KAVUAKA: A Low-Power Application-Specific Processor Architecture for Digital Hearing Aids

Von der Fakultät für Elektrotechnik und Informatik
der Gottfried Wilhelm Leibniz Universität Hannover
zur Erlangung des akademischen Grades
Doktor-Ingenieur
(abgekürzt: Dr.-Ing.)
genehmigte Dissertation

von Herrn
Dipl.-Ing. Lukas Gerlach

geboren am 12. Januar 1988
in Duisburg, Deutschland

2021

1. Referent: apl. Prof. Dr.-Ing. Guillermo Payá Vayá
2. Referent: Prof. Dr.-Ing. Alberto García-Ortiz

Tag der Promotion: 09.03.2021

Preface

This dissertation was written while I was working as a research associate at the Institute of Microelectronic Systems (IMS) of the Gottfried Wilhelm Leibniz Universität Hannover.

First of all, I would like to thank apl. Prof. Dr.-Ing. Guillermo Payá Vayá and Prof. Dr.-Ing. Holger Blume for kindly providing me with the topic, the professional and scientific support as well as the successful cooperation in various projects around the topic of hearing aid processors. I would also like to thank Prof. Dr.-Ing. Alberto García-Ortiz for his interest in the work and for taking on the role of 2nd examiner.

I would like to thank my colleagues and the students for the friendly cooperation at the institute. For the intensive cooperation I especially thank M. Sc. Christopher Seifert, M. Sc. Florian Giesemann, Dipl.-Ing. Julian Hartig, M. Sc. Tobias Stuckenberg, M. Sc. Fabian Stuckmann, M. Sc. Gia Bao Thieu, M. Sc. Fritz Webering and M. Sc. Moritz Weißbrich. I would like to thank Dipl.-Ing. Marc-Nils Wahalla and M. Sc. Jens Karrenbauer for the hard work of proofreading and helpful discussions.

The biggest thanks go to my parents and brothers. Thank you very much for the longtime support and encouraging words during my time as a research associate. Furthermore, I would like to thank my friends and relatives for their loving and diverse support.

Hannover, March 2021

Lukas Gerlach

Abstract

The power consumption of digital hearing aids is very restricted due to their small physical size and the available hardware resources for signal processing are limited. However, there is a demand for more processing performance to make future hearing aids more useful and smarter. Future hearing aids should be able to detect, localize, and recognize target speakers in complex acoustic environments to further improve the speech intelligibility of the individual hearing aid user. Computationally intensive algorithms are required for this task. To maintain acceptable battery life, the hearing aid processing architecture must be highly optimized for extremely low-power consumption and high processing performance.

The integration of application-specific instruction-set processors (ASIPs) into hearing aids enables a wide range of architectural customizations to meet the stringent power consumption and performance requirements. In this thesis, the application-specific hearing aid processor *KAVUAKA* is presented, which is customized and optimized with state-of-the-art hearing aid algorithms such as speaker localization, noise reduction, beamforming algorithms, and speech recognition. Specialized and application-specific instructions are designed and added to the baseline instruction set architecture (ISA). Among the major contributions are a multiply-accumulate (MAC) unit for real- and complex-valued numbers, architectures for power reduction during register accesses, co-processors and a low-latency audio interface. With the proposed MAC architecture, the *KAVUAKA* processor requires 16 % less cycles for the computation of a 128-point fast Fourier transform (FFT) compared to related programmable digital signal processors. The power consumption during register file accesses is decreased by 6 % to 17 % with isolation and by-pass techniques. The hardware-induced audio latency is 34 % lower compared to related audio interfaces for frame size of 64 samples.

The final hearing aid system-on-chip (SoC) with four *KAVUAKA* processor cores and ten co-processors is integrated as an application-specific integrated circuit (ASIC) using a 40 nm low-power technology. The die size is 3.6 mm^2. Each of the processors and co-processors contains individual customizations and hardware features with a varying datapath width between 24-bit to 64-bit. The core area of the 64-bit processor configuration is 0.134 mm^2. The processors are organized in two clusters that share memory, an audio interface, co-processors and serial interfaces. The average power consumption at a clock speed of 10 MHz is 2.4 mW for SoC and 0.6 mW for the 64-bit processor.

Case studies with four reference hearing aid algorithms are used to present and evaluate the proposed hardware architectures and optimizations. The program code for each processor and co-processor is generated and optimized with evolutionary algorithms for operation merging, instruction scheduling and register allocation. The *KAVUAKA* processor architecture is compared to related processor architectures in terms of processing performance, average power consumption, and silicon area requirements.

Key words: hearing aid, processor, ASIP, ASIC, low-power, system-on-chip

Kurzfassung

Die Leistungsaufnahme von digitalen Hörgeräten ist aufgrund ihrer kleinen physikalischen Bauformen sehr begrenzt und die Hardware-Ressourcen für die Signalverarbeitung sind limitiert. Es besteht jedoch ein Bedarf an mehr Rechenleistung, um zukünftige Hörgeräte nutzbringender und intelligenter zu machen. Zukünftige Hörgeräte sollen in der Lage sein, Sprecher und Sprecherinnen in komplexen akustischen Umgebungen zu orten, zu lokalisieren und zu erkennen, um das Sprachverstehen des einzelnen Hörgeräteträger und Hörgeräteträgerinnen weiter zu verbessern. Für diese Aufgabe sind rechnerisch komplexe Algorithmen erforderlich. Um eine akzeptable Batterielebensdauer zu erreichen, muss die Architektur der Hörgerätesignalverarbeitung für extrem niedrigen Stromverbrauch und hohe Rechenleistung optimiert sein.

Die Integration eines Prozessors mit anwendungsspezifischem Befehlssatz (ASIP) in Hörgeräten ermöglicht eine Vielzahl von Anpassungsmöglichkeiten der Architektur, um die strengen Anforderungen an Stromverbrauch und Leistung zu erfüllen. In dieser Arbeit wird der anwendungsspezifische Hörgeräteprozessor *KAVUAKA* vorgestellt, der mit Referenz-Hörgerätealgorithmen wie Sprecherlokalisierung, Rauschunterdrückung, Beamformer und Spracherkennung angepasst und optimiert wird. Spezialisierte und anwendungsspezifische Instruktionen werden entworfen und dem Basisbefehlssatz hinzugefügt. Zu den wichtigsten Anpassungen gehören eine MAC-Einheit für reelle und komplexwertige Datentypen, Architekturen zur Verlustleistungsreduzierung bei Registerzugriffen, Co-Prozessoren und eine Audioschnittstelle mit niedriger Audio-Latenz. Mit der vorgeschlagenen MAC-Architektur benötigt der *KAVUAKA* Prozessor 16 % weniger Zyklen für die Berechnung einer 128-Punkt schnellen Fourier-Transformation im Vergleich zu verwandten programmierbaren digitalen Signalprozessoren. Der Stromverbrauch bei Registerdateizugriffen wird durch Isolations- und Bypass-Techniken um 6 % bis 17 % verringert. Die hardwareseitige Audiolatenz ist 34 % niedriger im Vergleich zu verwandten Audioschnittstellen bei einer Fenstergröße von 64 Samples.

Das finale Hörgeräte System-on-chip (SoC) mit vier *KAVUAKA* Prozessorkernen und zehn Co-Prozessoren ist als ASIC unter Verwendung einer 40 nm Low-Power-Technologie integriert worden. Die Chip-Größe beträgt 3.6 mm^2. Jeder dieser Prozessoren und Co-Prozessoren enthält individuelle Anpassungen, Hardware-Features und die Datenpfadbreite variiert zwischen 24-bit to 64-bit. Die Fläche der 64-bit-Prozessorkonfiguration beträgt 0.134 mm^2. Die Prozessoren sind in zwei Clustern organisiert, die sich Speicher, eine Audioschnittstelle, Co-Prozessoren und serielle Schnittstellen teilen. Die durchschnittliche Leistungsaufnahme bei einer Taktfrequenz von 10 MHz beträgt 2.4 mW für das SoC und 0.6 mW für den 64-bit Prozessor.

Fallstudien mit vier Referenz-Hörgerätealgorithmen werden verwendet, um die vorgeschlagenen Hardware-Architekturen und Optimierungen herauszustellen und zu bewerten. Der Programmcode für jeden Prozessor und Co-Prozessor wird mit evolutionären Algorithmen für die

Zusammenführung von Operationen, den Befehlsablauf und die Registerzuweisung generiert und optimiert. Die Prozessorarchitektur von *KAVUAKA* wird mit verwandten Prozessorarchitekturen im Hinblick auf die Rechenleistung, den durchschnittlichen Stromverbrauch und den Siliziumflächenbedarf verglichen.

Schlagworte: Hörgerät, Prozessor, ASIP, ASIC, stromsparend, System-on-Chip

Contents

Preface iii
Abstract iv
Kurzfassung v
Contents viii
Abbreviations xiii

1 Introduction 1
1.1 Objectives 2
1.2 Overview 4

2 State-of-the-Art: Application-Specific Processor Architectures for Hearing Aids 5
2.1 State-of-the-Art Algorithms for Hearing Aid Devices 7
2.2 State-of-the-Art Hearing Aid Processor Architectures 8
2.2.1 Hard-Wired Architectures 11
2.2.2 Application-Specific Instruction-Set Processors 11
2.2.3 ASIPs with Hardware Accelerators 12

3 The KAVUAKA Hearing Aid Processor 15
3.1 Baseline Processor Architecture 15
3.1.1 Related ASIP Architectures 16
3.1.2 Baseline Instruction-Set Architecture 17
3.2 Specialization Towards Performance 22
3.2.1 Real- and Complex-Valued Multiply-Accumulate Functional Unit . . 23
3.2.2 Efficient Emulation of Floating-Point Arithmetic 36
3.2.3 Tightly Coupled Co-Processors 50
3.2.4 Operation Merging Extensions 61
3.2.5 Issue-Slot Based Predication Encoding Technique 65
3.3 Specialization Towards Low-Power 78
3.3.1 Configurable Datapath Width 79
3.3.2 Dummy Register and Register Address Isolation 89
3.3.3 Low-Level Low-Power Optimization Techniques 95
3.4 Low-Power Interfaces and Connectivity 103
3.4.1 A Low Latency Multichannel Audio Interface 103

