



**TECHNISCHE
UNIVERSITÄT
DRESDEN**

Faculty of Electrical and Computer Engineering

Chair of Highly-Parallel VLSI Systems and Neuro-Microelectronics

Diploma Thesis

Implementation of a Hardware-Accelerated Vibrotactile Codec for a RISC-V based Processing Element

Matthias von Wachter

Born on: 25th October 1991 in Munich

Course: Electrical Engineering

Discipline: Microelectronics

Matriculation number: 3849555

to achieve the academic degree

**DIPLOMINGENIEUR
(Dipl. Ing.)**

Supervisors

Dipl.-Ing. Matthias Jobst

Dr.-Ing. Johannes Partzsch

Supervising professor

Prof. Dr.-Ing. habil. Christian Georg Mayr

Submitted on: March 4, 2021



Topic for the Diploma thesis

for: **Matthias von Wachter**

Matriculation-no.: 3849555

Degree programme: **Diplom Elektrotechnik, 2012**

Subject: Implementation of a Hardware-Accelerated Vibrotactile Codec for a RISC-V based Processing Element

Objectives: Harnessing tactile information in addition to visual and auditive information is essential for a natural and intuitive human-machine interaction as well as convincing virtual reality systems. Especially the vibrotactile modality is important as it gives a sense of touch and surface structure. As many channels with low latency are required for an immersive experience, an IEEE working group is working on crafting a dedicated standard for efficient vibrotactile codecs. This work will research the matter of hardware acceleration and implementation of a vibrotactile codec using an existing RISC-V based processing element.

The project consists of the following tasks:

- Literature study of vibrotactile codecs and similar algorithms and existing hardware implementations thereof
- Implementation of a tactile codec in C on a RISC-V processor and characterization regarding memory and computational requirements
- Identification of the most complex parts of the codec and selection of an operation for hardware acceleration
- Implementation of a hardware accelerator for the selected operation and integration with the RISC-V core as well as verification of the functionality
- Characterization of the hardware accelerator regarding hardware effort and the achieved acceleration over the software implementation

The Diploma thesis work is to be written in English.

Supervisor(s): **Dr.-Ing. Johannes Partzsch**

Dipl.-Ing. Matthias Jobst

1st Examiner: **Prof. Dr.-Ing. habil. Christian Mayr**

2nd Examiner: **Dr.-Ing. Johannes Partzsch**

Handed out on: 31.07.2020

Submit until: 08.01.2021

Prof. Dr.-Ing. Steffen Bernet
Chairman of Examination Board

Prof. Dr.-Ing. habil. Christian Mayr
Supervising professor

Statement of authorship

I hereby certify that I have authored this document entitled *Implementation of a Hardware-Accelerated Vibrotactile Codec for a RISC-V based Processing Element* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, March 4, 2021

Matthias von Wachter

Abstract

Vibrotactile perceptions, which give a sense of touch and surface structure, will be enriching human-machine interfaces beyond the audio and visual media that are currently available. The lossy compression codec by Noll et al. uses a psychohaptic model to achieve high compression rates for vibrotactile signals with little reduction in perceived quality. This work implements the codec in the low-level programming language C. It runs on a *RI5CY* RISC-V processing element, which is configured without a floating point unit. The input signal is transformed into discrete wavelet coefficients. The codec adjusts the quantization resolution based on a psychohaptic model of the input signal spectrum. It applies deadzone quantization to the coefficients in multiple iterations to find the optimal distribution of the bit budget quantization resolution. A custom hardware accelerator improves the run time of the deadzone quantizer core function by $37\times$, cutting in half the time spent in the quantizer function block. Speedup of the complete codec ranges from 0.7 % to 3.4 %, rising with the bit budget. An increasing bit budget correlates with higher quality and resource usage as well as lower compression rates. The relatively small increase in overall performance is due to computationally complex parts of the codec, namely the psychohaptic model and the SPIHT encoder, for which hardware acceleration is less feasible. Synthesis of the System on Chip (SoC) on the 28nm SLP process by Globalfoundries, Inc. results in 1.09 % ($3450\text{ }\mu\text{m}^2$) area added by the accelerator to the SoC at a target frequency of 252.5 MHz. A system frequency of 310 MHz would enable processing vibrotactile signals (Sampling Frequency $f_s = 2800\text{ Hz}$) in real time. With an FFT accelerator (also developed at the HPSN institute) 250 MHz would suffice.

Zusammenfassung

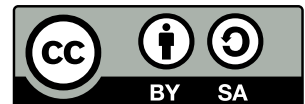
Vibrotaktile Sinneseindrücke, welche einen Eindruck von Berührung und Oberflächenstrukturen ermöglichen, werden Mensch-Maschine Schnittstellen jenseits der momentan eingesetzten audiovisuellen Medien erweitern. Der verlustbehaftete Kompressionscodec von Noll et al. nutzt ein psychohaptisches Modell, um hohe Kompressionsraten für vibrotaktile Signale bei geringer Reduzierung der wahrgenommenen Qualität zu erreichen. Diese Arbeit implementiert den Codec in der Low-Level-Programmiersprache C. Der Code wird auf einem RI5CY RISC-V Prozessorelement ausgeführt, das ohne Floating Point Unit (FPU) konfiguriert ist. Das Eingangssignal wird in diskrete Wavelet-Koeffizienten transformiert. Der Codec justiert die Auflösung der Quantisierung auf Grundlage eines psychohaptischen Modells, welches Erkenntnisse aus dem Spektrum des Eingangssignals gewinnt. Eine Deadzone-Quantisierung wird in mehreren Iterationen auf die Koeffizienten angewandt um eine optimale Verteilung der Quantisierungsauflösung zu erreichen. Ein für den Codec entworfener Hardware-Beschleuniger verbessert die Laufzeit der Deadzone-Quantisierer Kernfunktion um das 37-fache, was dazu führt, dass die Hälfte der Zeit im Quantisierer-Funktionsblock verbracht wird. Die Beschleunigung des gesamten Codecs reicht von 0.7 % bis 3.4 % und steigt mit dem Bit-Budget, also bei höherer Qualität, gesteigertem Ressourcenverbrauch und niedrigeren Kompressionsraten. Die relativ geringe Verbesserung der insgesamten Laufzeit auf rechenintensive Teile des Codecs zurückzuführen, insbesondere das psychohaptische Modell und der SPIHT-Encoder, für die eine Hardware-Beschleunigung weniger gut realisierbar ist. Bei der Synthese des System on Chip (SoC) auf dem 28nm SLP Prozess von Globalfoundries, Inc. verursacht der Beschleuniger 1.09 % ($3450\text{ }\mu\text{m}^2$) zusätzlichen Flächenverbrauch bei einer Zielfrequenz von 252,5 MHz. Eine Systemfrequenz von 310 MHz würde die Verarbeitung von vibrotaktile Signalen (Sampling Frequenz $f_s = 2800\text{ Hz}$) in Echtzeit ermöglichen. Mit einem FFT-Beschleuniger (ebenfalls am HPSN-Institut entwickelt) würden 250 MHz ausreichen.

Contents

List of Figures	vii
List of Tables	viii
Symbols	ix
Acronyms	x
1 Introduction	1
1.1 Vibrotactile Signals	1
1.2 RISC-V Processors	2
1.3 Thesis Overview	2
2 State of the Art	3
2.1 Characteristics of Vibrotactile Signals	3
2.2 Comparison to Audio- and Video Signals	4
2.3 Common Compression Techniques	5
2.3.1 Huffman Coding	5
2.3.2 Sampling and Quantization	6
2.3.3 Noise Shaping by Dithering	7
2.3.4 Using Perception Thresholds to Mask Noise	7
2.3.5 Time-Frequency Transformations	8
2.3.6 Set Partitioning in Hierarchical Trees (SPIHT)	10
2.3.7 Linear Predictive Coding (LPC)	12
2.4 Audio Compression with MP3	12
2.5 Compression Codecs for Vibrotactile Signals	13
2.6 A Rate-scalable Perceptual Wavelet-based Vibrotactile Codec	14
2.6.1 Overview	14
2.6.2 Psychohaptic Model	14
2.6.3 Quantization	15
2.6.4 SPIHT Coder	16
2.7 Hardware Accelerators	16
2.8 Consequences for this Work	17
3 Software Design	18
3.1 Development	18
3.1.1 Overview	18

3.1.2	Software Structure	19
3.1.3	External Libraries	20
3.1.4	Optimizations	20
3.1.5	Unit testing for desktop and embedded targets	21
3.1.6	Implementation Gaps	21
3.2	Software Results	22
3.2.1	Memory Requirements	22
3.2.2	Computational Performance	22
3.3	Search for Acceleration Potential	23
3.3.1	Profiling Tools and Techniques	23
3.3.2	Candidates for Optimization	25
4	Hardware Design	29
4.1	Data Flow	29
4.1.1	Constraining the Input	30
4.1.2	Multiplier	32
4.1.3	Rounding	33
4.2	Hardware Verification	34
4.2.1	Systemverilog Test Cases	34
4.2.2	Accuracy Impact of Hardware Optimizations	35
4.2.3	Using the Accelerator in Software	35
4.3	Synthesis	36
4.3.1	Frequency Characteristics	36
4.3.2	Resource Usage	37
4.3.3	Optimizations	38
4.4	Hardware Design choices	39
4.5	Hardware Results	40
4.5.1	Latency and Throughput	40
4.5.2	Impact of FFT accelerator	40
4.5.3	Hardware-accelerated Vibrotactile Codec Performance	41
5	Conclusion	43
5.1	Summary	43
5.2	Future Research Opportunities	44
	Bibliography	45
A	Appendix	50
A.1	Detailed Performance Analysis for x86	50
A.2	Code Listing: <code>generate_dzquantizer_tests.py</code>	54

This work is licensed under a Creative Commons “Attribution-ShareAlike 4.0 International” license.



List of Figures

2.1	Dimensions of tactile perception.	4
2.2	Huffman coding.	5
2.3	Signal sampling and quantization	6
2.4	Haar wavelet.	9
2.5	Frequency representation of a Discrete Wavelet Transform	10
2.6	Wavelet Filterbank	10
2.7	Set Partitioning in Hierarchical Trees (SPIHT)	12
2.8	Structure of a perceptual encoding/decoding system	13
2.9	Perceptual Deadband data reduction	14
2.10	Encoder structure of the vibrotactile codec.	14
3.1	Encoder structure of the vibrotactile codec.	19
3.2	Distribution of CPU time in the vibrotactile codec on x86.	24
3.3	Distribution of CPU cycles in the vibrotactile codec on RI5CY	24
3.4	Impact of reduced logarithm accuracy on the ST-SIM quality metric.	27
3.5	Impact of reduced logarithm accuracy on the ST-SIM quality metric. Difference.	28
4.1	Deadzone quantizer data flow graph.	31
4.2	Deadzone Quantizer FSM	32
4.3	Structure of a 32bit floating point number	32
4.4	Impact of reduced accuracy in floor() function.	35
4.5	Impact of reduced accuracy in mantissa multiplication on accuracy.	36
4.6	Area and leakage power over frequency.	37
4.7	Distribution of area use in the synthesized design.	38
4.8	Distribution of leakage power use in the synthesized design.	38
4.9	Impact of reduced accuracy in mantissa multiplication on deadzone quantizer area and leakage power.	39
4.10	RI5CY CPU Cycles needed to quantize 4960 samples.	41
4.11	Distribution of CPU cycles in execution of <code>Quantize()</code> on the RI5CY processor.	42

List of Tables

2.1	Overview of mechanoreceptors found in hairless skin.	3
2.2	Huffman Codes for the sentence “this is an example of a huffman tree”.	6
3.1	Absolute error thresholds for floating point numbers used in unit testing. . . .	21
3.2	Memory requirements	22
3.3	Software computational performance.	23
4.1	Latency and raw throughput of the deadzone quantizer hardware accelerator . .	40
4.2	Performance characteristics of the hardware-accelerated vibrotactile block en- coder.	41
4.3	Performance characteristics of the hardware-accelerated vibrotactile block en- coder without FFT.	42
A.1	Distribution of execution time on x86 with bit budget 8.	52
A.2	Distribution of execution time on x86 with bit budget 8.	53

Symbols

Notation	Description
a_p	Level of a signal peak
B	Frequency band
b	Bit budget
E	Exponent of a floating point number
$E_{M,B}$	Global masking threshold of a wavelet band
$E_{S,B}$	Signal energy in a wavelet band
f_p	Frequency of a signal peak
f_s	Sampling frequency
κ	Ratio of ΔS and S
M	Mantissa bits (fraction part) of a floating point number
$m_p(f)$	Peak dependent masking threshold
$\mathcal{F}(\omega)$	Fourier transform
p	Signal peak
Δ_{quant}	Quantization interval
S	Intensity of a reference stimulus
$S(t)$	Signal continuous in time and amplitude
ΔS	Difference of intensity between two stimuli
ΔS_{quant}	Quantization error
S_i	Signal with continuous amplitudes at discrete points in time
S_{quant}	Quantized signal
S	Sign bit of a floating point number
s	Wavelet scale factor
τ	Wavelet translation factor
$t(f)$	Absolute threshold of perception
T	Quantization/significance threshold
V_{th}	Threshold voltage
\hat{w}_{max}	Maximum quantized value

Acronyms

Notation	Description
ADAS	Advanced Driver-Assistance Systems
AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
AR	Augmented Reality
ASIC	Application Specific Integrated Circuit
CPU	Central Processing Unit
DCT	Discrete Cosine Transform
DFT	Discrete Fourier Transform
DSP	Digital Signal Processor
DWT	Discrete Wavelet Transform
ECG	electrocardiogram
EZW	Embedded Zero-tree Wavelet
FFT	Fast Fourier Transform
FLWT	Fast Lifting Wavelet Transform
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FSM	Finite State Machine
GPL	GNU Public License
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HPSN	Chair of Highly-Parallel VLSI Systems and Neuro-Microelectronics
IP	Intellectual Property
ISA	Instruction Set Architecture
JND	Just Noticeable Difference
LPC	Linear Predictive Coding
LSB	Least Significant Bit
MAU	Model Application Unit
MDCT	Modified Discrete Cosine Transform
MNR	Mask-to-Noise-Ratio
MPU	Model Providing Unit
NaN	Not a Number
PD	Perceptual Deadband
PDK	Process Design Kit

Notation	Description
PULP	Parallel Ultra Low Power
RAM	Random Access Memory
RTL	Register Transfer Logic
SAQ	Successive-Approximation Quantization
SIMD	Single Instruction, Multiple Data
SMR	Signal-to-Mask-Ratio
SNR	Signal to Noise Ratio
SoC	System on Chip
SPIHT	Set Partitioning in Hierarchical Trees
SRAM	Static Random Access Memory
SSE2	SSE2 (Streaming SIMD Extensions 2
ST-SIM	Spectral Temporal SIMilarity
TOP	teleoperator
TPTA	telepresence teleaction
VR	Virtual Reality
WSL	Windows Subsystem for Linux

1 Introduction

1.1 Vibrotactile Signals

Currently most communication from computers to humans happens via screens and loud speakers. Beyond these purely audio-visual stimuli lie other human sensations like smell and touch.

Mechanical buttons do still play a role for changing the sound volume of smart phones or controlling heavy machinery and cars, but the industry trend goes towards replicating those interactions with touch screens, sometimes with added vibrotactile feedback from a single motor vibrating the device. In the new SpaceX Crew Dragon vehicle astronauts rely solely on touch screens to change flight path parameters[Cra20]. The advantages of superior reliability compared to mechanical switches and a more modern, intuitive user interface come with the disadvantage of even more reliance on purely visual interaction.

One always needs to look at the “button” one wants to trigger instead of just feeling where it is and acting accordingly. This poses problems, for example in cars when drivers should be looking at the road instead of at touchscreens while adjusting the air conditioning.

Vibrotactile sensations can play a role for getting the attention of distracted human drivers in Advanced Driver-Assistance Systems (ADAS) and self-driving settings. For example the seat belt or seat itself could vibrate to indicate problems to the driver[PdB16].

Another application is Virtual Reality (VR), where tactile sensations would immensely increase immersion, compared to the current standard of one screen for each eye and one loud-speaker for each ear.

In contrast to audio signals, which humans perceive in their two ears, there are about 17 000 tactile receptors in one hand alone[Chu21; Sch01]. This makes multiple haptic data channels very desirable.

VR needs low latency processing of multiple such data streams for a realistic experience without motion sickness. Doing so purely in software, at a guaranteed latency, is a hard problem. Operating system level multitasking allots time slots to each process that wants to run “simultaneously”. If one process takes too long it is canceled in favor of the next one, lest it block the remaining processes.

Offloading some tasks to hardware accelerators with small embedded processors can help. Since they are made for one specific work load, like processing a signal with a certain algorithm, their performance is more predictable and can be optimized for greater efficiency and speed. Then delays caused by multitasking in the main processor do not directly effect stuttering in the data streams.

1.2 RISC-V Processors

Currently most embedded processors use core designs by ARM, where licensing costs for commercial products range from 1% to more than 2% of the chip cost[Shi13]. Since fees are calculated per core, using many small cores in separate accelerators quickly becomes very expensive. There is also no way to change or add instructions to the ARM Instruction Set Architecture (ISA), which prevents potentially useful modifications for custom accelerators.

The RISC-V ISA allows such flexibility by being open and free for all to use. While cores from commercial vendors like SiFive have licensing costs similar to ARM cores, purely open-source, free designs are also available. Here the hardware description in languages such as Systemverilog is available on open source platforms like Github. One example is the 32bit microcontroller RI5CY. Originally developed as part of the Parallel Ultra Low Power (PULP) platform at the ETH Zürich and University of Bologna, it has been under the stewardship of the OpenHW Group since February 2020 with the new identifier CV32E40P[21]. The lack of licensing costs makes it feasible to use many of these power and area efficient cores as parallel vibrotactile coders.

1.3 Thesis Overview

This work takes advantage of a RI5CY core maintained at the Chair of Highly-Parallel VLSI Systems and Neuro-Microelectronics (HPSN) chair of TU Dresden to run a vibrotactile codec. The encoder is implemented in C, a programming language suitable for embedded programming that can target the RI5CY. Profiling the software running on the RI5CY gives hints on parts of the codec amenable to hardware acceleration. After implementing a suitable hardware accelerator the RI5CY core is synthesized together with its subsystems and the accelerator. The resource usage added by the hardware accelerator to the RI5CY System on Chip (SoC) is explored in terms of area and power. System frequencies that guarantee real time encoding of vibrotactile signals under minimal power consumption will be evaluated.

Following this introduction in chapter 1, chapter 2 introduces existing knowledge about vibrotactile signals and their perception by humans, then gives a brief overview of some compression algorithms before focusing closer on one codec that processes vibrotactile signals. Chapter 3 explains how the software was developed and tested. Based on the knowledge gained about acceleration potential of the software a suitable part of the algorithm is chosen and accelerated in chapter 4. This chapter also elaborates on design choices made to keep power and area usage in check while keeping the potential for real time execution of the vibrotactile codec. After summarizing the main outcomes of this diploma thesis chapter 5 gives a preview of future research opportunities.

2 State of the Art

2.1 Characteristics of Vibrotactile Signals

Human haptic perception is based on a complex mix of sensations, commonly separated into two groups. Kinesthetic receptors track the activation, movement and position in z of muscles and joints. Phenomena like pressure, temperature, surface texture, among others, are registered by tactile receptors [Liu+17].

To quantify the human psychological perception of these physical features a coordinate system in which the sensations can be placed is needed. In a useful coordinate system the constituent unit vectors (dimensions) are orthogonal to each other. This means selecting a set of features, in which the magnitude of each dimension is (ideally) completely uncorrelated to all the other features. In a literature review Okamoto et al. search for a coordinate system to categorize sensations experienced while touching different materials. They propose using the five dimensions of hardness, thermal conductivity, friction, and micro/macro roughness [ONY13]. How those dimensions are reflected in real materials is shown in Figure 2.1.

Four different types of tactile receptors are each sensitive to another type of mechanical sensation, as listed in Table 2.1. Different kinds of stimuli correspond to dissimilar frequency ranges registered by the receptor types [Liu+17]. Pacinian corpuscles play the biggest role in perceiving vibrotactile signals [Lan+10][BHY05].

The remaining work focuses on this subset of human tactile sensations, which can be recorded and reproduced using vibration.

	Ruffini Ending	Merkel Cell	Meissner corpuscle	Pacinian corpuscle
Best stimulus	Stretch	Pressure, edges, corner, points	Lateral motion	High-frequency vibration
Example	Holding large objects	Reading Braille fonts	Sensing slippage of objects	
Frequency range (Hz)	N/A	0-100	1-300	5-1000
Most sensitive frequency (Hz)	N/A	5	50	200

Table 2.1: Overview of mechanoreceptors found in hairless skin (e.g. hands). Adapted from [Liu+17, Table 1]

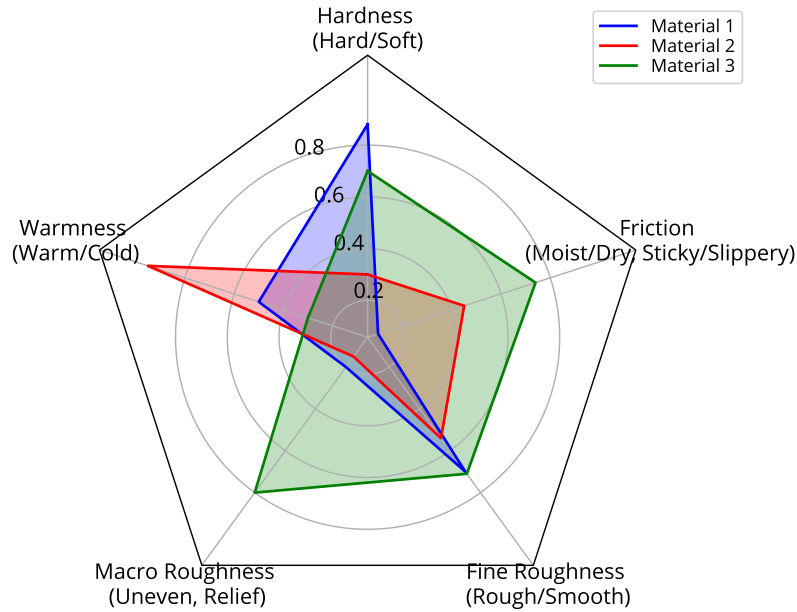


Figure 2.1: Dimensions of tactile perception (adapted from [ONY13, Fig. 1])

2.2 Comparison to Audio- and Video Signals

A fundamental difference between (single-channel) tactile and video signals is the dimensionality of their samples. The samples of video signals (commonly called frames) consist of 2D pixel matrices, where each pixel is represented by three color channels. In contrast, each sample of a vibrotactile or audio signal is just a 1D scalar. The remaining report focuses on those two signal types since the structure of video signals is significantly different from, and more complex than either audio or vibrotactile signals.

