the mean squared error

$$e(x) = \frac{1}{b-a} \int_a^b \left(x - \frac{1}{v} \right)^2 dv \tag{4.5}$$

where

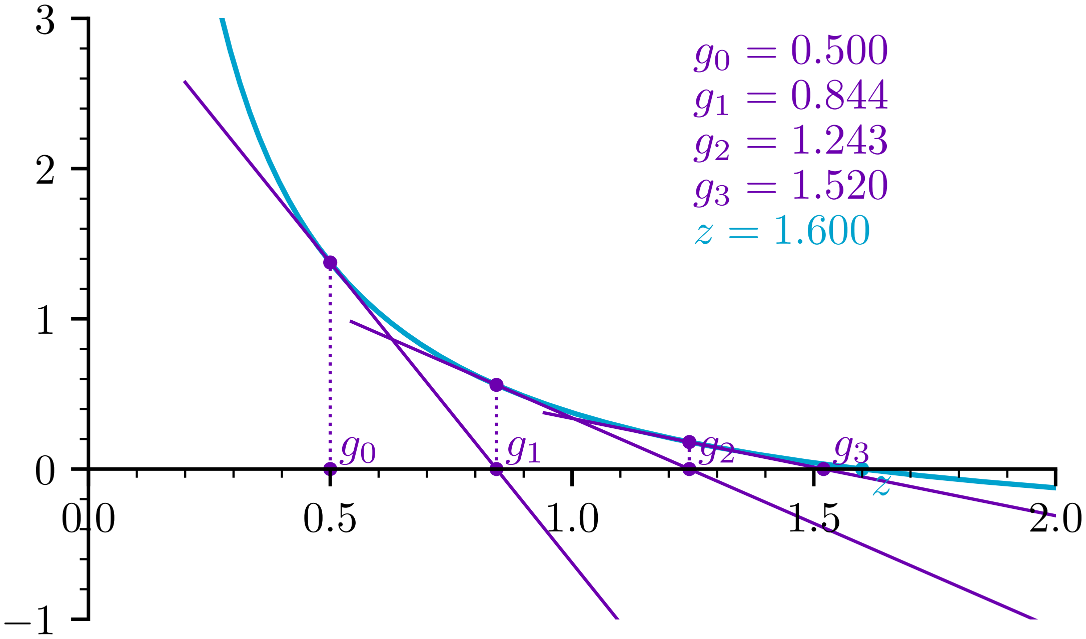


Figure 4.1: Three iterations of the Newton-Raphson method to compute $1/0.625$

- a is the lower bound of the interval.
- b is the upper bound of the interval.
- x is the estimate for the reciprocal on the interval $[a,b]$.
- $\frac{1}{v}$ is the true value for the reciprocal on the interval $[a,b]$.
- $e(x)$ is the mean squared error of the estimate x for the interval $[a,b]$.

To derive a formula for the optimal estimate, the integral is resolved first:

$$\begin{aligned} e(x) &= \frac{1}{b-a}\int_a^b \left(x^2 - 2x\frac{1}{v} + \frac{1}{v^2}\right) dv \\ &= \frac{1}{b-a}\left(x^2 v - 2x\ln v - \frac{1}{2v}\right)\Bigg|_a^b \\ &= \frac{1}{b-a}\left(x^2(b-a) - 2x(\ln b - \ln a) - \frac{1}{2b} + \frac{1}{2a}\right) \end{aligned} \tag{4.6}$$

In order to find the minimum of the error function, the derivative of equation 4.6 is taken with respect to x:

$$\begin{aligned} e'(x) &= \frac{1}{b-a}\left(2x(b-a) - 2(\ln b - \ln a)\right) \\ &= 2x - 2\frac{\ln b - \ln a}{b-a} \end{aligned} \tag{4.7}$$

Call	Output
`NSA(0000 0001 0111 0110)`	6
`NSA(1111 1111 1000 0001)`	8
`NSA(0001 0000 1111 0101)`	2

Table 4.1: Example calls to `normalize_shift_amount(x)`

Finally, equation 4.7 is set to zero:

$$2x - 2\frac{\ln b - \ln a}{b - a} = 0 \tag{4.8}$$

Solving for x now yields a formula to calculate the estimate with the minimum error on an interval $[a,b]$:

$$x = \frac{\ln b - \ln a}{b - a} \tag{4.9}$$

The lookup table is generated by splitting the range $[0.5, 1.0)$ into intervals of equal size and calculating the optimal estimate for each interval using equation 4.9. The length of the table determines the accuracy of the initial estimate. The Newton-Raphson method converges quadratically so the number of iterations can be determined from the accuracy of the estimate and the required accuracy of the result.

Full Algorithm

A C implementation of the division algorithm is shown in listing 4.1.

Lines 3-4 The first step of the algorithm is to normalize the denominator. The function `normalize_shift_amount(x)` computes the amount by which a number can be shifted to the left without destroying any meaningful bits. From the perspective of the function, the meaningful bits are the least significant bits of the input up to and including the first copy of the sign bit. The function works by scanning the input bits from left to right and counting how often the sign bit occurs. One is then subtracted from this count and the result is returned. Because the implementation runs on the Xtansa ISA, `normalize_shift_amount(x)` maps directly to the NSA instruction. Example calls to this instructions are shown in table 4.1.

By design, the inputs to the division algorithm are always positive and non-zero. This means that the result of shifting the denominator by it's normalization amount will always have $01\ldots$ in the most significant bits. In other words, those two lines of code transformed the denominator into a $Q1.31$ number that is always greater than 0.5, which is exactly what is supported by the generated lookup table. The algorithm can be easily converted to a signed division by taking the absolute value of the operands at the beginning and re-applying the appropriate sign at the end.

Lines 7-8 The table in the implementation has a length of 256 but other lengths are also possible. Having a length that is a power of 2 greatly simplifies calculating the index because bitwise operations can be used. In this case, the index are the eight bits following the most significant 01 bits of the normalized denominator. Assuming a 32-bit number format, these

```
1  signal_t fp_div(signal_t x, signal_t y) {
2      // Normalize denominator
3      int exponent = normalize_shift_amount(y);
4      signal_t y_normalized = y << exponent;
5
6      // Calculate index into the lookup table
7      int index = (y_normalized >> (32 - 10)) & 255;
8      signal_t estimate = lookup_table[index];
9
10     // Shift the estimate back into the format of signal_t
11     int correction_shift = exponent - (I - 1) + F - F_LUT;
12     if (correction_shift > 0) {
13         estimate <<= correction_shift;
14     } else {
15         estimate >>= correction_shift;
16     }
17
18     // Newton iterations
19     signal_t two = 2 << F;
20     for (size_t i = 0; i < iterations; i++) {
21         signal_t temp = (y * estimate) >> F;
22         estimate = estimate * (two - temp);
23         estimate >>= F;
24     }
25
26     // Multiply numerator by inverted denominator
27     return x * estimate;
28 }
```

Listing 4.1: Full implementation of the division algorithm

bits can be isolated by shifting the number by $(32-10)$ to the right and performing a bitwise AND with `0xff` (255).

Lines 11-16 Because the denominator was shifted to obtain the index, this shift has to be undone at this point. The correction shift has multiple parts to it: First, the obvious shift by `exponent` has to be reverted. Because this shift took place before the denominator was inverted through the lookup table, a left shift is needed here. Another more subtle shift also needs to be corrected. Reinterpreting the estimate as a $Q1.31$ number in lines 7-8 means that the value of the number was scaled by $2^{-(\mathtt{I}-1)}$ from the perspective of the lookup table. For example, reinterpreting the value 0.5 in $Q5.27$ format as a $Q1.31$ number will yield the value 0.03125. `I` and `F` is the number of integer and fractional bits respectively of the `signal_t` type.

Finally, for greater accuracy, the lookup table uses a different fixed point format from `signal_t`, with more fractional bits (`F_LUT`). The correction shift also needs to account for this difference in formats. All values except `exponent` on the right-hand side of line 11 are compile-time constants and can be folded into a single value during compilation.

```c
signal_t fp_hypot(signal_t a, signal_t b) {
    int64_t a_squared = a * a;
    int64_t b_squared = b * b;
    int64_t sum64 = a_squared + b_squared;
    int32_t sum32 = sum64 >> 32;
    return sqrt_q1_31(sum32);
}
```

Listing 4.2: Fixed point implementation of `hypot`

Because the `correction_shift` can be both positive or negative, an `if`-statement is needed to apply the shift. In practice, this `if`-statement is collapsed into a single call to a hardware-specific bi-directional shift, which would return the same result.

Lines 19-24 After having computed an initial estimate, the Newton-Raphson method is applied to improve that estimate. The implementation uses two `iterations` of the method, because this will produce a result which has almost 32 correct bits, due to the quadratic convergence of the algorithm. In general, because few iterations are generally needed to obtain a usable result, this loop is unrolled to save the additional overhead that the loop variable (which is unused) would incur. Line 19 also shows how an integer is transformed into a fixed point number by shifting it by the number of fractional bits of the fixed point format.

Line 27 The division is completed by multiplying the numerator with the reciprocal of the denominator and returning the result.

4.2.4 Hypot Function

The `hypot`-function is part of the C standard library and implements the formula

$$hypot(x,y) = \sqrt{x^2 + y^2} \tag{4.10}$$

which is used by the implementation primarily to calculate the absolute value of complex numbers. However, there are a few notable things about this function: Firstly, while it may seem convenient to simply use the standard library version of `hypot`, this version was not used in the implementation for the following reasons: C `hypot` only operates on floating point values, which means that an FPU would be required to not lose performance due a software implementation of floating point numbers. Furthermore, the C `hypot` function makes an effort to both prevent overflows to `Infinity` during the squaring of the input, as well make sure that the output is as close as possible to the mathematically correct result. These properties make using the C `hypot` function a trade-off between accuracy and speed [31].

It was determined that neither of these things are necessary for the implementation. Even though the larger fixed point values will most likely produce overflows, this is handled through the hardware by placing the result of the squaring (i.e. multiplication) operation into a register large enough to hold it. Furthermore, the loss of accuracy through rounding and truncation is still in a tolerable range. Finally, even with an FPU, using the C `hypot` was still be substantially slower than simply writing out equation 4.10 directly.

The fixed point implementation of `hypot` is shown in listing 4.2.

Lines 2-4 The first part of the algorithm is a straightforward implementation of equation 4.10. It should be noted however, that the multiplication used here is a special fixed point multiplication which shifts the result to the left by one bit. The contents of `sum64` are therefore scaled by 2^{F+1}.

Line 5 Because the square root function only operates on 32-bit values, the intermediate result is truncated by 32 bits on the right. The value in `sum32` is now scaled by $2^{\mathtt{F}+1-32}$.

Line 6 Lastly, the square root function is called. The function `sqrt_q1_31` is representative for a hardware-specific square root implementation. The important part is that this function always assumes a Q1.31 number as input. That means that the function sees the value in `sum32` as being scaled by $2^{\mathtt{F}+1+\mathtt{I}-1-32}$. Due to the fact that the number of fractional bits `F` and integer bits `I` always add up to 32 when using a 32-bit fixed point format, this scale factor completely cancels itself out to $2^0 = 1$. For this reason, no correction shift is necessary at the end of `fp_hypot`.

Some of the architectures to not have a Q1.31 square root function available. On these architectures, the CORDIC algorithm (cf. section 2.4.1) is used to compute the `hypot` function. First, the absolute values of both inputs is calculated in order to bring them in the definition domain for the algorithm. After that, ten iterations of the CORDIC algorithm are performed. The result is an approximation of the magnitude of the inputs.

4.3 ASIP Configurations

4.3.1 Base Architecture

The base architecture is a Cadence® Tensilica® Xtensa® LX7 processor (abbreviated as LX7), which is based on the Xtensa Instruction Set Architecture (ISA). This processor offers many options for customization, in order to tailor it to a specific algorithm. A user can both configure existing components of the architecture like the register file or the Load/Store-unit as well as add new custom Functional Units (FUs). An overview of the LX7 architecture is shown in figure 4.2.

The Xtensa ISA is a 32-bit architecture, hence the LX7 has a register file that is 32 bits wide. The number of entries in the register file can be configured by the user. Because some of the registers retain their values through subroutine calls, a higher amount of registers enables deeper call stacks, but also increase hardware costs.

The LX7 can read instructions either as 24-bit or as 16-bit instruction words. The 16-bit encoding is optional, but it can reduce the size of an executable at the cost of additional hardware. The encoding formats can also be mixed. The instruction word width is also configurable to be larger, in order to enable VLIW (cf. section 2.3.1). The coprocessors presented in the following sections make use of this feature and require specific widths for the instruction fetch unit.