3.4.2 A Serial Interface with Special DMA Capabilities 117
3.5 Hearing Aid System-on-Chip ASIC 120

4 Operation Merging, Instruction Scheduling and Register Allocation 129
4.1 Operation Merging 131
4.2 Instruction Scheduling 133
4.2.1 Issue-Slot Based Predication Register Allocation 135
4.2.2 Towards Power-Aware Instruction Scheduling 137
4.3 Register Allocation 138
4.3.1 Power-Aware Register Allocation 141

5 Evaluation and Design Space Exploration 147
5.1 Case Study: Beamforming Algorithms 147
5.2 Case Study: Speech Enhancement 162
5.3 Case Study: Speaker Localization 166
5.4 Case Study: Speech Recognition 175
5.5 Comparison to Other Related Hearing Aids 179
5.5.1 ASIC Technology and Supply Voltage 179
5.5.2 Power Consumption 182
5.5.3 Circuit Area 183
5.5.4 Operating Clock Frequency 184
5.5.5 Audio Datapath Width 185
5.5.6 On-Chip Memory 185

6 Conclusion 189

References 193

List of the Author’s Publications 217

Abbreviations

ADC analog-to-digital converter. 6, 79, 86, 98, 103, 107–109, 122, 124, 125, 167, 180, 183, 185

ADM adaptive directional microphone. 7, 9

AFB analysis filter bank. 7, 8, 11, 13

AFE analog front end. 12, 16, 103, 179–184

AGC automatic gain control. 183

ALU arithmetic logic unit. 18, 61, 80, 166

ASIC application-specific integrated circuit. iv–vi, 3, 4, 8, 11, 12, 16, 21, 36, 39, 57, 59, 60, 68, 72, 81, 83, 90, 96–100, 117, 120, 122, 124–127, 137, 143, 152, 158, 175, 179–181, 185, 191

ASIP application-specific instruction-set processor. iv, vi, 2–5, 7, 10–12, 15–17, 97, 104, 137, 141, 148, 157, 160, 179–186, 189–191

BCLK bit clock. 109, 125

BMF beamforming. 7, 9

CFU custom function unit. 25

CFX coolflux dsp. 17, 35

CLZ count leading zeros. 38

CMAC complex-valued multiply-accumulate. 26, 27, 29, 30, 32, 34–36, 96, 122, 172, 173

CMOS complementary metal-oxide-semiconductor. 16, 179, 181

CMUL complex-valued multiply. 27

CNN convolutional neural networks. 12, 13, 187

CORDIC coordinate rotation digital computer. 3, 25, 50–60, 78, 101, 102, 119–121, 124, 157, 158, 160, 161, 164–167, 173, 174, 176, 178

CP co-processor. iv, 2, 3, 12, 15, 18, 20, 22, 50–53, 55–60, 78, 120–122, 124–126, 157, 158, 160, 161, 164–167, 170, 173, 174, 178, 190, 191

CPU central processing unit. 50

CR condition read. 135, 136

CS condition set. 135, 136

DAC digital-to-analog converter. 6, 12, 103, 107–109, 122, 124, 125, 180, 185

DCU division co-processor unit. 22, 23

DDG data dependency graph. 133–136

DE instruction decode. 17, 20, 61, 97, 99, 100

DFT discrete fourier transform. 34

DIT decimation in time. 34

DLP data-level parallelism. 79, 80

DMA direct memory access. 18, 104–107, 114–120, 122, 124, 190

DMEM data memory. 20, 66, 101, 119

DNN deep neural network. 87, 101, 102, 175–179, 191

DRC dynamic range compression. 7, 9

DSP digital signal processor. iv, 2, 11–13, 16, 17, 23–25, 34, 38, 39, 44, 56, 68, 92, 104, 106, 107, 110, 117, 118, 157, 160, 166, 182, 187, 189, 190

EA evolutionary algorithm. 3, 130–135, 137, 138, 140, 141, 190

EX execution. 17, 20, 92, 97–100

FBC feedback cancellation. 7–9, 12, 13

FFT fast Fourier transform. iv, 12, 13, 16, 24–26, 28, 32, 34, 35, 46, 48, 49, 62, 64, 75, 84, 86–88, 101, 102, 123, 163, 165, 170, 172–176, 187, 189

FIFO first in first out. 105, 106, 109–118, 122, 124, 190

FIR finite impulse response. 12, 13, 16, 24, 25, 44, 46, 47, 72, 74, 84, 86–89, 102, 150, 182

FIREG file indirect register. 17, 18, 62, 65, 85, 101, 179

FLIX flexible length instruction extensions. 16

FPGA field programmable gate array. 125, 127

FU functional unit. 15, 17, 18, 20, 22–25, 29, 40, 61, 71, 72, 79, 91, 92, 95–97, 121, 129, 172, 173, 185

GMM gaussian mixture model. 166, 167, 169, 170, 174, 185

GSC generalized sidelobe canceller. 84, 87, 89, 90, 102

GTFB gammatone filter bank. 167, 168, 170–173

HDL hardware description language. 15

HFPM hybrid floating-point hardware modules. 39

HVT high threshold voltage. 21, 36, 57, 59, 60, 81, 83, 90, 96, 97, 99, 100, 120, 143, 158, 161, 175

HW hardware. iv, 2–5, 7, 11–13, 17, 22–24, 26, 30, 32, 34, 36–40, 43, 50, 51, 53–55, 61, 62, 66, 68, 78–80, 93, 101–105, 109, 112, 114, 120, 129, 137, 147, 148, 151, 156–158, 161, 163, 166, 169–171, 175, 176, 178, 179, 184, 185, 189, 191

I/O input/output. 6, 103, 117, 124–126

I^{2}C inter-integrated circuit. 6, 117–120, 124

I^{2}S inter-IC sound. 103, 107, 109, 124

IC integrated circuit. 107, 117, 124

IEEE institute of electrical and electronics engineers. 38–41, 44, 47–49

IF instruction fetch. 17, 97, 99, 100

IFFT inverse fast Fourier transformation. 102, 123, 163, 165

IIR infinite impulse response. 46–48, 72–74, 84, 87–89, 102, 123, 171, 182, 187

ILD interaural level difference. 166–170, 174

ILP instruction level parallelism. 66, 70, 75

IMEM instruction memory. 118–120

IPC instructions per cycle. 3, 17, 46–49, 53, 61, 73, 75, 80, 84, 156, 157, 164, 173, 178, 190

ISA instruction set architecture. iv, 16, 67, 68, 70, 129, 137, 160, 174, 178

ISNR intelligibility-weighted signal-to-noise ratio. 151–154, 157, 159, 190

ISR interrupt service routine. 104, 118

ITD interaural time difference. 166–170, 174

LMS least mean squares. 150

LRCLK left-right clock. 109, 125

LSB least significant bit. 62, 63, 66, 86

LUT lookup table. 25, 50, 51, 120, 174

MAC multiply-accumulate. iv, v, 12, 13, 17, 18, 23–28, 30, 32, 34–36, 38, 61, 62, 98, 120–122, 160, 161, 166, 171, 174, 176–179, 185, 189

MI micro-instruction. 17, 129, 132–135

MIMD multiple instruction, multiple data. 80

MO micro-operation. 17, 19, 61, 129, 131–135, 138, 164, 165

MSB most significant bit. 86

MUL multiply. 18, 26, 27, 29, 30, 36, 177, 178

MV move. 61, 62, 65, 67, 178, 179

NOP no operation. 20, 66, 75, 101, 112, 113, 124, 138, 139

NR noise reduction. 7–9, 11–13, 54, 55, 60

NT neural transduction. 170

OLSA Oldenburger Satztest. 151

PCB printed circuit board. 125, 127

PDM pulse density modulation. 107, 183

PESQ perceptual evaluation of speech quality. 3, 151–154, 157, 159

PGA programmable gain amplifier. 12, 180, 183

QFN quad flat no leads package. 125

RA register access. 17, 20, 97

RAM random-access memory. 187

REGA register allocation. 135, 140

RF register file. iv, 15–18, 20, 21, 25, 30, 61–63, 68, 69, 72, 80, 89–94, 97, 98, 129, 138, 140, 141, 144, 145, 156–158, 160, 161, 174, 175, 190

RISC reduced instruction-set computer. 16, 24, 38, 44, 160

ROM read-only memory. 12, 187

RTL register-transfer level. 112

SCL serial clock. 125

SDA serial data. 125

SDATA serial data. 125

SE speech enhancement. 9, 185

SFB synthesis filter bank. 7, 8, 11

SIMD single instruction, multiple data. 3, 15–18, 20, 22–30, 32–37, 39–46, 51, 53–55, 57, 59–61, 70–73, 79–86, 88, 102, 104–109, 111–113, 122, 124, 137, 156, 157, 160, 170, 173, 174, 177–179, 185, 189, 190

SLM straight line microcode. 66, 91, 129, 131–135, 137, 140, 164, 165

SNR signal-to-noise ratio. 3, 7, 9, 147, 163, 182, 185

SoC system-on-chip. iv, v, 3, 15–17, 101, 120, 122, 124–126, 137, 138, 141, 143, 180, 183, 191

SP single precision floating-point. 56

SR speech recognition. 9, 185

SRAM static random-access memory. 12, 17, 65, 66, 68, 83, 118, 120, 122, 124–126, 185, 187

SSL sound source localization. 9, 191

STOI short-time objective intelligibility. 147, 151–154, 157, 159, 185

SW software. 54, 71, 101, 102, 158, 164, 167, 173

TDM time-division multiplexing. 107, 109

TSMC taiwan semiconductor manufacturing company. 16, 17, 21, 36, 57, 59, 60, 72, 81, 83, 90, 96, 97, 99, 100, 120, 143, 158, 161, 175

TTA transport-triggered architecture. 175, 191

VAD voice activity detector. 9

VLIW very long instruction word. 3, 15–18, 22, 24, 34, 38, 39, 43, 44, 53, 55, 61, 65, 68–70, 91, 92, 104, 123, 129, 160, 175, 178, 190

VU vector unit. 17, 18, 20, 120

WB write back. 17, 20, 97

WDRC wide dynamic range compression. 11, 13

X2 operation merging. 17, 18, 20, 21, 61–66, 72, 92, 96, 120, 122, 129–132, 140, 164, 166

XML extensible markup language. 129

1 Introduction

According to the World Health Organization [1], approximately 466 million people worldwide, about 5 % of the world's population, suffer from disabling hearing loss. It is estimated that this number will rise to over 900 million people by 2050. An even larger group of people (15 %) is affected by mild or moderate hearing loss [2]. There are many causes of hearing loss. These include genetic causes, diseases, drugs, infections, injuries, excessive noise exposure or aging. The consequences of hearing loss are manifold. Major consequences are reduced communication skills and related social and emotional burdens, disorders in the speech development of children, and economic impacts, especially for developing countries [3].