The frequencies a normal human ear can hear vary between 20 Hz and 20 kHz, while tactile receptors can only sense frequencies up to 1 kHz (see table 2.1). According to the Nyquist-Shannon sampling theorem [Sha48a; Sha48b] the sampling frequency f_s needs to be at least twice the recorded bandwidth to guarantee perfect signal reconstruction [Liu+17]. Therefore the minimum f_s to record the full perceptible range of audio signals is 40 kHz compared to the 2 kHz needed for tactile signals. The tactile codec proposed in [Nol+20] (and hardware accelerated for this diploma thesis) uses $f_s = 2800$ Hz.

Latency requirements are stricter for tactile signals than for audio. Humans generally do not notice an audio signal delay when it is smaller than 70-80 ms [BSK05]. But test subjects noticed tactile latency when it was only around 41 ms and reported a changed experience of a stimulus when it was delayed by a mean of 59 ms [Oka+08].

The number of audio channels needed for even sophisticated surround sound recording setups like 5.1 and 7.1 is less than ten. Theoretically only two should be needed since the human body registers sound only with its two ears. In contrast a multitude of tactile receptors is embedded in the human skin. For realistic VR and Augmented Reality (AR) applications hundreds or thousands of simultaneous vibrotactile channels could be needed. Still useful scenarios can be covered with far less channels.

Although humans can roughly pinpoint the source of an audio signal by comparing the slightly different delays with which an air pressure wave arrives at the ears, no direction of the air vibration itself can be detected. For vibrotactile signals the Pacinian corpuscle has a different sensitivity depending on the orientation of the vibration towards the skin. Vibrations perpendicular to the skin surface are felt a bit more strongly than vibrations parallel to the skin [BHJ99]. Landin et al. show a method to map vibrations recorded along three axes to a

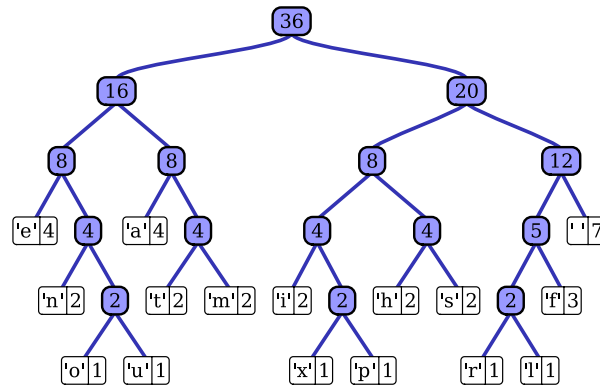


Figure 2.2: Huffman coding of the sentence “this is an example of a huffman tree”. Numbers on blue background denote the number of occurrences represented by that branch, those on white background say how often that symbol occurred in the sentence. See Table 2.2 for the resulting codes of each letter. ©[Met07]

one dimensional signal with minimal information loss [Lan+10].

2.3 Common Compression Techniques


Most multimedia codecs use a combination of lossless and lossy compression. While the former enables perfect signal reconstruction, the latter sacrifices some part of the data for the goal of smaller file sizes.

2.3.1 Huffman Coding

Huffman Coding compresses arbitrary binary data without loss. First the source data is divided into multiple symbols, for example the letters, number and signs of the extended ASCII alphabet. Then the Huffman coder counts how often each of the symbols occurs in the data and ranks them accordingly. The two symbols that occur least often in the data are combined into a new auxiliary symbol, after which the process is repeated with the two symbols that now have the lowest probability of coming up in the data. The number of binary merges necessary to combine one symbol with all the others is equal to the number of bits needed to encode that symbol. The message code for that symbol is produced by assigning a binary 0 or 1 depending on the branch taken at each each of the combinations [Huf52].

Figure 2.2 illustrates the process for the sentence “this is an example of a huffman tree”. All of the letters are sorted by the number of uses in the sentence. The total number of symbols is 36 and forms the root of the decision tree. At each of the branches one bit of the code is determined. In this example taking the right branch is mapped to 1 and the left to 0. Thus the letter “a” is mapped to the code 10 as one reaches it from the root of the tree by first taking the left branch, then the right and again the left branch. Table 2.2 shows the codes derived in this manner for all the letters of the sentence. Using those codes the original sentence would take up only 135 bits of memory if the structure of the tree does not have to be saved.. Were it encoded with an extended ASCII alphabet, in which each symbol has eight bits, the Huffman coded sentence would take up $8 \text{ bit} * 36 = 288 \text{ bit}$. The achieved reduction in file size is then $\frac{288 \text{ bit} - 135 \text{ bit}}{288 \text{ bit}} \approx 53 \%$.

Char	Freq	Code	Char	Freq	Code	Char	Freq	Code
space	7	111	i	2	1000	l	1	11001
a	4	010	m	2	0111	o	1	00110
e	4	000	n	2	0010	p	1	10011
f	3	1101	s	2	1011	r	1	11000
h	2	1010	t	2	0110	u	1	00111
						x	1	10010

Table 2.2: Huffman Codes for the sentence “this is an example of a huffman tree”. Adapted from [Dco07].

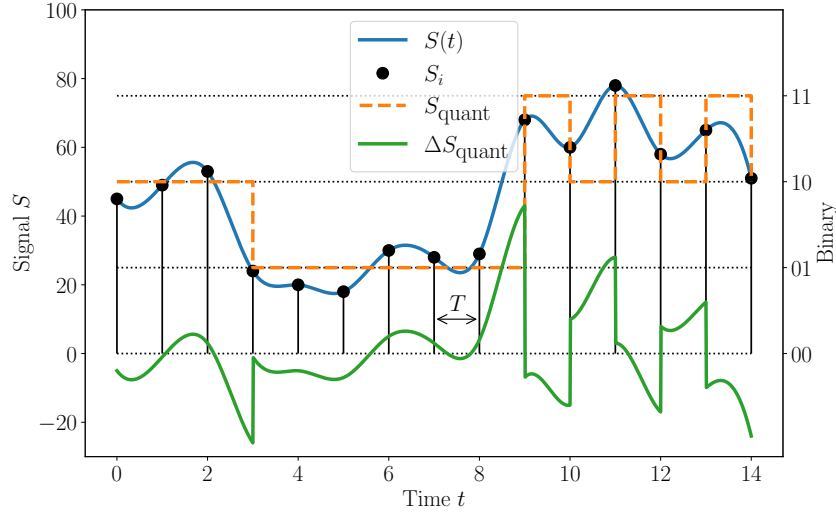


Figure 2.3: Sampling and quantization of a continuous signal $S(t)$ at intervals $T = 1/f_s$ resulting in the signal samples S_i . Quantization of S_i returns the discrete signal S_{quant} with the quantization error $\Delta S_{\text{quant}} = S_{\text{quant}} - S(t)$

2.3.2 Sampling and Quantization

The first step in transforming a continuous analog signal into a discrete digital signal is sampling and quantization. Sampling measures the amplitude of an analog signal $S(t)$ at fixed intervals $T = 1/f_s$ with the sampling frequency f_s . Quantization maps the continuous amplitude of the resulting samples S_i to a signal S_{quant} consisting of discrete values (see Figure 2.3). The difference between the analog source signal $S(t)$ and the quantized signal S_{quant} is the quantization error ΔS_{quant} .

$$\Delta S_{\text{quant}} = S_{\text{quant}} - S(t) \quad (2.1)$$

Quality of the reproduced signal can be traded for more efficient compression (smaller file sizes) in two fundamental ways. Lowering the sampling rate f_s leads to reduced bandwidth. Decreasing the number of bits allocated to each sample diminishes the dynamic range and distorts the signal. The choice of quantization intervals is essential to most compression techniques.

2.3.3 Noise Shaping by Dithering

Quantization noise is the main distortion introduced by digitizing and compressing an analog signal. It occurs when the discrete values of digital data samples do not match up exactly with the continuous values of the source signal. Since the rounding operation to the nearest discrete quantized value is deterministic, this quantization noise is statistically dependent on the source signal [Ben48].

An example quantization scheme always rounds to the nearest integer. Then according to Equation 2.3.2 the quantization error for $S(t) = 4.8$ and $S_{\text{quant}} = 5.0$ is $\Delta S_{\text{quant}} = -0.2$. This is the case every time the fractional part of $S(t)$ is 0.8. Such a deterministic relationship of the quantization error with the source signal is more noticeable than statistically independent noise overlaid on the signal.

Dithering is used to make the quantization error ΔS_{quant} statistically independent of the signal $S(t)$. In its most basic form white noise X , a random signal that has equal intensity at all frequencies, is added to the source signal. Then the quantization error with such dither is

$$\Delta S_{\text{quant}} = (S + X)_{\text{quant}} - X - S_{\text{quant}} \quad (2.2)$$

By subtracting the dithering noise from the quantized dithered signal the average amount of noise introduced by quantization stays the same [JR72, p. 1295f].

Noise Shaping shifts noise energy to different frequencies to make it less noticeable to the human ear. The first and simplest implementation was described by C. C. Cutler in a 1954 patent [Cut54]. In it the quantization error of a previous sample $i - 1$ is subtracted from the next signal sample.

$$S_{\text{quant},i} = (S_i - \Delta S_{\text{quant},i-1}) + \Delta S_{\text{quant},i} \quad (2.3)$$

While this does effectively double the cumulative error amplitude if the errors at neighboring points in time are uncorrelated, it also averages out quantization errors over a longer period of time. Therefore noise is shifted to higher frequencies. At those higher frequencies noise is less perceptible to humans and can be reduced or eliminated with a low-pass filter.

2.3.4 Using Perception Thresholds to Mask Noise

One form of lossy compression, Perceptual Encoding, uses insights from psychological research about human perception of physical stimuli, also called psychophysics, like audio (psychoacoustics) or haptic signals (psychohaptics) to make the difference between the original and the reconstructed signal imperceptible [Bra99].

The first to quantitatively explore perception thresholds was Ernst Weber in 1834, who published the results of his experiments in 1851 [Web51]. He claims a mathematical relationship between the physical magnitude of a stimulus and its perceived intensity [Ste+11, p.90]. The resulting Weber's Law says that the smallest possible perceived difference of two stimuli intensities is not an absolute difference of physical magnitudes, but depends on the ratio κ of the difference between the stimuli ΔS and the absolute value of the reference stimulus S .

$$\frac{\Delta S}{S} = \kappa = \text{constant} \quad (2.4)$$

ΔS is also called the Just Noticeable Difference (JND).

Parts of the signals whose amplitude or frequency differs less than the perception threshold ΔS from a previous signal element can be either set to zero or left as pure noise. This process is called masking since a mask covering only the relevant parts is applied to the signal.

2.3.5 Time-Frequency Transformations

After transforming a signal from the time to the frequency domain it can be filtered into different parts of the spectrum. This frequency view often corresponds better to how human hearing works than the time domain [Dau92, p.6].

Fourier Transforms

The basis of most time-frequency transformations is the Fourier Transform $\mathcal{F}(\omega)$. For any function $f(t)$ that satisfies Dirichlet's Test and for which $\int_{-\infty}^{+\infty} |f(t)|dt$ converges the Fourier Integral is

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} e^{i\omega(t-\tau)} f(\tau) d\omega d\tau \quad (2.5)$$

at every value where $f(t)$ is continuous.

The Fourier Transform is then defined as

$$\mathcal{F}(\omega) = \int_{-\infty}^{+\infty} e^{-i\omega t} f(t) dt \quad (2.6)$$

A function $f(x_\nu)$ with a period of 2π that is only defined at N discrete points in time denoted as

$$x_\nu = \nu h \quad (\nu = 0, 1, \dots, N), \quad h = \frac{2\pi}{N} \quad (2.7)$$

can be described by a Fourier Series known as Discrete Fourier Transform (DFT)

$$g(x) = \sum_{k=-n}^n c_k e^{ikx_\nu} \quad \text{where } c_{-k} = \bar{c}_k \quad (2.8)$$

with the complex Fourier coefficients

$$c_k = \frac{1}{N} \sum_{\nu=0}^{N-1} f(x_\nu) e^{-ikx_\nu} \quad (2.9)$$

\bar{c}_k is the complex conjugate of c_k , so that if $c_k = a + b * i$ then $\bar{c}_k = a - b * i$.

Computing all coefficients of $g(x)$ therefore requires $\mathcal{O}(N^2)$ operations. Fast Fourier Transform (FFT) can reduce the problem complexity to $\mathcal{O}(N \log N)$. This is achieved by using the fact that $e^{-\frac{2\pi i}{N}} = e^{-\frac{2\pi i}{N/2}}$ for even values of N and computing odd and even-numbered coefficients separately to sum up the addends of equation 2.8 in a way that requires only $N \log N$ calculations [Bro12, p.1006ff]. Fast Fourier Transform (FFT) forms a part of many compression algorithms, including the Vibrotactile Perceptual Codec proposed by Noll et al. in [Nol+20].

Discrete Cosine Transform (DCT)

Discrete Cosine Transform (DCT) uses schemes based on FFT to transform a data sequence $X(m)$, $m = 0, 1, \dots, (M-1)$ into the frequency domain via cosine functions [ANR74].

$$G_x(k) = \frac{2}{M} \sum_{m=0}^{M-1} X(m) \cos \frac{(2m+1)k\pi}{2M}, \quad k = 1, 2, \dots, (M-1) \quad (2.10)$$

The advantage lies in improved energy compaction, which means that most of the energy of a common real world signal (skewed towards one frequency range) can be expressed with relatively few coefficients. Originally developed for pattern recognition, its main use nowadays is

in the lossy image compression algorithm JPEG [Wal92]. Modified Discrete Cosine Transform (MDCT) uses overlapping windows to avoid artifacts occurring at boundaries.

$$X(k, n) = w(n) \sqrt{\frac{2}{M}} \cos \left[\frac{\pi}{M} \left(n + \frac{M+1}{2} \right) \left(k + \frac{1}{2} \right) \right], \quad k = 0, 1, \dots, (M-1), \quad n = 0, 1, (2M-1) \quad (2.11)$$

The application specific window function $w(n)$ is $2M$ wide [Luo09, p.590]. Modified Discrete Cosine Transform (MDCT) is used by most modern audio codecs, including MP3, AAC, [Bra99] and Opus [Vos+13].

Discrete Wavelet Transform (DWT)

The distinction of wavelets from normal waves is as follows: whereas waves go on forever in the temporal domain, wavelets are defined only around a certain interval and are zero elsewhere. The most simple wavelet, called Haar wavelet (pictured in 2.4) is similar to a rectangular function. Ingrid Daubechies and others have developed more complex forms, among them the CDF 9/7 wavelet, described as a biorthogonal wavelet with four vanishing moments [Dau92, p.278ff].

Wavelets are constructed from base wavelet, the Mother Wavelet $\psi(t)$, by scaling it by a scale factor s and shifting it relative to the axis origin by a translation factor τ .

$$\psi_{s,\tau}(t) = \frac{1}{\sqrt{s}} \psi \left(\frac{t - \tau}{s} \right) \quad (2.12)$$

In contrast to the transformations mentioned above, Wavelet Transforms retain not only information about the frequencies the original signal is composed of (through the scale dimension s), but also at which time those frequencies occur (through the translation dimension τ) [Dau92, p.1]. They do this by performing a convolution of each input signal section with a wavelet. If the wavelet and the input signal have similar frequencies their stronger correlation is reflected in the transform coefficients. Thus the wavelet acts like a band-pass filter and can be modeled as such. Perfect analysis and reconstruction of a signal is possible with filter banks using sub band coding modeled after these wavelets [Val13].

Figures 2.5 and 2.6 illustrate the analysis process for a three level Discrete Wavelet Transform (DWT). At each level the spectrum is divided in half and the coefficients determined by the part with the higher frequencies are saved.

The resulting data sequences are downsampled by a factor of two: The number of samples in the lower and upper part is half the number of samples of the original signal, keeping the total number of samples constant. The high-pass filtered samples are also called the *detail* part of the signal and the low-pass filtered the *approximation* part. Synthesis reverses this process by upsampling and combining the *detail* and *approximation* components to reconstruct the original signal.

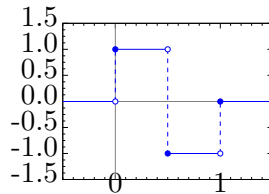


Figure 2.4: Haar wavelet ☹️👤👤[Ome06].

The CDF 9/7 Wavelet has found the most uses, among them lossy compression in the FBI fingerprint database [Bri98], the JPEG2000 image codec [CSE00] and the Perceptual Wavelet-based Vibrotactile Codec by Noll et al. [Nol+20].

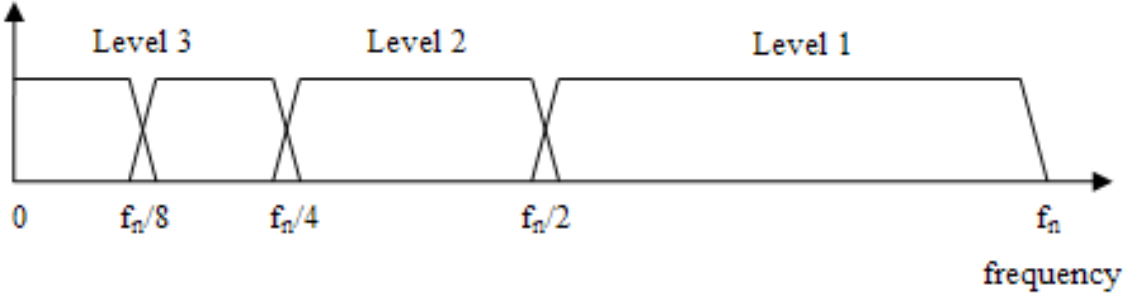


Figure 2.5: Frequency representation of a three level Discrete Wavelet Transform. ©[Joh05]

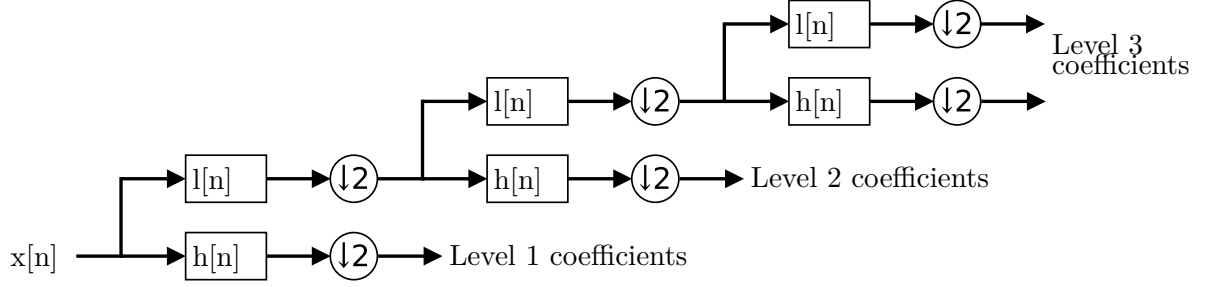


Figure 2.6: Wavelet transform analysis filterbank of a three level Discrete Wavelet Transform. ©[Fal18]

For some wavelets, among them CDF 9/7, a simplified algorithm for calculating the DWT can be used. Daubechies et al. accomplish it with only two simple step types, called *predict* (for odd numbered samples) and *update* (even numbered) [DS97]. Both of these add the sum of two neighboring samples x_{i+1} and x_{i-1} , multiplied by a step-specific factor λ , to a sample x_i .

$$x_{i,\text{new}} = x_{i,\text{old}} + \lambda(x_{i-1,\text{old}} + x_{i+1,\text{old}}) \quad (2.13)$$

At the end of the Fast Lifting Wavelet Transform (FLWT) the samples x_{new} are either divided (odd numbered) or multiplied (even numbered) by a scale factor K .

2.3.6 Set Partitioning in Hierarchical Trees (SPIHT)

The Embedded Zero-tree Wavelet (EZW)[Sha93] and Set Partitioning in Hierarchical Trees (SPIHT)[SP96] algorithms compress wavelet coefficients through ordering them by importance and encoding the location of insignificant (quantized to zero) values in a zero tree. An encoded data stream can be cut off at any point and still be decoded. The length of the bit sequence only determines the amount of distortion introduced by the compression, a feature called progressive or embedded coding. Compression can be (almost) lossless if all coefficients are encoded at their full accuracy. It is not completely lossless because wavelets used in the DWT have non-integer taps, which can only be represented with finite precision during computation.

Zero trees work on the assumption that low/insignificant values of higher frequency components of a DWT performed on real world data are predictive of corresponding lower frequency coefficients also being low or zero. This can be seen in 2D image data where a lack of edges (*detail*/high frequency) is often associated with larger uniform patches (*approximation*/low frequency). In 1D audio, haptic or electrocardiogram (ECG)[Poo+04] data a lack of spikes (*detail*/high frequency) is correlated with longer time periods during which the signal is nearly constant (*approximation*/low frequency). (The following elaborations will focus on 1D SPIHT as described by [Poo+04] and [ZJ10], since the topic of this thesis is compressing haptic signals.) The location of all the zeroed (insignificant) coefficients are saved as a highly efficient

zero tree, as zeroes form a big part of the total number of coefficients. A zero tree encodes an insignificant coefficient as either a zero root, a descendant of a root, or an isolated zero. A lower frequency coefficient descends from a higher frequency coefficient if it stems from an earlier frequency division. In figure 2.6 the Level 2 and 3 coefficients descend from the Level 1 coefficients. A zero that descends from a zero root need not be saved. If an insignificant coefficient does not descend from a root it is either an isolated zero (some of its descendants are significant), or a new root (no significant descendants).