The Xtensa ISA has a number of optional instructions which can be enabled or disabled when designing a processor. Depending on the algorithm, a programmer might want to add some instructions to speed up the execution and leave out other instructions to save hardware costs. An optional instruction that was already presented in section 4.2.3 is the Normalize Shift Amount (NSA) Instruction. Other optional instructions include different forms of hardware multiplication as well as hardware division. Finally, if more specific instructions are needed, custom Functional

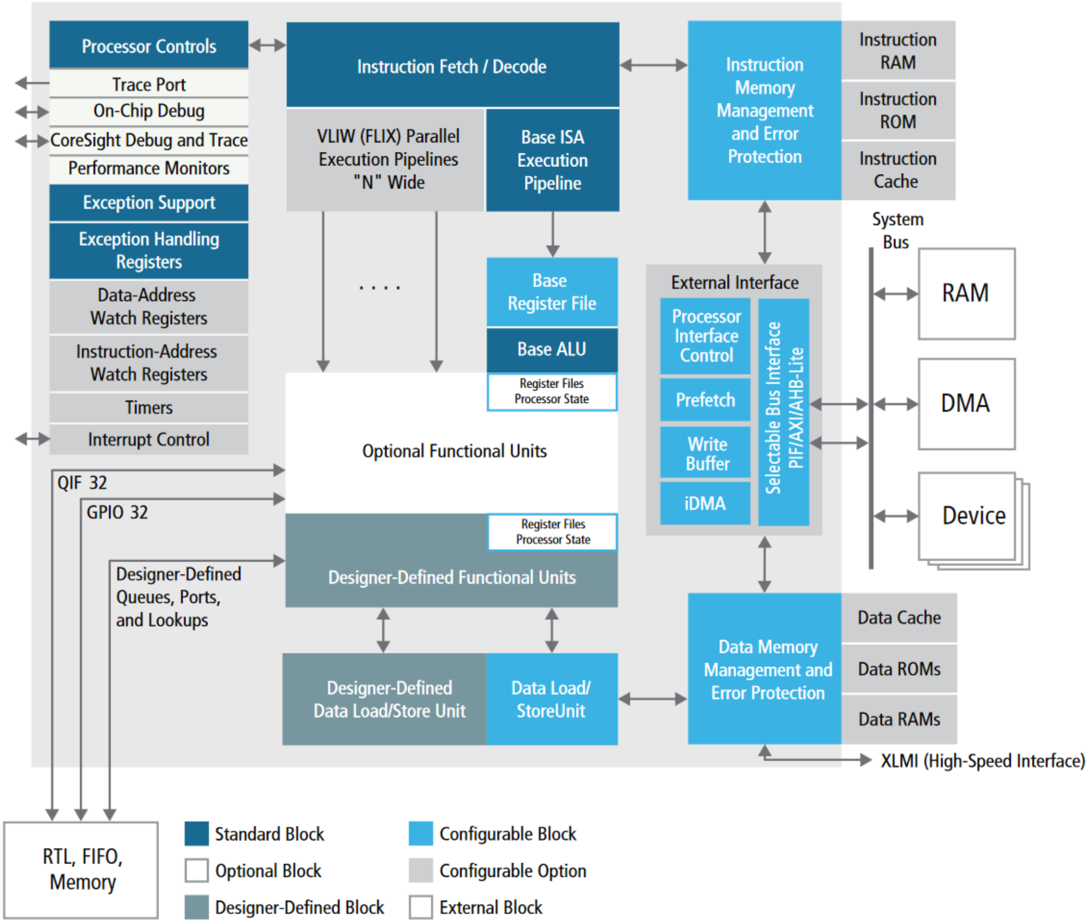

Processor Controls
Trace Port
On-Chip Debug
CoreSight Debug and Trace
Performance Monitors
Exception Support
Exception Handling Registers
Data-Address Watch Registers
Instruction-Address Watch Registers
Timers
Interrupt Control
Instruction Fetch / Decode
VLIW (FLIX) Parallel Execution Pipelines "N" Wide
Base ISA Execution Pipeline
Base Register File
Base ALU
Register Files Processor State
Optional Functional Units
Register Files Processor State
Designer-Defined Functional Units
Designer-Defined Data Load/Store Unit
Data Load/ StoreUnit
Instruction Memory Management and Error Protection
Instruction RAM
Instruction ROM
Instruction Cache
System Bus
External Interface
Processor Interface Control
Prefetch
Write Buffer
iDMA
Selectable Bus Interface PIF/AXI/AHB-Lite
RAM
DMA
Device
Data Memory Management and Error Protection
Data Cache
Data ROMs
Data RAMs
XLMI (High-Speed Interface)
QIF 32
GPIO 32
Designer-Defined Queues, Ports, and Lookups
RTL, FIFO, Memory
Standard Block
Optional Block
Designer-Defined Block
Configurable Block
Configurable Option
External Block

Figure 4.2: Overview of the LX7 architecture [3]

Units can be added. These new Functional Units are designed in a hardware description language and are automatically synthesized and integrated into the processors data path.

Another feature of the LX7 and the Xtensa ISA are *zero-overhead loops*. These are a dedicated set of instructions and registers to manage fast execution of program loops. A zero-overhead loop is first initialized by calling one of the `LOOP`, `LOOPGTZ` or `LOOPNEZ` instructions, which sets up the related hardware states. Three special registers control the start and end of a loop, as well as the number of iterations. When a loop is running, the processor constantly checks whether the next instruction would be the end of the loop and if so, jump back to the beginning and decrement the loop counter. All of this happens transparently and therefore saves the overhead of manually managing a loop variable and checking the termination condition. Because there is only one set of special registers, zero-overhead loops cannot be nested. A programmer can however manually transform nested loops into a single loop, to make zero-overhead loops possible [32, 3].

The filter system is tested with three different configurations of the LX7 which differ mainly in the implementation of the multiplication. Because the LX7 registers are only 32 bit wide, a fixed point multiplication needs to be designed to work with these registers. This means that either the width of the operands has to be reduced so the result fits into 32 bits or the operation has to be split in multiple parts to then construct a final 32-result. Both of these approaches are represented in the configurations. The LX7 configurations are also the only configurations where `signal_t` and `coeff_t` use the same fixed point format:

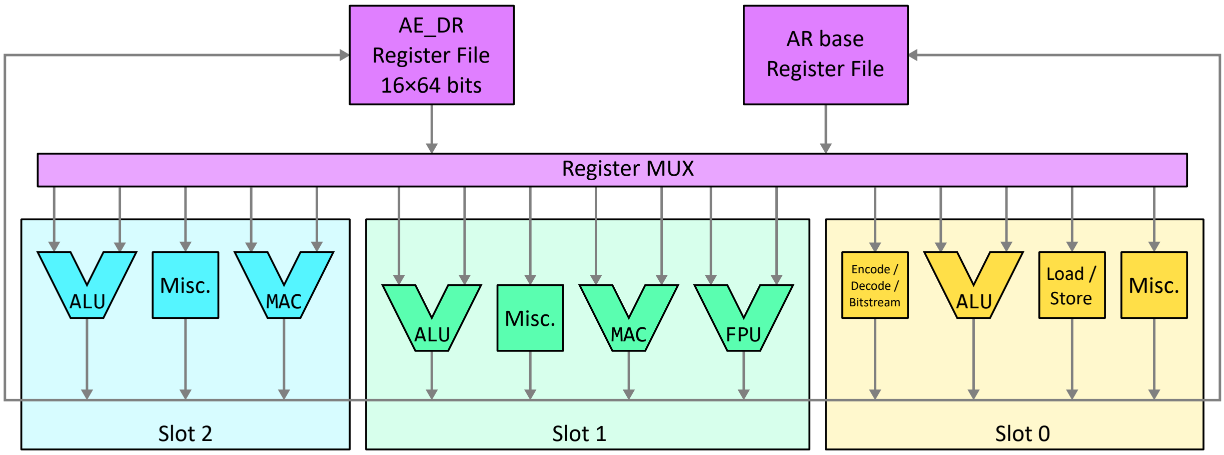


Figure 4.3: HiFi3 DSP Coprocessor [1]

LX7 base: The first configuration uses a 16×16 bit multiplier, to synthesize a 32×32-bit multiplication. Writing a 32-bit number like $a \cdot 2^{16} + b$ allows to synthesize a multiplication as follows:

$$(a \cdot 2^{16} + b) \cdot (c \cdot 2^{16} + d) = ac \cdot 2^{32} + (ad + bc) \cdot 2^{16} + bd \tag{4.11}$$

Therefore, four 16×16-bit multiplications are necessary to perform a 32×32-bit multiplication. However, additional logic is required to calculate the correct sign for the result.

LX7+MUL16: The second configuration uses the same multiplier but discards the 16 least significant bits from the factors, leaving only the 16 most significant bits for a single 16×16-multiplication. This results in a lower cycle count at the expense of accuracy, as the multiplication effectively drops 16 fractional bits from the operands.

LX7+MUL32: The third configuration uses additional hardware to provide dedicated instructions for 32×32-bit multiplication. Because the result still has to fit in 32-bits, two separate instructions are used to calculate the least significant and most significant halves of the result. This means that the multiplication takes twice as long as the one from *LX7+MUL16*, but computes a full 64-bit result. The output is therefore identical to the *LX7 base* configuration, but runs faster due to the additional hardware.

All configurations use a 32-entry register file and a 5-stage pipeline. They also all support the NSA-instruction and zero-overhead loops.

4.3.2 First Coprocessor

The performance of the implementation will be evaluated on multiple different coprocessors. The first is the Cadence® Tensilica® HiFi 3 DSP (abbreviated as HiFi 3), a coprocessor which is designed for audio processing [1]. The HiFi 3 is VLIW architecture which can issue groups of up to three parallel instructions. The first slot is capable of accessing memory through the load/store-unit while the other two slots perform mainly arithmetic operations. The second slot also has an option for a Floating Point Unit, although this is not used in this configuration.

The HiFi 3 adds a 16-entry register file with 64-bit registers. These registers can hold either a single 64-bit value, two 32-bit values or four 16-bit values, depending on which instructions are used. Most instructions have SIMD-capabilities, allowing for parallel processing of multiple data items in a single instructions. Specifically, the HiFi 3 can perform up to four Multiply-Accumulate (MAC) operations in one cycle, depending on the bit widths used. This property can speed up many operations in the implementation, most notably the FIR-filters. Most instructions also have scalar variants, which reduces the overhead of highly sequential code.

The load/store-unit can access up to 64 bit of memory in one instruction and supports multiple different addressing modes. One notable mode allows the programmer to specify a range of memory that should be treated as a circular buffer. When accessing memory in this range, the register containing the memory address is automatically updated and wrapped to stay in the defined memory region. This can for example be used to manage the state of an FIR filter. The HiFi 3 also supports other addressing modes like post-update load and store operations or indexed addressing.

The hardware features of the HiFi 3 are presented to the programmer in form of extensions to the programming language: The 64-bit registers which can hold multiple values receive dedicated data types depending on how the register is split, e.g. `ae_int32x2` for two 32-bit integers in a single register. This helps to convey the programmers intention to the compiler. Individual instructions can be called through *intrinsics* (compiler-supplied functions), but using the HiFi 3 data types with regular C operators is also possible.

The implementation for the HiFi 3 uses 16 bits for the coefficients with format Q1.15. The HiFi 3 can perform four 32×16 bit MAC operations per cycle, therefore using 16 bits for `coeff_t` and 32 bits for `signal_t` results in the highest throughput.

4.3.3 Second Coprocessor

The second coprocessor is the Cadence® Tensilica® HiFi mini DSP (abbreviated as HiFi mini). This coprocessor is a reduced version of the HiFi 3 which is designed for low-power applications. It has two separate register files: one 8-entry register file with 48-bit registers and one 4-entry register file with 56-bit registers. The 56-bit registers are used to store the results and accumulators of MAC-instructions. Like the HiFi 3, the HiFi mini is a VLIW-architecture with SIMD-instructions, however, the HiFi mini features one less issue slot than the HiFi 3 and SIMD-arithmetic is only available in one of the slots.

The main operand size of the HiFi mini is 24 bits, i.e. half of one register. Being a 24-bit architecture, the HiFi mini interfaces with the 32-bit LX7 by either truncating or extending values. When a 32-bits value from the LX7 is transferred to the HiFi mini, the programmer has the option to either truncate 8 bits from the right (useful for fixed point values) or 8 bits from the left. When storing a 24-bit register to memory, the store is aligned on 32-bits, with the actual value of the register being stored in either the lower or upper 24 bits. Because of the different bit width, the HiFi mini implementation uses 24-bit fixed point formats, instead of 32-bit or 16-bit.[4].

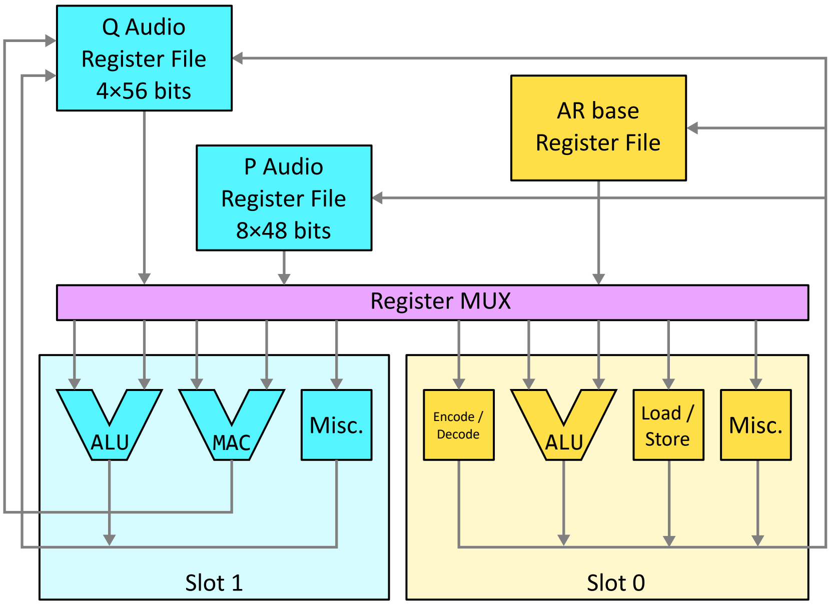


Figure 4.4: HiFi mini DSP Coprocessor [4]

4.3.4 Third Coprocessor

The third coprocessor that the implementation will be tested on is the Cadence® Tensilica® Fusion G3 DSP (abbreviated as Fusion G3) [5]. This coprocessor extends the base processor by providing a 4-way VLIW architecture with SIMD functionality. It has two load units to load up to 256 bits from memory at once, as well as store 128 bits. The load and store units are extended by different addressing modes, similar to the HiFi 3. Like the the HiFi 3, the Fusion G3 supports using special registers to form circular buffers, which can be used by the load/store units. Other addressing modes include post-increment or decrement with immediate values or register contents. Unaligned memory access is also possible, although the compiler generally ensures that memory allocations are already properly aligned.