Certain effects and consequences of a hearing loss can be reduced with modern hearing aids by improving general hearing and speech intelligibility through digital signal processing. Although a large number of hearing impaired people would benefit from the use of hearing aids, only about 10 % of the global need is covered by current manufacturing and service supply [3]. New processors and algorithms are essential to make current hearing devices more affordable and to increase their benefits. However, the small physical size of hearing aids and battery operation limit the digital and advanced signal processing. Due to new findings from research and continuous technological developments, the signal processors of digital hearing aid systems are becoming increasingly powerful, smaller and more energy efficient [4]. This enables the use of novel and computationally intensive signal processing algorithms that improve speech intelligibility especially in complex acoustic scenarios, like in a cafeteria. For example, future hearing aids will be able to recognize speaking people in the environment and filter out the relevant speakers from the background noise.

Future and present hearing aid processors must meet various requirements:

- **Low-power consumption** for an acceptable battery life.
- **High processing performance** for computationally intensive and advanced algorithms.
- **Small physical size** for user preferences and physical design requirements.
- **High programming flexibility** to achieve short innovation cycles and to reduce manufacturing costs.
- **Low audio delay** for an acceptable synchronicity of audio-visual perception.
- **etc.**

The power consumption of hearing aids is limited due to small battery sizes [5]. Despite advances in semiconductor technology, the power consumption of the hearing aid's signal processing system is crucial, as there is the trend for more computationally intensive algorithms [5]. A challenging task for future hearing aids will be binaural signal processing, e.g., detect, localize and identify target speakers in complex acoustic environments [6]. Another example of an advanced algorithm is given in [7], where deep-learning techniques are used for noise reduction. These algorithms require more processing performance compared to existing algorithms, but may increase the speech intelligibility of the individual hearing aid user. In order to make these algorithms applicable and to maintain an acceptable battery life, the hearing aid processing architecture has to be optimized and customized for low-power consumption and high processing performance.

Different processing architectures are feasible for hearing aids. Dedicated hardware solutions have advantages in terms of energy and area efficiency, although they offer less flexibility compared to programmable architectures. Digital signal processors (DSPs) are programmable, but it is essential to reduce their power consumption by hardware customizations and optimizations. The integration of application-specific instruction-set processors (ASIPs) in hearing aid devices is a current research topic [7–12]. Their advantages are high flexibility and programmability. However, compared to dedicated or analog hearing aids [13–15], their power consumption is still too high. To compensate for this drawback, new customizations on the hardware and software side are required in order to meet the stringent power consumption constraints and processing performance requirements. In many cases, these ASIP architectures are optimized on different design levels and are enhanced with dedicated hardware accelerators. This combination offers a good trade-off between flexibility through programmability and efficiency in terms of processing power per watt. However, the design space is large and the exploration of the architectures and their evaluation based on hearing aid algorithms is challenging.

1.1 Objectives

The main objective of this thesis is a novel hardware architecture for low-power hearing aids, which is called *KAVUAKA* [1]. Secondary objectives are depicted in Figure 1.1. The general focus lies in the architecture of the programmable application-specific instruction-set processor (ASIP) and the co-processors. The processor architecture is based on the *Moai4k2* architecture, a scalable and configurable ASIP architecture for multimedia applications [16]. In this thesis, it is customized and optimized for hearing aid applications. To enable efficient use of the application-specific hardware architecture, new software techniques for optimization and

---

[1] Partial results of this thesis were presented in previous publications, which are included in the List of the Author's Publications.

software frameworks for evaluation are developed. The goal is to improve the overall efficiency of hearing aid processing in terms of processing performance per watt and silicon area. The efficiency of the *KAVUAKA* architecture is compared to related research and commercial architectures.

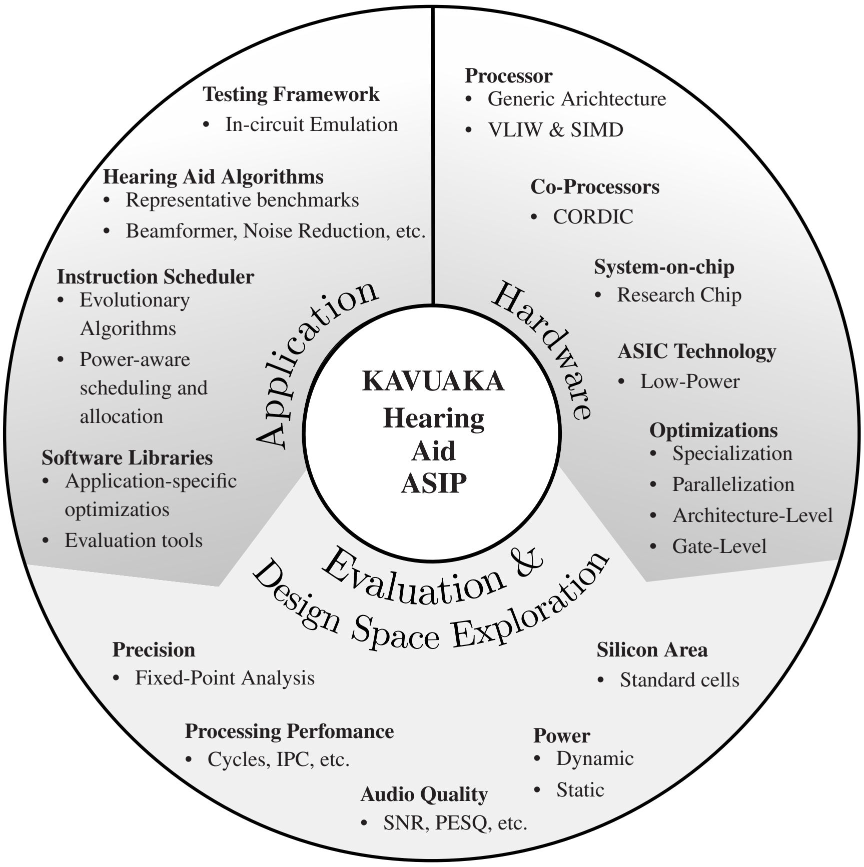


Figure 1.1: Objectives of this thesis.

Due to the ASIP-based hearing aid processor design, optimized software libraries are essential for the efficient use of available and custom hardware resources. Therefore, customized instruction scheduler techniques are developed. The extended instruction scheduler will use evolutionary algorithms and hardware-related information for instruction scheduling, operation merging, and register allocation. It will support new hardware features such as issue-slot based predication and bypass register techniques for a power-aware instruction scheduling and register allocation. In addition, optimization techniques for fixed-point and emulated floating-point arithmetic software libraries for VLIW-SIMD processors and co-processors will be developed.

Another important part of this work is a design space exploration of the developed hardware and software components. Therefore, the hardware will be synthesized and manufactured using application-specific integrated circuit (ASIC) technology. The hardware customizations will then be evaluated based on the silicon area requirements. Power consumption will be accurately predicted by hardware simulations and physical measurements. Fixed-point software models of the hearing aid algorithms will be used to evaluate the processing performance and precision of the algorithms.

1.2 Overview

In Chapter 2 the state-of-the-art application-specific instruction-set processor (ASIP) architectures for hearing aids are introduced. Subsequently, the proposed hearing aid processor architecture called *KAVUAKA* is described in detail in Chapter 3. The instruction scheduler extensions for operation merging, instruction scheduling and register allocation are described in Chapter 4. The evaluation and design space explorations are presented in Chapter 5. This thesis is concluded with Chapter 6.

2 State-of-the-Art: Application-Specific Processor Architectures for Hearing Aids

Modern hearing aids, as shown in Figure 2.1, must meet a variety of technical requirements. First of all, the power consumption of hearing aids is limited. To achieve an acceptable battery life, the average power consumption of hearing aids needs to be in the range of a few milliwatts. The reason for the low energy budget is the small physical size of battery-powered hearing aids. At the same time, the demand for more audio processing performance and memory capacity is steadily growing. There are newly developed algorithms with more and improved features and increasing demands for audio quality. In addition, it is essential that the hearing aids can be individually fitted by the audiologist or the hearing aid user himself, adapt to constantly changing environmental conditions and connect wirelessly to other electronic devices. These requirements, i.e., low-power consumption and a high degree of flexibility and programmability, imply extensive design space explorations and trade-offs between the optimization goals (Figure 2.3).

Figure 2.2 shows the system components and peripherals of a state-of-the-art hearing aid. Typically, hearing aids contain a central processing unit that provides the functionality and connects all other components such as the receiver and microphones. The design and implementation of the processor is demanding and involves numerous trade-offs due to the wide range of requirements and design space. The implementation alternatives in a multidimensional design space are shown in Figure 2.3. Different hearing aid processor architectures and implementations have been presented in the literature. There are analog, mixed-signal or purely digital hearing aid signal processing architectures. Some use hard-wired processing and control circuits, others use fully programmable application-specific instruction-set processors (ASIPs) with custom instructions and hardware accelerators.