Significance of coefficients is not measured against a single threshold T but thresholds T_i specific to each step of a Successive-Approximation Quantization (SAQ). Each threshold T_i is half of the previous threshold T_{i-1} : $T_i = T_{i-1}/2$. The initial threshold T_0 needs to satisfy $|X_j| < 2T_0$ for all transform coefficients x_j .

During a run of EZW coefficients are sorted into two lists: the *dominant* and the *subordinate* list. A *dominant* list contains the *coordinates*/indices of coefficients that have not yet been determined to be insignificant. The values of those coefficients are saved in the *subordinate* list. During each *dominant* pass coefficients in the *dominant* list are compared to the significance threshold T_i . If their absolute value is below T_i their location is saved in the zero tree. If not, those significant coefficients are put into one of two lists according to their sign. Those lists are ordered first by scale s from coarse to fine and in those scales by magnitude from large to small [Sha93].

In the original EZW symbols are then encoded separately for positive and negative significant coefficients, zero roots and isolated zeros. The symbols are sorted in order of falling importance. In combination with the commonly known property of real numbers in binary representation, where cutting digits at the end only reduces the accuracy, this delivers the aforementioned quality of progressive/embedded encoding. A precise bit rate can be set by simply stopping the encoding/decoding process at a suitable digit.

Said et al. improve on the Embedded Zero-tree Wavelet (EZW) algorithm by Shapiro et al. [Sha93] with Set Partitioning in Hierarchical Trees (SPIHT) [SP96]. The input data vector \mathbf{p} is transformed to the wavelet coefficient vector \mathbf{c} , with the corresponding values for reconstruction after (lossy) compression $\hat{\mathbf{p}}$ and $\hat{\mathbf{c}}$.

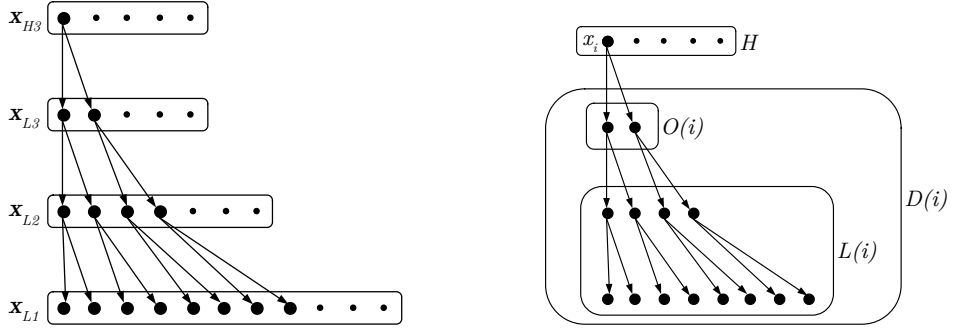
Coefficients are sent ordered by magnitude, as in EZW, but with the addition of bit counting. That means that only the number of sequential bits is recorded, not all of those bits themselves.

The zero tree concept is augmented to form a *spatial orientation tree* in the form of a pyramid. Significance is decided by whether a coefficient is above or below a certain threshold. This threshold is halved for every new iteration. Insignificant coefficients are not sorted into the three categories of root, (discarded) descendant of a root and isolated zero but into four groups, also called sets, illustrated in Figure 2.7b. Three sets are in reference to a root x_i [Poo+04]:

- $\mathcal{O}(i)$: set of two coefficients that are direct offspring of next higher sub band of x_i
- $\mathcal{D}(i)$: set of all coefficients that descend from x_i
- $\mathcal{L}(i)$: $\mathcal{D}(i) - \mathcal{O}(i)$

\mathcal{H} is the set of all spatial orientation tree roots, meaning the nodes at the highest level of the pyramid. The set partitioning process works as follows [SP96]:

1. The initial partition consists of the sets $\{x_i\}$ and $\mathcal{D}(i)$ for $i \in \mathcal{H}$.
2. If elements of $\mathcal{D}(i)$ are significant, they are partitioned into $\mathcal{L}(i)$ plus two single element sets $\{x_k\}$ with $k \in \mathcal{O}(i)$.
3. If elements of $\mathcal{L}(i)$ are significant, they are partitioned into $\mathcal{L}(i)$ plus two single element sets $\{x_k\}$ with $k \in \mathcal{O}(i)$.



(a) SPIHT three level 1D DWT spatial orientation tree. Adapted from [Poo+04, Fig. 1]
(b) SPIHT Set Definitions. Adapted from [Poo+04, Fig. 2]

Figure 2.7: Set Partitioning in Hierarchical Trees (SPIHT)

The full coding algorithm does not save the whole tree structure explicitly, but makes use of the encoder and decoder following the same decision criteria in packing and unpacking the wavelet coefficients.

2.3.7 Linear Predictive Coding (LPC)

Linear prediction uses past signal samples $X(n-i)$ to predict a current signal sample $\hat{X}(n)$.

$$\hat{X}(n) = \sum_{i=1}^p a_i * X(n-i) \quad (2.14)$$

p controls how many past signal samples are used for prediction. Linear Predictive Coding (LPC) approaches differ in how the predictor coefficients a_i are determined.

When LPC is used in lossless audio compression codecs like FLAC, the difference between the actual data sample and the prediction $X(n) - \hat{X}(n)$ is saved alongside the base samples and predictor coefficients [Mui+17].

A very simple approach by Hinterseer et al. uses a first order linear equation in the prediction of haptic signals [HSC06]. It determines the current sample $\hat{X}(n)$ based on the slope of two past samples $X(n-p)$ and $X(n-q)$. ($p > q > 1$).

$$a_n = 1 + \frac{X(n-q) - X(n-p)}{p-q} * \frac{n-q}{X(n)}, \quad a_i = 0 \text{ for } i \neq p, i \neq q \quad (2.15)$$

Steinbach et al. only save a new base sample when the difference between the actual and the predicted value is higher than the Just Noticeable Difference (JND) [Ste+11].

2.4 Audio Compression with MP3

MPEG-1/2 Layer-3, commonly known as MP3, is one of the first and most popular audio compression methods. Its basic elements are shown in Figure 2.8. First the audio signal is separated into smaller pieces, called frames. An analysis filterbank transforms the signal from the time to the frequency domain. For compatibility reasons the signal is first separated into 32 sub bands and then transformed via MDCT (see section 2.3.5). The perceptual model works with input from the temporal and frequency domain to determine limits of perception below which humans do not notice noise, also called masking thresholds (see section 2.3.4). The signal is then quantized as needed for quantization noise to stay below masking thresholds

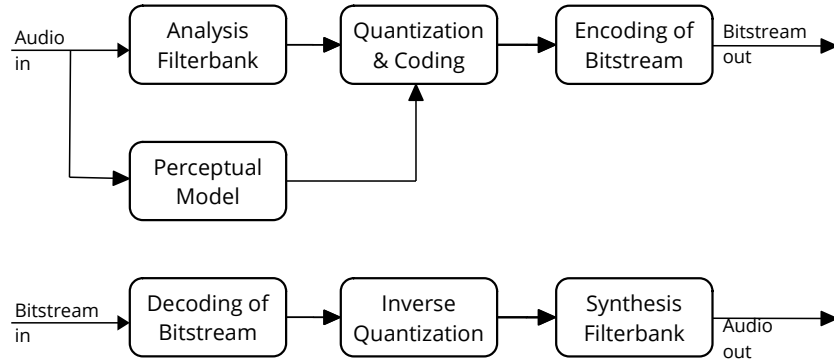


Figure 2.8: Structure of a perceptual encoding/decoding system, such as MP3. Adapted from [Bra99, Figure 1].

(see section 2.3.3). Those quantized values are further compressed via lossless Huffman Coding (see section 2.3.1). Finally meta information, like the sampling rate or compression level, is encoded with each frame and forms its header. The actual encoded audio data is referred to as the data block[Bra99].

2.5 Compression Codecs for Vibrotactile Signals

Lossless compression methods like Huffman coding often form the second stage of vibrotactile codecs since they do not depend directly on signal features and the human perception thereof, but simply remove redundancy in binary data.

Steinbach et al. cover haptic data compression in the context of telepresence teleaction (TPTA) applications [Ste+11]. TPTA scenarios include surgical procedures in which the doctor is not physically present or needs help from machines to do minimally invasive surgery, controlling robots in dangerous environments and teachers guiding students from afar. Systems enabling such interactions of a human operator with a remote teleoperator (TOP) primarily need force feedback as a haptic interface. Controlling a remote robot in an effective manner requires a stable connection with minimal transmission delays. Suitable compression techniques therefore need to have minimal or no algorithmic delay. This sets them apart from compression for audio and video transmissions, where delay requirements are typically less stringent. In a naive implementations data is sent over a packet-based network (like the internet) the moment it is sampled. The resulting high packet rates of up to 1 kHz pose practical challenges for network transmission.

Early approaches use various sampling and quantization techniques to reduce the data rate. For lossy compression Borst et al. [Bor05] choose quantization intervals in a manner that makes the resulting quantization noise inaudible to the human ear. None of these approaches ameliorate the problem of high packet rates.

Here, the concept of Perceptual Deadband (PD) helps by only sending a new data packet when the sampled data differs by a certain amount from the most recently sent value. On the receiving side the output force is held constant until the next signal arrives. The range in which no new packet is sent needs to be so small that humans cannot perceive any difference to the previously sent signal. The Just Noticeable Difference (JND) (see section 2.3.4), determines the size of the PDs. Figure 2.9 shows the compression algorithm in action.

Okamoto et al. use Discrete Cosine Transform (DCT) to transform haptic signals into the frequency domain [OY10].

The approaches discussed so far are not as complex and encompassing as the “A Rate-scalable Perceptual Wavelet-based Vibrotactile Codec” by Noll et al. which will be discussed in the next section.

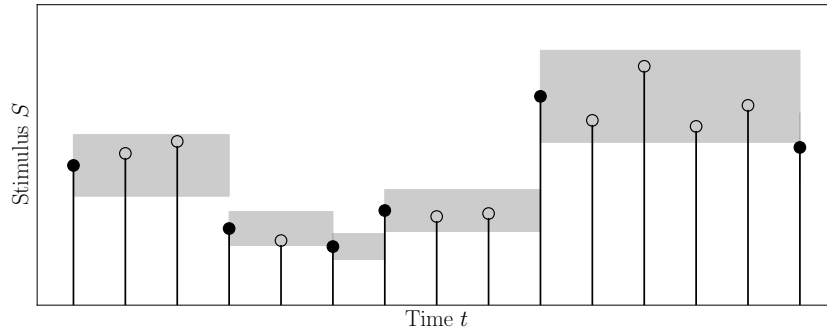


Figure 2.9: Perceptual Deadband data reduction. Adapted from [Ste+11, Fig. 3], with $\kappa = 0.22$ from [BHY05, p.839]

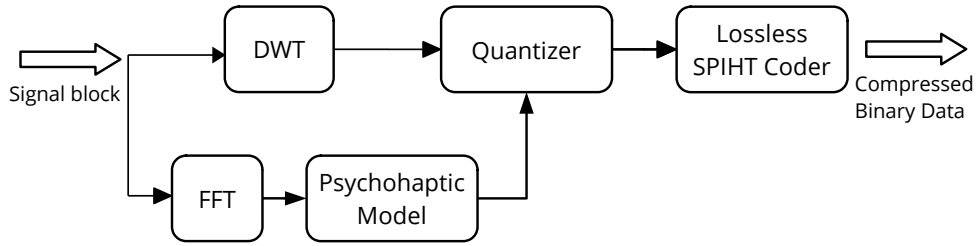


Figure 2.10: Block Encoder structure of the vibrotactile codec. Adapted from [Nol+20, Fig. 1].

2.6 A Rate-scalable Perceptual Wavelet-based Vibrotactile Codec

2.6.1 Overview

Noll et al. propose a vibrotactile codec to the IEEE P1918.1.1 standardization group [Nol+20]. Its general structure seen in figure 2.10 mirrors the MP3 audio codec described in Section 2.4.

First the codec divides a discrete input signal into data blocks. Those blocks are transformed separately with a DFT and a DWT using the CDF 9/7 wavelet (see section 2.3.5). A psychohaptic model uses the DFT coefficients to determine suitable masking thresholds for quantizing the DWT coefficients. The quantized DWT coefficients are further compressed with an SPIHT algorithm (see section 2.3.6) in lossless configuration. A header encodes the length chosen for the data block and details on the maximum value and quantization of the DWT coefficients. In the resulting bitstream a header is followed by the data block it describes.

Decoding works by extracting the blocks from the bitstream with help from the headers, applying the SPIHT algorithm in reverse and running the dequantized wavelet coefficients through an inverse DWT.

2.6.2 Psychohaptic Model

The psychohaptic model applies two types of mask: The *absolute threshold of perception* depends only on the frequency while the relative *masking threshold* is based on the frequency f_p and level a_p of peaks p in the signal. Those masks are first determined by the Model Providing Unit (MPU) and applied to the signal by the Model Application Unit (MAU) of the psychohaptic model. The masks will be used to distribute quantization noise so that is less perceptible by humans. In this way it is comparable to the noise shaping through dithering discussed in section 2.3.3.

The authors base the *absolute threshold of perception* on observations in which humans were

found to be most susceptible to haptic signals at a frequency of about 250 Hz, leading to a perception threshold of -77 dB. At a frequency close to 0 Hz they assume a level of -15 dB and for the upper end of the frequency range near 1000 Hz a threshold above 40 dB. Using those value pairs as regression points a function $t(f)$ approximating the true threshold is specified.

$$t(f) = \left| \frac{62 \text{ dB}}{\left(\log_{10}\left(\frac{6}{11}\right)\right)^2} \left[\log_{10}\left(\frac{f}{550 \text{ Hz}} + \frac{6}{11}\right) \right]^2 \right| - 77 \text{ dB} \quad (2.16)$$

For the relative *masking threshold* peaks need to be extracted from the magnitude spectrum. The criterion for a peak is that it must have a prominence of at least 15 dB and a minimum level of -42 dB. The recommended minimum distance of 10 Hz between peaks is optional. The masking thresholds $m_p(f)$ for each peak p are approximated with quadratic functions centered around each peak's frequency f_p .

$$m_p(f) = a_p - 5 \text{ dB} + 5 \text{ dB} \frac{2f_p}{f_s} - \frac{30 \text{ dB}}{f_p^2} (f - f_p)^2 \quad (2.17)$$

f_s is the sampling frequency.

The power additive combination of $t(f)$ and $m_p(f)$ is the global masking threshold $E_{M,B}$ and forms the output of the MPU.

In parallel to the work of the MPU, the MAU computes the signal energy $E_{S,B}$ contained in each wavelet frequency band B . The Signal-to-Mask-Ratio (SMR) in dB is

$$\text{SMR}_B = \frac{E_{S,B}}{E_{M,B}}. \quad (2.18)$$

The $E_{S,B}$ and SMR_B values for all bands B are input to the quantizer.

2.6.3 Quantization

During quantization bits are allocated to the bands according to their Mask-to-Noise-Ratio (MNR). The MNR is the ratio of Signal to Noise Ratio (SNR) to SMR.

$$\text{MNR}_B = \text{SNR}_B - \text{SMR}_B \quad (2.19)$$

This happens in a loop where a preset number of bits n is allocated to all bands of a block. If the MNR of a band is low, it indicates that a lot of noise is above the masking threshold. Therefore more bits should be allocated to it than to bands with higher MNRs. It is therefore served first, up to a maximum of 15 bits per band.

The quantizer used by the authors is called an embedded deadzone quantizer. It begins with finding the largest wavelet coefficient in the band w_{\max} and quantizing it to the next highest fixed point number \hat{w}_{\max} . Either 7 fraction bits and no integer bits if $\hat{w}_{\max} < 1$, or 3 integer and 4 fraction bits. With the bit budget b and \hat{w}_{\max} the quantization interval Δ_{quant} is calculated for each band.

$$\Delta_{\text{quant}} = \frac{\hat{w}_{\max}}{2^b} \quad (2.20)$$

This leads to the quantization algorithm for the wavelet coefficients w of a band with an added sign bit.

$$\hat{w}_{\max} = \text{sgn}(w) \left\lfloor \frac{w}{\Delta_{\text{quant}}} \right\rfloor \Delta_{\text{quant}} \quad (2.21)$$

After all wavelet coefficients of all bands have been quantized to w_{quant} they are scaled to integers depending on the highest bit budget b_{\max} allocated to a band.

$$w_{\text{quant,int}} = w_{\text{quant}} \frac{2^{b_{\max}}}{\hat{w}_{\max}} \quad (2.22)$$

2.6.4 SPIHT Coder

Those quantized integer wavelet coefficients are processed by an SPIHT algorithm configured for lossless compression: It loops over all existing bit planes, (corresponds to quantization thresholds) until all quantized wavelet coefficients are encoded.

2.7 Hardware Accelerators

There are multiple avenues to improving the performance of an algorithm once optimization of the software running on a Central Processing Unit (CPU) is no longer sufficient.

One can adapt the code to let it run on a Graphics Processing Unit (GPU), which has many small compute cores working in parallel. This brings advantages for problems that are inherently parallel, meaning that they can be split up into many smaller tasks which do not depend on each other for results. Using existing customer or enterprise grade GPUs has the advantage that no new hardware needs to be developed and manufactured so that developing and deploying a new algorithm is relatively cheap and easy. While each unit of a GPU has a lot of parallel compute power, it is also quite expensive and energy hungry.

In Field Programmable Gate Arrays (FPGAs) the logic structure itself can be programmed and changed with Hardware Description Languages (HDLs) such as Verilog or VHDL after it has been manufactured. This added flexibility enables custom architectures that may be more efficient than programming GPUs. FPGA manufacturers such as Xilinx offer tools and Intellectual Property (IP) cores for components such as CPUs and interconnects to facilitate development and shorten time-to-market. The disadvantage of such a reliance on proprietary intellectual property is less transparency into how the device actually works and possible licensing issues when publishing or selling designs. But it is also possible to use plain HDLs to design core functionality. In this way the FPGA can be used as a prototyping platform for later implementation in Application Specific Integrated Circuits (ASICs).

ASICs are custom silicon chips. The initial cost of developing and manufacturing them is very high compared to the approaches mentioned above, but it can pay off in superior efficiency and performance as well as low unit costs if many devices are produced.

There has been a multitude of designs meant to accelerate the decoding of MP3 audio files (see section 2.4) and lower the associated energy consumption. Bang et al. propose a custom Digital Signal Processor (DSP) ASIC with special instructions to decompress Huffman encoded files [Ban+02]. Lin et al. focus on accelerating the inverse MDCT and poly-phase synthesis, also in an ASIC design [LH07]. None of the authors report having actually manufactured the chips.

Song et al. use GPUs to accelerate decompressing satellite images with 1024×1024 pixels. Those images have been compressed via DWT and SPIHT. (In addition to Reed-Solomon error correction coding.) Profiling the software on a CPU they find that performing an inverse DWT takes 14 times longer than SPIHT decoding. Another reason for continuing to run SPIHT on a CPU is that the process of ordering coefficients by importance is necessarily serial and therefore cannot exploit the parallel computation resources of a GPU. In contrast, the DWT lifting scheme (FLWT) can be easily calculated in parallel, as only neighboring samples depend on each other. They further simplify the lifting scheme by substituting the floating point factors λ (see section 2.3.5) with integers divided by a power of two. Division or multiplication by a power of two can be implemented as bit shifting. Therefore all calculations can be executed with fixed-point numbers and the hardware only needs units for integer addition/subtraction and multiplication, but no floating point unit[SLH11].

In his Master's Thesis Zarifar develops a DWT accelerator for a Xilinx Virtex II FPGA platform. The DWT is implemented as a Fast Lifting Wavelet Transform (FLWT) scheme. Possible performance improvements for CDF 9/7 wavelets were investigated, but not actually implemented. The DWT functionality was added in the form of new instructions to a MIPS

ISA. The design files were distributed with the original thesis on a CD-ROM and are therefore no longer publicly available[Zaf+02].

Many authors use proprietary Xilinx tools to design their System on Chips (SoCs) for DWT acceleration, including [Bor+06; RS11; Sai+12].

Kuzmanov et al. also implement the lifting scheme on a Xilinx Virtex II FPGA [Kuz+02]. They provide a detailed overview of their architecture which takes advantage of the structure of FLWT to reuse data (the step-specific factors λ) and parallelize and pipeline data processing (calculations of all even even- or odd-indexed coefficients do not depend on each other). In addition some Xilinx-specific FPGA features are leveraged.

Barua et al. show that symmetric extension at the beginning and end of FLWT input data is possible without any additional mathematical operations [Bar+04].

2.8 Consequences for this Work

The proposal for a vibrotactile codec by Noll et al. [Nol+20] is by far the most advanced and comprehensive compression codec published so far. It builds on research about the human perception of haptic and tactile stimuli for a psychohaptic model to implement lossy compression that is combined with lossless SPIHT encoding. Relevant features are extracted via DWT and additional information for the psychohaptic model and quantizer is provided by a DFT of the input signal.

Since a FFT accelerator has already been developed at the HPSN the DWT and SPIHT codecs are natural candidates for hardware acceleration in this thesis. Song et al. give hints that SPIHT is not very amenable for acceleration in hardware, at least for their case of decompressing 2D images with GPUs [SLH11]. In their experiments it took 12 times more CPU time to calculate the inverse DWT of a 1024×1024 pixel image than to reverse SPIHT compression of it. They also remark that the execution of the SPIHT algorithm is mostly serial since coefficients are ordered by magnitude (importance) in one long succession to form the final embedded bitstream.