The Fusion G3 contains several register files, most importantly it provides a 32-entry vector register file containing 128-bit vectors and a wide vector register file with four 320-bit vectors. The vector registers can be split into 8- 16- or 32-bit integers or single or double precision floating point values. Through the ALUs, various SIMD operations are available. In addition, the Fusion G3 also has support for Multiply-Accumulate (MAC) operations, the result of which are stored in the wide vector register file.

Because the result of the MAC operation is saved in a register with a higher bit width than necessary, the operation is said to have *guard bits*. Guard bits have the following function: Assuming a longer sequence of operations (e.g. using several MAC instructions to perform a convolution), the final result may fit into a narrow register. However, some intermediate values might be too large for the target register. An example would be summing the sequence $127, 127, 127, -128, -128, -128$, where the final result is -3, but the highest intermediate value is 381 (when adding from left to right), which needs two more bits than the values of the sequence. To prevent these intermediate values from overflowing and invalidating the result, guard bits are added to correctly represent these values [13].

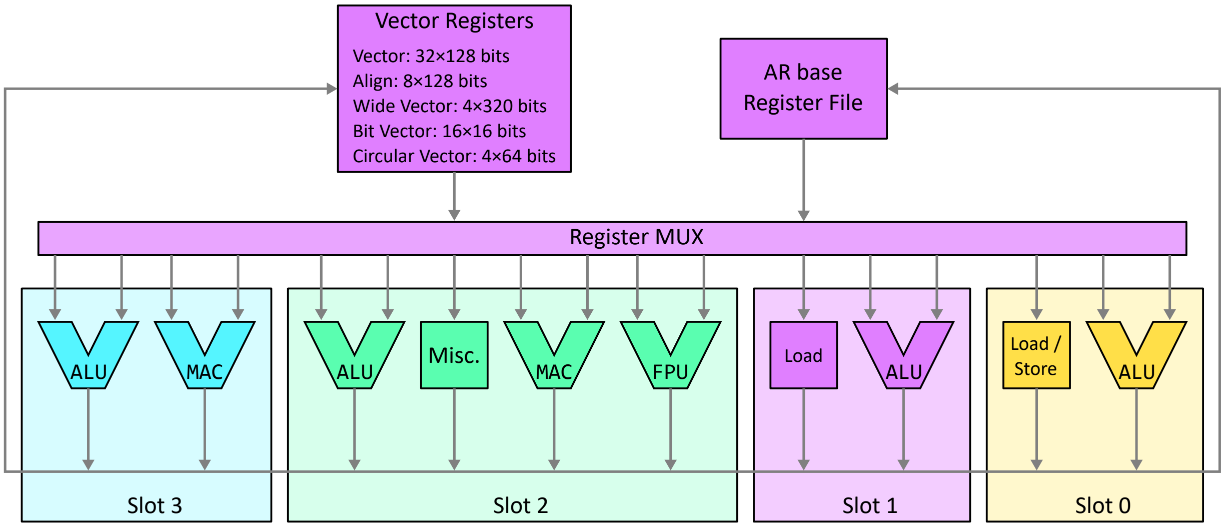


Figure 4.5: Fusion G3 DSP Coprocessor [5]

The Fusion G3 provides eight different VLIW-formats with up to four slots The full architecture is shown in figure 4.5. Having multiple formats allows the compiler to choose the format with the least bits to represent a group of instructions. The encoding scheme ranges from 16-bit instructions to 128-bit instruction words. Therefore the size of an executable image is reduced. Furthermore, the compiler will attempt to recognize vectorizable parts of the code and automatically insert corresponding SIMD instructions. In addition to that, various hardware features and instructions are also exposed to the programmer through intrinsics, data types and overloaded operators, similar to the HiFi 3.

The Fusion G3 implementation uses 32 bits for both samples and coefficients. The coefficients always use a $Q1.31$ fixed point format.

5 Evaluation and Optimization

This chapter presents the methods and results of the evaluation and optimization of the proposed filter system implementation. First, the setup used for the evaluation is presented, along with the test data sets. Following that, the performance results of each of the tested architectures are given and explained. Next, the different properties of the hardware are compared. The chapter ends with a discussion of the evaluation results which also highlights possible further optimizations.

5.1 Design Space Exploration

The aim of this thesis is to provide a design space exploration of processor architectures on which the filter system presented in chapter 3 might run. Being an exploration, there is no specific metric that is optimized or constraint that must be met. Instead, the focus is to provide an overview of possible processor architectures along with the individual benefits and drawbacks. This helps to set expectations by showing which kinds of performance results are possible and which efforts and costs have to be taken to reach these results.

5.2 Evaluation Setup

The proposed implementation is evaluated through the profiling tools built into the Xtensa IDE. The profiling data contains the number of processor cycles spent in each function. Because the individual components all have a single function as an entry point, the cycle counts for each component can be deduced directly from that data. The profiling was limited to the critical path of the filter system by adding switches in the code which enable or disable the profiler. This means that certain parts of the implementation are not included in the profile. Firstly, any initialization code for the components is skipped in the profile. This mainly contains the calculation of the various FIR filter coefficients. Secondly, any I/O-related code is also not included in the profile as this may change depending on the execution context and input format.

To evaluate the throughput of the filter system components, counters are included which count the number of samples that are passed through the individual components. The number of samples changes throughout the system, as components like the decimation and interpolation filter have a different number of output samples than input samples. The implementation keeps track of how many samples were input into each component and reports the total number of samples at the end of the evaluation.

Furthermore, the output of the filter system is saved. This data will be used to evaluate the influence of different fixed point formats on the accuracy of the output by comparing them to a reference output. This reference data was generated by running the proposed filter system implementation using a single precision floating point type for `signal_t` and `coeff_t`. In

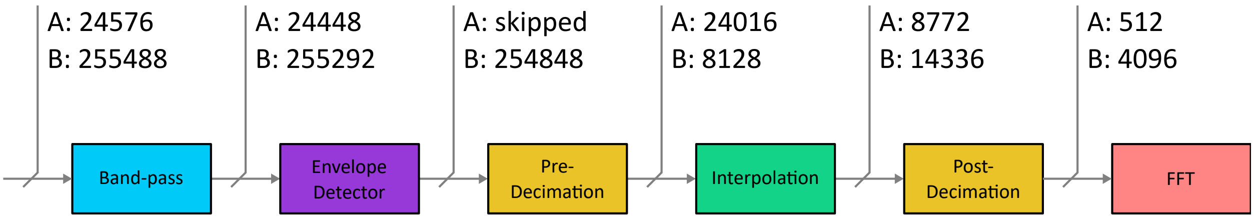


Figure 5.1: Number of samples at the input of each component

theory, the used fixed point formats are independent of the processor configuration, however each architecture has some formats which run faster than others due to the availability of different Functional Units and register bit widths.

Finally, some intrinsic properties of the processor hardware are analysed, specifically, the size of the processor core and the power consumption. This information is available through an estimation by the Xtensa IDE.

5.2.1 Test Data Sets

The proposed implementation is evaluated with two different data sets. The first data set contains $24\,861$ samples and is a recording of the coast-down of a machine. Because of it's short length, this was the data set which was used most during development for testing purposes. The provided filter system configuration for this data set features a decimation factor of one for the first decimation filter, so the filter system will skip this component completely. The oversampling factor and therefore also the second decimation factor is six. Being a coast-down, the x-values in this data set get closer to each other towards the end of the data. This has a very noticeable effect on the interpolation filter. This setup will be referred to as *test setup A* in the following sections.

The second data set is longer than the first one, with $256\,000$ samples. Unlike test setup A which skipped the first decimation, the second data set configures a first decimation factor of 31, resulting in a very large anti-aliasing filter. The raw oversampling factor is only slightly greater than one (1.068) but is still rounded up to 2 by equation 3.40. The x-values also change mostly linearly which might make the measured throughput values for the interpolation filter more representative with this data set. This setup will be referred to as *test setup B* in the following sections.

The actual number of samples which are passed through the individual components varies. Figure 5.1 shows the number samples at the input of each of the component. It should be noted that the band-pass filter and envelope detector also change the amount of samples slightly. The reason for this is that the transient, which is the the initial part of the output during which the FIR filter fills it's internal state, is discarded by the implementations. The numbers shown in figure 5.1 are for the LX7 base implementation. Because the samples are passed between the components in blocks and the different architectures need these blocks to be of a specific size, the number of samples varies slightly for the other architectures. Furthermore, the frame buffer, which is located between the second decimation filter and the FFT, also discards samples when it's buffer is full or when it waits for a trigger to start collecting samples. The frame buffer is currently triggered when the x-values wrap around to 0.

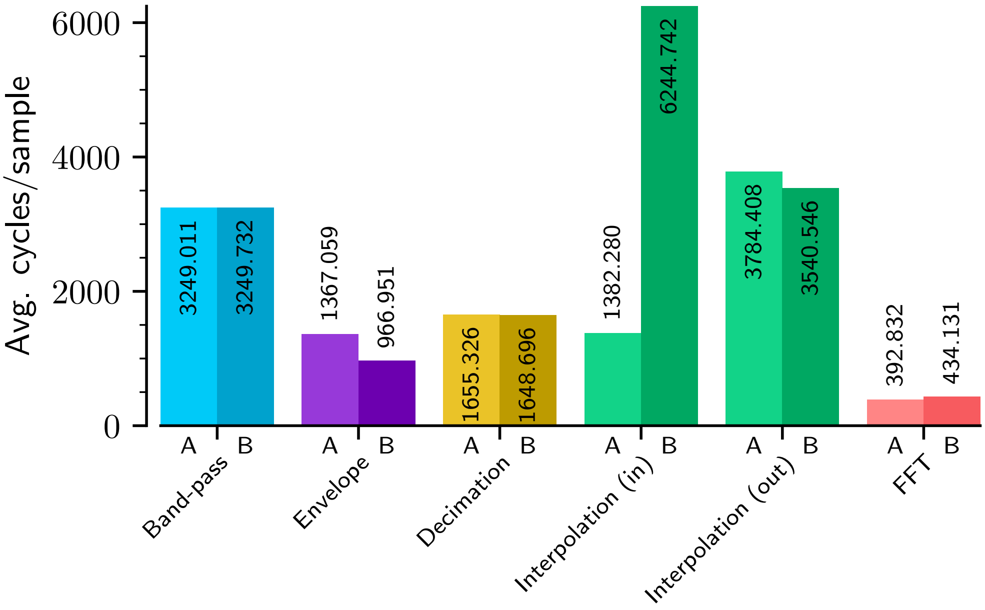


Figure 5.2: Average cycles per sample for the LX7 baseline on test setups A and B

5.3 LX7

Three different LX7 configurations were evaluated which were presented in section 4.3.1. The main difference between the configurations is the implementation of the multiplication operation. However, because this operation is so ubiquitous throughout the filter system, the effects of the different implementations should be very noticeable. The expected result is that the LX7+MUL16 configuration should be the fastest, as it uses the fewest cycles per multiplication. However, the accuracy of the output should also be the lowest, due to the reduced bit widths of the operands. LX7+MUL32 should take at most twice the amount of cycles as LX7+MUL16, as this configuration needs to perform two multiplications for a full 32×32-bit multiplication, but does not need to shift any bits out of the operands. Finally, the LX7 base configuration is expected to run the slowest, as it performs four smaller multiplications per 32×32-bit multiplication and also needs to calculate the sign of the result separately.

Figures 5.2, 5.3 and 5.4 show the average cycle counts per sample for the three LX7 configurations. The differences in multiplication speed become apparent when looking at the average cycle counts for the band-pass filter. Because this component is essentially a straight convolution with no additional computations and the filter size is fixed to 128, the cycle counts are almost identical for test setups A and B. LX7+MUL16 has the lowest average of the three configurations for the band-pass filter, LX7+MUL32 is about 65% slower and LX7 base is slowest, requiring over five times the number of cycles of the LX7+MUL16.

The envelope detector requires less cycles overall, since it's FIR filter is shorter. It's length is also dependent on the filter system configuration, which is why there is a more noticeable difference between test setups A and B. The ratio of average cycle counts between the processor configurations is similar to the band-pass filter, however, the differences are less pronounced. This is because, aside from the FIR filter, the envelope detector also computes the absolute value of the analytic signal, which for the LX7 configurations is performed by the CORDIC algorithm.