A study [22] published in 2016 presents state-of-the-art signal processing techniques in hearing aids. Among the studied algorithms are feedback reduction, directional microphones, noise reduction, and environment recognition. Current limitations and future trends, like binaural and music processing, are highlighted. Binaural processing as a future hearing aid processing technique is also addressed in the two surveys [5, 23] from 2005 and 2009. Both papers present the state-of-the-art, challenges and future trends of signal processing in hear-

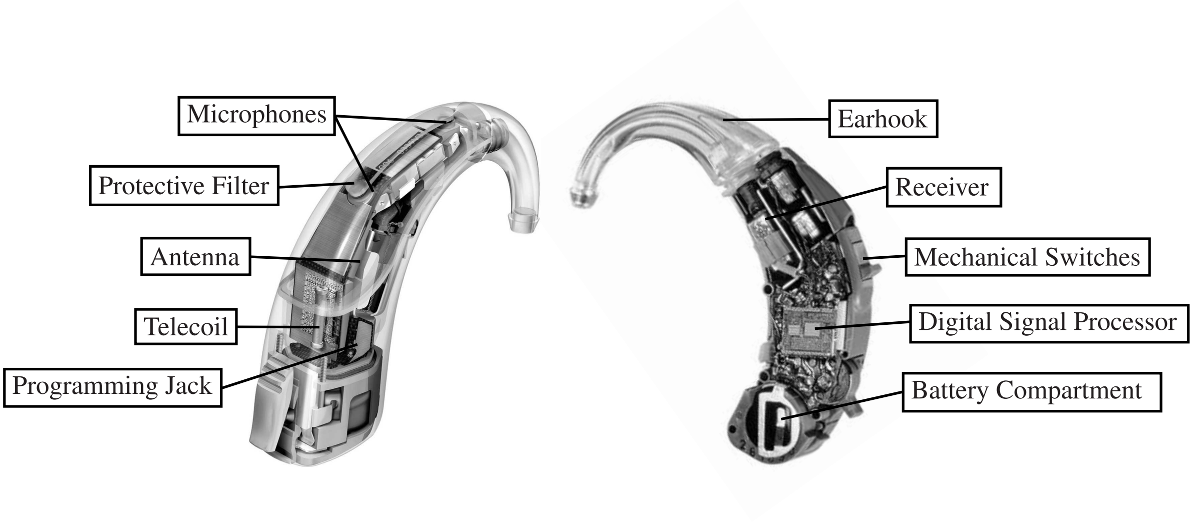


Figure 2.1: Schematic view and photograph of a behind-the-ear (BTE) hearing aid [17, 18].

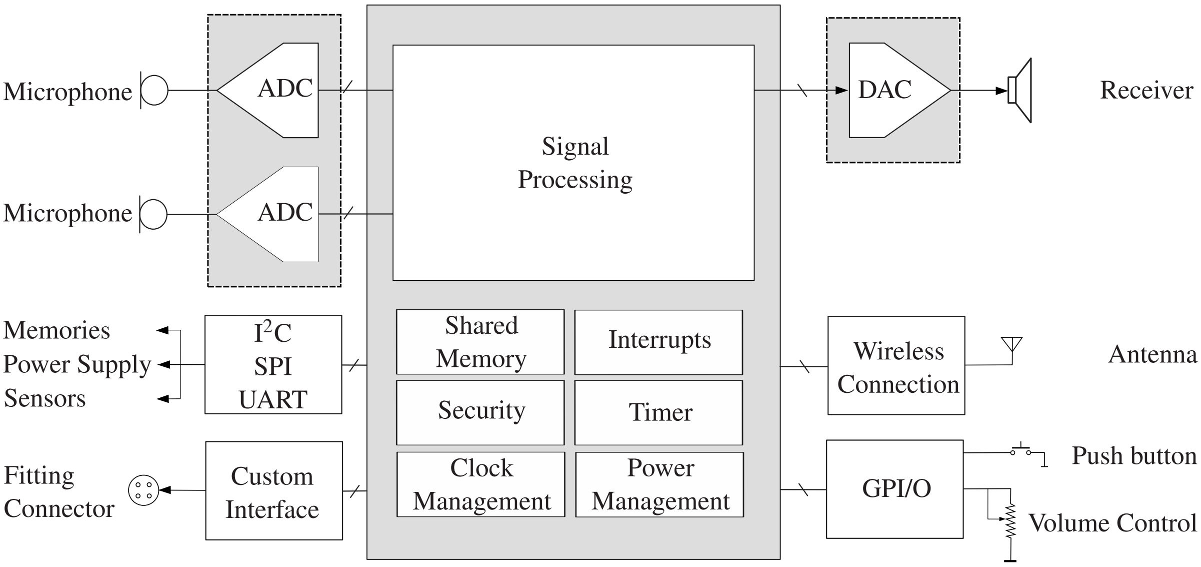


Figure 2.2: System components and peripherals of a state-of-the-art hearing aid [19, 20].

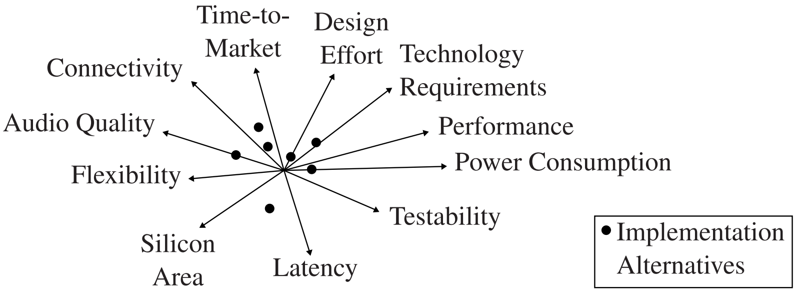


Figure 2.3: Implementation alternatives in a multi-dimensional design space [21].

ing aids. Physiological requirements due to hearing impairment are described as well as the different audio processing methods such as directional microphones, noise reduction, acoustic feedback suppression, classification, and compression. No related work focuses on the hardware perspective. Only little information is given about hardware architectures and circuit implementation and the different design methods. This chapter gives an overview of the current processor architectures. A particular focus is on application-specific instruction-set processors (ASIPs).

This chapter is structured as follows: Section 2.1 contains a list of algorithms implemented on the hearing aid processors that are part of this overview. The hearing aid processors are described in detail in Section 2.2. The differences between the architectures of hard-wired, ASIP-based and heterogeneous (ASIP-based with accelerators) hearing aid processors are discussed.

2.1 State-of-the-Art Algorithms for Hearing Aid Devices

A typical high-end hearing aid processing is shown as a block diagram in Figure 2.4. Multiple microphones enable directional filtering. Therefore, beamforming (BMF) and adaptive directional microphone (ADM) algorithms are the first in the chain and aim to increase the signal-to-noise ratio (SNR) by performing directional filtering. Feedback is then suppressed with a feedback cancellation (FBC) algorithm by analyzing the output signal and detecting feedback loops. The algorithms that process frequency domain data, such as the noise reduction (NR) and dynamic range compression (DRC) algorithms, require an analysis filter bank (AFB) and a synthesis filter bank (SFB). Classification algorithms generally generate control signals for the processing chain. A list of typical algorithms is included in Table 2.1. This list contains exclusively algorithms that are part of a processing chain in state-of-the-art hearing aid processors from literature and commercial products. Publications with the implementation, optimization and application of these algorithms are included in Table 2.1. There is a trend towards algorithms, that are computationally more demanding. In the last years, algorithms for machine learning and deep learning [24–26] and binaural processing algorithms [27] have been used. Recently proposed algorithms of this type [28–30], that have not yet been implemented on a hearing aid, are not listed in Table 2.1.

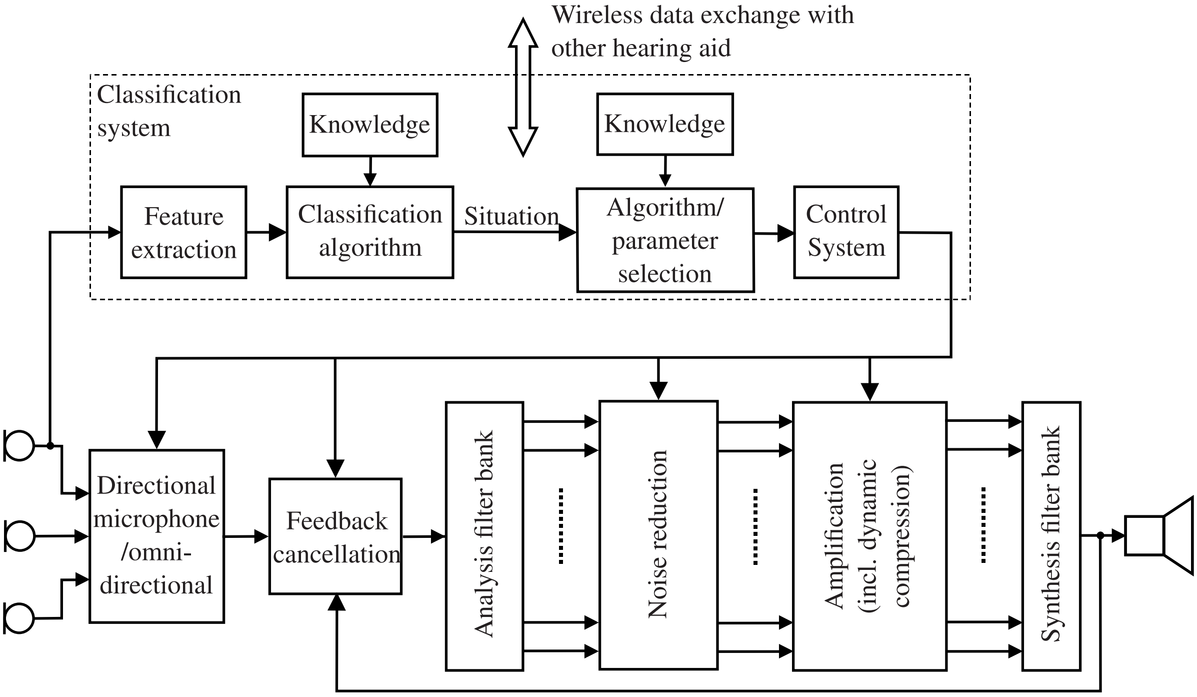


Figure 2.4: Simplified block diagram of the signal processing chain of state-of-the-art high-end digital hearing aids [5, 31].