At the same time DWT, especially in its lifting implementation, can be easily run in parallel, and takes an order of magnitude more processing resources for compressing 2D data than SPIHT. These findings could change when applied to relatively small (32 bits to 512 bits) 1D samples, as used in the codec by Noll et al. The lifting implementation of DWT, as opposed to its more direct execution, holds great promise. Franco et al. [Fra+10] saw a $14\times$ faster execution of the lifting scheme on GPUs than the direct implementation. This work will explore whether the lifting scheme is compatible to the reference implementation of Noll et al. and whether it is as advantageous for smaller 1D data as it is for larger 2D data.

The psychohaptic model of the codec is not as suited to hardware acceleration because it needs many conditional branches and is subject to further change if and when the codec gets improved in the future.

The quantizer processes selects the quantization resolution of DWT bands based on inputs from the psychohaptic model. Its core is the deadzone quantizer, a well defined mathematical algorithm. This is another potential avenue for hardware acceleration.

First the codec will be implemented in the C programming language on a conventional x86-64 architecture with C standard libraries to then be ported to the RISC-V ISA with C libraries for embedded computing. Code execution will be profiled to find those function calls that take up the most processing time. The parts of the codec that need the most CPU time and where the overhead of communicating with the accelerator is acceptable will be accelerated in hardware. The accelerator will be implemented as an additional module to be connected via an on-chip data bus with the RISC-V RISCY that is already set up at the institute. Benefits of using the FFT module to accelerate execution of the vibrotactile codec on the RISC-V CPU will be analyzed.

3 Software Design

In the previous chapter the vibrotactile codec by Noll et al.[Nol+20] (see section 2.6) was chosen as the tactile codec that will now be implemented as software for an embedded computer. This chapter will describe the steps of developing and testing the program and then finding performance bottlenecks that can be hardware accelerated.

3.1 Development

3.1.1 Overview

The software implementing the vibrotactile codec was developed in this work to run on the RI5CY [Gau+17; 21] processor. It is based based on the Matlab software by Andreas Noll[Nol21] which he developed for the vibrotactile codec presented in [Nol+20]. Some proprietary Matlab functions that were used in the Matlab reference code such as `fft` and `findpeaks` had to be either replaced with an open-source equivalent or implemented from scratch. The Matlab code was applied to reference data representing 280 vibrotactile signals with each 2800 samples, also supplied by Andreas Noll, to create data to test against in unit testing of the embedded software implemented in this work.

This 32bit RISC-V CPU supports the basic integer instructions (I), multiplication and division extension (M) and compressed instructions (C), leading to the moniker RV32IMC. The optional floating point extension (F) is not enabled in the processor which this project targets. It therefore had to fall back to software emulation of floating point instructions. This memory-restricted, bare metal computing environment without an operating system precluded usage of the C++ standard library and numerical template libraries like Eigen[GJ+10]. Therefore all data types not included in standard C, like vectors, and accompanying methods to manipulate them, e.g. multiplying their elements by a scalar value, were custom written for the vibrotactile codec. They had the general structure of a pointer `data` to an array of the values specified by the data type, and an additional variable `length` of type `size_t` that specified the number of elements in the array.

The cross platform build system CMake[Kit] helped create two separate build targets, one for double precision (64 bit) floating point values, and one for single precision (32 bit) values. In a C header shared by all relevant source files, a string called `VIB_SCALAR` was either defined as `double` or `float`. With this simple form of polymorphism, C Code could be run in 64 bit mode on the x86 development machine to directly check correctness of the results compared to the reference data, and also in the exact same 32 bit configuration that is compiled for the RI5CY embedded CPU.

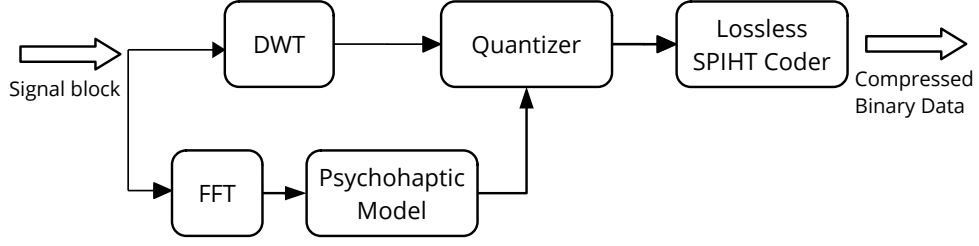


Figure 3.1: Block Encoder structure of the vibrotactile codec. Adapted from [Nol+20, Fig. 1].

RISC-V binaries were compiled on a server at the HPSN. They were then executed by a Register Transfer Logic (RTL) simulation of the RI5CY core.

3.1.2 Software Structure

Section 2.6 has explained the vibrotactile codec as developed by Noll et. al. Following is the structure of the software implementing this codec, which follows figure 2.10. Also repeated is figure 3.1.

The C software in this project works on the block level. This means that its input is a vector of floating point values at a certain fixed length; the block lengths allowed by the codec are 32, 64, 128, 256 and 512.

Thus `BlockEncoder()` is the main function that takes this input, processes the data with help of the function blocks and returns as output a C `struct` with the block header and a pointer to the actual block of compressed binary data. Apart from managing the data flow between the different parts of the vibrotactile codec, the function encodes the header data with values such as the block length, the maximum quantized wavelet coefficient and the length of the output binary data. Additional input to `BlockEncoder()`, apart from the data blocks themselves is the desired bit budget.

The first function called by `BlockEncoder` is `DwtCdf97()` which transforms the input signal into CDF9/7 wavelet coefficients with a lifting implementation of DWT. DWT is applied multiple times to slices of the input block, which results in several frequency bands.

`PsychohapticModel()` also takes the input signal, but puts it through an FFT and analyzes the resulting spectrum. This includes finding peaks of a certain prominence and calculating the masking effect they have on neighboring samples. To ensure full equivalence with the results of the reference Matlab code the exact definition of peak prominence used by the proprietary Matlab `findpeaks` function needed to be adhered to. This definition is explained in [Mat20]. Another input for the masking behavior is the so called perceptual threshold which depends on the frequency of the analyzed samples.

`Quantize()` uses the input from the `PsychohapticModel()`, namely the signal energy of the DWT bands and the Signal-to-Mask-Ratio (SMR) to find an optimal allocation of the bit budget between the DWT bands of coefficients from `DwtCdf97()`. The bit budget determines the quantization resolution. If the bit budget of a certain band is three, then all of its quantized coefficients can be at most one of $2^3 = 8$ different values. By calculating the difference between the original and quantized coefficients a Signal to Noise Ratio (SNR) is computed. The difference between SMR and SNR is the Mask-to-Noise-Ratio (MNR). In an iterative loop the bit budget is distributed between the bands by allocating one more bit to the budget of the band with the lowest MNR. It is then deadzone quantized again with the new, higher resolution to increase its MNR. The goal is to have an evenly distributed MNR so that most of the quantization noise is masked.

`Spiht1dEncode()` compresses the quantized coefficients to generate the final binary data

block.

For a fully functional software an `Encoder()` function, which processes signals of arbitrary length and outputs a binary data stream of concatenated blocks, would still need to be implemented. This is left for future work on this topic.

3.1.3 External Libraries

The KISS FFT [Bor21] library by Mark Borgerding was used for the FFT functionality. One of its main advantages is the lean code base with no external dependencies. Only a couple C source files needed to be copied into the project and some defines set to use it inside the `PsychohapticModel()` function. Another advantage is its liberal “BSD-3-Clause” license, which is compatible with “copy-left” licenses like the GNU Public License (GPL)[20] as well as commercial, closed source code.

As the bare metal RI5CY execution environment lacks some functions of a standard `libc`, a resource-efficient routine for memory allocation with the `malloc()` and `free()` was required. The alternative would have been purely static memory allocation, but using worst-case estimates for processing a block with 512 samples would have resulted in a very bloated memory footprint when encoding blocks with shorter lengths, like 32.

The `nano-malloc()` function in the RISC-V newlib library, also licensed under “BSD-3-Clause”, works very well in the low-memory RI5CY environment. The one thing that was adapted is the `sbrk()` function for increasing program data space: A `static char` array called `memory` is allocated at compile time. All of the memory needed during execution of the codec needs to fit within `memory`. The heap is defined as the amount of memory that is allocated to the program at run time. `sbrk()` is called with the desired size of program memory increase. If the increase fits inside `memory`, `sbrk()` return a pointer to the end of the previously allocated heap. (For the first call of the function a pointer to the start of the `memory` array is returned.) Then the requested increase is added to the `heap_end` variable. If the memory is exhausted it prints a warning message and returns “-1” cast to a pointer.

3.1.4 Optimizations

The dynamic allocation of the `data` section of custom vectors with `malloc` led to excessive usage of the `malloc()` and `free()` function calls in some loops where the lengths of interim result vectors changed in almost every iteration. Here the maximum space these vectors could take up was allocated at compile time and only the `length` parameter was changed during execution.

Some calculations that were part of the original reference Matlab code were not actually necessary for correct execution of the codec.

The unit of the original Signal-to-Mask-Ratio (SMR) (see equation 2.18) is dB and therefore needs a logarithm in its calculation. But since SMR is used only to select the wavelet band with the lowest value (fidelity) to quantize it with a higher resolution, not applying the logarithm does not change the result of that relative comparison. This works because the logarithm is a monotonously increasing function that preserves the relative order of the values it is applied to.

The quantizer also does not need to quantize all wavelet bands anew after a bit allocation, but only the one whose resolution was changed in the previous step.

3.1.5 Unit testing for desktop and embedded targets

Unit testing helps ensure correctness of a software project by running automated tests after each change in the code.

Google Test[Goo21b] is a widely used C/C++ test suite that has good integration in the CMake/CTest framework. It could only be run on the author’s x86 development machine, but the binaries it checked were compiled out of the same source code also used to generate the RISC-V binaries.

A reference implementation of the vibrotactile codec[Nol+20] (see section 2.6), written in Matlab script, was thankfully provided to the author by Andreas Noll and his colleagues. Using this basis, the different functions constituting the vibrotactile codec were executed separately to generate test data. This data was saved in the proprietary Matlab `.mat` format, and then imported into the Google Test[Goo21b] C++ test suite using Matlab’s C/C++ Application Programming Interface (API).

Since floating point values are almost never exactly equal some error thresholds are needed. Table 3.1 shows the absolute error thresholds the code was tested against and passed. Those differ for the psychohaptic model and the remaining functions, since the psychohaptic model operates on the spectrum produced by a FFT of the original signal and the rest is concerned with DWT coefficients. They have different value ranges and impact on the coder result. The psychohaptic model influences the bit budget allocation and therefore the resolution in which DWT bands are quantized. Inaccuracies only affect the outcome when the relative positioning of SMR values changes. DWT coefficients form the output, albeit only at the quantization resolution.

The error thresholds were chosen empirically and represent the accuracy that could be achieved by the software. The fact that there were only very few errors in the quantized samples, which form the output of the encoder, compared to the Matlab reference code indicates that the chosen error threshold guarantee good functional accuracy. Out of 1400 blocks processed in 64bit precision, only 13 blocks had deviations in one of their quantized and scaled coefficients, and each of those integers differed only by one from the correct result. In 32bit precision

Data Domain	32 bit precision	64 bit precision
Psychohaptic Model	3.5×10^{-3}	1×10^{-11}
DWT Coefficients	7.6×10^{-5}	5×10^{-9}

Table 3.1: Absolute error thresholds for floating point numbers used in unit testing.

The data in the Matlab reference is calculated in 64bit accuracy, but the target RI5CY processor supports only 32 bit numbers. Tests were therefore run both in the 64bit desktop and the 32bit embedded configuration. To account for the unequal precision different error thresholds are used.

3.1.6 Implementation Gaps

Some functionality is still not fully implemented in the C code.

It only encodes single blocks, but does not do the work of padding and cutting longer input signals into blocks.

The `BlockEncoder()` sets values in a `struct` for the codec header, but this does not represent the exact bit-for-bit structure envisioned by the vibrotactile codec specification. The variable length of the bits encoding the block length is especially challenging here.

Similarly the output blocks are not merged into a single continuous data stream.

Although all function blocks were checked with unit tests to give the same output in 64bit precision as the reference Matlab code, errors do still occur in the SPIHT encoder.

The C code also lacks a decoder. While DWT can already be reversed by passing a negative level to `DwtCdf97()`, this is not the case for SPIHT. Also missing is logic for decoding the codec headers.

3.2 Software Results

3.2.1 Memory Requirements

The memory requirements in table 3.2 represent the the minimum memory allocated by the custom `Malloc()` function required to run the codec at all block sizes (up to 512). They were determined by lowering the size of the `char` array that is used as a memory pool by `Malloc()` until it runs out of memory to allocate during execution of the codec at the specified block length.

For the longest block length tested, 512, 99 KiB are needed. For smaller block sizes the memory requirements are considerably lower, down to about 3 KiB.

Reducing the bit budget has little effect: 512 samples long blocks need 2 % less memory at a bit budget of 8 instead of 64 (97 KiB instead of 99 KiB), and with a block length of 32 there are no detectable differences in memory usage at all.

Block Length	Max. Memory Usage [KiB]
32	2.4
64	4.8
128	11
256	36
512	99

Table 3.2: Memory requirements at bit budget 64 and 32bit precision.

This is not all the memory required to run the vibrotactile codec. Instruction memory and memory for pre-assigned variables is also needed. The hardware described in chapter 4 is synthesized with a total of 128 KiB RAM divided into 96 KiB data memory and 32 KiB instruction memory. This is enough to test the `Quantize()` function and the deadzone quantizer hardware accelerator in isolation, but not the whole codec. The complete `BlockEncoder()` was tested with 1 MiB, of which 768 KiB was data memory 256 KiB instruction memory.

3.2.2 Computational Performance

Results for speedup of the C code are based on processing $4960 * 1000$ samples, as the first blocks of five signals with five different block lengths were processed in a loop 1000 times. In Matlab all the blocks of five different padded signals with original length 2800 were processed with five different block lengths, resulting in $(4 * 2816 + 6 * 512) * 5 = 71\,680$ samples used for measuring performance. The values for the C code were recorded as “real” (wall clock) time spent between beginning and end of program run via the UNIX `time` utility, while the `tic/toc` function in Matlab measured the average time spent processing one block. Software was run either as a Matlab script in Windows 10 or as a Linux binary compiled with `GCC 9.3` and flag `-O2` in Windows Subsystem for Linux (WSL) 2 on the same computer with an Intel Core i5-3337U CPU and 10 GB RAM. The C binary executed three times for each of the four bit budgets to average the run time recorded by `time`.

There are two reasons for the considerable performance improvements of $149.7\times$ to $216.1\times$ of the C code compared to the original Matlab Script. One is algorithmic improvements like

not computing the logarithm to calculate and compare SMR and MNR values. Those led to a halving of cycles needed on the simulated RI5CY processor. The other is speed advantages inherent to the programming languages. C is very low-level and close to the hardware. Among others, it lacks the abstraction of memory management; all memory needs to be explicitly allocated and freed. The Matlab programming language on the other hand provide many conveniences to the programmer which come at the cost of efficiency. High level functions like finding peaks in a spectrum are included while they need to be custom written in C. Types like `int`, `float` or arrays need not be stated explicitly as they are inferred by Matlab. Matlab scripts are either interpreted directly at run time or just-in-time compiled. Thus a major speedup was achieved at the cost of higher development effort and (currently) some feature gaps, described in section 3.1.6.

Bit Budget	Sample Rate C [$10^3/s$]	Sample Rate Matlab [$10^3/s$]	Speedup C/Matlab
8	958.1	6.40	149.7
16	763.9	6.05	126.3
32	556.7	3.17	175.6
64	460.3	2.13	216.1

Table 3.3: Computational performance of vibrotactile codec software implemented in either C or Matlab script language.

The sample rate refers to the number of samples that were processed by the software per second in 64bit precision by a release (non-debug,optimized) build of the codec.

The 32bit mode used for the RI5CY embedded CPU introduces some discrepancies to the Matlab reference, but those are mostly minor. There is at least one value error in six out of 1400 blocks processed by `Quantize()`, but those are merely off-by-one errors in four-digit integers. This is the only type of error that actually matters, since these integers form the output of the vibrotactile coder, after having been SPIHT compressed. All the floating point inaccuracies only matter insofar as they impact the quantized coefficients. The 32bit accuracy therefore suffices for most use cases of the vibrotactile codec.

3.3 Search for Acceleration Potential

To look for parts of the algorithm that would benefit from hardware acceleration, the software had to be profiled in more detail.

3.3.1 Profiling Tools and Techniques

Profiling helps to find bottlenecks that slow down execution of the codec as a whole. Two separate approaches were used, one more detailed method on the x86 platform and another approach on the targeted RI5CY RISC-V CPU itself.

Profiling on an x86 Platform

Valgrind and *gperftools* were used to measure the relative performance of the functions on an x86 development machine with an Intel Core i5-3320M CPU and 16 GiB RAM.

Valgrind[NS07] is a popular framework for dynamic analysis of software. It was very helpful for finding memory leaks, and its *Callgrind* tool helped estimate the performance of code by measuring how many instructions were executed by each function. While *Callgrind* does not measure effects like memory access and caching, those phenomena are not very relevant for the embedded application targeted by this diploma thesis. One test with *Cachegrind* gave cache miss rates of at most 0.1 %.

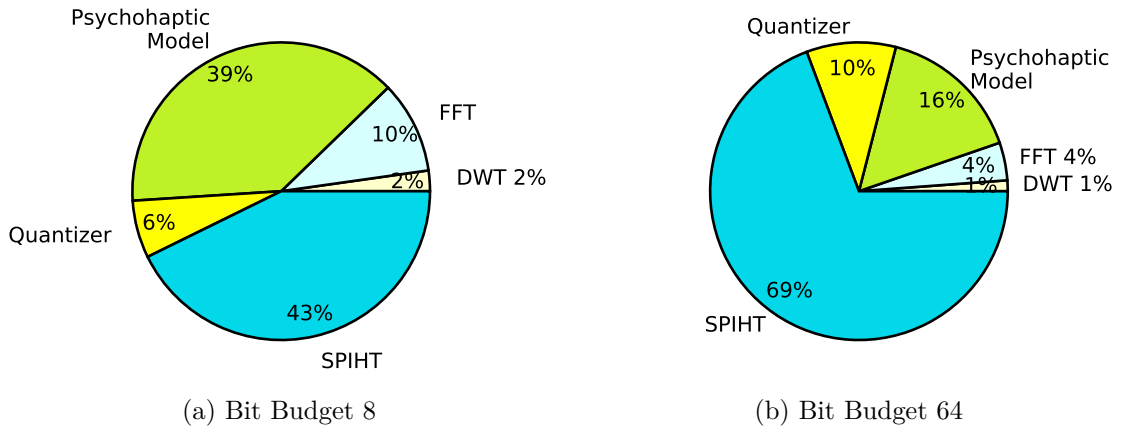


Figure 3.2: Distribution of CPU time spent in function blocks of the vibrotactile codec on x86 as measured by the *gperftools*.

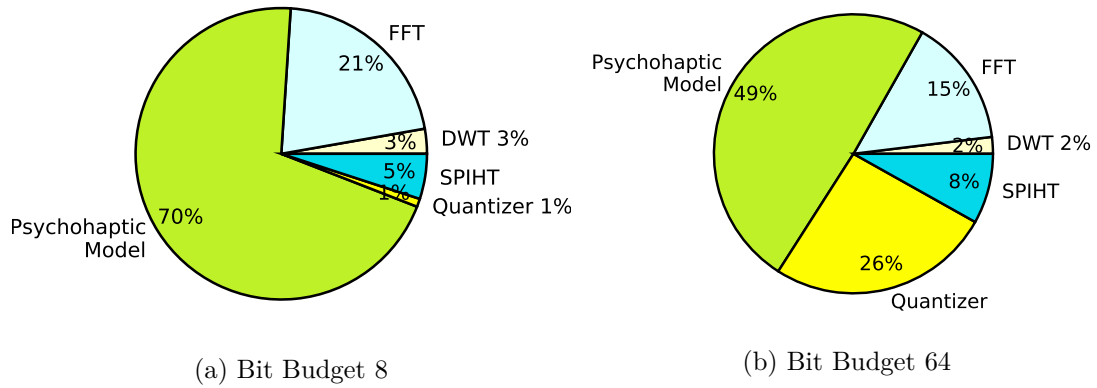


Figure 3.3: Distribution of CPU cycles spent in function blocks of the vibrotactile codec on RI5CY.

Another tool for measuring performance on x86 systems is the CPU profiler of *gperftools* [Goo21a] (also known as Google Performance Tools). It obtains a runtime-based estimation of the functions' relative resource use by sampling how often a function is active at certain time intervals during execution of the program. Figure 3.2 shows the distribution of CPU time spent in the function blocks of the the vibrotactile codec as measured by the *gperftools*. A more detailed breakdown of the most resource-intensive functions on 32bit x86 is in the appendix A.1. They are very similar to those determined by *Valgrind*, suggesting that memory accesses and caches do not significantly impact performance of the x86 binary.

Performance Measurement on the RI5CY Embedded Platform

For more direct performance measurement of the RISC-V binary running on the emulated RI5CY CPU a Systemverilog test bench used the time between memory bus accesses to a certain address to determine the time spent in a certain part of the algorithm. Those memory accesses were triggered in the C code at the beginning and end of a (partial) function of interest. Figure 3.3 shows how the distribution of computational complexity differs for a low bit budget (signal quality) of 8 and a high bit budget of 64.

Comparison of Performance on x86 and RISC-V Embedded

The x86 execution environment has many important differences to the RI5CY processor, among them hardware acceleration of floating point operations, instruction-level parallelism (superscalar architecture) and the use of caches to ameliorate the big latency differences between modern CPU speeds and memory accesses. Especially the lack of a Floating Point Unit (FPU) leads to a big discrepancy in the relative performance impact of the individual components. While only 60 % of the time is spent outside SPIHT on x86, this increases to 90 % on the RISC-V core without an FPU.