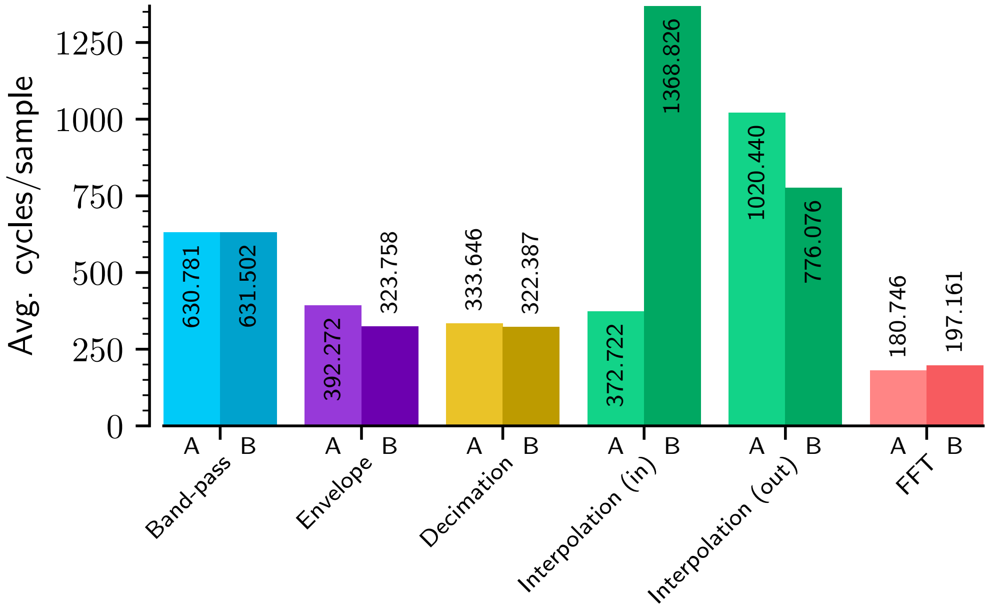


Figure 5.3: Average cycles per sample for the LX7+MUL16 on test setups A and B

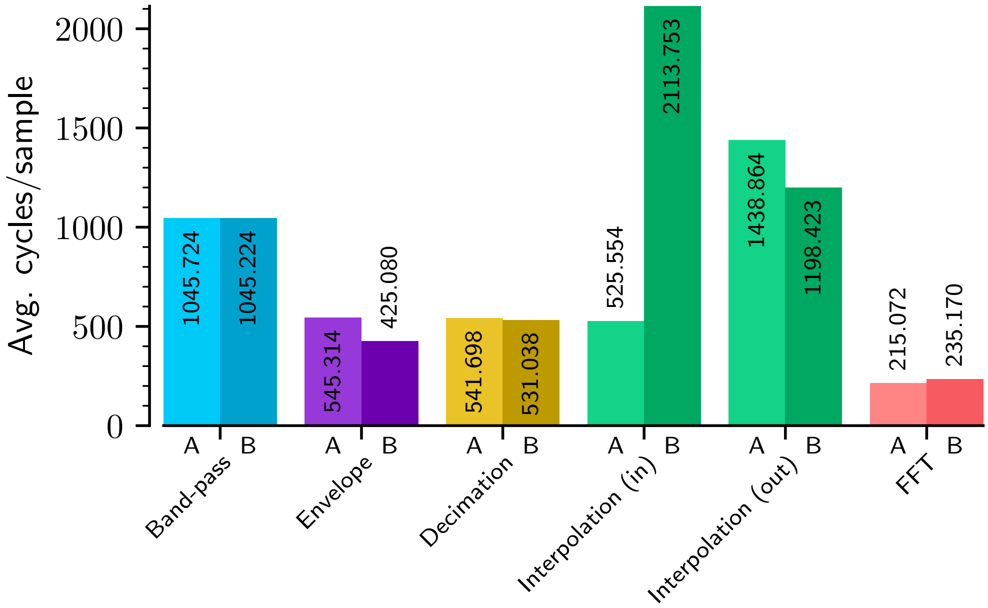
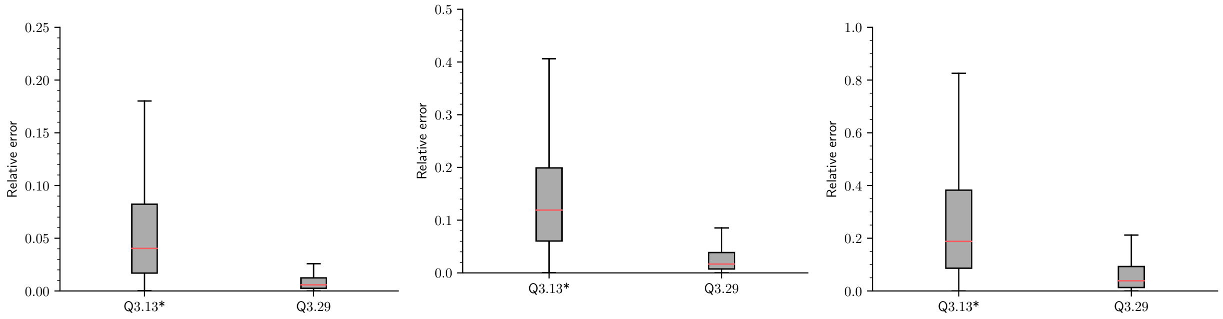


Figure 5.4: Average cycles per sample for the LX7+MUL32 on test setups A and B



(a) Test setup A

(b) Test setup B, with $ref > 0.005$

(c) Test setup B, full

Figure 5.5: Relative error for the LX7 configurations

The CORDIC algorithm at it's core does not use multiplications, so the different multiplication implementations have no effect on it.

The decimation filter, like the band-pass filter, shows little differences between the two test setups. While it may seem that the decimation filter should take much longer for test setup B due to the large anti-aliasing filter, this is not the case. The first decimation filter for file B has a decimation factor of 31 which, by equation 3.36, demands an anti-aliasing filter of length $31 \times 64 - 1 = 1983$. However, a decimation factor of 31 also means that only every 31st output sample is actually calculated. In total, this gives the decimation filter an overhead comparable to a 64-tap FIR filter, regardless of the actual length of the filter. This can also be seen from the fact that the decimation filter needs about half as many cycles as the band-pass filter which with 128 taps has double that length.

The interpolation filter is the most computationally expensive component. It's performance is also highly dependent on the input data, as can be seen by the large difference in cycle counts between test setups A and B. In an attempt to still give a representative picture of the filter's performance, the average is taken in regards to both the number of input samples and the number of output samples from the interpolation filter. Since test setup A represents a coast-down of a machine, the number of output samples decreases quickly as the x-values get closer together. The result is a very low average for the cycles per input sample as well as a high average for the cycles per output sample. The machine from which test setup B was recorded ran at a more constant speed, so the x-values are mostly equally spaced.

The interpolation filter also contains a large amount of instructions that do not involve multiplication, as can be seen by the fact that the average cycles per output sample for the *LX7 base* configuration are just slightly higher than the value of the band-pass filter, but this difference is increased for *LX7+MUL16* and *LX7+MUL32*. Like the band-pass filter, the fractional delay filters in the interpolation filter have a fixed length of 128 taps.

All three configurations are tested with a fixed point format with three integer bits. Because the multiplication for the LX7+MUL16 uses only 16 bits for the multiplication but 32 bits everywhere else, this configuration is assigned the format Q3.13*. This notation signifies that the high amount of multiplication operations in the filter system make this configuration effectively use a Q3.13 format with additional bits in other operations. The configurations LX7 base and LX7+MUL32 both use a full 32-bit format with three integer bits. These two configurations use the same format and perform identical computations, with the main difference being the amount of cycles required per multiplication.

Figure 5.5 shows the accuracy of the output for the fixed point formats of the LX7 configurations. The accuracy is presented as the relative error between the output of the filter system and the floating point reference. Because test setup B contains a high amount of small values, two different errors are considered. For very small values, small discrepancies like quantization noise will produce a large relative error even though the absolute difference is still very small. For this reason the accuracy for test setup B is both shown for the full data as well as only for values where the reference has a value greater than the threshold 0.005.

For test setup A, the relative error for the $Q3.13$* format has its median at around 4% while the $Q3.31$ format has a lower median at about 0.5%. The range between the 25th and 75th percentiles is also smaller for the $Q3.31$ format. The relative errors for test setup B are higher overall. A probable cause for this is that more calculations are performed on the data due to the longer FIR filters. The more operations are performed, the more noise through rounding of samples and coefficients is introduced into the data. Overall, the relative errors for test setup B are about twice as high as those for test setup A while still keeping the same relation between the formats. Including the full reference for test setup B again doubles the relative error, however, as was pointed out, the absolute errors are still small in this case.

5.4 HiFi 3

The first coprocessor configuration on which the filter system was implemented is the HiFi 3. Because the HiFi 3 supports up to four MAC operations per cycle, a speedup of at least 4 over the *LX7+MUL16* is expected. The accuracy should be slightly higher than the one from the $Q3.13$* format, since the HiFi 3 uses the full 32 bits for the samples but still only 16 bits for coefficients. Furthermore, the HiFi 3 is tested with four different fixed point formats for the sample type `signal_t`, with increasing number of fractional bits. The accuracy is expected to improve the more fractional bits are available.

5.4.1 Optimization Process

Throughout the optimization process, the filter system was profiled using test setup A. Figure 5.6 shows a timeline of the cycle count improvements at various points during development.

The initial implementation only replaced the `signal_t` and `coeff_t` data types with HiFi 3 types, as well as implemented the corresponding arithmetic functions. At this point, no explicit SIMD instructions were used, but the compiler already grouped instructions into VLIW instruction bundles. This version took about $57{,}5M$ cycles to process test setup A.

The HiFi 3 has a dedicated signal processing library called *NatureDSP Signal*, which was used to optimize the HiFi 3 implementation [27]. As a first step, the inner loops of the convolutions of the filter system were replaced by an optimized vector dot product function. This change resulted in the cycle count dropping below half of the original count to only needing around $26.3M$ cycles. The vector dot product was used in the band-pass and decimation filters, the envelope detector and the interpolation filter. For the interpolation filter, this improvement already marks the last

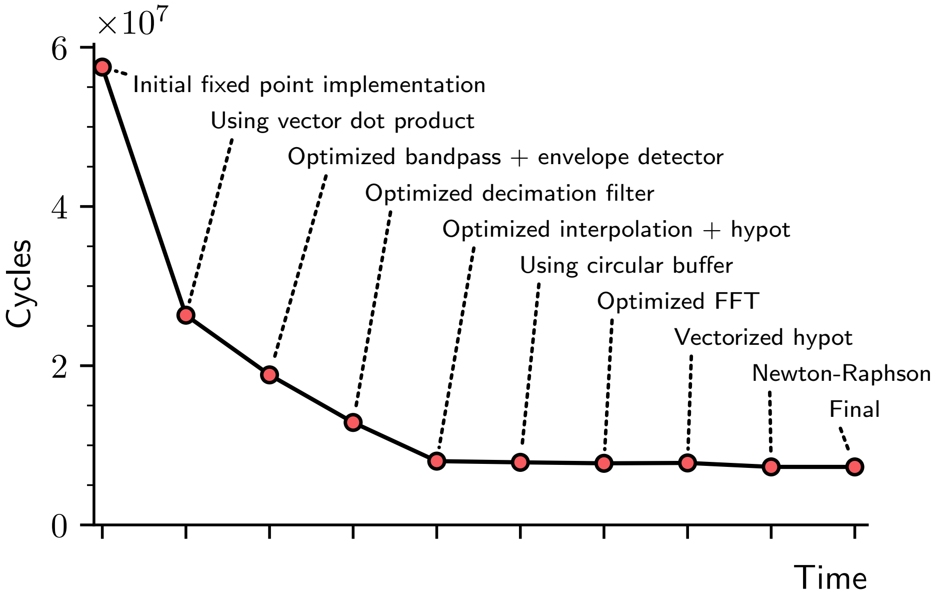


Figure 5.6: Cycle count improvements of the HiFi 3 implementation over time

major cycle saving, as a lot of the code inside the filter is sequential and and has branches which are difficult to remove without knowledge of the input data.

The remaining convolutions are now replaced again with dedicated library functions. These functions provide a greater speedup over the vector dot product by making use of more opportunities to benefit from the HiFi 3's VLIW/SIMD architecture. By having more independent instructions in the function, more instructions can be bundled into Very Long Instruction Words and less cycles are needed. The library provides functions for FIR filters which are used in the band-pass filter and the Hilbert filter of the envelope detector, as well as a decimation filter implementation.

The next step was to implement the `hypot` function described in section 4.2.4. Previously, this function made use of a slower, iterative method for calculating the square root, which also involved additional scaling to prevent overflows and to keep the fixed point format. The new method uses the HiFi 3's 64-bit register file to store intermediate results as well as use a $Q1.31$ square root function to remove the correction shift.

By this point, the cycle count for test setup A has dropped to around $8.01M$, which corresponds to a speedup of 7.18. The remaining optimizations are more minor, like using the HiFi 3's hardware support for a circular buffer for the filter state of the interpolation filter or adding a vectorized version of the `hypot` function. The final cycle count for the HiFi 3 implementation for test setup A is $7.29M$, which corresponds to a speedup of 7.89.