2.2 State-of-the-Art Hearing Aid Processor Architectures

In the literature, various hearing aid processors have been proposed over the last decades. All hearing aid processors are subject to similarly stringent requirements with respect to limited energy budget, available chip area, and performance requirements. However, a wide range of different architectures, algorithms, approaches, and technologies have been introduced and used to meet these stringent requirements. 30 research and commercial processors published between 1996 and 2020 are listed in Table 2.2. This table provides a comparison of the architecture, ASIC technology, supply voltage, average power consumption, silicon area, and operating clock frequency of the various hearing aid systems.

The processor architectures are designed and optimized to efficiently execute particular hearing aid algorithms listed in Table 2.1. The architectures of these processors can be grouped

Table 2.1: List of hearing aid algorithms implemented on related hearing aid processors.

Year	Publication	Class	Application in a hearing aid processor
1982	[32]	Beamforming (BMF)	[8, 33]
1984	[34]	Speech enhancement (SE)	[11]
1985	[35]	Noise reduction (NR)	[19]
1995	[36]	Noise reduction (NR)	[8]
1996	[37]	Speech enhancement (SE)	[8]
1997	[38]	Voice activity detector (VAD)	[39]
1997	[40]	Feedback cancellation (FBC)	[41]
1999	[42]	Feedback cancellation (FBC)	[19, 43]
2001	[44]	Adaptive directional microphone (ADM)	[45]
2001	[46]	Digital filter	[13, 15, 39, 47–49]
2001	[50]	Noise reduction (NR)	[11]
2002	[51]	Adaptive SNR Monitor	[49]
2002	[52]	Voice activity detector (VAD)	[53]
2002	[54]	Noise reduction (NR)	[19, 41, 43]
2002	[55]	Beamforming (BMF)	[56]
2005	[57]	Dynamic range compression (DRC)	[8]
2005	[58]	Speech enhancement (SE)	[56]
2006	[59]	Noise reduction (NR)	[19, 43]
2007	[60]	Noise reduction (NR)	[53]
2008	[61]	Dynamic range compression (DRC)	[62]
2008	[63]	Feedback cancellation (FBC)	[11]
2008	[63]	Dynamic range compression (DRC)	[11, 19, 41, 43]
2008	[63]	Adaptive directional microphone (ADM)	[11]
2011	[27]	Sound source localization (SSL)	[56]
2013	[64]	Speech enhancement (SE)	[33]
2013	[65]	Feedback cancellation (FBC)	[8]
2016	[26]	Speech enhancement (SE)	[7, 12]
2017	[66]	Speech enhancement (SE)	[25]
2019	[24]	Speech recognition (SR)	[56]

Table 2.2: Related hearing aid processors and systems.

Year	Publication	Proc. Arch.	Analog front end	Tech. [nm]	Supply [V]	Power [mW]	Area [mm^2]	Clock [MHz]
1996	[67]	hard-wired	yes	1200	1.30	1.300	28.00	0.200
1997	[68, 69]	ASIP	yes	800	1.30	1.950	35.00	1.024
1999	[70]	hard-wired	yes	1200	1.40	0.100	1.10	0.640
1999	[71]	ASIP	no	500	1.00	0.800	28.00	2.000
2000	[72]	ASIP	no [73]	250	1.05	0.660	20.00	2.500
2001	[74]	ASIP+accelerator	no	250	1.00	0.011	5.00	0.192
2003	[13]	hard-wired	yes	600	1.10	0.290	12.00	2.560
2004	[14]	hard-wired	yes	120	1.00	0.300	7.50	0.150
2005	[45]	hard-wired	no	250	1.25	0.045	0.67	—
2006	[49]	hard-wired	no	180	0.90	0.010	0.30	0.064
2006	[47]	hard-wired	yes	180	0.90	0.096	2.70	0.032
2007	[48]	hard-wired	yes	180	0.90	0.096	3.08	0.032
2008	[15]	hard-wired	yes	180	0.90	0.107	3.74	0.032
2010	[62]	hard-wired	no	90	0.60	1.095	3.10	8.000
2011	[8]	ASIP	no	65	0.80	0.964	0.49	11.000
2012	[9]	ASIP+accelerator	no	65	0.80	1.300	3.60	10.000
2013	[39]	hard-wired	no	180	0.90	0.025	0.50	0.032
2013	[75]	ASIP	no	130	1.00	0.863	—	8.000
2014	[53]	hard-wired	no	90	0.60	0.083	1.53	6.000
2014	[76]	hard-wired	yes	130	1.00	1.600	5.24	—
2014	[43]	ASIP+accelerator	yes	130	1.00	0.860	9.50	8.000
2015	[19]	ASIP+accelerator	yes	130	1.00	1.200	9.50	8.000
2016	[10]	ASIP+accelerator	yes	130	1.00	1.100	9.30	8.000
2018	[20]	ASIP+accelerator	yes	65	1.18	0.870	—	10.240
2018	[11]	ASIP+accelerator	yes	130	1.00	1.100	9.30	2.000
2018	[12]	ASIP+accelerator	no [7]	28	0.55	4.000	9.00	50.000
2019	[41]	ASIP+accelerator	no	65	1.00	1.300	2.71	8.000
2019	[56]	ASIP+accelerator	no	40	1.10	0.600	3.60	10.000
2020	[25]	ASIP+accelerator	no	40	0.60	2.170	4.20	20.000
2020	[33]	ASIP+accelerator	no	40	0.70	1.500	0.30	10.500

into three main classes: hard-wired with dedicated processing blocks, ASIPs, and ASIPs with hardware accelerators.

2.2.1 Hard-Wired Architectures

In case of a hard-wired hearing aid processing architecture, all components of the hearing aid processing chain are implemented by dedicated circuits. Their basic function is fixed and can only be changed before manufacturing. The algorithm parameters or control signals can be changed while the hearing aid is in operation. There are pure analog [14, 70, 76], mixed signal [13,15,47,48,67], or pure digital [39,45,49,53,62] hardwired hearing aid signal processing architectures.

A digital hardwired hearing aid [53] is highlighted in the following. This architecture, which is originally proposed in 2014, is remarkable for the flexibility that the dedicated architecture offers compared to related architectures. It comprises a core-based architecture consisting of a memory management unit for data exchange, a control unit and an arithmetic unit for processing. Therefore, the processing is easier to control compared to other architectures. The authors of [53] propose a sample-based perceptual multiband noise reduction algorithm for the design of a digital hearing aid architecture. This noise reduction algorithm is part of the hearing processing chain. An analysis and synthesis filter bank (AFB and SFB), noise reduction (NR) [52, 60], insertion gain (IG) and wide dynamic range compression (WDRC) are integrated on the hearing aid chip with an average power consumption of 83.7 µW. A 90 nm ASIC technology is used and the digital core voltage is 0.6 V.

2.2.2 Application-Specific Instruction-Set Processors

Application-specific instruction-set processor (ASIP) architectures include a digital signal processor (DSP) for signal processing [8,68,69,71,72,75]. The DSP architecture is optimized for the typical hearing aid algorithms, therefore it is here also denoted as an application-specific instruction-set processors (ASIPs). The target algorithms can be replaced by changing the program code. This offers greater flexibility compared to hard-wired architectures. However, due to the more flexible processor architecture and memory requirements, the power consumption and silicon area requirements are higher than for hard-wired architectures. Instruction-level and data-level optimizations improve the efficiency of signal processing. New custom instructions increase performance.

A hearing aid with a DSP for signal processing is presented in [72]. This hearing aid publication is highlighted as the authors propose a silicon flow algorithm in addition to the proposed hearing aid chip. This flow is integrated into the chip design flow and supports accurate and fast simulation, ASIC synthesis, optimization and verification [72]. These tools are useful for managing overall complexity and reduce design time, if the underlying ASIC technology is changed. The DSP architecture consists of a datapath with several general purpose execution units, a complex-valued multiplier, and a controller with a program read-only memory. The clock frequency of the DSP is reduced by increased parallelism and reduced memory accesses. A fast Fourier transform (FFT) algorithm case study for the architecture shows how a radix-8 implementation can minimize memory accesses and increase the number of parallel operations. Over 20 operations per cycle are achieved. In addition to clock gating and low voltage operation techniques, the authors propose to partition the datapath and the read-only memory (ROM) of the complete architecture. The underlying concept is that there are different types of operations that do not require the same hardware resources. This partitioning is implemented for reasons of power consumption optimization and depends on the operation mode: FFT and non-FFT. Consequently, only one of the ROMs needs to be accessed in each clock cycle, which reduces the power consumption of the DSP by about 40 %. The DSP is embedded in the hearing aid chip with the instruction read-only memorys (ROMs) and the parameter static random-access memorys (SRAMs). The final chip consumes on average 0.66 mW at a core voltage of 1.05 V. The analog front end including a digital-to-analog converter (DAC), programmable gain amplifier, and a serial interface is integrated on a separate chip [73].

2.2.3 ASIPs with Hardware Accelerators

There are hearing aid processing architectures that combine ASIPs with dedicated hard-wired accelerators. These accelerators are used for frequent, computationally intensive tasks. The flexibility and complexity of these accelerators varies. Examples of accelerators for hearing aids can be found in Table 2.3. The hearing aid processing task is mapped to either the ASIP or the accelerator. The goal is to process the intensive computing task on the accelerator, while the ASIP controls the accelerator processing [12].

In [25], an arithmetic unit with a dual MAC and butterfly unit is introduced, which can operate either in FFT-mode or in CNN-mode. By sharing hardware resources, 42 % of hardware complexity can be saved. In [12], a streaming DSP hardware accelerator is introduced, which can compute applications such as keyword recognition or other algorithms for classifications. Each of the co-processors in [56] can be disabled by clock gating, however these operations are elementary and often used in hearing aid applications. This also applies to the FIR filter accelerators presented in [9, 74]. The accelerators presented in [9–11, 19, 20, 41, 43] are more complex and specific, because they implement complete algorithms, such as noise reduction (NR), feedback cancellation (FBC) in hardware. An advantage is, that the computing efficiency is increased by the hardwired implementation. This leads to lower power consumption. However, the silicon area requirement can be higher. In case of algorithm changes, it is possible to use ASIP processing resources instead of the accelerators.

Table 2.3: Related hardware accelerators for hearing aid systems.