3.3.2 Candidates for Optimization

Good candidates for hardware acceleration do not only consume a significant part of the total resources needed by a program, but also fulfill some additional criteria:

- Self-contained and simple basic operation
- Not likely to change in the future
- No further software optimizations possible
- Ideally: suitable for parallel execution

Another significant consideration is the lack of a FPU in the RI5CY core targeted by this project. This means that many functions whose floating point operations are emulated in software and hence slow would profit from doing these operations in hardware. The hardware could be either a custom hardware accelerator for the specific function, or a general purpose Floating Point Unit (FPU). Custom hardware accelerators should consequently use substantially less chip resources than a general purpose FPU, because otherwise a FPU that benefits all floating operations would have a substantially bigger impact on overall codec performance.

Discrete Wavelet Transform

The software implementation of the (more efficient) lifting scheme for Discrete Wavelet Transform (DWT) with CDF 9/7 wavelets took up only two to three percent of the total time needed to execute the codec. Other parts of the codec, such as the psychohaptic model and quantizer are more resource intensive. Compared to that, 1D DWT is not a fruitful target for hardware acceleration. Although many hardware accelerators for DWT exist (see sections 2.3.5 and 2.7) they are all built for handling 2D image data, where processing is more resource intensive.

Psychohaptic Model

The psychohaptic model not only needs a FFT, but also multiple passes over the coefficients of the resulting spectrum to find peaks of a certain prominence and calculate masking thresholds of the signal. It is especially prone to change when the vibrotactile codec evolves, since the parameters for finding specific peaks and weighting their influence in masking aspects of the vibrotactile signal might be optimized in the future. This is in contrast to most of the other operations in this section, which mainly consist of well established algorithms like DWT, the deadzone quantizer and SPIHT. Hardware acceleration is less beneficial when it could become obsolete in future iterations of the vibrotactile codec.

Simple Arithmetic Operations

Simple arithmetic operations applied to a vector, for example multiplication or division of vector with/by a scalar, also took up a considerable amount of CPU time. Hardware acceleration could theoretically apply such functions to each member of a vector in parallel, albeit at a huge cost in power and area of the chip. It would also necessitate direct and parallel memory access for the accelerator. Since multipliers and especially dividers are by far the biggest components of a floating point unit, it would make more sense to enable the optional floating point unit of the RI5CY core instead of adding a custom accelerator with at least the same hardware cost and not much additional benefit.

The logarithm function was among those functions responsible for the most instruction executions, even when profiling the software with *Callgrind* running on x86 CPU with a FPU. Figures 3.4 and 3.5 show that even with artificially reduced 8bit accuracy of the logarithm there is no significant deviation in the Spectral Temporal SIMilarity (ST-SIM) quality metric. (The ST-SIM “Measure for Vibrotactile Quality Assessment” is developed and explained in [HS20].) They were plotted with help of the Matlab Reference code by inserting an artificially inaccurate logarithm function. This opens the door to rough logarithm approximations like using the exponent bits of a floating point number as its logarithm. Since such optimizations can also be applied in software and any errors introduced by hardware accelerators should be avoided where possible, this approach was not pursued further.

Quantizer

The quantizer algorithm tests many different quantization resolutions on its way to the optimal allocation of the bit budget between wavelet bands, based on the SMR calculated by the psychohaptic model.

The deadzone quantizer, as described in section 2.6.3 with equations 2.20 to 2.22, is the innermost loop of the quantizer algorithm. It is applied to each coefficient of a wavelet band when distributing the bit budget. It is therefore an often called function in the algorithm. Between 1.3 % and 3.3 % of total execution time is spent in this function. The lack of a floating point unit is one of the factors behind this relatively contribution. There is also considerable potential for optimization and simplification: Since Δ_{quant} and \hat{w}_{max} stay constant during processing of a wavelet band (consisting of 4 to 32 coefficients), they need to be transmitted only at the start of a new band. No division is needed if the inverse of Δ_{quant} ,

$$\Delta_{\text{quant},\text{inverse}} = \frac{1}{\Delta_{\text{quant}}} \quad (3.1)$$

is calculated in software, once for each band. The ranges of Δ_{quant} and \hat{w}_{max} are either limited by features of the vibrotactile codec itself, or by the value range of common vibrotactile signals, which will be explained in more detail in section 4.1. This opens the possibility of simpler hardware, namely a floating point multiplier that does not need to implement some edge cases of the IEEE 754 [IEE19] floating point standard, such as subnormal numbers and infinity. A reduction in precision of the constituent parts without negative impact on the accuracy of the deadzone quantizer’s results might be possible as well.

SPIHT

SPIHT (see chapter 2.3.6) takes up about half of the processing time in the current C implementation of the vibrotactile codec on an x86 processor with a FPU. The difficulty lies in the different lists coefficients are sorted in. A normal C array is merely a succession of values in memory with a pointer to the address of the first value. In the SPIHT algorithm some values need to be cut from the middle of their lists. With the C array described above, all numbers

following the cut out number need to be copied to the address of their respective predecessor. This inefficient procedure can only be avoided by introducing a new data structure, the linked list. Here each value is associated with pointers to its predecessor and successor. To cut out or insert a value, only the pointers of the preceding and following number need to be adjusted. Linked lists are not yet implemented in the software developed for this project and would also be hard to realize in a hardware accelerator. Either the accelerator needs a big portion of internal memory or independent access to shared memory to save the different lists of coefficients. The size of the lists is variable, therefore either dynamic memory allocation or static pre-allocation of a worst case size each list can have with a 512 samples long block is needed.

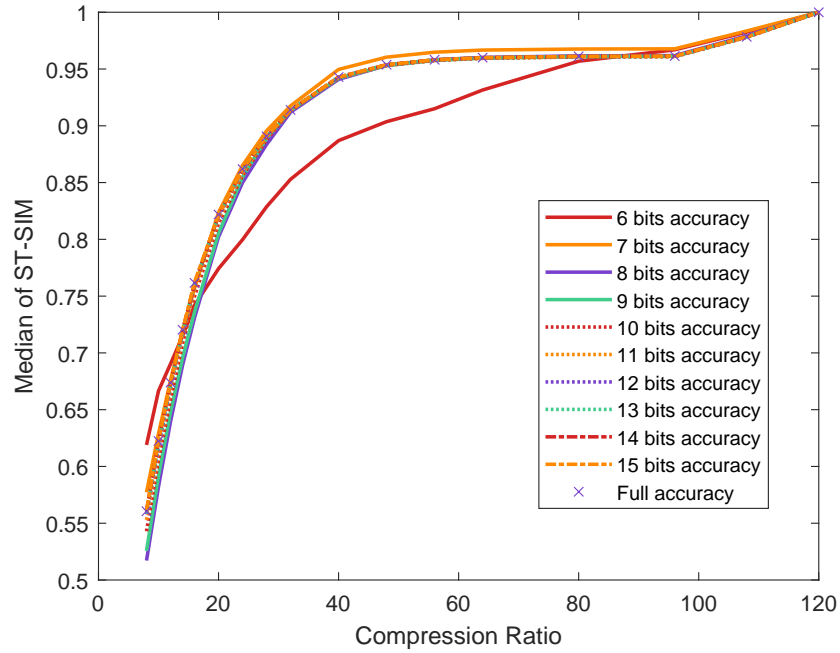


Figure 3.4: Impact of reduced logarithm accuracy on the ST-SIM quality metric.

Choice of Acceleration Target

As described in the section about the quantizer, the deadzone quantizer algorithm is one of the most often executed functions in the codec and has other qualities that facilitate hardware acceleration. It is therefore chosen as the target for the hardware accelerator, whose design and performance results will be detailed in the next chapter.

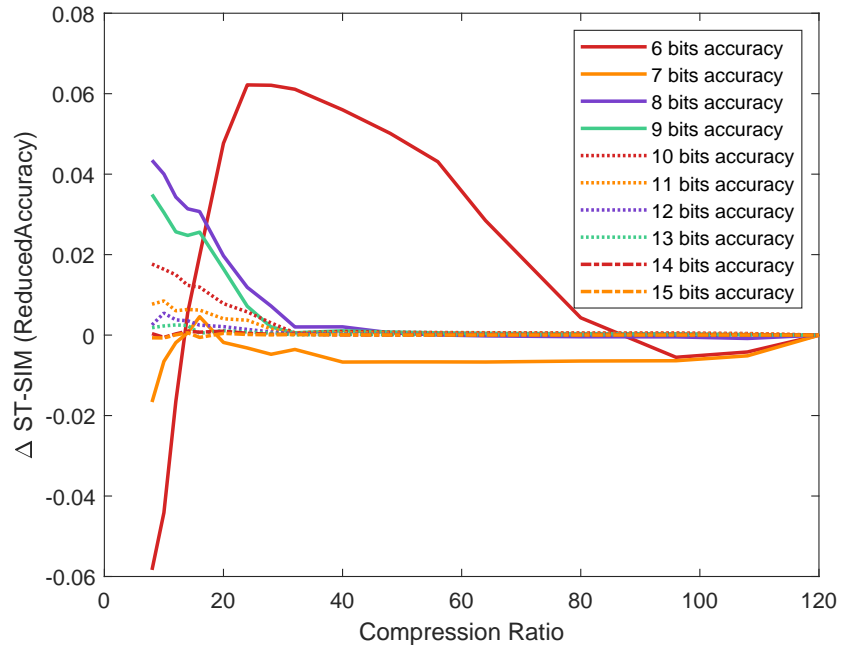


Figure 3.5: Impact of reduced logarithm accuracy on the ST-SIM quality metric. Difference: $\text{STSIM}_{\text{FullAccuracy}} - \text{STSIM}_{\text{ReducedAccuracy}}$

4 Hardware Design

In the previous chapter the decision was made to accelerate the deadzone quantizer algorithm contained in the quantizer function block of the vibrotactile codec. This quantization algorithm was described in section 2.6.3. The hardware accelerator processes 32bit floating point numbers, also referred to as single precision or `float` values in C, because processing the vibrotactile codec works best with floating point numbers and the RI5CY is a 32bit CPU. Systemverilog was chosen as hardware description language because the RI5CY core itself is also written in this modern HDL.

Equations 2.21 to 2.22, which describe the deadzone quantizer algorithm and its input, are repeated here for convenience.

The quantizer begins with finding the largest wavelet coefficient in the current DWT band and quantizing it to the next highest fixed point number \hat{w}_{\max} (`max` in the data flow graph). From the bit budget b that is currently allocated to the band and \hat{w}_{\max} the quantization interval Δ_{quant} (`delta`) is calculated for each band.

$$\Delta_{\text{quant}} = \frac{\hat{w}_{\max}}{2^b} \quad (4.1)$$

These values are then fed to the deadzone quantization algorithm for the wavelet coefficients w (`x`) of a band with an added sign bit.

$$\hat{w}_{\max} = \text{sgn}(w) \left\lfloor \frac{w}{\Delta_{\text{quant}}} \right\rfloor \Delta_{\text{quant}} \quad (4.2)$$

After all wavelet coefficients of all bands have been quantized to w_{quant} they are scaled to integers depending on the highest bit budget b_{\max} allocated to a band.

$$w_{\text{quant,int}} = w_{\text{quant}} \frac{2^{b_{\max}}}{\hat{w}_{\max}} \quad (4.3)$$

4.1 Data Flow

The general structure of the data flow, shown in figure 4.1, has four inputs, but only one of them, the value x that is to be quantized, changes for each quantization operation. The remaining three parameters, the quantization interval Δ_{quant} (`delta [31:0]`), its inverse $\frac{1}{\Delta_{\text{quant}}}$ (`1/delta [31:0]`) and the maximum quantized value \hat{w}_{\max} (`max`) stay the same for each DWT. These input registers are written to via an Advanced Microcontroller Bus Architecture (AMBA) Advanced High-performance Bus (AHB) bus from which the result register

`x_quant` can also be accessed. Precalculating the inverse of Δ_{quant} in software eliminates the need for a resource-intensive floating point divider.

IDLE waits for writes to the input registers and checks whether there has been at least one valid write to the three parameter registers. In that case, `x` is transformed into its absolute value $|x|$ by leaving out the sign bit at position 31 (see 4.3) for further processing and appending it to the final result. The first state after IDLE, `CHECK_RANGE`, decides whether `x` is outside of the value range prescribed by the vibrotactile codec and found in realistic vibrotactile signals. None of the 280 signals with 2800 samples had a larger amplitude than 3.5, therefore a value range of the original signal from -5 to 5 is assumed. The quantizer in the vibrotactile codec processes wavelet coefficients obtained by applying DWT to the input signal. The maximum product of the lifting and scale factors occurring in FLWT for CDF 9/7 wavelets is about 1.6, therefore the maximum magnitude of valid x is $5 * 1.6 = 8 = 2^3$. If x is bigger than that, an erroneous input is assumed and Not a Number (NaN) returned as the output value. Since Δ_{quant} and \hat{w}_{max} are also value constrained, as indicated in figure 4.1, assumptions about the output after the rounding function can be made. If `x_div_delta` is below 1.0, it will be rounded to 0 by the `floor(x)` function. The highest possible value for $1/\text{delta}$ is 2^{18} . Given `x_constrained` $\leq 2^{-18}$ then

$$\text{x_div_delta} = \text{x_constrained} * 1/\text{delta} \leq 2^{-18} * 2^{18} = 1 \quad (4.4)$$

Therefore any exponent of `x` that is smaller or equal 2^{-18} leads to a result of 0, irrespective of the other variables. These shortcuts to `X_NAN` and `X_ZERO` are taken based on the exponent of the floating point number alone, resulting in little additional hardware effort. Figure 4.2 shows the (slightly simplified) Finite State Machine (FSM) implementing the described data flow. At the end the original sign bit is appended to the result as a wire from the input register for x .

Most of the edge cases of the IEEE754[IEE19] floating point standard can be ignored in further processing, since subnormal numbers and zero ($\leq 2^{-127}$) are covered by `X_ZERO` and infinity as well as the different NaN variants ($\geq 2^{128}$) lead to `X_NAN`. Especially the multiplier benefits from such reductions in complexity and resource consumption.

4.1.1 Constraining the Input

The input `x` must either be smaller than `max`, or be replaced with a value slightly smaller than `max`. In the Matlab reference code this value `max_almost` is chosen as $0.999 * \text{max}$. For the hardware implementation the multiplier

$$\frac{1023}{1024} \cong 0.999023438 \quad (4.5)$$

is more advantageous since it enables replacing one multiplication by a far less resource-intensive shift and subtraction operation:

$$\text{max_almost} \cong \frac{1023}{1024} * \text{max} = \frac{1024 - 1}{1024} * \text{max} = \text{max} - \frac{1}{1024} * \text{max} = \text{max} - 2^{-10} * \text{max} \quad (4.6)$$

$2^{-10} * \text{max}$ can be calculated by shifting `max` ten digits to the right.

In this `max_almost` is always calculated at the beginning of processing with minimal resource usage.

`x_constrained` is the output of a multiplexer which chooses between `x` and `max_almost` depending on the sign of the subtraction result `max - x`.

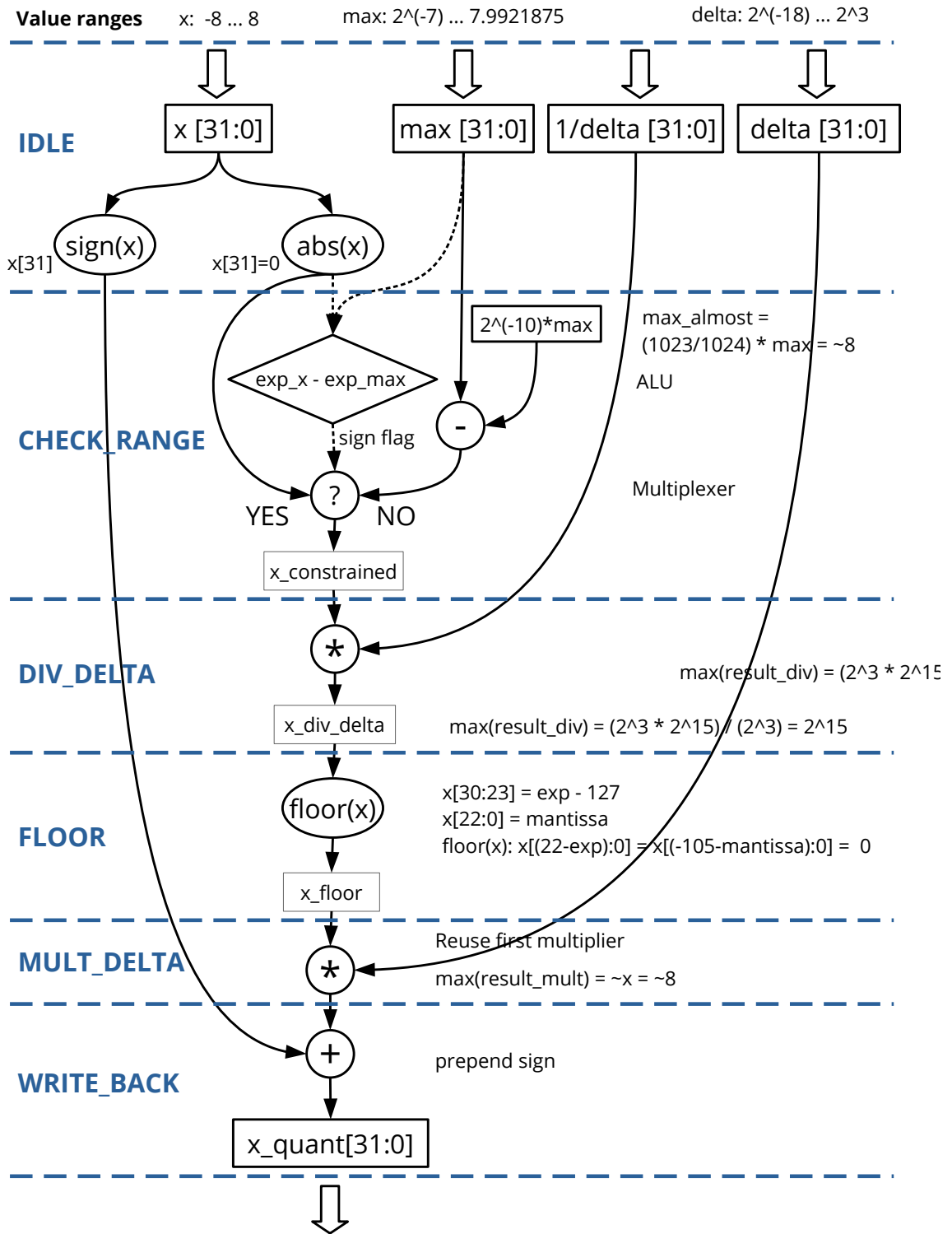


Figure 4.1: Deadzone quantizer data flow graph.

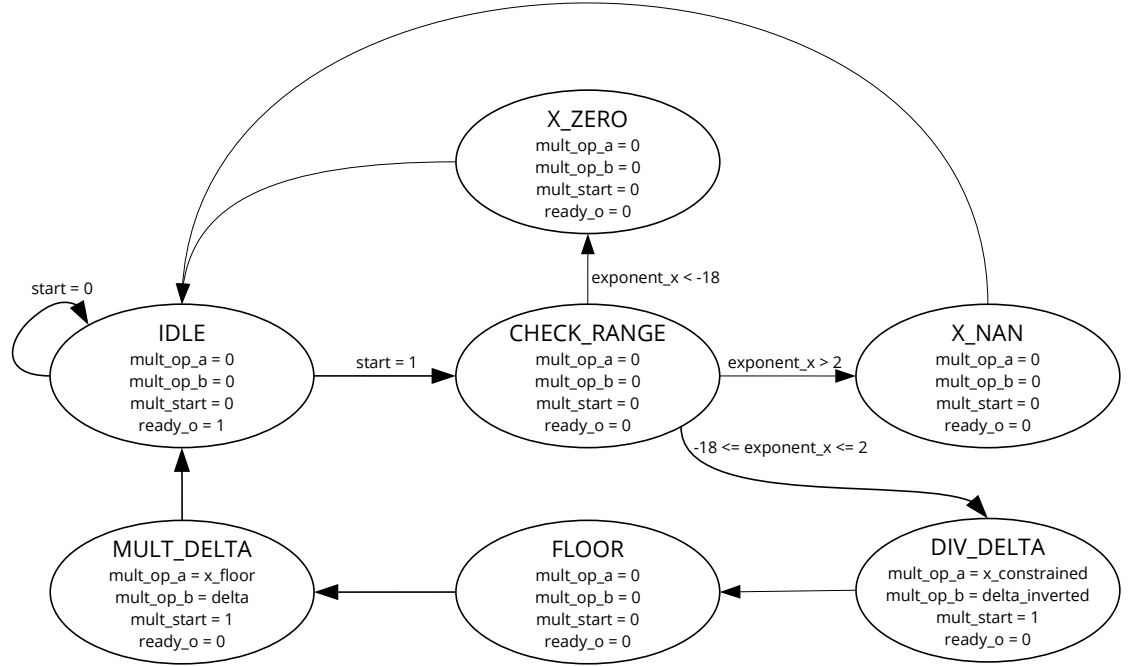


Figure 4.2: Finite State Machine (FSM) of the deadzone quantizer core logic

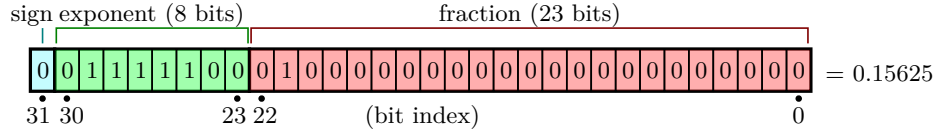


Figure 4.3: Structure of a 32bit floating point number with the sign bit S , the eight exponent bits E and 23 fraction bits (mantissa) M . [FS07] © ⓘ

4.1.2 Multiplier

The multiplier processes two 32bit floating point numbers, as described in IEEE754[IEE19]. The structure of such numbers starts with a sign bit S at position 31 (see figure 4.3), an 8bit exponent and 23 bits for the fraction part, also called the mantissa M . A 127 offset is subtracted from the exponent E for an effective range from -127 to 128 . As mentioned above, special modes apply when the exponent is either -127 (subnormal numbers and zero) or 128 (infinity and NaN). The exponent is chosen so that there is always a single “1” before the fractional dot. Since that is now a given, it no longer needs to be encoded in the mantissa M leading to

$$\text{float} = (-1)^S * 2^{E-127} * 1.M \quad (4.7)$$

The product of two floating numbers is then

$$\text{float1} * \text{float2} = (-1)^{S1+S2} * 2^{E1+E2-127} * (1.M1 * 1.M2) \quad (4.8)$$

As the deadzone quantizer operates on absolute values, the multiplier's input operands are assumed to be positive. Calculating the exponent is a simple addition of the input exponents E_a and E_b , taking the offset into account:

$$E_{\text{result}} = E_{\text{a}} + E_{\text{b}} - 127 \quad (4.9)$$

The mantissa is more complicated since the leading 1 needs to be considered and the position of the fractional dot may change:

$$1.M_a * 1.M_b = 1 + 0.(M_a + 0.M_b) + (0.M_a * 0.M_b) \quad (4.10)$$

The resulting number can either have a one, two or three as integer part before the fractional point.