5.4.2 Final Evaluation of the HiFi 3 Implementation

Figure 5.7 shows the average cycle counts per sample for the HiFi 3 implementation. Because the HiFi 3 is an architecture that uses both SIMD and VLIW, algorithms which contain a high number of independent instructions achieve a greater speedup. This can be seen by the fact that the band-pass filter, envelope detector and decimation filter all have low average cycle counts while the FFT and especially the interpolation filter have higher averages. Another noteworthy observation

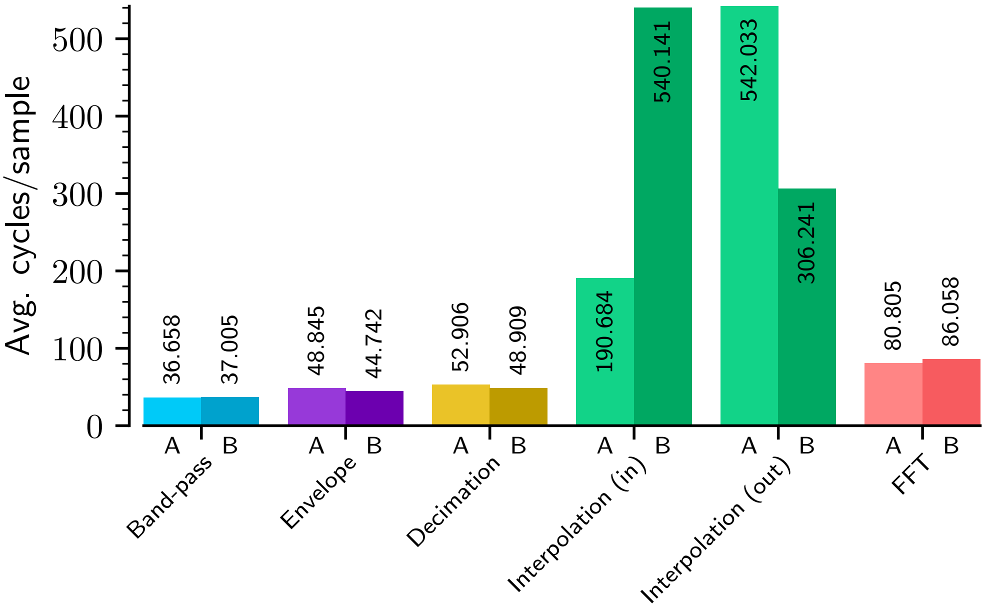


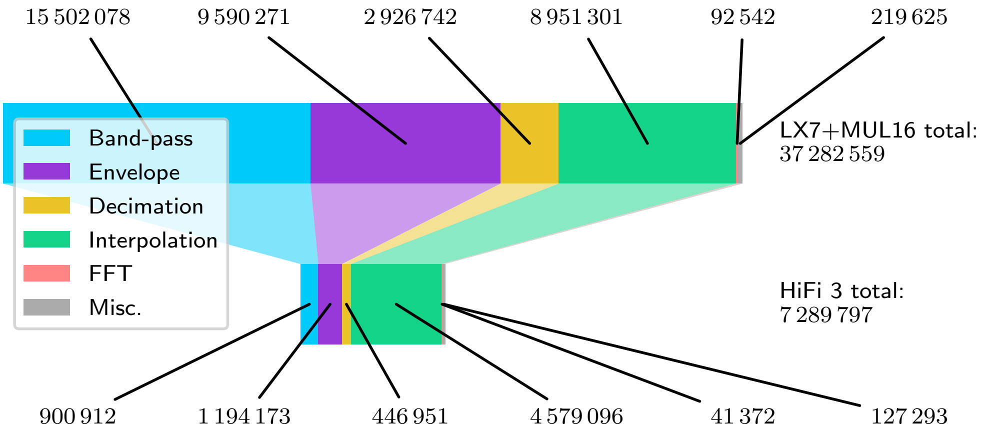
Figure 5.7: Average cycles per sample for the HiFi 3 on test setups A and B

is that the band-pass filter, which was previously slower than the envelope detector and decimation filter, is now faster. This is because these components perform additional computations aside from the convolution. The envelope detector computes the absolute value of the analytic signal and the decimation filter has to decide which values to output while keeping the x and y-values in sync.

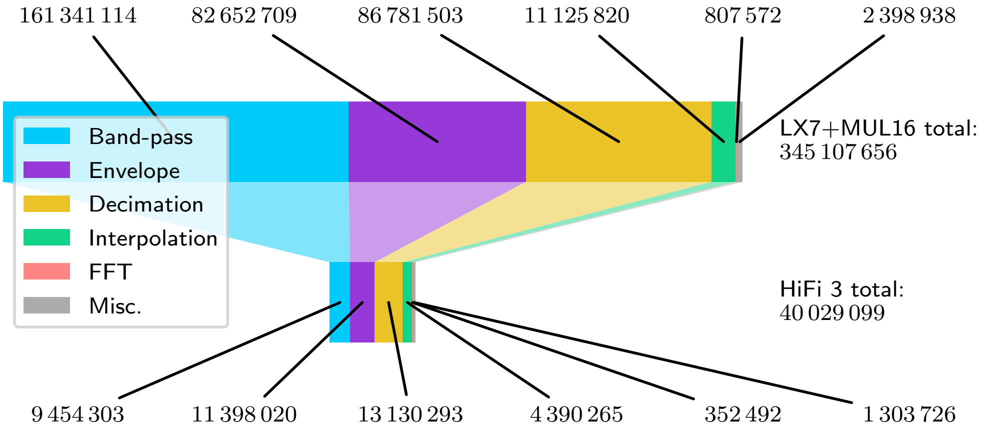
The interpolation filter is still the slowest component and still shows the discrepancy between the two test setups and the number of input and output samples. However, because the convolution aspect of the filter is now parallelized, this discrepancy is even more pronounced than for the LX7 configurations. When compared to the *LX7+MUL16*, the average cycles per input sample for the interpolation filter for test setup A only show a speedup of about 1.95 despite the HiFi 3's parallelism features. Many parts of the interpolation filter's code contain sequential and dependent instructions as well as more complex control flow. An example of which is the division function which was presented in section 4.2.3. Almost every line uses a result from the line directly preceding it, which greatly reduces the compilers ability to create VLIW bundles.

Upon comparing the HiFi 3's results to those of the other architectures, it was discovered that the library implementation of the decimation filter does not use the HiFi 3 to it's full potential. The HiFi 3 reaches it's highest throughput by issuing two dual multiplications in one cycle for a total of four multiplications. However, when viewing the generated assembly for the decimation, the majority of VLIW bundles inside the innermost loop only contain one dual multiplication, effectively halving the throughput. Without considering the x-values, the computation of the y-values reaches an average cycle count per sample of 36.906, which is almost identical to the one of the band-pass filter. However, as was previously pointed out, the band-pass filter has double the overhead of the decimation filter and thus the decimation filter should ideally need about half the cycles.

While the average cycles per sample are an easy way to quickly assess the efficiency of a component, they have to be put into perspective by considering the total number of samples which are passed through each component. For example, the purpose of the decimation filters is to reduce the



(a) Test setup A

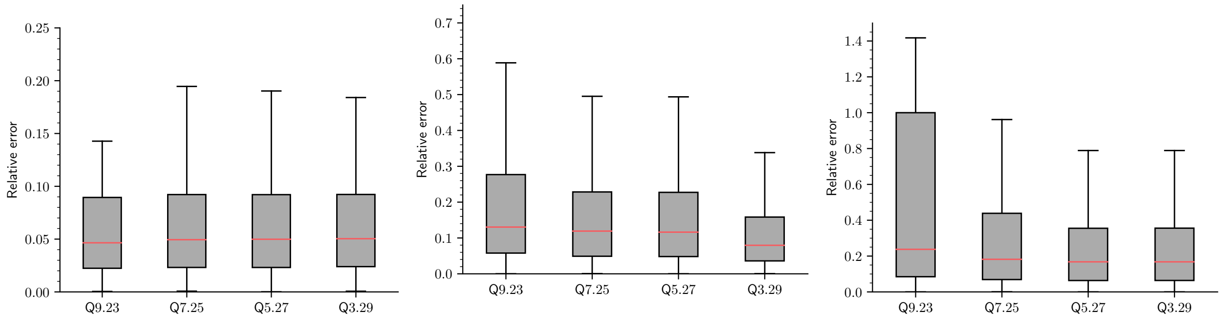


(b) Test setup B

Figure 5.8: Cycles per component for the HiFi 3 and LX7+MUL16

number of samples so that later components need to do less computations. Figure 5.8 shows the total number of cycles that the HiFi 3 implementation saves over the LX7+MUL16. From this figure can be seen that the largest cycle saving for both test setups occurs in the band-pass filter. At this point, no samples have been decimated yet so the full file is passed through the filter. The envelope detector also receives the full file, but it's FIR filter is shorter, so less cycles are saved here.

For the decimation and interpolation filters, the differences between the two filter system configurations become apparent again: test setup A skips the first decimation filter, so no cycles are saved in this component as nothing is executed for either processor configuration. However, this also means that more samples reach the interpolation filter compared to test setup B. A higher sample count at this stage means that more cycles are saved in total by the HiFi 3's parallel computation capabilities. Conversely, test setup B has a very high first decimation factor and the anti-aliasing filter has almost 2000 taps. Therefore, a higher percentage of the cycles are saved by parallelizing the decimation filter for this test setup. On the other hand, the high decimation



(a) Test setup A

(b) Test setup B, with $ref > 0.005$

(c) Test setup B, full

Figure 5.9: Output error for the HiFi 3

factor results in very few samples passing through the interpolation filter, so less cycles overall are saved here.

The HiFi 3 implementation was tested with four different fixed point formats with increasing numbers of fractional bits. Figure 5.9 shows the relative error for these formats on test setups A and B. For the $Q3.29$ format, the input values from the data set are scaled down before being converted to fixed point, because otherwise some values in the test data would not fit into the format. A fixed point format with three integer bits can only hold values in the range $[-4.0, 4.0)$, however, the test data contains values outside this range. The reference data to which the result is compared was also scaled in the same way.

The results for test setup A show little difference between the formats, with the formats with more fractional bits actually performing slightly worse. The overall results for this test setup are comparable to the $Q3.13$* format from the LX7+MUL16 configuration. Conversely, test setup B shows improvements in output accuracy for the formats with higher precision, albeit only very slight. The greatest improvement occurs when jumping from $Q5.27$ to $Q3.29$. However when considering the full reference for test setup B, this improvement vanishes. Furthermore, the relative error increases dramatically for the $Q9.23$ format, most likely because this format does not have enough fractional bits to resolve the smaller values.

A possible explanation for the lack of significant improvement between the formats for test setup B might be that the accuracy is limited by the $Q1.15$ format of the filter coefficients. If the bit width of the coefficients is a bottleneck for the output accuracy, a wider format for the samples will not change the result significantly.

5.5 HiFi mini

The second coprocessor on which the filter system was implemented is the HiFi mini. Being a low-power architecture, the HiFi mini has less capabilities for parallelism, with fewer issue slots and narrower registers. Performance-wise, the results should lie in between those of the LX7+MUL16 and the HiFi 3. Since the HiFi mini implementation uses 24-bit registers for both samples and coefficients, the output accuracy should be higher than the one of the LX7+MUL16. The speed of the implementation is also expected to lie between that of the LX7+MUL16 and the HiFi 3, as

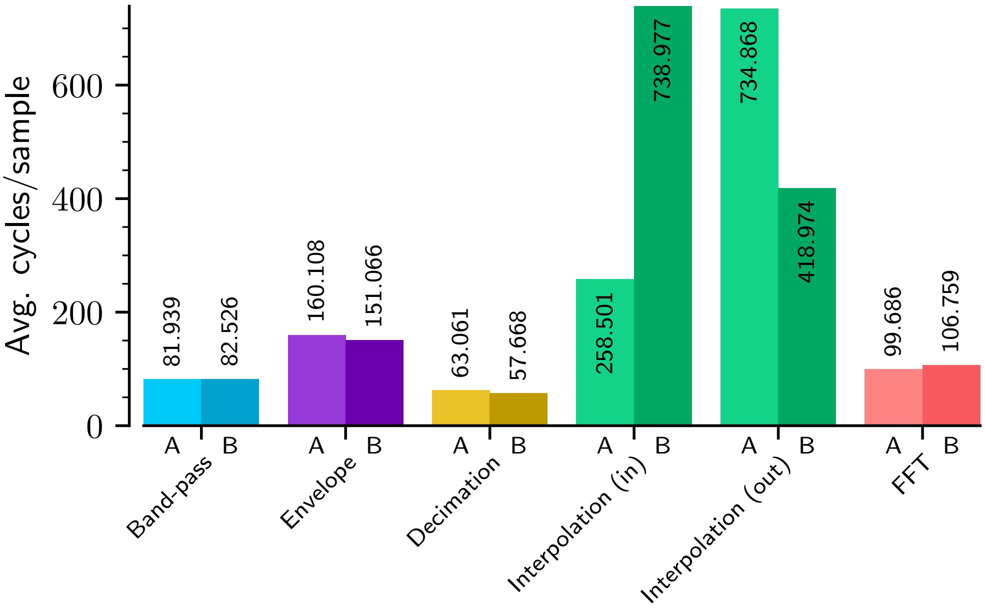


Figure 5.10: Average cycles per sample for the HiFi mini on test setups A and B

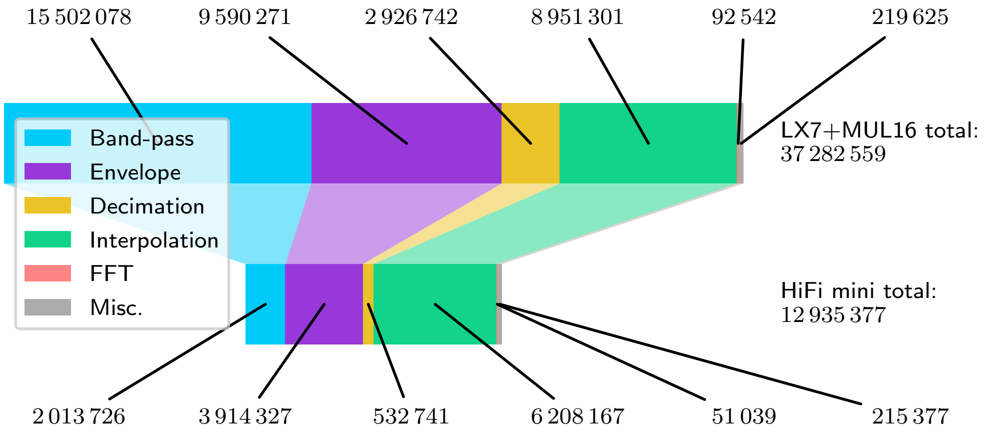
the HiFi mini does have VLIW and SIMD features, however, they are narrower than the ones from the HiFi 3.