Publication	Accelerators
[9, 20, 74]	Finite impulse response (FIR) filter accelerators
[25]	Convolutional neural networks (CNN) and fast Fourier transform (FFT) accelerators for speech enhancement
[12]	Streaming DSP for voice code word detection
[56]	Co-processors for hyperbolic and trigonometric functions
[33]	Noise reduction (NR) accelerator
[33]	Multiply-accumulate (MAC) unit accelerator
[33]	Fast Fourier transform (FFT) accelerator
[9–11, 19, 20, 41, 43]	Analysis filter bank (AFB) accelerator
[9–11, 19, 20, 41, 43]	Noise reduction (NR) accelerator
[9–11, 19, 20, 41, 43]	Feedback cancellation (FBC) accelerator
[9–11, 19, 20, 41, 43]	Wide dynamic range compression (WDRC) accelerator

3 The KAVUAKA Hearing Aid Processor

The design, architecture and optimization of *KAVUAKA* hearing aid processor are described in this section. The architecture is based on the *Moai4k2* architecture, a scalable and configurable ASIP architecture for multimedia applications [16].

Previous studies [8, 10, 19, 20, 43, 68, 74, 75, 77–80], which investigate application-specific instruction-set processors (ASIPs) for hearing aids, presented promising results. Compared to dedicated and hard-wired hearing aid implementations [9, 13–15, 81], ASIP implementations offer more flexibility, while maintaining the limits of low-power consumption and processing performance.

In this section, new architectures and design methodologies for hearing aid application-specific instruction-set processors (ASIPs) are presented and evaluated. The section starts with a description of the baseline processor architecture (Section 3.1) and continues with subsections on processing performance (Section 3.2) and low-power optimizations (Section 3.3). A low latency audio interface and a serial data interface for hearing aids are presented in Section 3.4. A hearing aid system-on-chip (SoC) implementation is described in Section 3.5.

3.1 Baseline Processor Architecture

Within the *RAPANUI* project [16], a design space exploration framework for application specific VLIW-SIMD processors (ASIPs) was introduced. This framework comprises a pre-built configurable VLIW-SIMD processor architecture, which is designed and implemented using a hardware description language (HDL). The proposed generic baseline processor architecture can be customized in accordance with application-specific specifications and requirements, such as processing performance, silicon area, and power consumption. Not only the integration of new user-defined operations, e.g., new functional units or co-processors, is possible, but also fundamental changes within the architecture, e.g., changes of pipeline stages, register file, or external bus interfaces. In addition to the processor architecture, there is a corresponding instruction scheduler. This instruction scheduler can be quickly adapted to the given processor configuration. This section introduces the baseline VLIW-SIMD processor architecture, the instruction-set architecture (Section 3.1.2), and the related ASIP architectures (Section 3.1.1).

3.1.1 Related ASIP Architectures

The author of [77] presents an ASIP implementation based on the *Cadence Tensilica Xtensa LX4* processor for a digital hearing aid system. The baseline ASIP architecture consists of a 32-bit RISC architecture with configurable instruction and data cache sizes. The number of instructions executed per clock cycle in parallel (called flexible length instruction extensions (FLIX)) is also configurable. In addition, the *Xtensa* framework offers the possibility to add user-defined instructions to the instruction set architecture (ISA) using Tensilica instruction extensions (*TIE*). However, the *Xtensa* framework has limitations when it comes to changing certain parts of the underlying *LX4* processor architecture. Major changes to the pipeline architecture, e.g., significant reduction of the pipeline levels, or the register file architecture, e.g., changing the number of ports of the general purpose register file, and extensions of the forwarding path are not possible. It is shown that the most efficient *LX4* configuration for processing a complex modulated filter bank [82] and a noise reduction algorithm [58] requires several TIEs and no FLIX extensions. The silicon area requirement is 0.623 mm^2 (TSMC 40 nm low-power). The processor requires 13.24 MHz for real-time processing, while it consumes about 2 mW.

The authors of [19] present a fully integrated system-on-chip with a digital signal processor, low-power accelerators, an analog front end (AFE) and a class-D amplifier (switching amplifier). The 24-bit DSP and the dedicated accelerators are flexible and power-efficient with respect to the evaluated hearing aid algorithms, i.e., wide dynamic range compression, noise reduction, and feedback cancellation. The system was fabricated using a 130 nm CMOS technology and consumes approximately 1.2 mA current at 8 MHz clock frequency with a 1 V power supply. 0.86 mA of this current is consumed by the DSP, which is 71 % of the total current.

The authors of [8] present a modified *Silicon Hive Pearl* 16-bit three issue-slot VLIW ASIP architecture with a 32-bit datapath, 40-bit intermediate result registers, separate instruction and data memories, fixed-point arithmetic units and user-defined instructions. The algorithms executed are beamforming, feedback suppression, a FIR filter bank, compression and noise reduction. With voltage and frequency scaling, the processor requires 0.964 mW average power and a silicon area of 0.49 mm^2. The application-specific integrated circuit (ASIC) technology used is the TSMC C65G and the required and applied clock frequency is 11 MHz.

The authors of [75] propose a low-power fixed-point DSP containing a 16/32-bit datapath and twelve simultaneous 16-bit multipliers. The processor is optimized for FFT computation with up to 128-bit wide SIMD instructions, optimized instruction scheduling, and reverse bit addressing for improved FFT processing. The executed algorithms are a FFT-based filter bank, compression, and feedback suppression. After an optimization process, the DSP requires less than 37.5 % of the total processing time for these algorithms. The total processing time is specified as the real-time processing time at a clock frequency of 8 MHz.

A commercial hearing aid system-on-chip (SoC) with the name *Ezario 7100* is presented in [78]. This system comprises three hardware modules, a general purpose ARM core for system management, an energy efficient 24-bit coolflux dsp (CFX) digital signal processor and a dedicated hardware accelerator called *HEAR*. This system therefore offers a compromise between software flexibility and hardware efficiency. The system was manufactured in a 65 nm TSMC mixed-signal process, including 100 kB of SRAM memory. At a maximum clock frequency of 10.24 MHz and at 1.25 V, the peak power consumption is 5 mW when four MACs are used in parallel and the average power consumption is between 1 mW and 2 mW.

3.1.2 Baseline Instruction-Set Architecture

The pipeline architecture of the baseline VLIW-SIMD ASIP, as proposed in [16] and [83], is shown in Figure 3.1. A vector unit (VU) is depicted. The vector unit is divided into five or six pipeline stages in this baseline configuration: instruction fetch (IF), instruction decode (DE), register access (RA), execution (EX1 and EX2) and write back (WB).

Within the instruction fetch (IF) stage, instructions are fetched from the local instruction memory. The instruction address is generated by the program counter (PC). Two instructions, so called micro-operations (MOs), are encoded within a 64-bit micro-instruction (MI) [16]. One MI represents one very long instruction word (VLIW). The encoding of one of the micro-instructions is shown in Figure 3.2. The instruction-set is orthogonal to keep the complexity of the hardware low. It includes several common instructions such as arithmetic, logic, permutation, multiplication, control flow, and load/store instructions. It can be extended to match the requirements of the applications.

The presented vector unit contains two issue slots, labeled *issue 0* and *issue 1*, so that two 32-bit wide MOs are executed in parallel. Therefore, two almost independent instruction-decoders are part of the instruction decode (DE) stage. To further increase the instructions per cycle (IPC) value, two MOs can be merged into one X2-instruction, which is represented as X2 mode in Figure 3.1 [83, 84]. In this mode, two identical instructions, which share the same opcode (command and command modifier), are merged into one single instruction. The merged X2-instruction controls two functional units of the same type, which are located in the execution stage. The addressed registers differ only in their last significant address bit. The corresponding implementation of the X2-mode register file is shown in Figure 3.4. Therefore, more issue-slots are created virtually, because a X2-command needs only one issue-slot, although it executes the command on duplicated functional units. Because only one instruction is decoded, only minor changes to the instruction decode stage and register file architecture are required to enable X2-mode execution with low overhead. A code example for the X2-mode is shown in Figure 3.3.

Furthermore, the instruction decode stage contains file indirect registers (FIREGs), which are shared between the two issue slots. These registers store address pointers, which point to