The minimum is one and not zero since we have already filtered out potential zeros before the multiplier and can therefore rely on both input operands having a leading one:

$$(1.M_{\min})^2 = 1.0_{\text{bin}}^2 = 1.0_{\text{bin}} \quad (4.11)$$

The maximum integer part is three since a binary number with a leading one has a maximum slightly below 2:

$$(1.M_{\max})^2 = (1.111\,111\,111\,111\,111\,111\,111\,111\,11_{\text{bin}})^2 \cong 11.111\,111\,111\,111\,111\,111\,111\,111\,11_{\text{bin}} \quad (4.12)$$

If the leading digit is two or three, the exponent needs to be incremented by one and the mantissa is shifted right by one.

Multiplication of two 23 bit long mantissas leads to a 46bit value that needs to be cut down to 23 bits for a 32bit floating point result. To accomplish this in full compliance with the IEEE754 standard its default rounding mode “Round to Nearest - Tie To Even” was implemented. Here the deciding digit is the one following the desired cutoff after the 23rd bit. If it is 0, nothing needs to be done, since the nearest value is expressed exactly by the first 23 bits. If the 24th bit and any of the following bits is one, the nearest value is one higher than the first 23 bits, therefore they need to be incremented by one. The “Tie” case occurs when the 24th bit is one and all the following digits are zero. Here the 23rd bit determines whether we are already at an even number or need to add one to it and thus round up the tie towards an even number.

4.1.3 Rounding

To accomplish the desired reduction in accuracy/resolution in the deadzone quantizer x is rounded to the next lowest integer (`floor(x)` function) after having been scaled up by $\frac{1}{\Delta_{\text{quant}}}$. The position of the fractional dot depends on the exponent E . E , expressed in thermometer code, is used to mask the mantissa. (Thermometer Code expresses an integer as the count of leading ones in a vector.) If it is 0, the fractional point is before the first mantissa bit and therefore all mantissa bits need to be set to zero. This is accomplished by applying a logical AND operation to the mantissa and exponent thermometer code (zero, therefore a vector of 23 zeros). When the exponent is one, the first mantissa bit needs to be kept, which a mask with one at the first position achieves, and so on. Exponents smaller than zero do not occur at this stage, since input values that would produce such a result have already been filtered out, leading to state `X_ZERO`. Exponents bigger or equal 23 necessitate keeping all mantissa bits, since such floating point values can already only be saved and displayed as integers in 32bit floating point precision.

Finally, `x_floor` is multiplied by `delta`, scaling it to an integer value.

4.2 Hardware Verification

Correct operation of the accelerator is verified at several levels and with two approaches.

The multiplier and the calculating core of the deadzone quantizer are stimulated and their output is in Systemverilog test benches.

The overall accelerator, which includes the connection to the AHB bus, is tested by using it in software running on the simulated RI5CY CPU.

4.2.1 Systemverilog Test Cases

Multiplier

The multiplier unit has tests separate from the deadzone quantizer top unit. They are mostly simple base cases like $1.0 * 1.0$ or $1.0 * 0.0$, but also a multiplication such as $0.9912185669 * 1032.0629921259842 = 1023.000000006$. The latter multiplication is special because the rounding behavior determines whether integer part of the result is 1022 or 1023. Since the integer part often determines the output of the `floor` function in the deadzone quantizer, minuscule rounding errors at this stage can lead to deviations compared to correct implementations of the floating point math in the IEEE754 standard.

Deadzone Quantizer

The majority of input values for the deadzone quantizer are generated randomly by a biased (subtracting one centers it around zero) log-normal distribution with $\mu = 0$ and $\sigma = 1.5$ to achieve similar characteristics to the vibrotactile reference data. In particular, most values should be in the range of DWT coefficients for vibrotactile signals (-8.0 to 8.0 , see section 4.1), clustering around 0. The advantage of taking the logarithm of a normal distribution is that the decades of number space are covered more evenly, which better represents floating point numbers with varying exponents.

To simulate the processing of all values in a DWT band, vectors of length 32 are created as just described, its maximum value selected as `max`, and all members of the vectors set as input `x` to the accelerator, one after the other. The input variables `max`, `delta` and `1/delta` are set only at the beginning of each band, emulating the intended use in the vibrotactile codec software. From equations 2.21 and 2.22 follows

$$\text{delta} = \Delta_{\text{quant}} = \frac{2^{b_{\text{max}}}}{\hat{w}_{\text{max}}} = \frac{2^{\text{bit_budget}}}{\text{max}} \quad (4.13)$$

The maximum bit budget spent on one DWT band is 15. Test cases therefore iterated over all bit budgets between 0 to 15 to calculate `delta` with equation 4.13.

Some edge cases are also covered. These include 0, values just below or above the threshold that leads to `X_ZERO` (2^{-19} and 2^{-18}), the highest number possible in the vibrotactile codec's signed 8bit fixed point format (7.992 187 5) and values above the valid value range like 16.0 or NaN.

In sum, this mix of custom edge cases and randomly generated cases leads to 15 499 separate executions of the deadzone quantizer, all of which succeed in the final version of the accelerator. In this configuration two bits are ignored in the mantissa multiplication.

A Systemverilog `task` is called by the Python-generated Systemverilog test bench with deadzone quantizer's input and expected output values. The `task` writes the values to the accelerator's input registers, sets the control signals, waits for the output signaling completion and then compares the output register of the quantizer with the expected result. A mismatch triggers a warning message and increases the error counter of the current test run.

The source code of the Python script with more details is in appendix A.2.

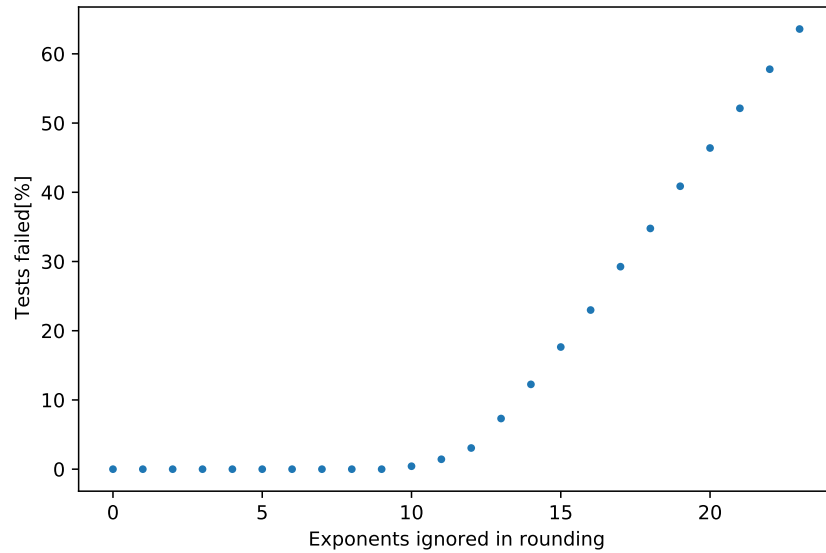


Figure 4.4: Impact of reduced accuracy in `floor()` function.. No errors in 15 499 tests of the deadzone quantizer occur if at most 9 exponents are ignored / not handled exactly.

4.2.2 Accuracy Impact of Hardware Optimizations

Two hardware parameters were adjusted to see their impact on resource-consumption and accuracy.

The number of exponents considered for rounding can be lowered from 23 to 14 with no loss in accuracy, shown in figure 4.4. Theoretically no number at the `floor(x)` stage should be higher than 2^{15} , therefore all exponents above 15 can be ignored, but during tests ignoring an exponent of 15 did not lead to any error either.

How many bits of the mantissas are actually processed by the multiplier, the most resource-intensive part of the accelerator, has a more significant impact on resource usage. Figure 4.5 shows the impact on accuracy of this simplification. To test this the a varying number of mantissa bits, starting with the Least Significant Bit (LSB), is set to to zero, and only the remaining bits are multiplied. The hardware multiplier produced by the synthesis then only needs to process numbers with fewer digits and is therefore smaller.

In 15 499 tests run no test failed if up to two mantissa bits were ignored, and only when ignoring more than 9 mantissa bits did the error rate rise above 0.13 %. The next error rate after that threshold is 1.4 % and rises rapidly from then on.

The hardware optimization described in section 4.1.1, in which the multiplier 0.999 is approximated as $\frac{1023}{1024} \approx 0.9961$, led to 16 tests failing out of 15 499. Since 0.999 is a somewhat arbitrarily chosen factor to produce a number slightly lower than `max`, this should not significantly impact real world fidelity of the hardware-accelerated codec.

4.2.3 Using the Accelerator in Software

To test the AHB wrapper the `Quantize()` C function block was run on the simulated RI5CY CPU with the accelerator connected to its periphery via the AHB bus. Correctness of the outputs of `Quantize()` was checked against the results from the reference Matlab code and data. A Matlab script generated the C source code and headers that contained and used the relevant data. The first blocks of five different signals were processed with the five possible block lengths of the vibrotactile codec (32, 64, 128, 256 and 512) and four different bit budgets (8, 16, 32, 64). In sum 100 blocks with a total of $5 \cdot (32 + 64 + 128 + 256 + 512) \cdot 4 = 19\,840$ samples were quantized without additional errors introduced by the hardware deadzone quantizer.

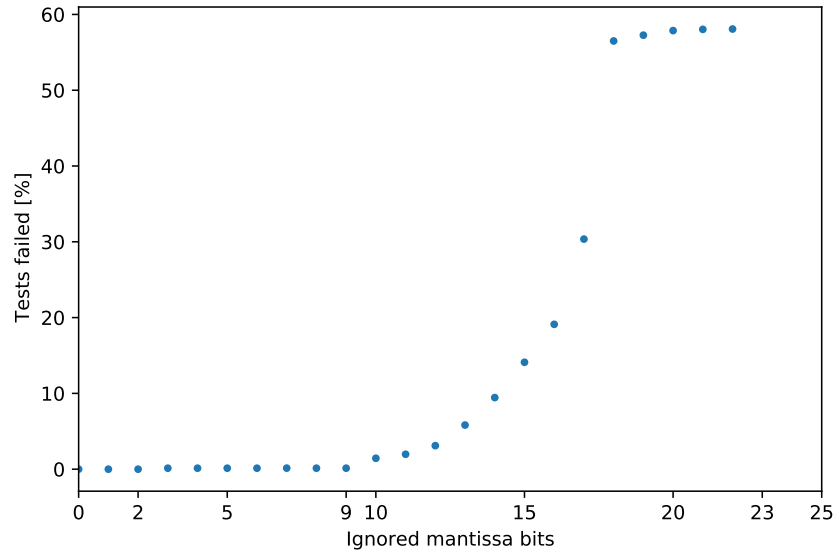


Figure 4.5: Impact of reduced accuracy in mantissa multiplication on accuracy. No errors in 15 499 tests of the deadzone quantizer occur if 0, 1, or 2 mantissa bits are ignored.

4.3 Synthesis

The design was synthesized using the Process Design Kit (PDK) for the 28SLP (Super Low Power) process by Globalfoundries Inc., configured with worst-case process conditions for a 0.9 V supply voltage[19] and an operating temperature of -40°C . All warnings emitted during synthesis related to the deadzone quantizer were fixed. Only synthesis of the RTL description to a net list of transistors was performed, not the steps (floor planning, place and route, ...) that would be needed for a tape out after that. Therefore all area values refer to cell area, not the actual die area of a finished chip.

The synthesized design is composed of the RI5CY CPU core, deadzone quantizer (hardware accelerator), 128 KiB Static Random Access Memory (SRAM) and the AHB bus connecting those elements.

4.3.1 Frequency Characteristics

Figure 4.6 demonstrates the positive correlation between system frequency and area or leakage power, respectively, of the finished design. The irregular decrease of area at the highest achieved frequency might be caused by the need for short critical paths and the use of low V_{th} cells which impact only power but not area.

The pace of leakage power increases significantly after 225 MHz, indicating the need for low V_{th} cells with higher leakage currents to reach those higher frequencies.

The highest frequency that was achieved without timing violations is 260 MHz.

Synthesis was also attempted with the FFT accelerator included that was already developed for the RI5CY at the HPSN institute.

Here only 225 MHz were reached, not enough for real time operation even if the FFT accelerator were used by the codec. The problem is in the FFT accelerator where the input data from the SRAM is multiplied directly for its window function without buffering it first. Since the SRAM alone has a 3 ns latency, this leaves little room for higher frequencies. This could be fixed with an additional register bank, but is outside the scope of this work.

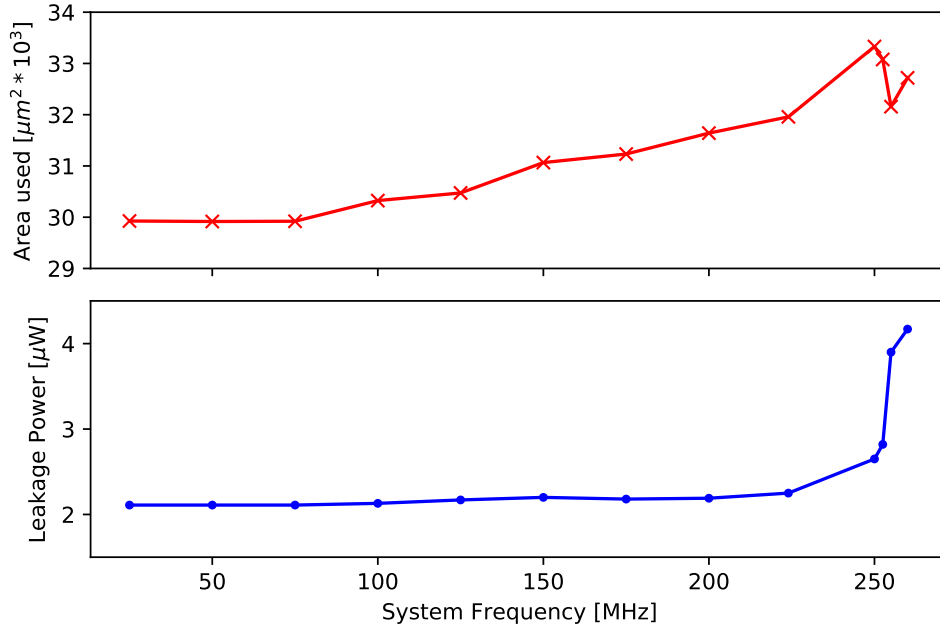


Figure 4.6: Area (without SRAM) and leakage power dependency on system frequency of RI5CY SoC with deadzone quantizer hardware accelerator.

4.3.2 Resource Usage

Area use is dominated by the SRAM's 90 %, as shown in figure 4.7a. A smaller part of the leakage power budget(70 %) in figure 4.8a is consumed by the SRAM, probably because the efficient SRAM macros used are more power optimized than the remaining custom units.

Since the size of the SRAM is configurable, it makes sense to look at the remaining parts of the design in isolation. The RI5CY CPU core takes up 70 % of non-SRAM area, as expected for the processing core that does the bulk of the work executing the codec. The memory cross bar connects the 4 KiB SRAM banks with each other and handle connections to the four IO ports. The AHB connects the CPU with the hardware accelerators. The memory cross bar has a leakage power that is disproportionately higher than its area, presumably because high-performance, low threshold voltage transistors are needed for quick data transfer between the core and SRAM. The deadzone quantizer hardware accelerator takes up 1.09 % of total area and 1.94 % of leakage power. It uses roughly the same amount of area and power as the AHB bus connecting it, leaving little room to further reduce overall resource usage by modifying the accelerator.

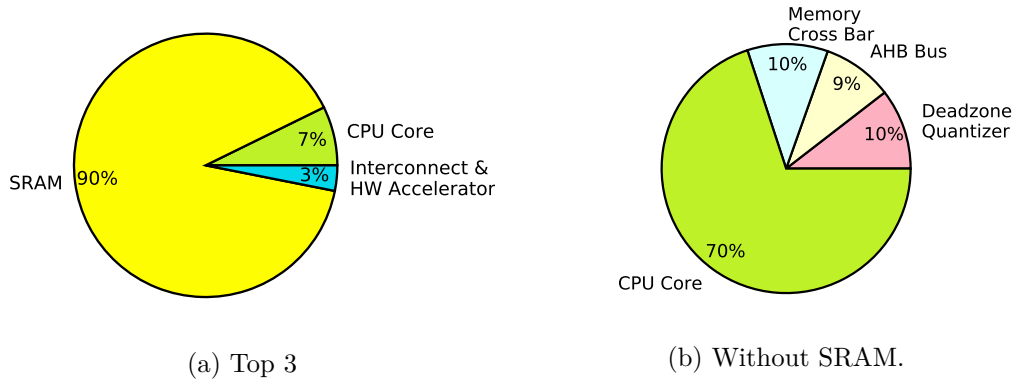


Figure 4.7: Distribution of area use in the synthesized design.
252.5 MHz, two ignored bits in mantissa multiplication.

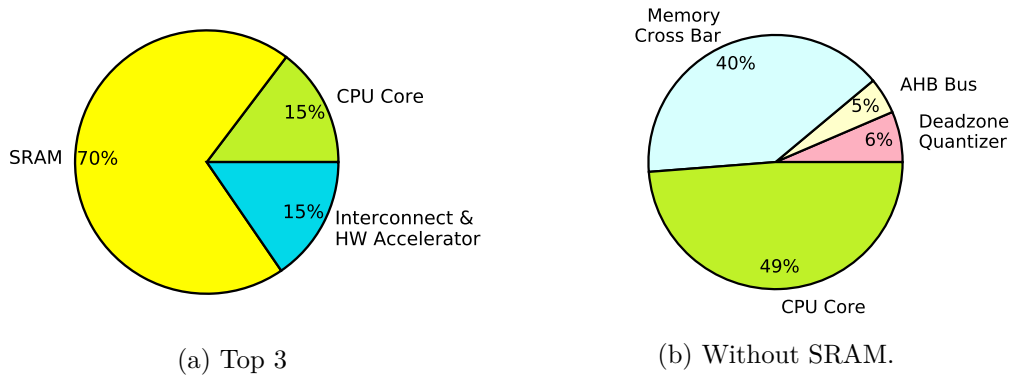


Figure 4.8: Distribution of leakage power use in the synthesized design.
252.5 MHz, two ignored bits in mantissa multiplication.

4.3.3 Optimizations

The floor rounding optimization, for which not all possible exponents were considered to determine which mantissa bits should be set to zero, lead to only modest improvements of $8 \mu\text{m}^2$ deadzone quantizer area saved and 1.2 nW reductions in its leakage power. These changes are not very significant, but since no additional errors are introduced by this optimization it is still put into effect.

Optimizing mantissa multiplication has a bigger impact. Area falls in figure 4.9 as the mantissa multiplication is performed with less bits; leakage power has a less clear downwards trend. There are some points where power rises when ignoring one more bit, namely with two or six ignored bits at 252.5 MHz. This may be an artifact of the iterative synthesis and optimization process. Changed mantissa bits might, for example, make one part of the design the critical path at a certain stage during synthesis that is then optimized further, leading to suboptimal power results in the finished design. Power is heavily influenced by the choice of cells with different threshold voltages V_{th} . To close timing lower threshold voltages are needed which have the side-effect of increased leakage power.

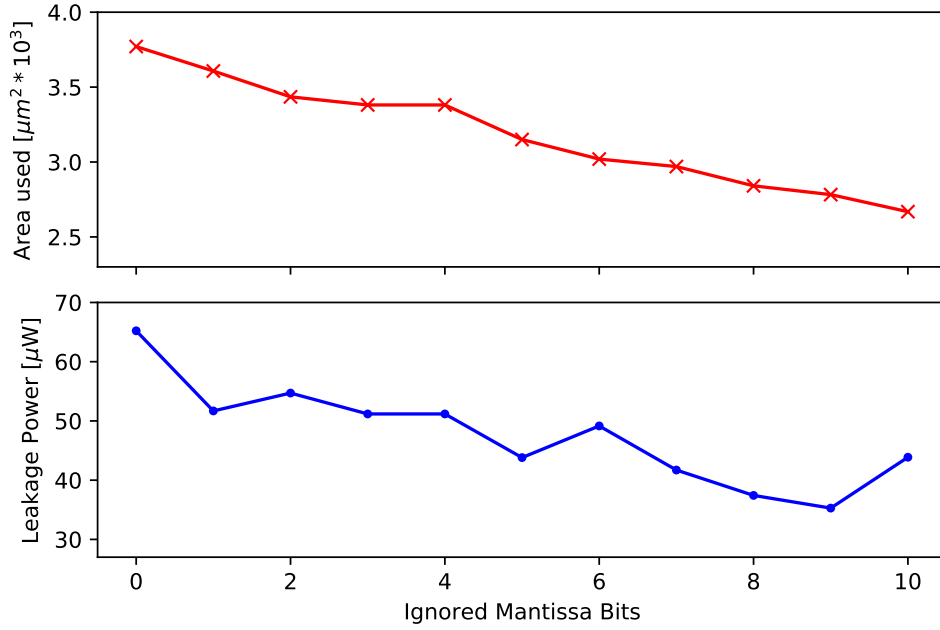


Figure 4.9: Impact of reduced accuracy on deadzone quantizer area and leakage power at 252.5 MHz.