The average cycles per sample are shown in figure 5.10. Like the HiFi 3, the envelope detector is slower than the band-pass or decimation filter, however, the difference is more noticeable on the HiFi mini. There are two reasons for this: First, the HiFi mini's FIR filter implementation has a higher overhead for the outer loop of the convolution, which means that shorter filters like the Hilbert filter run slower. Second, the HiFi mini does not use a vectorized `hypot` function, but instead uses CORDIC. The overhead of the CORDIC implementation is comparable to a non-vectorized `hypot` function.

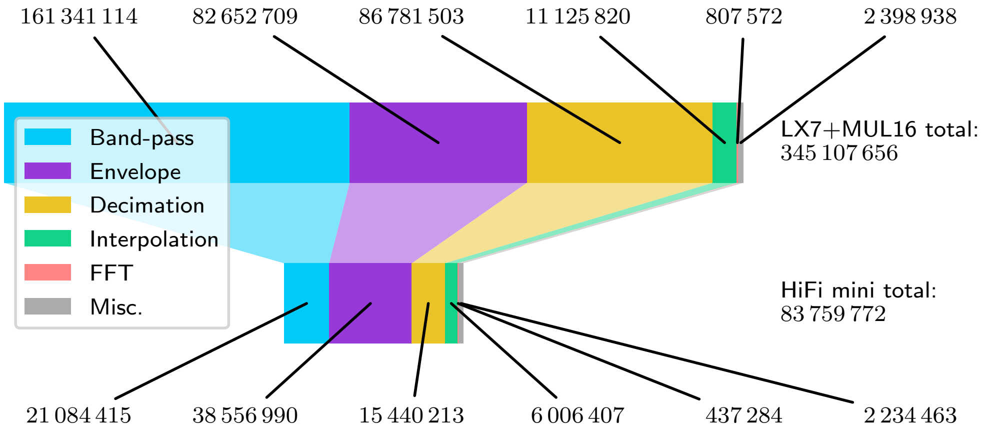
The average cycle count of the FFT implementation is closest to the HiFi 3's value. This was to be expected, since the FFT contains the least amount of SIMD instructions and therefore entirely relies on VLIW to speed up execution. The additional issue slot of the HiFi 3 speeds up the FFT by about 20 cycles per sample over the HiFi mini.

Another thing to note is that on the HiFi mini, the decimation filter requires less cycles than the band-pass filter, while on the HiFi 3 it is the other way around. This is more in line with the Fusion G3 and LX7 implementations, which supports the assumption that HiFi 3's decimation is the outlier in this case. The slower decimation on the HiFi 3 has already been attributed to the fact that the implementation of the decimation filter does not fully utilize the HiFi 3's VLIW slots. The HiFi mini has one issue slot less than the HiFi 3 and the implementation combines load instructions in the first slot and dual MAC instructions in the second slot.

Since the HiFi mini has less hardware parallelism than the HiFi 3, the cycles savings over the LX7+MUL16 are also lower. Figure 5.11 shows the cycles per component for the HiFi mini implementation on test setups A and B. For test setup A, the savings for the decimation filter and interpolation filter are closer together than for the HiFi 3, since with less hardware parallelism, the sequential part of the interpolation filter makes less of a difference.



(a) Test setup A

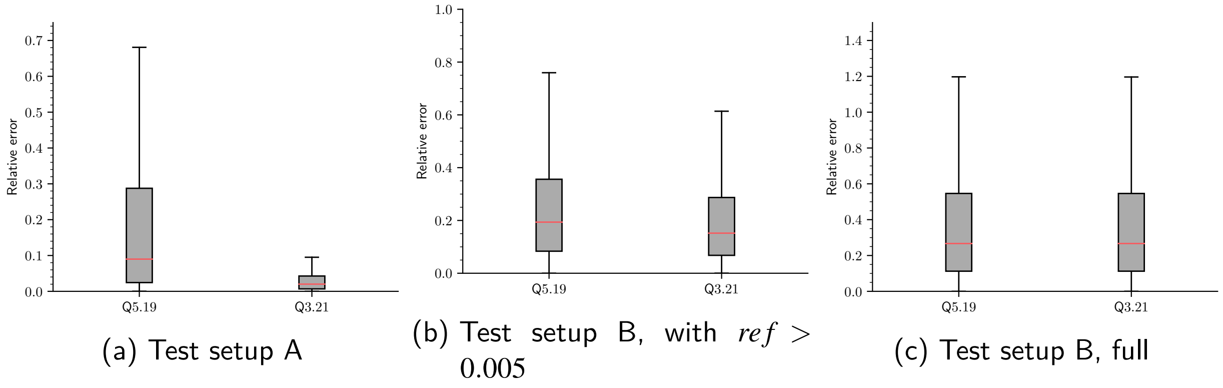


(b) Test setup B

Figure 5.11: Cycles per component for the HiFi mini and LX7+MUL16

The biggest difference in cycles compared to the HiFi 3 is in the envelope detector, due to the differences in FIR filter implementation and lack of a vectorized `hypot` function. On the other hand, the savings for the decimation filter are closer to those of the HiFi 3, which may be because of the HiFi 3s slower decimation filter. Overall, the HiFi mini needs about 77.4% more cycles on test setup A and 109.2% more cycles on test setup B. Since the HiFi mini performs half the multiplications of the HiFi 3 in one cycle, this doubling in cycle requirements is what was expected. However the uneven distribution of the additional cycles over the individual components is an indication that the HiFi mini still has some potential for further optimization.

The HiFi mini was tested with two 24-bit fixed point formats, of which the Q3.21 applies scaling to the input values to make them fit the format. The relative errors for these formats are shown in figure 5.12. The most notable result is that of the Q3.21 format on test setup A, which has a very low relative error compared to the other results. The other formats and test setups have a higher relative error. This highlights the importance of the bit width of the sample data type, as all other architectures have a 32-bit format for `signal_t`. The most probable component in which a lot of accuracy is lost is the FFT, which produces a high number of small values. Because



(a) Test setup A

(b) Test setup B, with $ref > 0.005$

(c) Test setup B, full

Figure 5.12: Output error for the HiFi mini

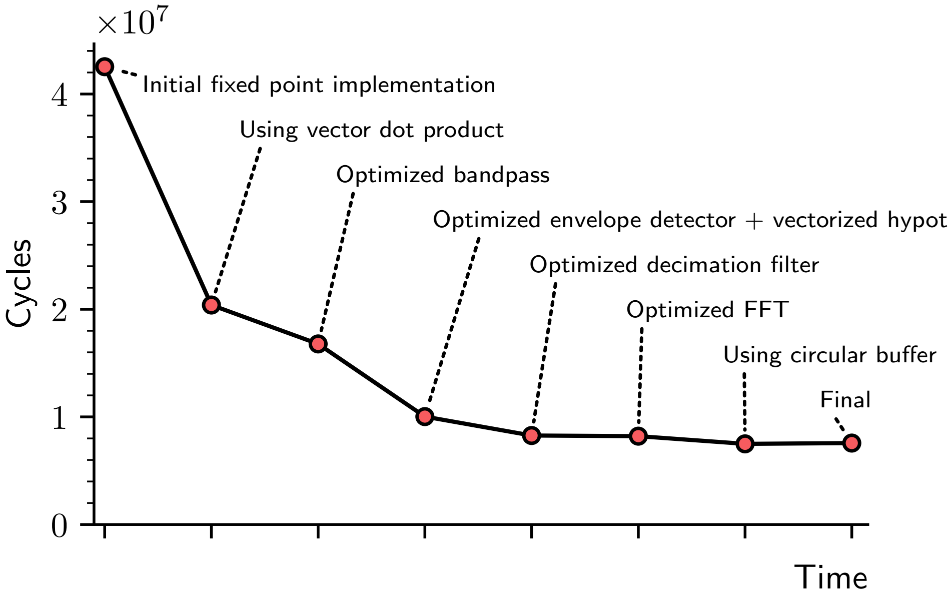


Figure 5.13: Cycle count improvements of the Fusion G3 implementation over time

of the recursive nature of the FFT, rounding errors in these values effect a large number of other values further into the computation.

5.6 Fusion G3

The third coprocessor for which the filter system was implemented is the Fusion G3. Similar to the HiFi 3, the Fusion G3 provides both VLIW and SIMD. The key difference between the two implementation is that the Fusion G3 achieves the same throughput as the HiFi 3 but with wider registers for the coefficients. Therefore, the expected result is that the speed of the filter system is comparable to that of the HiFi 3, but the accuracy of the output should be higher.

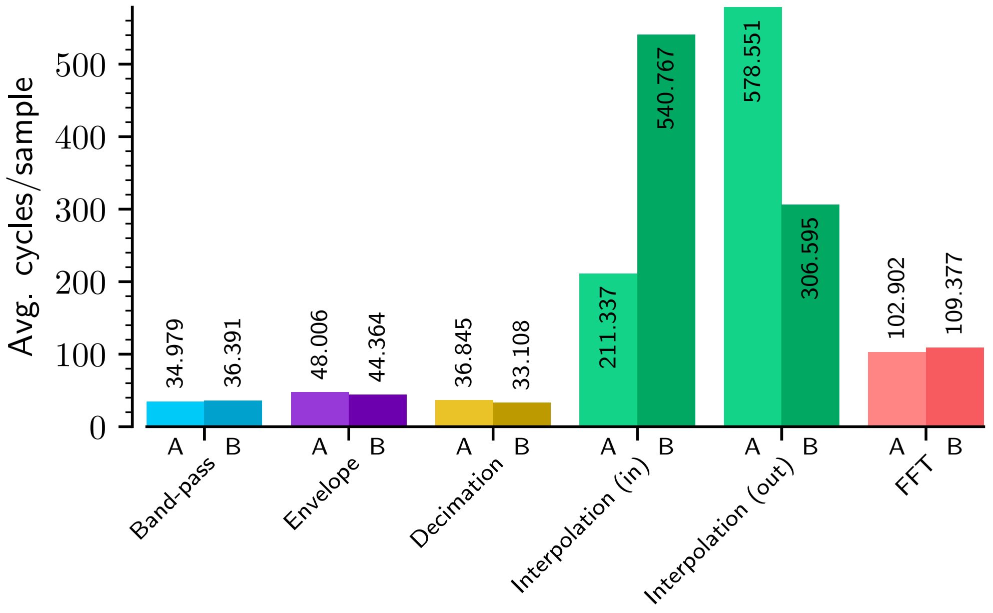


Figure 5.14: Average cycles per sample for the Fusion G3 on test setups A and B

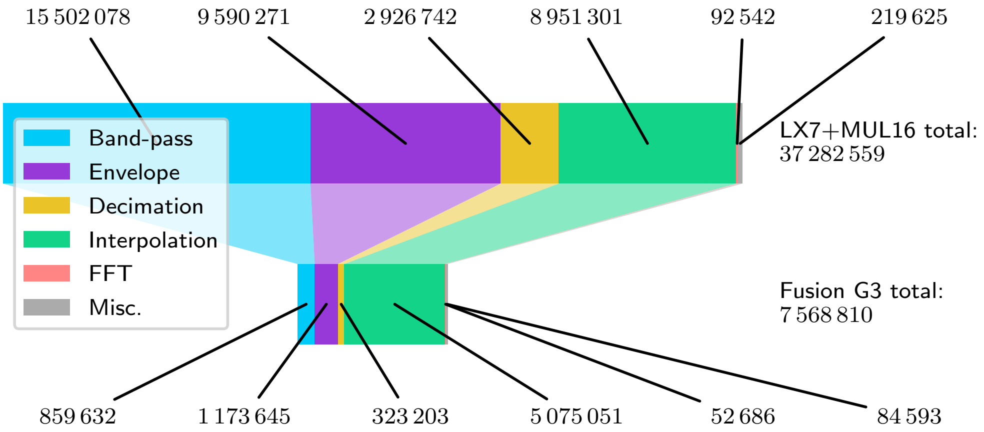
5.6.1 Optimization Process

The optimization process for the Fusion G3 was similar to the one for the HiFi 3. Some optimizations which were added at later stages in the HiFi 3 implementation were added sooner for the Fusion G3. Figure 5.13 shows a timeline of the implementation process for the Fusion G3. Like the HiFi 3 implementation, the first step was to implement fixed point arithmetic operations on the Fusion G3's integer type. The initial cycle count for test setup A was $42.5M$ cycles which is less than what the HiFi 3 needed at this stage. Following this, the convolutions were replaced by a dedicated function for the vector dot product. Like the HiFi 3, this reduced the cycle count to less than half of the original value. Continuing with the optimization, the band-pass filter, Hilbert filter and decimation filters were replaced with optimized functions from the DSP Library [28]. These optimizations further reduced the cycle count to around $8.27M$. The remaining optimizations were implementing arithmetic on complex number using the Fusion G3's integer types to speed up the FFT, as well as using the Fusion G3's hardware support for circular buffers. The Fusion G3 supports multiple circular buffers at once, by means of a dedicated register file for the address ranges. This makes it possible to use circular buffers in more places which would have conflicted on the HiFi 3.