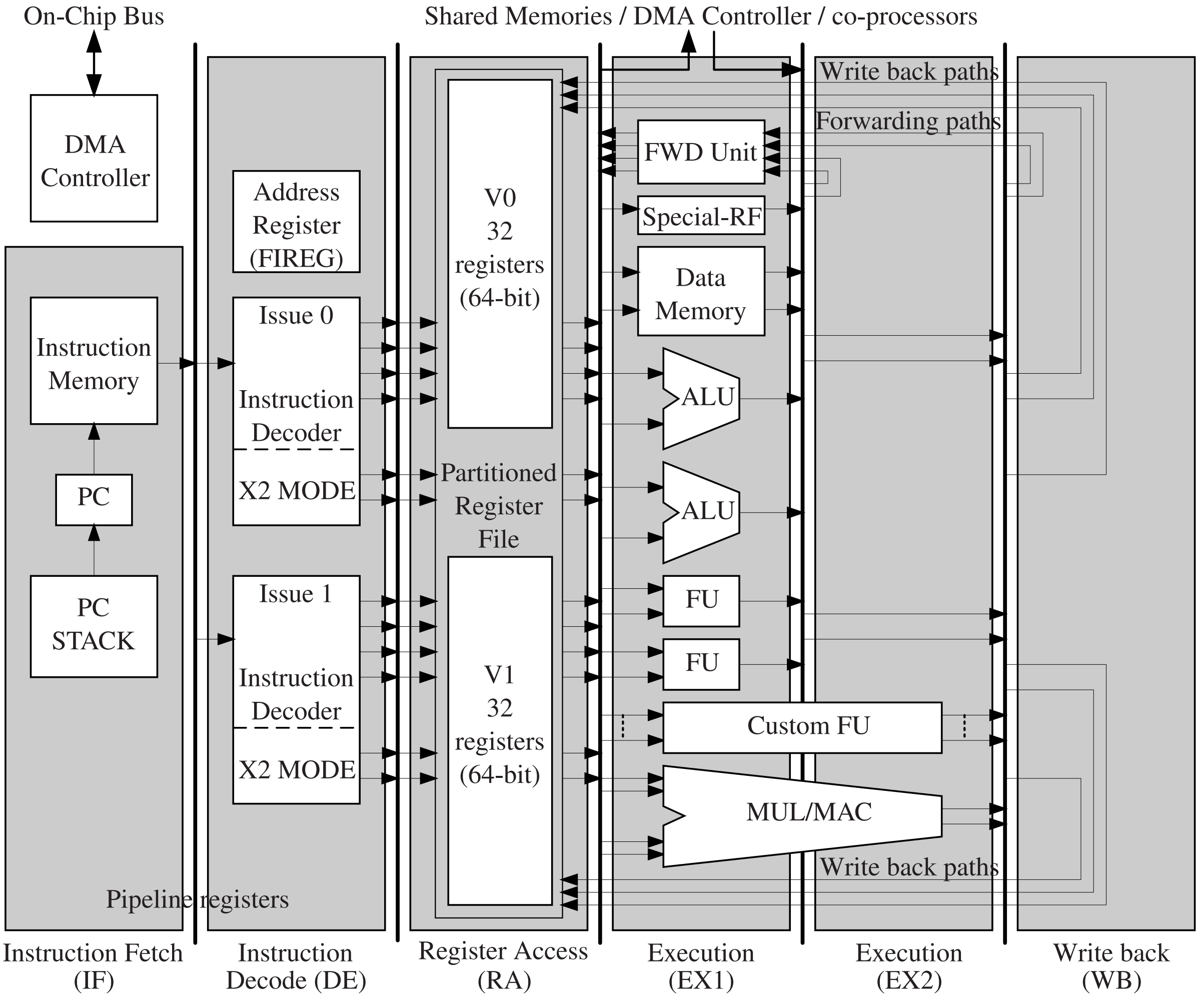
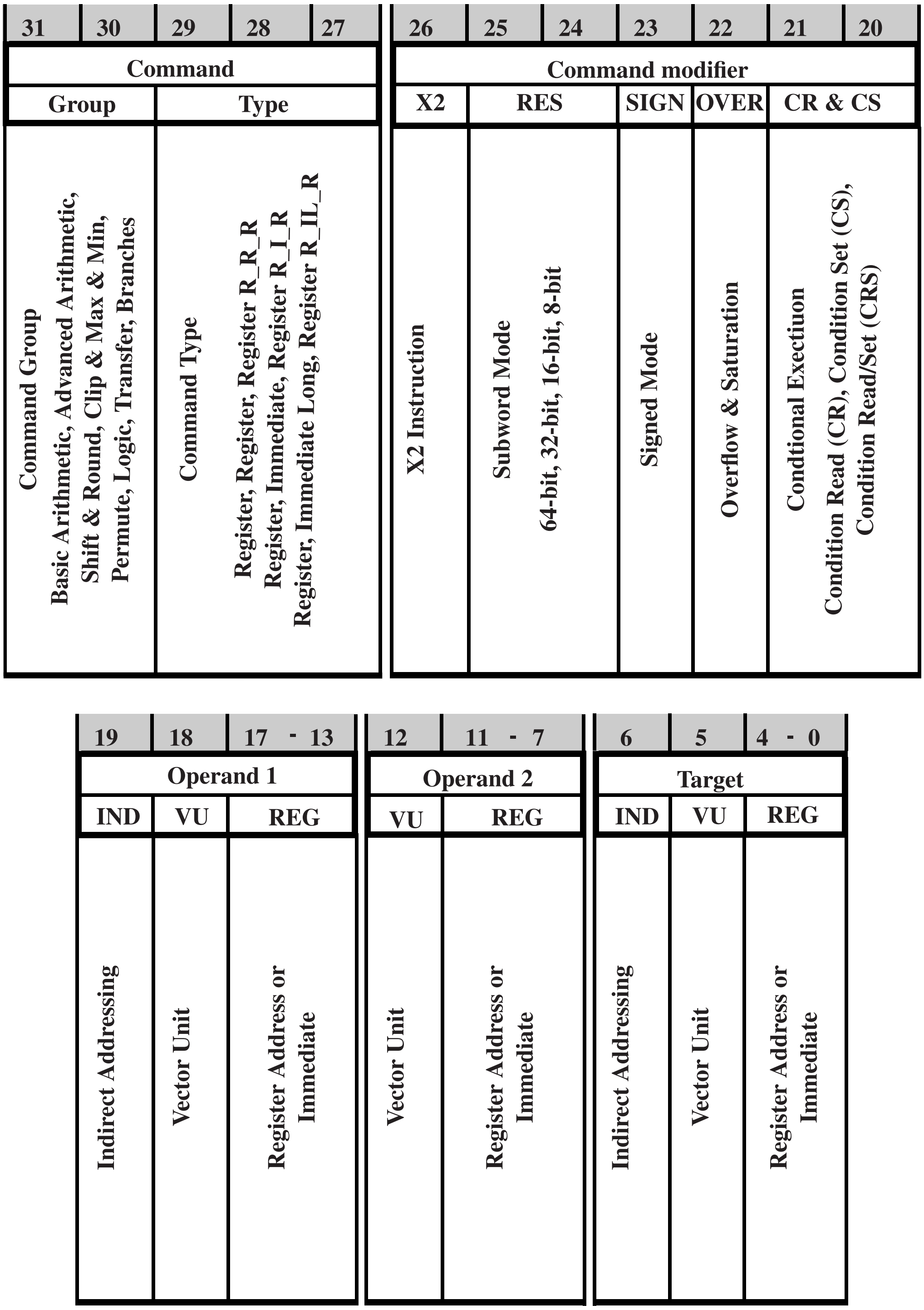


Figure 3.1: Vector unit (VU) of the baseline VLIW-SIMD processor [16, 83].



31 30	29 28 27	26	25 24	23	22	21 20
Command		Command modifier				
Group	Type	X2	RES	SIGN	OVER	CR & CS
Command Group Basic Arithmetic, Advanced Arithmetic, Shift & Round, Clip & Max & Min, Permute, Logic, Transfer, Branches	Command Type Register, Register, Register R_R_R Register, Immediate, Register R_I_R Register, Immediate Long, Register R_IL_R	X2 Instruction	Subword Mode 64-bit, 32-bit, 16-bit, 8-bit	Signed Mode	Overflow & Saturation	Condtional Exectiuon Condition Read (CR), Condition Set (CS), Condition Read/Set (CRS)

19	18	17 - 13	12	11 - 7	6	5	4 - 0
Operand 1			Operand 2		Target		
IND	VU	REG	VU	REG	IND	VU	REG
Indirect Addressing	Vector Unit	Register Address or Immediate	Vector Unit	Register Address or Immediate	Indirect Addressing	Vector Unit	Register Address or Immediate

Figure 3.2: 32-bit MO. The MO is partitioned into the command, the command modifier, two operands and one target register.

(a)

```
//    Issue slot #0                           ; Issue slot #1
(0x00) ADD_16 V0R0,V0R2,V0R4                  ; NOP
(0x01) ADD_16 V0R1,V0R2,V0R5                  ; NOP
(0x02) ADD_16 V0R6,V0R8,V0R10                 ; NOP
(0x03) ADD_16 V0R7,V0R9,V0R10                 ; NOP
```

(b)

```
(0x00) ADD_16 V0R0,V0R2,V0R4                  ; ADD_16 V0R1,V0R2,V0R5
(0x01) ADD_16 V0R6,V0R8,V0R10                 ; ADD_16 V0R7,V0R9,V0R10
```

(c)

```
(0x00) ADD_16_X2 V0R0+V0R1,V0R2,V0R4+V0R5     ; NOP
(0x01) ADD_16_X2 V0R6+V0R7,V0R8+V0R9,V0R10    ; NOP
```

Figure 3.3: Example of a scheduled assembler code using the X2-mode [83]. (a) Four additions (ADD) executed with one arithmetic unit sequentially. Registers V0R0, . . . , V0R10 are used. (b) Replication of arithmetic unit. (c) Using X2-mode. Two unused issue-slots are present (filled with NOPs), when using X2-mode. Those can be used with other instructions, which do not use the already used arithmetic unit.

any address in the global address space, including the main data memory (DMEM), the coprocessors or the register file. Indirect addressing can be used with post increment, decrement and offset addressing.

Both issue-slots, belonging to one vector unit, share a partitioned register file [85], which is located in the register access (RA) stage. In the configuration, shown in Figure 3.1, each partition, V0 and V1, contains 32 registers of 64-bit. Figure 3.4 shows a register file configuration with four read and two write ports. The read and write ports are controlled by the instruction decoders or the write back and forwarding paths. Since the number of register file ports can be the bottleneck, that prevents dense code compaction, the register file is partitioned into two small register files with 32 registers each. Table 3.1 lists different available register file configurations, comparing normalized silicon area requirements and scheduling flexibility.

The functional units are in the execution stage. The output of the functional units is written back to the register file or to local memory. Forwarding is used to avoid data hazard stalls. The execution stage of the processor architecture can be customized to the application by adding custom functional units, replacing existing functional units, or by duplicating them. Adding new custom functional units is an approach to specialization, while duplication offers more performance, e.g., the X2 mode. Functional units can be extended to implement 64-bit SIMD commands using subword parallelism, i.e., the 64-bit operands can be divided into two 32-bit, four 16-bit or eight 8-bit operands to perform the same operation on all these subwords si-

Table 3.1: Available register file configurations (4r2w = four read and two write ports, 2r1w = two read and one write port) [83]. Results from an ASIC synthesis with the TSMC 40 nm HVT low-power ASIC technology at 50 MHz.