4.4 Hardware Design choices

The performance characteristics of the hardware-accelerated block encoder in table 4.2 show that, depending on the configured bit budget, CPU frequencies between 284 MHz and 310 MHz are needed to encode blocks of samples in real time. This hypothetical frequency is based on the sampling frequency f_s for the vibrotactile signals:

$$f_{\text{realtime}} = f_s * \frac{\text{cycles}}{\text{sample}} = 2800 \text{ Hz} * \frac{\text{cycles}}{\text{sample}} \quad (4.14)$$

Leaving room for overhead added by padding and splitting the input signal into blocks as well as variation in signal complexity makes a system frequency of 320 MHz desirable. Unfortunately the maximum system frequency that leads to a synthesized design without timing violations is 260 MHz, as explained in section 4.3.1.

Benchmark numbers for the hypothetical scenario of performing vibrotactile encoding with help of a fixed FFT accelerator are in table 4.3. The bit budget has a significant impact on the CPU cycles needed to process a sample, while the other relevant parameter, block length, influences mainly the amount of memory needed.

To allow for a little bit of margin over the 249.9 MHz needed for the most computationally intensive configuration while minimizing additional power use a system frequency of 252.5 MHz (clock period $T = 3.96 \text{ ns}$) is chosen.

Total leakage power is raised from $2.65 \mu\text{W}$ at 250 MHz to $2.82 \mu\text{W}$. Choosing 255 MHz or 260 MHz would incur a more consequential increase to $3.9 \mu\text{W}$ or $4.17 \mu\text{W}$, respectively.

With this frequency and the optimizations described in sections 4.3.3 and 4.4 the processor combined with the hardware accelerator has a leakage power of $2.82 \mu\text{W}$ in a cell area of $318011 \mu\text{m}^2$.

While reducing the number of exponents considered for rounding has only minor impact on area and leakage power, there is no downside to ignoring exponents larger than 14. Therefore this is part of the final configuration.

Ignoring bits in the mantissa multiplication leads to more significant savings, as figure 4.9 demonstrates. The two sweet spots here are ignoring two or nine bits. The former leads to no

regressions in result accuracy, while the latter causes only a 0.13 % error rate. Ignoring only one additional bit multiplies the error rate to more than 1 %. Two mantissa multiplication bits less save 9.0 % in deadzone quantizer area and 14.1 % in deadzone quantizer leakage power. Nine bits give 26.2 % area reduction and 45.9 % power savings. In the interest of not impeding error free operation the final configuration leaves out only two bits.

4.5 Hardware Results

The deadzone quantizer itself needs $37\times$ less CPU cycles than the equivalent software routine, roughly halving the effort for the overall `Quantize()` routine. This speedup, seen in figure 4.10, stays the same for different bit budgets since the workloads themselves scale linearly with the quantization resolution.

4.5.1 Latency and Throughput

Looking at the raw throughput numbers in table 4.1 we see that the latency of the hardware accelerator depends on its operating mode and whether the input values lead to any short cuts. Changing all three parameters, listed and explained in section 4.1, for each new quantization of x , takes an additional six cycles over the eleven cycles used for reading and processing x . If x is not a valid input, or so small that the output must be 0.0, the accelerator takes a shortcut to the states `X_NAN` and `X_ZERO` to deliver the output for these scenarios in only six cycles. `X_ZERO` is also activated if a larger value is rounded down to 0.0, skipping three cycles compared to normal execution. For the hypothetical case that samples are merely processed by the deadzone quantizer, the operating frequency would have to be only 47.6 kHz if all parameters change for each calculation or 30.8 kHz if they stay the same.

Mode	Cycles/Sample	Throughput [$\frac{10^6 \text{Samples}}{\text{s}}$]	f_{realtime} [kHz]
Change all parameters	17	14.85	47.6
Only change x	11	22.95	30.8
Only change x : <code>X_NAN</code> and <code>X_ZERO</code>	6	42.08	16.8
Only change x : Round to 0.0	8	31.56	22.4

Table 4.1: Latency and raw throughput of the deadzone quantizer hardware accelerator.

Theoretical throughput at 252.5 MHz. f_{realtime} is calculated for a sampling frequency $f_s = 2800$ Hz.

4.5.2 Impact of FFT accelerator

Aside from the deadzone quantizer hardware accelerator designed in this diploma thesis, the FFT accelerator for the RI5CY that was already developed at the HPSN chair is also useful for the vibrotactile codec. As shown in figure 3.3 the FFT function block takes up between 15 % and 21 % of CPU cycles on the RI5CY processor. The results in table 4.3 exclude the time needed for the software FFT. This assumes that the execution time of the FFT accelerator is negligible compared to the overall codec. The results in table 4.3 show significant improvements compared to table 4.2 where the FFT is computed without a hardware accelerator.

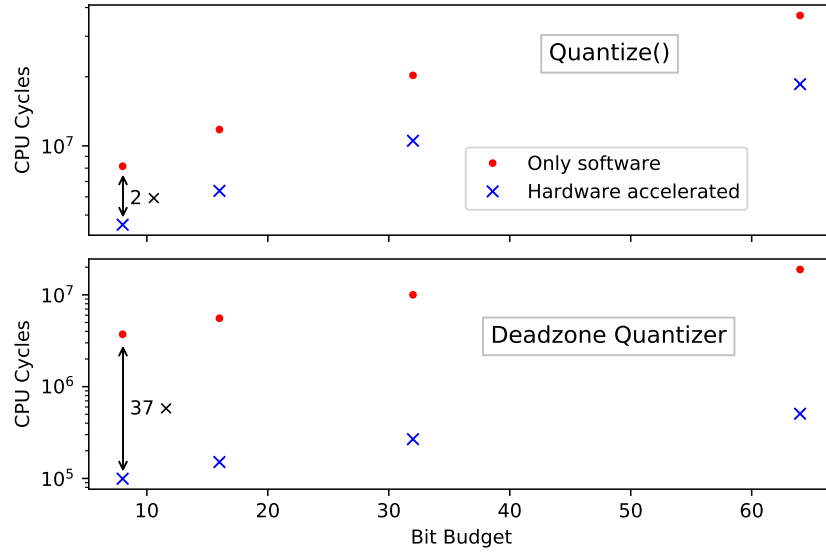


Figure 4.10: RI5CY CPU Cycles needed to quantize 4960 samples. Five different signals with five different block lengths (32, 64, 128, 256, 512) were processed in either the whole `Quantizer()` function block of the vibrotactile codec, or only the hardware-accelerated deadzone quantizer part. The observed speedup is $2\times$ and $37\times$, respectively.

Bit Budget	Cycles/Sample	f_{realtime} [MHz]	Cycles saved with HW Accelerator [%]
8	101 190	283.3	0.72
16	103 312	289.3	1.06
32	106 983	299.6	1.85
64	110 677	309.9	3.37

Table 4.2: Performance characteristics of the hardware-accelerated vibrotactile block encoder. f_{realtime} is the minimum frequency needed to encode samples in real time (live) for a sampling frequency of $f_s = 2800$ Hz

4.5.3 Hardware-accelerated Vibrotactile Codec Performance

Table 4.2 gives key performance characteristics of the RI5CY running the vibrotactile codec with help of the deadzone quantizer hardware accelerator. The accelerator saves between 0.72% and 3.27% CPU time compared to software-only calculations.

Figures 4.11 and 3.3 illustrate the reasons for the five-fold difference in speedup and the relatively small impact on overall performance.

The bulk of algorithmic complexity is in the psychohaptic model, with its peak finding and perceptual mask calculating procedures. Unfortunately it does not easily lend itself to hardware acceleration, as was explained in section 3.3.2.

The bit budget parameter, which controls the quantization resolution, influences only the quantizer and SPIHT encoder function blocks. Higher resolution leads to more coefficients quantized by the deadzone quantizer and a bigger data stream encoded in SPIHT. Thus the hardware accelerator is needed more often when the bit budget is larger. DWT and FFT transforms apply to the original signal, forming inputs to the psychohaptic model and the quantizer, respectively.

Integrating the FFT accelerator that was already designed at the HPSN into the overall codec execution would bring significant benefits, explored in section 4.5.2.

Bit Budget	Cycles/Sample	$f_{realtime}$ [MHz]	Cycles saved with HW Accelerator including FFT [%]
8	79 751	223.3	27.8
16	81 874	229.2	27.5
32	85 545	239.5	27.3
64	89 238	249.9	28.2

Table 4.3: Performance characteristics of the hardware-accelerated vibrotactile block encoder without FFT. $f_{realtime}$ is the minimum frequency needed to encode samples in real time (live) for a sampling frequency of $f_s = 2800$ Hz

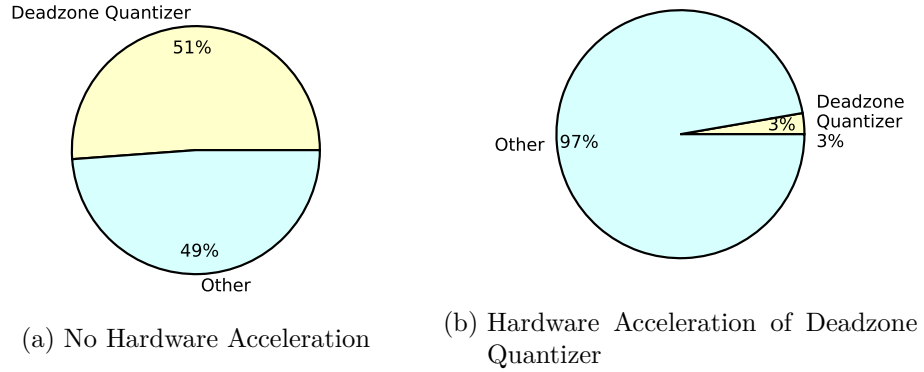


Figure 4.11: Distribution of CPU cycles in execution of `Quantize()` on the RI5CY processor. The first block of five signals with five different block lengths (32, 64, 128, 256, 512) were processed, resulting in a total of $5 * (32 + 64 + 128 + 256 + 512) = 4960$ encoded samples.

5 Conclusion

5.1 Summary

Among the approaches for recording and compressing vibrotactile signals, the codec by Noll et al. is by far the most advanced[Nol+20]. Thanks to the Matlab code and vibrotactile reference provided by Andreas Noll (Chair of Media Technology of Technical University Munich) [Nol21], an in depth understanding of the vibrotactile codec could be translated into a new C implementation with unit tests against the reference.

After optimizing the code several parts with especially high computational complexity were detected. Due to the lack of a Floating Point Unit (FPU) on the RI5CY as configured at the institute, all kinds of floating point operations take a long time on this RISC-V CPU. In the x86 application issues in the SPIHT coder were more prevalent, owing to the use of lists with fluctuating size and bit-level operations. Linked lists, which are not yet implemented in the C code, could help with the lists used by the SPIHT codec. On the RI5CY, the psychohaptic model took up most of the execution time. But since this functionality might easily change in future iterations of the codec and is not amenable to hardware acceleration in other aspects as well, the quantizer was hardware accelerated instead. Here the inner loop is the deadzone quantizer which is applied to each sample of DWT band. It is called especially often because bands are often quantized not just once but several times to achieve an optimal distribution of Signal-to-Mask-Ratio (SMR) as calculated by the psychohaptic model. $37\times$ less time is spent on the deadzone quantizer when using the hardware accelerator, cutting in half the time needed for the overall quantizer. Speedup of the complete codec ranges from 0.7 % to 3.4 %, rising with the bit budget. An increasing bit budget correlates with higher quality and resource usage as well as lower compression rates.

Synthesis targets the 28nm SLP process by Globalfoundries, Inc. for 0.9 V operating voltage and an operating temperature of -40°C . The synthesized hardware needs 3.18 mm^2 of area and $2.82\text{ }\mu\text{W}$ leakage power at the system frequency which would enable real time processing of samples at 252.5 MHz if a FFT accelerator with a sufficiently short critical path were used. The area added by the accelerator is $3435\text{ }\mu\text{m}^2$ (1.08 % of total area).

The hardware solution described above encodes vibrotactile signal blocks. With no licensing costs for the RI5CY open-source RISC-V CPU and the total area for this project of about 3.18 mm^2 , it would be feasible to use multiple cores at once for encoding vibrotactile data streams in parallel. It was hard to find a part of the codec where hardware acceleration can have a big impact with an acceptable effort. Although the deadzone quantizer is the routine that is called most often in the program, its contribution to total CPU time is not as large. The following section gives ideas for future improvements, among them accelerating the FFT with a design already developed at the HPSN institute.

5.2 Future Research Opportunities

Software Improvements There is still work to be done in fixing the SPIHT encoder and implementing a `Encoder()` function merging the data blocks with their headers into one continuous data stream. Using linked lists that do not need as many memory allocation calls to manage the SPIHT lists as the currently implemented linear C arrays will help. Creating the C data structures necessary for serial encoding of such bit streams is not trivial.

Another approach for improving performance is reducing the accuracy of the logarithm function, which is quite resource intensive even with a FPU.

Software that targets environments that are less constrained than the embedded, bare-metal 32bit RI5CY CPU would profit from a different development approach which uses the C++ standard library and numerical template libraries like Eigen[GJ+10] to streamline the code. Then most of the custom function implementing varying mathematical operations on vectors containing different scalar data types would no longer be necessary as they could be replaced by well-tested and optimized standard functions and data types. The Eigen library, for example, has optimizations for the vector processing units of modern 64bit x86 and ARM CPUs. Using established C++ data structures and methods would also ease handling of binary data, described in the above paragraph about SPIHT, and interactions with file systems.

Potential impact of an FFT accelerator As mentioned in section 4.5.2, including a FFT accelerator would achieve the goal of live, real time encoding, if the issue of the too long critical path were solved.

Improvements through a Floating Point Unit The RI5CY CPU core, now under development under the CV32E40P name, can be configured with a Floating Point Unit (FPU). While this would presumably add quite a bit to total area, it opens the door to lower operating frequencies - and the accompanying power savings - than with only the hardware accelerators mentioned so far. Then the performance characteristics would resemble more closely those measured on an x86, seen in section 3.3.1 and in more detail in appendix A.1, where the SPIHT algorithm dominates execution time.

Bibliography

- [ANR74] N. Ahmed, T. Natarajan, and K.R. Rao. “Discrete Cosine Transform”. In: *IEEE Transactions on Computers* C-23.1 (Jan. 1974), pp. 90–93.
- [Ban+02] Kyoung Ho Bang et al. “Design and VLSI implementation of a digital audio-specific DSP core for MP3/AAC”. In: *IEEE Transactions on Consumer Electronics* 48.3 (Aug. 2002), pp. 790–795.
- [Bar+04] S. Barua, J. E. Carletta, K. A. Kotteri, and A. E. Bell. “An efficient architecture for lifting-based two-dimensional discrete wavelet transforms”. In: *Proceedings of the 14th ACM Great Lakes symposium on VLSI*. GLSVLSI ’04. New York, NY, USA: Association for Computing Machinery, Apr. 2004, pp. 61–66.
- [Ben48] W. R. Bennett. “Spectra of Quantized Signals”. In: *Bell System Technical Journal* 27.3 (July 1948), pp. 446–472.
- [BHY05] Sliman Bensmaïa, Mark Hollins, and Jeffrey Yau. “Vibrotactile intensity and frequency information in the Pacinian system: A psychophysical model”. en. In: *Perception & Psychophysics* 67.5 (July 2005), pp. 828–841.
- [Bor21] Mark Borgerding. *mborgerding/kissfft*. original-date: 2017-10-25T13:55:40Z. Feb. 2021.
- [Bor+06] Simone Borgio et al. “Hardware DWT accelerator for MultiProcessor System-on-Chip on FPGA”. In: *Modeling and Simulation 2006 International Conference on Embedded Computer Systems: Architectures*. July 2006, pp. 107–114.
- [Bor05] C.W. Borst. “Predictive coding for efficient host-device communication in a pneumatic force-feedback display”. In: *First Joint Eurohaptics Conference and Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems. World Haptics Conference*. Mar. 2005, pp. 596–599.
- [Bra99] Karlheinz Brandenburg. “MP3 and AAC explained”. In: *Audio Engineering Society Conference: 17th International Conference: High-Quality Audio Coding*. Audio Engineering Society. 1999.
- [BHJ99] A. J. Brisben, S. S. Hsiao, and K. O. Johnson. “Detection of Vibration Transmitted Through an Object Grasped in the Hand”. In: *Journal of Neurophysiology* 81.4 (Apr. 1999), pp. 1548–1558.
- [Bri98] Christopher M. Brislawn. “The FBI Fingerprint Image Compression Specification”. en. In: ed. by Pankaj N. Topiwala. *The International Series in Engineering and Computer Science*. Boston, MA: Springer US, 1998, pp. 271–288.
- [Bro12] Bronstejn. *Taschenbuch der Mathematik*. Frankfurt am Main: Harri Deutsch, 2012.

- [BSK05] Douglas S Brungart, Brian D Simpson, and Alexander J Kordik. “The detectability of headtracker latency in virtual audio displays”. In: Georgia Institute of Technology. 2005.
- [CSE00] C. Christopoulos, A. Skodras, and T. Ebrahimi. “The JPEG2000 still image coding system: an overview”. In: *IEEE Transactions on Consumer Electronics* 46.4 (2000), pp. 1103–1127.
- [Chu21] Eric H. Chudler. *Brain Facts and Figures*. 2021. URL: <https://faculty.washington.edu/chudler/facts.html> (visited on 02/12/2021).
- [Cra20] Leah Crane. “SpaceX’s first crewed flight is a go”. In: *New Scientist* 246.3281 (2020), p. 17.
- [Cut54] Cassius C. Cutler. “Transmission Systems employing quantization”. US2927962. Apr. 26, 1954.
- [Dau92] Ingrid Daubechies. *Ten Lectures on Wavelets*. Society for Industrial and Applied Mathematics, Jan. 1992.
- [DS97] Ingrid Daubechies and Wim Sweldens. “Factoring wavelet transforms into lifting steps”. In: *Wavelets in the Geosciences*. Springer-Verlag, 1997, pp. 131–157.
- [Dco07] Dcoetzee. *Huffman tree*. May 15, 2007. URL: https://commons.wikimedia.org/wiki/File:Huffman_tree.svg.
License: <https://creativecommons.org/publicdomain/zero/1.0/deed.en>.
- [Fal18] FalseCondition. *Wavelet filterbank*. June 24, 2018. URL: https://commons.wikimedia.org/wiki/File:Wavelet_filterbank.svg.
License: <https://creativecommons.org/licenses/by-sa/4.0/deed.en>.
- [Fra+10] Joaquin Franco et al. “The gpu on the 2d wavelet transform, survey and contributions”. In: *In proceedings of Para 2010: State of the Art in Scientific and Parallel Computing* (2010), pp. 173–183.
- [20] *Frequently Asked Questions about the GNU Licenses*. Free Software Foundation. 2020. URL: <https://www.gnu.org/licenses/gpl-faq.en.html> (visited on 03/04/2021).
- [FS07] Fresheneesz and Stannered. *File:Float example.svg*. Apr. 2007. License: Attribution-Share Alike 3.0 Unported (<https://creativecommons.org/licenses/by-sa/3.0/deed.en>).
- [Gau+17] Michael Gautschi et al. “Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (Oct. 2017). RISCY, pp. 2700–2713.
- [Goo21a] Google, ed. *gperftools/gperftools*. original-date: 2014-01-28T11:15:07Z. Feb. 2021.
- [Goo21b] GoogleOpenSource. *google/googletest*. original-date: 2015-07-28T15:07:53Z. Jan. 2021.
- [GJ+10] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. <http://eigen.tuxfamily.org>. 2010.
- [HS20] R. Hassen and E. Steinbach. “Subjective Evaluation of the Spectral Temporal SIMilarity (ST-SIM) Measure for Vibrotactile Quality Assessment”. In: *IEEE Transactions on Haptics* 13.1 (Jan. 2020), pp. 25–31.
- [HSC06] P. Hinterseer, E. Steibach, and S. Chaudhuri. “Model based data compression for 3D virtual haptic teleinteraction”. In: *2006 Digest of Technical Papers International Conference on Consumer Electronics*. ISSN: 2158-4001. Jan. 2006, pp. 23–24.
- [Huf52] David Huffman. “A Method for the Construction of Minimum-Redundancy Codes”. In: *Proceedings of the IRE* 40.9 (Sept. 1952), pp. 1098–1101.