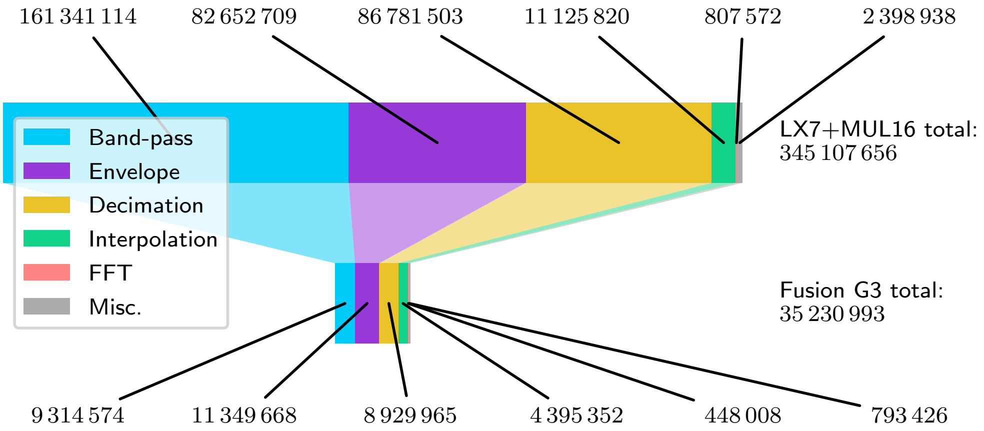
The final implementation for the Fusion G3 takes about $7.57M$ cycles for test setup A, which is slightly more than the HiFi 3 needed.

5.6.2 Final Evaluation of the Fusion G3 Implementation

Figure 5.14 shows the average cycles per sample for each of the components for the Fusion G3 implementation. It shows clear parallels to the average cycles counts of the HiFi 3 implementation, in that the Fusion G3 has very similar results. The band-pass and decimation filters are faster although the difference is very minor. The envelope detector on the Fusion G3 achieves almost identical average cycle counts to the HiFi 3. For the interpolation filter, the values for test setup



(a) Test setup A

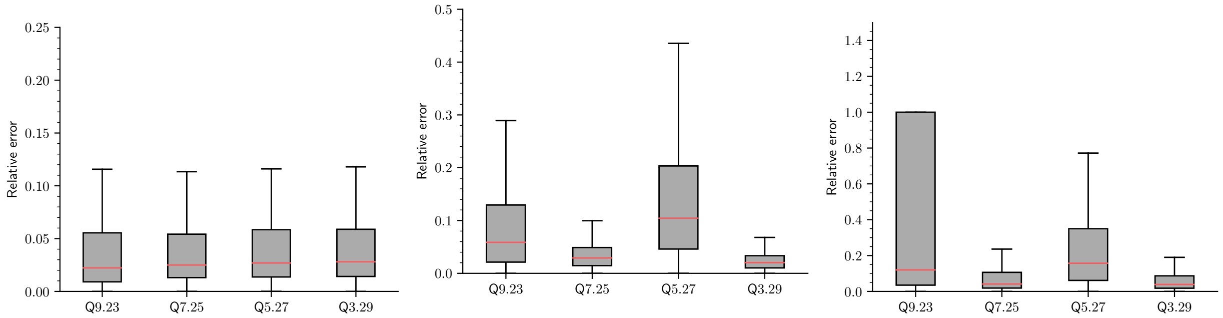


(b) Test setup B

Figure 5.15: Cycles per component for the Fusion G3 and LX7+MUL16

B are also nearly identical to the HiFi 3 implementation, however, the averages for test setup A are higher on the Fusion G3. Furthermore, the average cycles per sample for the FFT are also noticeably higher. The reason for this is that these components both contain some amount of non-parallelizable code. The Fusion G3 is optimized for processing large amounts of data in parallel, however instructions to operate on single data items are scarce on this processor. Unlike the HiFi 3, the Fusion G3 imposes some restrictions on it's scalar and vector data types, which prevent easy conversions between them. When writing sequential code, the additional overhead required for type conversion quickly adds up.

When comparing the Fusion G3's total cycle counts to the LX7+MUL16, the results are again similar to the HiFi 3. Figure 5.15 shows the cycle counts for the Fusion G3. The most cycles are again saved in the band-pass filter. For test setup A, the envelope detector saves more cycles than the decimation filters, while for test setup B, the decimation filter saves slightly more cycles. The FFT, while having a noticeably slower average cycle count, does not lose as many cycles because few samples actually reach the FFT. Overall, the Fusion G3 is about $279k$ cycles slower for test setup A, but about $4.8M$ cycles faster for test setup B.



(a) Test setup A

(b) Test setup B, with $ref > 0.005$

(c) Test setup B, full

Figure 5.16: Output error for the Fusion G3

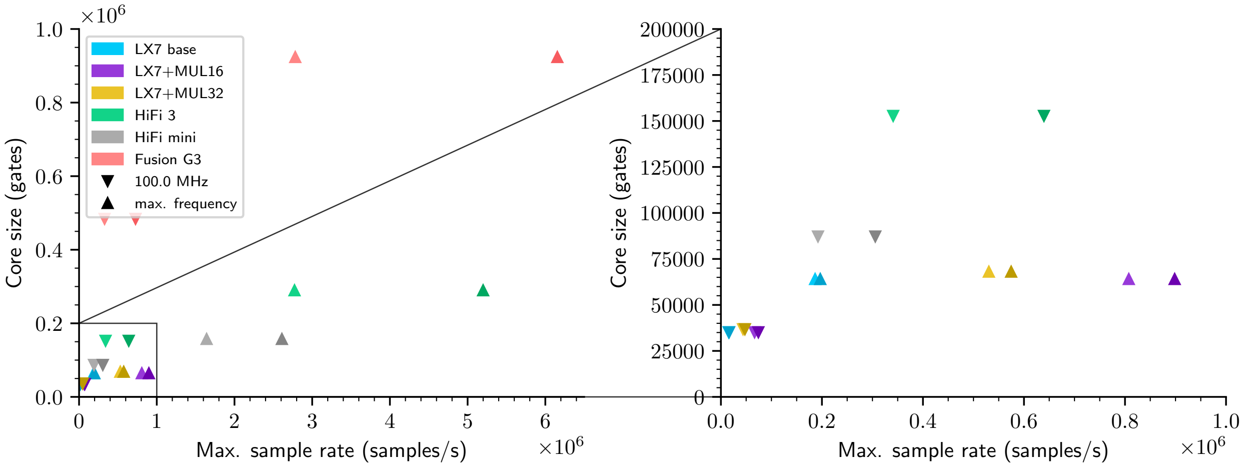
The Fusion G3 and the HiFi 3 have similar capabilities when it comes to parallel execution, which is why the results for the cycle counts are so close to each other. The difference between the two coprocessors is that the Fusion G3 achieves the same speedup while retaining the ability to use a full 32 bits for all operands.

The implementation was tested with the same fixed point formats as the HiFi 3. The relative errors for the Fusion G3 are shown in figure 5.16. For test setup A, the Fusion G3 achieves an even more uniform result than the HiFi 3. The error for all formats is, with a median of about 2%, lower than the one from the HiFi 3. For test setup B, the results are less uniform. For the Q9.23 format, the median of the relative error is about 6%, which is lower than error for the same format on the HiFi 3. The relative error drops further for the Q7.25 and Q3.29 formats. However, for the Q5.27 format, the relative error is significantly higher than for the other formats. This can be an indication of a missed overflow error in one of the components, in a section which is probably not part of a convolution. The Fusion G3 does provide guard bits which should protect against overflows during multiplication. When considering the full test setup B, the relative error increases a lot for the Q9.23 format when comparing to the other formats. 23 fractional bits appear to be the threshold for this test setup, below which some of the samples can no longer be properly resolved.

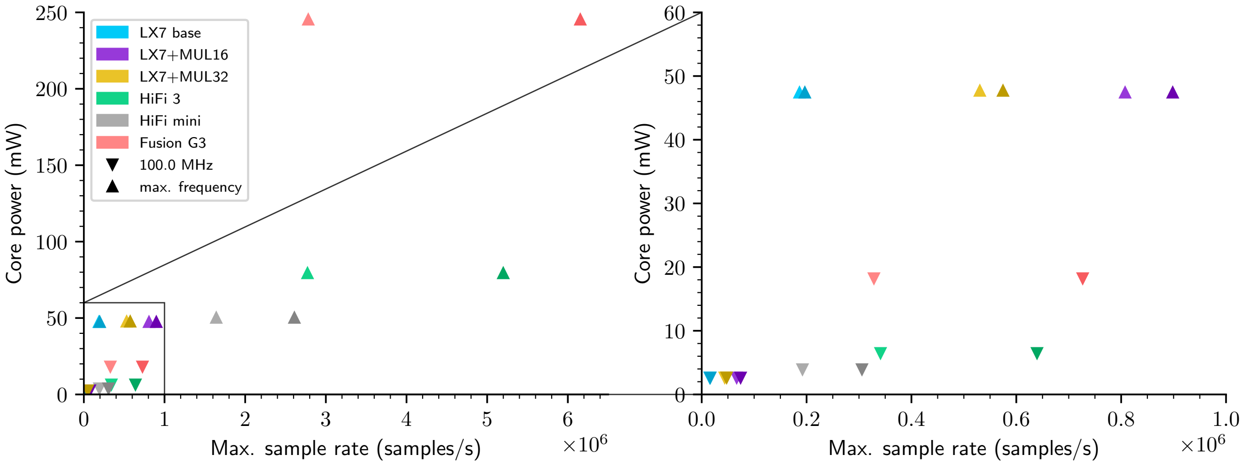
A possible explanation for the different results for the two test setups, is that the values in test setup A are overall greater than the ones in test setup B. For example, in the reference data for test setup A, around 72.8% of the values are greater than 2^{-6}. This means that for a Q9.23 format, 72.8% of the values have at least 18 significant fractional bits. However for test setup B, the number of values greater than 2^{-6} is only 18.8%. The consequence of this is that an increase in fractional bits has a more significant effect on test setup B, because a larger portion of values can now be resolved better.

5.7 Hardware Properties

To further assess the performance results presented in the previous sections, the cost of the required hardware has to be considered. The two metrics presented in this section are the size of the core in gates and the power usage of the core in mW.



(a) Core size vs. speed



(b) Core power vs. speed

Figure 5.17: Comparison of hardware properties

Figure 5.17 compares the speed of the implementation on the different processor architectures to the size and power consumption. The speed of each architecture is given as the theoretical maximum input sample rate in samples per second that the architecture would be able to handle. This value is given for both test setups A and B (which have different filter system configurations that affect the speed) and for two different clock frequencies for the processors. One of the two clock frequencies is the maximum frequency supported by the processor configuration, the other frequency is fixed at 100.0 MHz. The value of 100.0 MHz was chosen because it is the smallest value the Xtensa IDE provides when estimating the hardware properties. Except for the LX7 base configuration, all other architectures are capable of processing the test setups in real-time at this clock frequency (cf. table 5.1). The maximum clock frequencies for each architecture are given in table 5.1. The formula to calculate the sample rates is based on the average performance on a given filter system configuration:

$$f_s = f_{clk} \frac{N_{samples}}{N_{cycles}} \tag{5.1}$$

where

Configuration	max. Frequency (MHz)
LX7 base	1,211
LX7+MUL16	1,211
LX7+MUL32	1,210
HiFi 3	813
HiFi mini	854
Fusion G3	847

Table 5.1: Maximum clock frequency supported by the processor configurations

Configuration	Setup A (MHz)	Setup B (MHz)
LX7 base	166.234	157.606
LX7+MUL16	38.392	34.511
LX7+MUL32	58.395	53.907
HiFi 3	7.507	4.003
HiFi mini	13.320	8.376
Fusion G3	7.794	3.523

Table 5.2: Minimum clock frequency required for real-time processing

- f_{clk} is the clock frequency of the processor
- $N_{samples}$ is the number of samples in a given test setup. Test setup A contains 24861 samples and test setup B contains 256000 samples.
- N_{cycles} is the number of cycles needed by the architecture to process the given test setup.

The Fusion G3 has both the highest area and power consumption of the configurations when running at maximum frequency, due to it's large register files and wider Functional Units. The HiFi 3 has smaller register files and therefore achieves has a much lower size and power requirement. However, the performance of the HiFi 3 is still not far from that of the Fusion G3. Both the Fusion G3 and the HiFi 3 feature very noticeable differences in speed for the two test setups, showing that the overall performance of the implementation is very dependent on the input.

The LX7 configurations (LX7 base, LX7+MUL16 and LX7+MUL32) are all very similar to each other in terms of core size and power consumption. LX7 base and LX7+MUL16 use identical processor configuration, so the difference between them is mainly in speed. LX7+MUL32 has an additional instruction to compute the most significant word of a 32×32-bit multiplication. However, the hardware cost added by this instruction is small compared to the cost of using a coprocessor.

The HiFi mini, being designed specifically to have low power consumption, has the smallest size and power requirements of the coprocessors. It's power consumption is only slightly higher than that of the LX7 configurations, but the HiFi mini achieves greater speeds. It's core size and speed values are also about halfway in between those of the LX7 cores and the HiFi 3.

5.8 Discussion

Six different processor architectures were evaluated in this chapter. Most importantly, all of the presented architectures are able to process the tested data in real-time, whereby additional overhead through I/O can be compensated for by increasing the processors clock frequency.

The LX7 configurations benefit the least from parallelizable code, as can be seen by the overall high cycle counts and the differences in average cycle counts for the components. For example, the band-pass filter has effectively twice the length of the decimation filter and thus needs about twice the number of cycles. When using hardware parallelism, this difference would be smaller because the decimation filter also includes overhead to decimate the x-values, which would offset the overall cycle count to be closer to the band-pass filter. The LX7+MUL16 configuration is up to five times faster than the LX7 base but also has a lower output accuracy. LX7+MUL32 is between two and three times faster, but retains the output accuracy from the LX7 base.

The HiFi 3 uses both VLIW and SIMD and thus achieves a much lower cycle count than the LX7 configurations due to parallel execution. This also means that instructions that cannot be parallelized have a greater effect on the average cycle counts. The interpolation filter and the FFT reach the smallest speedup compared to LX7+MUL16 while the band-pass, envelope detector and decimation filter have higher speedups. The HiFi 3 uses 32 bits for samples but 16 bits for coefficients. Therefore the accuracy of the output is lower than that of the LX7 base and LX7+MUL32 configurations which use 32 bits for both samples and coefficients. Furthermore, the accuracy is not significantly improved by adding fractional bits to the sample data type.

The HiFi mini can perform only two multiplications per cycle compared to the HiFi 3's four multiplications. As a result, the HiFi mini has a higher cycle count for the filter system components. Furthermore, the HiFi mini uses a non-vectorized CORDIC implementation for the envelope detector which is slower than the vectorized `hypot` implementation of the HiFi 3. The HiFi mini also has only 24-bit wide registers which means that the output accuracy is lower than that of the LX7 configurations.

The Fusion G3 is largest of the evaluated processors regarding the size. It's performance varies with input data, as the cycle counts are comparable to the HiFi 3's for test setup A, but are lower than the HiFi 3's cycle counts for test setup B. The Fusion G3 can perform the same number of multiplications per cycle as the HiFi 3, but still use 32 bits for all operands. Therefore, it's output accuracy is higher than the one from the HiFi 3 and more comparable to the accuracy from the LX7 base.

It has been observed that the overall performance of any of the architectures is highly dependent on the input data and filter system configuration. Of the tested architectures, the Fusion G3 achieves the highest speed on the test data, the LX7 base has the lowest size and power requirements and the HiFi 3 and HiFi mini both balance speed and hardware costs in their own ways. The Fusion G3 can take the most advantage of highly parallelizable code, but at the same time also requires that kind of code to run efficiently. The HiFi 3 and HiFi mini reach a more even support for both parallel and sequential code, with the HiFi 3 aiming for higher speed and the HiFi mini for lower hardware costs. The LX7 configurations have the lowest hardware costs but also run the slowest, due to their lack of hardware-parallelism.

Regarding the fixed point formats, the accuracy generally increases with an increase in fractional bits. However, the results from the HiFi 3 also showed that the accuracy can get bottlenecked when a format with less fractional bits is used throughout a computation. Furthermore, an increase in fractional bits does not always result in a gain in accuracy, as some of the test values had to be scaled down to fit the fixed point formats with fewer integer bits. This scaling step means that fewer bits are actually used to increase the precision of the numbers. For example, adding two fractional bits but needing to scale the input values by 0.5 results in a net gain of only one bit of precision.

5.9 Future Work

The current implementation still has opportunities for further optimizations. These optimization include removing some of the remaining sequential code from the components, like the interpolation filter or the FFT. Especially the interpolation filter would benefit greatly from more opportunities for parallelism, as it processes a larger amount of samples. On the other hand, the FFT could also be optimized to retain a higher output accuracy. Some parts of the implementation already 'extract an exponent' from their numbers using the NSA instruction to simulate a floating point number. This is usually done to avoid overflows or to retain more bits for precision. The FFT may be able to use a similar technique as it often contains very small values, which, in a fixed point format, do not resolve very well.

Regarding the fixed point format, another optimization would be to convert the whole implementation to use a $Q1.31$ format. While this will most likely not increase the accuracy by much, it does remove the need for some of the correction shifts as many of the coprocessors have dedicated instructions for dealing with $Q1.31$ numbers. Some of the more architecture-specific optimizations are to improve the decimation filter of the HiFi 3 to make better use of the VLIW slots, or to implement a vectorized version of `hypot` for the HiFi mini.

Finally, there are some more experimental optimizations: It may be possible to combine the interpolation filter and the second decimation filter into a single component, by adding a cutoff frequency to the fractional delay filters. A combination similar to this is presented in [33], although a different approach is used to arrive at these filters. A disadvantage of this combination is that the filter coefficients need to be recalculated when the filter system configuration changes. In the current implementation, the interpolation filter uses hardcoded values due to the high number of coefficients required for all the fractional delay filters. However, the computation of the coefficients would be subject to the same benefits that the presented processor architectures already brought to the other components and thus the additional overhead may be tolerable.

Lastly it remains to be tested, whether using a Floating Point Unit after all would be an good way to increase the output accuracy or if the hardware costs and possible reductions in execution speed outweigh these accuracy gains.

6 Conclusion

This thesis presented a design space exploration for a filter system for order analysis. The reference implementation of the filter system was analysed for it's function and mathematical background. An implementation of this filter system was created for six ASIP configurations which are based on the Xtensa ISA. To save hardware costs and increase execution speed, these implementations have floating point numbers omitted from their code and instead use fixed point. The processor configurations have different capabilities for hardware parallelism, register bit widths and core size and power requirements.

The implementation features custom arithmetic for the fixed point formats which allow changing of the format at compile time. Compiler intrinsics and optimized signal processing functions were used to access the hardware-specific instructions of the different architectures and to make use of the provided SIMD capabilities. Through this process, the base implementation was adapted to each of the architectures and makes use of their specific features to increase the overall performance.

The implementations were tested, optimized and evaluated using provided data sets to assess the advantages and disadvantages of each of the configurations. The evaluation compares the different configurations in terms of both performance and cost. Effects of hardware parallelism, fixed point formats and other implementation differences were highlighted and linked to hardware features and implementation specifics.

All evaluated architectures are able run the filter system in real-time. Both the Fusion G3 and the HiFi 3 achieve significantly higher speeds than the LX7+MUL16 architecture with a speedup of up to 9.8 for the Fusion G3 and up to 8.6 for the HiFi 3. The difference between these two architectures is that the Fusion G3 achieves this speedup while retaining a full 32 bits for all samples, resulting in the relative error being about half of that of the HiFi 3. On the other hand, the HiFi 3 requires only 31.6% of the core size and 35.6% of the power of the Fusion G3. The HiFi mini also displays promising results for performance with two 24×24-bit multiplications per cycle. Of the coprocessor architectures, the HiFi mini has the lowest core size and power requirement with a power requirement of 1.49 times that of the LX7 base configuration.

Each of the architectures has specific fixed point formats, which they can process faster than others. It has been observed that the overall accuracy of the output increases with a higher number of fractional bits, but it will also reach saturation when the input values no longer fit into the format without prior scaling. The results of the HiFi 3's evaluation have also shown that, if multiple fixed point formats are used together, the accuracy is limited by the format with the least number of fractional bits.

The evaluation highlighted the differences between the architectures and their implications on the performance of the filter system. It was also shown which parts of the filter system are possible starting points for further optimizations. The information from the design space exploration give an indication of which kinds of results can be reached by the filter system with a dedicated ASIP.

Bibliography

[1] *HiFi 3 DSP User's Guide*, Cadence Design Systems, Inc., 2566 Seely Ave., San Jose, CA, 2019.

[2] Y.-J. Kim, H.-H. Kim, and H.-S. Lee, "Design of a radix-8/4/2 variable fft processor for ofdm systems," *Jouranl of Digital Convergence*, vol. 11, 01 2013.

[3] *Xtensa LX7 Processor*, Cadence Design Systems, Inc., 2566 Seely Ave., San Jose, CA, 2016.

[4] *HiFi mini DSP User's Guide*, Cadence Design Systems, Inc., 2566 Seely Ave., San Jose, CA, 2019.

[5] *Fusion G3 DSP User's Guide*, Cadence Design Systems, Inc., 2566 Seely Ave., San Jose, CA, 2019.

[6] S. Xu, F. Xing, R. Wang, W. Li, Y. Wang, and X. Wang, "Vibration sensor for the health monitoring of the large rotating machinery: review and outlook," *Sensor Review*, 2018.

[7] A. Brandt, T. Lagö, K. Ahlin, and J. Tůma, "Main principles and limitations of current order tracking methods," *Sound & vibration*, vol. 39, pp. 19–22, 03 2005.

[8] N. Robinson, "Ten reasons to optimize a processor," Cadence Design Systems, Inc., 2015, https://ip.cadence.com/uploads/770/TIP_WP_10Reasons_Customize_FINAL-pdf.

[9] K. Wang and P. S. Heyns, "A comparison between two conventional order tracking techniques in rotating machine diagnostics," in *2011 International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering*, 2011, pp. 478–481.

[10] D. Liu and J. Wang, *Application Specific Instruction Set DSP Processors*. Springer Science+Business Media LLC., 07 2010, pp. pp 415–447.

[11] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005.

[12] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.

[13] S. Arar, "Fixed-point representation: The q format and addition examples," 2017, https://www.allaboutcircuits.com/technical-articles/fixed-point-representation-the-q-format-and-addition-examples/.

[14] J. M. Conrad, "Fixed-point math and other optimizations," 2007, https://webpages.uncc.edu/~jmconrad/ECGR6185-2007-01/notes/UNCC-IESLecture23%20-%20Fixed%20Point%20Math.pdf.

[15] K. Shirriff, "Extracting rom constants from the 8087 math coprocessor's die," 2020, http://www.righto.com/2020/05/extracting-rom-constants-from-8087-math.html.

[16] J. S. Walther, "A unified algorithm for elementary functions," in *Proceedings of the May 18-20, 1971, Spring Joint Computer Conference*, ser. AFIPS '71 (Spring). New York, NY, USA: Association for Computing Machinery, 1971, p. 379–385. [Online]. Available: https://doi.org/10.1145/1478786.1478840

[17] S. A. Tretter, "Communications design laboratory," accessed November 23rd 2020, https://user.eng.umd.edu/~tretter/commlab/c6713slides/ch5.pdf.

[18] J. O. Smith, *Mathematics of the Discrete Fourier Transform (DFT)*. http://ccrma.stanford.edu/~jos/mdft/, accessed November 22nd 2020, ch. Analytic Signals and Hilbert Transform Filters, online book, 2007 edition.

[19] J. P. Wheeler, "Assigning value to the valueless... the cauchy principle value method and related ideas in divergent series," 2010.

[20] J. O. Smith, *Spectral Audio Signal Processing*. http://ccrma.stanford.edu/~jos/sasp/, accessed November 22nd 2020, ch. Primer on Hilbert Transform Theory, online book, 2011 edition.

[21] A. V. Oppenheim, J. R. Buck, and R. W. Schafer, *Discrete-time signal processing. Vol. 2*. Upper Saddle River, NJ: Prentice Hall, 2001.

[22] C. E. Shannon, "Communication in the presence of noise," *Proceedings of the IRE*, vol. 37, no. 1, pp. 10–21, 1949.

[23] V. Välimäki, *Discrete-Time Modeling of Acoustic Tubes Using Fractional Delay Filters*. Helsinki University of Technology, 1998, ch. Fractional Delay Filters.

[24] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[25] "Rad studio c++ builder," Embarcadero Inc., https://www.embarcadero.com/products/cbuilder.

[26] *Tensilica Software Development Kit (SDK)*, Cadence Design Systems, Inc., 2014, https://ip.cadence.com/uploads/103/SWdev-pdf.

[27] *NatureDSP Signal for HiFi3 with VFPU v3.1.0*, IntegrIT Technologies Ltd., 2017.

[28] *DSP Library for Fusion G3 v2.0.0*, IntegrIT Technologies Ltd., 2017.

[29] *NatureDSP Signal v1.22*, IntegrIT Technologies Ltd., 2013.

[30] S. Oberman and M. Flynn, "An analysis of division algorithms and implementations," Stanford University, Tech. Rep. CSL-TR-95-675, 02 1970.

[31] "C11 standard iso/iec 9899:201x," ISO/IEC, p. 247, 2010.

[32] *Xtensa® Instruction Set Architecture (ISA)*, Tensilica Inc., 3255-6 Scott Blvd., Santa Clara, CA 95054, 2010.

[33] A. Petrovsky, A. Stankevich, M. Omieljanowicz, and G. Rubin, "Digital order tracking analysis for rotating machinery monitoring. theory and implementation," 01 2008.