- [IEE19] IEEE. “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (July 22, 2019), pp. 1–84.
- [JR72] N. S. Jayant and L. R. Rabiner. “The Application of Dither to the Quantization of Speech Signals”. In: *The Bell System Technical Journal* 51 (No. Feb. 15, 1972), pp. 1293–1304.
- [Joh05] John Teslade. *Wavelets - DWT Freq.* July 15, 2005. URL: https://en.m.wikipedia.org/wiki/File:Wavelets_-_DWT_Freq.png (visited on 07/18/2020).
- [Kit] Kitware. *Overview / CMake*. en-US.
- [Kuz+02] Georgi Kuzmanov, Bahman Zafarifar, Prarthana Shrestha, and Stamatis Vassiliadis. “Reconfigurable DWT unit based on lifting”. In: *Proc. 13th Annual Workshop on Circuits, Systems, and Signal Processing (ProRISC2002)*. 2002. 2002, pp. 325–333.
- [Lan+10] Nils Landin, Joseph M. Romano, William McMahan, and Katherine J. Kuchenbecker. “Dimensional Reduction of High-Frequency Accelerations for Haptic Rendering”. en. In: *Haptics: Generating and Perceiving Tangible Sensations*. Ed. by Astrid M. L. Kappers, Jan B. F. van Erp, Wouter M. Bergmann Tiest, and Frans C. T. van der Helm. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 79–86.
- [LH07] Yi-Ting Lin and Ing-Jer Huang. “Enhanced 32-bit Microprocessor-based SoC for Energy Efficient MP3 Decoding in Portable Devices”. In: *2007 Digest of Technical Papers International Conference on Consumer Electronics*. ISSN: 2158-4001. Jan. 2007, pp. 1–2.
- [Liu+17] Xun Liu, Mischa Dohler, Toktam Mahmoodi, and Hongbin Liu. “Challenges and opportunities for designing tactile codecs from audio codecs”. In: *2017 European Conference on Networks and Communications (EuCNC)*. June 2017, pp. 1–5.
- [Luo09] Fa-Long Luo, ed. *Mobile Multimedia Broadcasting Standards: Technology and Practice*. en. Springer US, 2009.
- [Mat20] Mathworks. *findpeaks - Matlab R2020b Documentation*. 2020.
- [Met07] Meteficha. *Huffman tree 2*. Oct. 7, 2007. URL: https://commons.wikimedia.org/wiki/File:Huffman_tree_2.svg. License: <https://creativecommons.org/publicdomain/zero/1.0/deed.en>.
- [Mui+17] Fathiah Abdul Muin, Teddy Surya Gunawan, Mira Kartiwi, and Elsheikh M. A. Elsheikh. “A review of lossless audio compression standards and algorithms”. In: *AIP Conference Proceedings* 1883.1 (Sept. 2017), p. 020006.
- [NS07] Nicholas Nethercote and Julian Seward. “Valgrind: A framework for heavyweight dynamic binary instrumentation”. In: vol. 42. June 2007, pp. 89–100.
- [Nol21] Andreas Noll. Chair of Media Technology, Technical University Munich. 2021. URL: <https://www.ei.tum.de/lmt/team/mitarbeiter/noll-andreas/> (visited on 02/17/2021).
- [Nol+20] Andreas Noll, Basak Gülecüyüz, Alexander Hofmann, and Eckehard Steinbach. “A Rate-scalable Perceptual Wavelet-based Vibrotactile Codec”. In: *2020 IEEE Haptics Symposium (HAPTICS)*. ISSN: 2324-7355. Mar. 2020, pp. 854–859.
- [Oka+08] Shogo Okamoto, Masashi Konyo, Satoshi Saga, and Satoshi Tadokoro. “Identification of cutaneous detection thresholds against time-delay stimuli for tactile displays”. In: *2008 IEEE International Conference on Robotics and Automation*. ISSN: 1050-4729. May 2008, pp. 220–225.

- [ONY13] Shogo Okamoto, Hikaru Nagano, and Yoji Yamada. “Psychophysical Dimensions of Tactile Perception of Textures”. In: *IEEE Transactions on Haptics* 6.1 (2013), pp. 81–93.
- [OY10] Shogo Okamoto and Yoji Yamada. “Perceptual properties of vibrotactile material texture: Effects of amplitude changes and stimuli beneath detection thresholds”. In: *2010 IEEE/SICE International Symposium on System Integration*. Dec. 2010, pp. 384–389.
- [Ome06] Omegatron. *Haar wavelet*. June 3, 2006. URL: https://de.wikipedia.org/wiki/Datei:Haar_wavelet.svg (visited on 07/20/2020). License: <https://creativecommons.org/licenses/by-sa/3.0/deed.de>.
- [21] *openhwgroup/cv32e40p*. original-date: 2016-02-18T18:21:33Z. Jan. 2021.
- [19] *pb-28slp-12-web*. GLOBALFOUNDRIES Inc., 2019.
- [PdB16] S. M. Petermeijer, J. C. F. de Winter, and K. J. Bengler. “Vibrotactile Displays: A Survey With a View on Highly Automated Driving”. In: *IEEE Transactions on Intelligent Transportation Systems* 17.4 (2016), pp. 897–907.
- [Poo+04] Mohammad Pooyan, Ali Taheri, Morteza Moazami-Goudarzi, and Iman Saboori. “Wavelet compression of ECG signals using SPIHT algorithm”. In: *International Journal of signal processing* 1.3 (2004), p. 4.
- [RS11] Takkiti Rajashekar Reddy and Rangu Srikanth. “Hardware implementation of DWT for image compression using SPIHT algorithm”. In: *Int. J. Comput. Trends Technol* 2.2 (2011), pp. 58–62.
- [SP96] A. Said and W.A. Pearlman. “A new, fast, and efficient image codec based on set partitioning in hierarchical trees”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 6.3 (June 1996), pp. 243–250.
- [Sai+12] Taoufik Saidani, Mohamed Atri, Yahia Said, and Rached Tourki. “Real time FPGA acceleration for discrete wavelet transform of the 5/3 filter for JPEG 2000”. In: *2012 6th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT)*. Mar. 2012, pp. 393–399.
- [Sch01] H.R. Schiffman. *Sensation and Perception: An Integrated Approach*. New York: John Wiley & Sons, Inc, 2001.
- [Sha48a] C. E. Shannon. “A mathematical theory of communication”. In: *The Bell System Technical Journal* 27.3 (July 1948), pp. 379–423.
- [Sha48b] C. E. Shannon. “A mathematical theory of communication”. In: *The Bell System Technical Journal* 27.3 (July 1948), pp. 623–656.
- [Sha93] J.M. Shapiro. “Embedded image coding using zerotrees of wavelet coefficients”. In: *IEEE Transactions on Signal Processing* 41.12 (Dec. 1993), pp. 3445–3462.
- [Shi13] Anand Lal Shimpi. *How ARM’s Business Model Works. How ARM makes money*. Anandtech. June 28, 2013. URL: <https://www.anandtech.com/show/7112/the-arm-diaries-part-1-how-arms-business-model-works/2> (visited on 12/17/2019).
- [SLH11] Changhe Song, Yunsong Li, and Bormin Huang. “A GPU-Accelerated Wavelet Decompression System With SPIHT and Reed-Solomon Decoding for Satellite Images”. In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 4.3 (Sept. 2011), pp. 683–690.
- [Ste+11] Eckehard Steinbach et al. “Haptic Data Compression and Communication”. In: *IEEE Signal Processing Magazine* 28.1 (Jan. 2011), pp. 87–96.

- [] *The 3-Clause BSD License*. Open Source Initiative. URL: <https://opensource.org/licenses/BSD-3-Clause> (visited on 03/04/2021).
- [Val13] Clemens Valens. *A Really Friendly Guide To Wavelets – PolyValens*. en-US. 2013. URL: <https://web.archive.org/web/20191221091855/http://www.polyvalens.com/blog/wavelets/theory/> (visited on 07/23/2020).
- [Vos+13] Koen Vos, Karsten Vandborg Sørensen, Søren Skak Jensen, and Jean-Marc Valin. “Voice Coding with Opus”. English. In: Audio Engineering Society, Oct. 2013.
- [Wal92] G.K. Wallace. “The JPEG still picture compression standard”. In: *IEEE Transactions on Consumer Electronics* 38.1 (Feb. 1992), pp. xviii–xxxiv.
- [Web51] Ernst Heinrich Weber. *Die Lehre vom Tastsinne und Gemeingefühle auf Versuche gegründet*. German. Braunschweig: Friedrich Vieweg und Sohn, 1851. 143 pp.
- [Zaf+02] Bahman Zafarifar et al. “Micro-codable discrete wavelet transform”. In: *Computer Engineering Laboratory, Delft University of Technology* (2002).
- [ZJ10] Zhang Zhi-hui and Zhang Jun. “Unsymmetrical SPIHT Codec and 1D SPIHT Codec”. In: *2010 International Conference on Electrical and Control Engineering*. June 2010, pp. 2498–2501.

A Appendix

A.1 Detailed Performance Analysis for x86

This section will do a more detailed performance analysis of the vibrotactile codec running on an x86 CPU, as measured with the *gperftools* profiling software. Since the CPU has a Floating Point Unit (FPU), it gives an impression how the codec would perform if the RI5CY RISC-V CPU was equipped with its optional FPU.

gperftools samples a program while it is running 100 times per second and records which function is executed, and by which mother function it was called at that instant. The analysis differentiates between the number of samples when the function itself was running or when one of its child functions was running (cumulative). This cumulative value gives a good impression of the distribution of CPU time between major function blocks like `PsychohapticModel()` or `Quantize()`. The individual view is helpful to identify the specific function the most execution time is spent on. Optimizing them - or using less of them - gives the greatest overall benefit to program runtime.

The C implementation of the vibrotactile codec processed five vibrotactile signals with 2800 samples each 1000 times with five different block lengths, leading to $5 * 2800 * 5 = 70\,000$ samples processed in total. The performance analysis was run on a Intel Core i5-3320M CPU with 16 GiB RAM with code configured to use floating point values in 32bit precision and `malloc` routines optimized for memory constrained embedded environments.

Tables A.1 and A.2 display the 50 functions that use up the most cumulative CPU time, with some irrelevant functions like the `libc` routine bootstrapping the binary and subroutines of `kiss_fft` left out. Functions are grouped by the function blocks they are called by, as described in section 3.1.2 and displayed in figure 3.1. At the top of both lists are `main()` and `BlockEncoder()` since those are the entry function to the C program and the mother function calling all the other parts of the code, respectively. When functions are called in multiple function blocks, they are summarized under “Multiple Callers”. This is most often the case for the combination of `PsychohapticModel()` and `Quantize()`, because the quantizer uses decibel values from the psychohaptic model. Therefore they both need to calculate logarithms, etc.

When using the smaller bit budget of 8 (table A.1) the `PsychohapticModel()` function block uses the most computing resources, as it or its child functions it calls, e.g. `GlobalMaskingThreshold()` and `VibFft()`, were active during 48.2% of samples taken by *gperftools*. `VibFft()` is a wrapper function for the open source KISS FFT (`kiss_fft`) library by Mark Borgerding [Bor21]. `PowScalarToVibVector()` applies the power function ($\text{scalar}^{\text{vectorelement}}$) to each element of vector. The `__powf_sse2` function uses the SSE2 (Streaming SIMD Extensions 2 (SSE2) processor extension with Single Instruction, Multiple Data (SIMD) to take the power of multiple 32bit float values in parallel. Since it calls no

other functions the individual execution time and cumulative time are both 3.0 % of the total.

At the high end, a bit budget of 64 (table A.2), lets the `Spiht1dEncode()` block dominate with 69.2 % of CPU time spent in it or one of the functions it calls, because the bitstream it encodes grows with the higher quantization resolution. The sub function `MaxDescendant()` alone consumes almost half the resources with 49.9 % cumulative execution time. Since SPIHT uses lists with integer indices that change size, `VibVectorIntAppendInt()` and `VibVectorIntCutValue()` make up a substantial amount of the cumulative execution time. By far the most calls of `MallocNano()`, `FreeNano()` etc. also come from `Spiht1dEncode()`, but of course not exclusively. SPIHT as the final coder stage produces a binary data stream, which is formed by appending bits to it with `VibVectorBitsAppendBit()`. The proportion of `Quantize()` is also higher at a bit budget of 64 (9.7 %) than at a bit budget of 8 (6.2 %), because it has to find the optimal distribution of a higher bit budget and the accompanying finer quantization resolution.

The execution of `DwtCdf97()` and `PsychohapticModel()` are not influenced by the bit budget, leading to a lower relative ratio of execution time (proportional to *gperftools* samples) at a higher bit budget. The lifting implementation of DWT is very efficient so that processing a 1D signal takes little time compared to the other function blocks described above.

The most potential for future improvement lies in the `Spiht1dEncode()` and `PsychohapticModel()` function blocks. The former would profit immensely from pre-allocated data structures with linked lists, saving on calls to `malloc` routines and drastically decreasing the computational complexity of cutting a value from the middle of a list. The latter's complexity mainly lies in the algorithms prescribed by the vibrotactile codec itself. Maybe there will be some simplifications of the psychohaptic model in future iterations of the vibrotactile codec.

Function Name	Execution Time [%]	Cumulative Execution Time [%]
main	0.0%	100.0
BlockEncoder	0.0	99.8
PsychohapticModel	0.0	48.2
GlobalMaskingThreshold	2.3	29.6
FreqSpectrum	0.0	13.0
VibFft	0.5	9.9
PowScalarToVibVector	0.6	9.5
__powf_sse2	8.9	8.9
FindPeaks	0.1	6.5
PeakMask	2.5	6.5
SumEnergy (inline)	0.0	5.4
kiss_fft	0.0	5.4
PeakProminence	4.6	4.8
__sincosf_sse2	3.0	3.0
PerceptualThreshold	0.0	2.8
VibVectorAddScalar	1.8	1.8
FindAllPeakLocations	0.9	1.1
Spiht1dEncode	1.5	42.2
MaxDescendant	6.3	36.3
VibVectorIntAppendInt	3.0	13.9
__memcpy_sse2_unaligned_erms	4.1	4.1
VibVectorIntCutValue	1.2	1.4
VibVectorIntMaxAbs	1.3	1.3
VibVectorBitsAppendBit	1.0	1.0
Quantize	0.3	6.2
QuantizeSingleDwtBand	2.2	3.6
QuantizeDwtBands	0.0	2.5
DwtCdf97	0.0	3.0
ForwardDwtCdf97	1.8	2.2
Multiple Callers		
VibVectorIntRealloc	4.3	18.7
ReallocNano	3.8	14.4
MallocNano	7.8	7.8
FreeNano	5.7	6.6
VibVectorLog10	0.2	6.6
__ieee754_log10f	2.7	5.8
VibVectorIntMalloc	0.8	4.5
__logf_sse2	3.1	3.1
VibVectorMultScalar	3.1	3.1
VibVectorMax	2.9	2.9
VibVectorIntFree	0.6	2.2

Table A.1: Distribution of execution time on x86 with bit budget 8.
70 000 vibrotactile samples were processed by the vibrotactile codec.
gperftools recorded 4162 samples at a sampling frequency of 100 Hz

Function Name	Execution Time [%]	Cumulative Execution Time [%]
main	0.0	100.0
BlockEncoder	0.0	99.9
Spiht1dEncode	3.7	69.2
MaxDescendant	8.7	52.7
VibVectorIntAppendInt	3.6	19.4
VibVectorIntCutValue	2.6	3.6
__memcpy_sse2_unaligned_erms	3.1	3.1
VibVectorBitsAppendBit	2.9	2.9
VibVectorIntMaxAbs	2.7	2.7
VibVectorIntDuplicate	0.1	1.1
PsychohapticModel	0.0	19.8
GlobalMaskingThreshold	1.0	12.2
FreqSpectrum	0.0	5.3
PowScalarToVibVector	0.1	4.1
VibFft	0.2	4.1
__powf_sse2	4.0	4.0
FindPeaks	0.0	3.0
PeakProminence	2.2	2.3
PeakMask	0.9	2.2
SumEnergy (inline)	0.0	2.2
kiss_fft	0.0	2.0
PerceptualThreshold	0.0	1.1
__sincosf_sse2	1.3	1.3
Quantize	0.5	9.7
QuantizeSingleDwtBand	4.4	6.8
QuantizeDwtBands	0.0	1.0
DwtCdf97	0.0	1.2
Multiple Callers		
VibVectorIntRealloc	4.8	24.7
ReallocNano	3.9	20.0
MallocNano	16.3	16.3
FreeNano	11.8	12.9
VibVectorIntMalloc	1.7	9.7
VibVectorIntFree	0.8	5.4
VibVectorLog10	0.0	2.3
__ieee754_log10f	0.9	2.1
VibVectorDivRight	1.5	1.5
__logf_sse2	1.1	1.1
VibVectorMax	1.1	1.1
VibVectorIntSum	1.0	1.0

Table A.2: Distribution of execution time on x86 with bit budget 64.
70 000 vibrotactile samples were processed by the vibrotactile codec.
gperftools recorded 9574 samples at a sampling frequency of 100 Hz

A.2 Code Listing: generate_dzquantizer_tests.py

```

import struct
import random
import numpy as np          # Needed for NaN

# Adapted from https://stackoverflow.com/a/16444778/14385893
def Binary(num):
    return (''.join('{:0>8b}'.format(c) for c in struct.pack('!f', num)))

def WriteTestcase(fn, delta, max_coeff_quant, x, result):
    with open(fn, 'a') as tc:
        print(f'// $display ("x={x}, \delta={delta}, \backslash'
delta_inverted={1/delta}, \max\_coeff\_quant={max\_coeff\_quant}, \backslash'
result_expected={result}");', file=tc)
        print(f"x_test={32'b{Binary(x)}};", file=tc)
        print(f"delta_test={32'b{Binary(delta)}};", file=tc)
        print(f"delta_inverted_test={32'b{Binary(1/delta)}};", file=tc)
        print(f"max_test={32'b{Binary(max\_coeff\_quant)}};", file=tc)
        print(f"result_test={32'b{Binary(result)}};", file=tc)
        print(f"test_quantize(x_test, \delta_test, \delta_inverted_test, \backslash'
max_test, \result_test, \{x}, \{delta}, \{1/delta}, \backslash'
{max\_coeff\_quant}, \{result}");", file=tc)
        print("", file=tc)

"""Generate testcase as it would be done in the vibrotactile codec
coeff_max should be positive
"""

def WriteTestcaseAutomatic(fn, num_bits, coeff_max, x):
    # Calculate results in 32bit precision to ensure equivalence with
    # hardware accelerator
    num_bits = np.int32(num_bits)
    coeff_max = np.single(coeff_max)
    x = np.single(x)
    max_coeff_quant = np.single(0)
    if (coeff_max >= 1.0):
        max_coeff_quant = MaxQuant(coeff_max, np.int32(3), np.int32(4))
    else:
        max_coeff_quant = MaxQuant(coeff_max, np.int32(0), np.int32(7))
    delta = max_coeff_quant/np.float_power(np.int32(2), num_bits)
    if (x >= 8.0):
        result = np.nan
    else:
        result = DeadzoneQuant(x, max_coeff_quant, num_bits)
    WriteTestcase(fn, delta, max_coeff_quant, x, result)

def DeadzoneQuant(x, max_coeff_quant, num_bits):
    x_abs = abs(x)
    delta = max_coeff_quant/np.float_power(np.int32(2), num_bits)
    if (x_abs >= max_coeff_quant):
        # x_abs = max_coeff_quant * np.single(0.999)
        x_abs = max_coeff_quant * np.single(1023/1024)
    x_div_delta = x_abs / delta

```

```

x_floor = np.floor(x_div_delta)
x_quant = np.sign(x) * x_floor * delta
return x_quant

def MaxQuant(coeff_max, integer_bits, fraction_bits):
    max_coeff_quant = np.float_power(np.int32(2), integer_bits+fraction_bits) - 1
    max_coeff_quant /= np.float_power(np.int32(2), fraction_bits)
    if (coeff_max >= max_coeff_quant):
        # coeff_max = np.sign(coeff_max) * max_coeff_quant * np.single(0.999)
        coeff_max = np.sign(coeff_max) * max_coeff_quant * np.single(1023/1024)
    delta = np.float_power(np.int32(2), -fraction_bits)
    quant_max = np.ceil(np.abs(coeff_max) / delta) * delta
    return quant_max

fn = "quantizer_testcases_long.sv"

random.seed(a=8234709847509234458)

# Initialise file
with open (fn, 'w') as testcase:
    print('//$display("Automatically_generated_test_cases:");', file=testcase)

with open (fn, 'a') as testcase:
    print('//$display("*****Basic_cases*****");', file=testcase)

delta_default = 1.0 # Normal rounding behavior
max_default = 7.9921875 # Maximum possible max_coeff_quant
WriteTestcase(fn, delta_default, max_default, 1.0, 1.0)
WriteTestcase(fn, delta_default, max_default, 0.0, 0.0)
WriteTestcase(fn, delta_default, max_default, 0.5, 0.0)
WriteTestcase(fn, delta_default, max_default, -0.5, -0.0)
WriteTestcase(fn, delta_default, max_default, 7.9921875, 7.0)
WriteTestcase(fn, delta_default, max_default, -7.9921875, -7.0)
WriteTestcase(fn, delta_default, max_default, np.nan, np.nan)
WriteTestcase(fn, delta_default, max_default, 16.0, np.nan)
WriteTestcase(fn, delta_default, max_default, -16.0, -np.nan)
WriteTestcase(fn, delta_default, max_default, np.float_power(2, -19), 0.0)
WriteTestcase(fn, delta_default, max_default, np.float_power(2, -18), 0.0)

BIT_BUDGET = 15
NUM_RAND_ARRAYS = 30

for b in range(0, BIT_BUDGET+1):
    with open (fn, 'a') as testcase:
        print(f'//$display("*****{b}_bits_resolution*****");', file=testcase)
    # Generate random numbers centered around 0 with a log normal distribution
    with open (fn, 'a') as testcase:
        print('//$display("*****Random_cases*****");', file=testcase)
    for i in range(0, NUM_RAND_ARRAYS):
        coeffs = np.random.lognormal(0, 1.5, 32)
        coeffs -= 1

```

```

        coeff_max_rand = max(coeffs)
        for j in range(0, len(coeffs)):
            x_rand = coeffs[j]
            WriteTestcaseAutomatic(fn, b, coeff_max_rand, x_rand)
# Test the edge cases for each bit resolution
with open (fn, 'a') as testcase:
    print(f'//_${display("*****_Edge_Cases_*****")};',
          file=testcase)

# Normal test cases
WriteTestcaseAutomatic(fn, b, 8, 1.0)
WriteTestcaseAutomatic(fn, b, 8, 0.0)
WriteTestcaseAutomatic(fn, b, 8, 0.5)
WriteTestcaseAutomatic(fn, b, 8, -0.5)
WriteTestcaseAutomatic(fn, b, 8, 7.9921875)
WriteTestcaseAutomatic(fn, b, 8, -7.9921875)
WriteTestcaseAutomatic(fn, b, 8, np.float_power(2, -19))
WriteTestcaseAutomatic(fn, b, 8, np.float_power(2, -18))

with open (fn, 'a') as testcase:
    print(f'//_${display("_*****_Test_cases_done_*****")};',
          file=testcase)

# Replace 'nan' with 8.684406692798715e+76 (2**255 * 1.5)
# Adapted from https://stackoverflow.com/a/17141572/14385893
with open (fn, 'r') as file:
    filedata = file.read()

filedata = filedata.replace('nan', '8.684406692798715e+76')

with open (fn, 'w') as file:
    file.write(filedata)

```

This work is licensed under a Creative Commons “Attribution-ShareAlike 4.0 International” license.