	Port config.	Port width	Parallel X1-accesses	Parallel X2-accesses	Normalized area	Scheduling flexibility
RF1	4r2w	64	2	1	1.00	normal
RF2	4r2w	128	2	2	1.06	high
RF3	2r1w	128	1	1	0.76	low

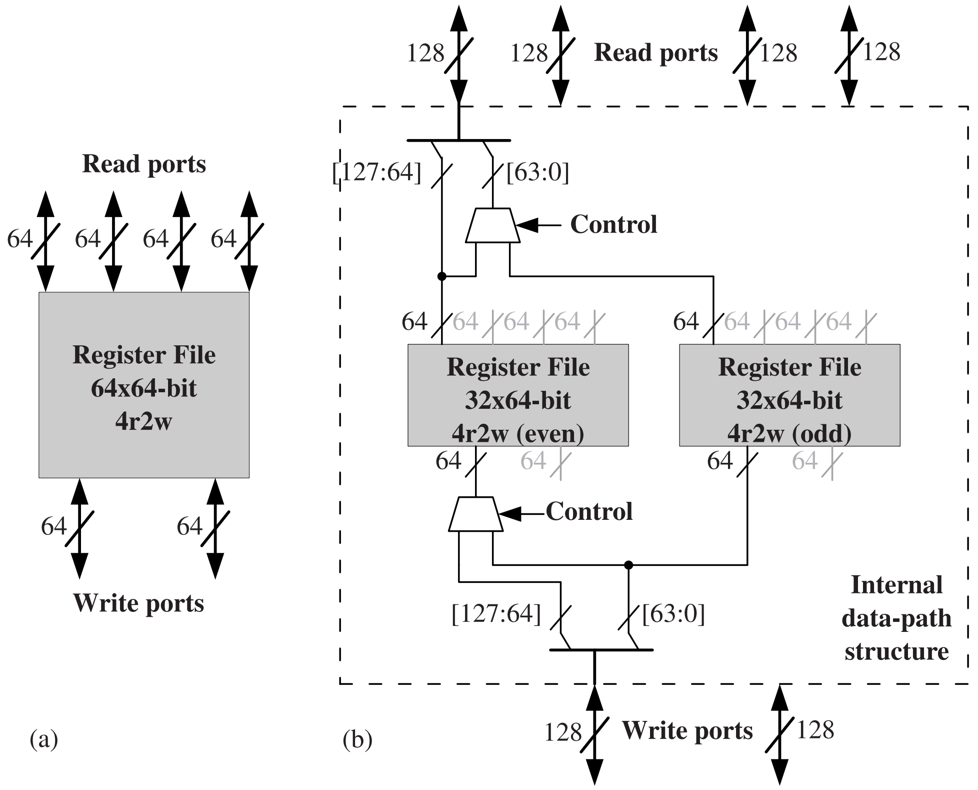


Figure 3.4: Monolithic and partitioned register file (RF) configurations [83]. In (a) a monolithic RF with 64×64-bit registers with four read and two write ports is shown. In (b) a partitioned register file is shown. The port width is doubled for reading and writing 128-bit. Two internal 32×64-bit RF are used. In case of a X2-instruction as shown in Figure 3.3, the upper and lower 64-bit of the 128-bit data is written/read to/from the even or odd register file.

multaneously. Low-power mechanisms such as operand isolation and clock or power gating are applicable. Additionally, conditional execution is used to convert control flow dependencies into data dependencies. This results in higher code compression by avoiding conditional branching, as shown in Figure 3.5. Since the critical path is usually in the execution stage, complex functional units (FUs) with two or more pipeline stages can be implemented for performance purposes.

```
for (i=0; i<8; i++){
  if (V0R0[i*8+7:i*8] == 0){
    V0R3[i*8+7:i*8] = V0R1[i*8+7:i*8] + V0R2[i*8+7:i*8];
  }
}
```

```
SMVI V0CONDSEL,#COND_ZERO   // choose evaluated condition
SUBICS_8 V0R4,V0R0,#0       // store operation flags (CS)
ADDCR_8 V0R3,V0R1,V0R2      // cond. execution, using flags and cond. implicitly (CR)
```

Figure 3.5: Assembler example for subword parallelism and conditional execution [83]. The addition is only executed for the 8-bit subwords for which the condition is set.

Co-processors (CPs) can be used for more complex operations [83]. Co-processors can be tightly attached to the processor via an external bus interface. By default the VLIW SIMD architecture includes a 2×32-bit SIMD fixed-point division co-processor unit (DCU), which implements an iterative, non-restoring division algorithm in dedicated hardware [86–88]. After saving the divisor operand, the processor starts the division and increases the accuracy of the result in each cycle. Thus, the desired minimum accuracy can be controlled by the user-defined number of cycles between saving the last operand and loading the result, as shown in Figure 3.6.

3.2 Specialization Towards Performance

The application-specific hardware and software specialization for performance and power consumption optimization reasons is of great significance in the design of hearing aid processors. One of the central challenges is to provide sufficient processing performance for given and future hearing aid algorithms under strict power consumption and silicon area constraints. In this section, new hardware and software architectures for increased processing performance for application-specific hearing processors are presented.

(a)

```
//store dividend
STOREDCU HW_ADDRESS_DCU0_0,V0R0,#0
//store divisor / start
STOREDCU HW_ADDRESS_DCU0_1,V0R1,#4
//delayed load of result
LOADDCU  V0R2,HW_ADDRESS_DCU0_0
```

(b)

```
(0x00) STOREDCU 0x200,V0R0,#0
(0x01) STOREDCU 0x201,V0R1,#4
(0x02) NOP
(0x03) NOP
(0x04) NOP
(0x05) LOADDCU  V0R2,0x200
```

Figure 3.6: Exemplary use of division co-processor unit (DCU). (a) Coded assembler operations (b) Scheduled assembler operations. After both operands are stored, the DCU is enables. The user-defined delay (#4), which determines the iteration counts of the DCU. The *LOADDCU* instruction is scheduled with a minimum delay of (#4) after the DCU starts the computation.

3.2.1 Real- and Complex-Valued Multiply-Accumulate Functional Unit

Multiplication and addition operations are among the most frequently used operations in digital signal processing [89]. Fundamental signal processing tasks like correlations, filtering or transformations require a large number of these operations. Hardware functional units for multiply-accumulate (MAC) operations are therefore an essential part of digital signal processors (DSPs) [9, 19, 74, 90–123]. The combination of a hardware multiplier and an adder in a functional unit reduces the number of execution cycles required for the mentioned signal processing tasks. The result of this architecture decision enables higher processing performance or energy efficiency. However, due to the continuing demand for higher processing performance in digital signal processing applications, further approaches, architectures, and extensions for MAC units have been presented. The goal is to further parallelize and specialize MAC operations. Parallelism is increased by duplicating the MAC unit in the DSP architecture [74, 90, 96, 97, 100, 101, 104, 106, 110, 115] and by using single instruction, multiple data MAC units [91–96, 98–101, 103–105, 107–111, 113, 114, 116–119, 121, 123]. Complex-valued digital audio processing, such as for filter banks, is required [124, 125]. More specialized MAC units also process complex-valued data [93, 96, 102, 103, 110, 115, 126, 127].

In this section, a new architecture of a real- and complex-valued multiply-accumulate (MAC) functional unit for digital signal processors is introduced [128]. The new MAC architecture described here is SIMD-capable for parallel processing and is specialized for complex-valued operations. The functionality of related real-valued SIMD-MAC units is maintained. By reusing existing hardware adders and multipliers of the real-valued SIMD-MAC architecture, complex-valued MAC operations and butterfly operations are implemented for the FFT computation. The area overhead for these specialized complex-valued operations is discussed and compared to the overhead of related complex-valued architectures.

Related Multiply-Accumulate Architectures

Hardware functional units for multiply-accumulate (MAC) operations are an essential component in digital signal processors [9, 19, 74, 90–123]. These MAC architectures can be classified by the following architectures and use cases:

A single MAC unit is used in [10, 19, 43, 112, 120, 129, 130]. With these architectures a real-valued multiplication and accumulation per instruction per cycle can be computed. Several consecutive instructions are required to compute a complex-valued MAC operation. Independent MAC units, implemented as a *dual MAC unit* architecture, are used in VLIW-DSP architectures to compute MAC operations in parallel [74, 90, 92, 96, 97, 100, 101, 104, 106, 110, 115], reducing the number of processing cycles. In [92], the authors present a DSP consisting of four identical data paths, each containing a MAC unit. The implementation of two almost identical MAC units, the so-called *RMAC* and *IMAC* units, is presented in [96] and [115]. With these two units, the imaginary and the real part of a complex-valued operation are processed in parallel.

SIMD MAC units are proposed in [75, 91, 93–96, 98–101, 103–105, 107–114, 116–119, 121, 123]. These SIMD-MAC units can be used to process multiple real-valued operands by executing a single instruction, thereby increasing processing performance. Complex-valued MAC operations require multiple successive instructions [106, 110, 119, 123, 131]. In [110], a DSP core is equipped with a dual MAC architecture. This MAC architecture is optimized for the computation of digital filters. An input port of one MAC is the delayed input of the other MAC unit. Both MAC units support multiple subwords. This architecture decreases data access requests for algorithms, that use the same operands for many MAC operations. The authors of [106] and [115] introduce DSPs with two MAC units. In both cases the MACs can reuse previously latched output values. These architectures are specially optimized for finite impulse response (FIR) filter calculations. In [119] and [100], SIMD-MAC units are used for a two-way superscalar RISC architecture and a DSPs with low-power consumption. In both cases, the butterflies of a FFT are processed in parallel to increase performance.

Specialized complex-valued MAC units support complex-valued or both real- and complex-valued operations [93, 126, 127, 132, 133]. Complex-valued operations can be computed with these units with fewer instructions compared to the computation with real-valued MAC units, where real and imaginary part have to be computed separately. Some of these architectures are also extended for a butterfly operation to accelerate the computation of FFT algorithms [93, 126, 132, 133].

In [127], a DSP is equipped with a specialized complex-valued multiplier. This functional unit cannot be used for real multiplications and does not support SIMD operations.

The MAC unit introduced in [93] supports real- and complex SIMD operations for single- and full-precision 16-bit MAC operations. This unit is optimized for real- and complex-valued FIR convolutions. A great acceleration for real- and complex-valued filter computation can be realized by this architecture. Besides the efficient implementation of FIR filters with these MAC architectures, there are disadvantages for the use in other cases. Parts of the computed results, which are either real- or complex-valued, are stored in different accumulator registers, which are part of the MAC architecture. Not all these registers are directly accessible in every cycle. The output multiplexer of the MAC also limits the number of transferred words to the register file. The complex-valued multiplication results of this MAC unit are used by the butterfly processor architecture. One butterfly per cycle can be calculated.

The authors of [126] propose a data processing unit for DSPs. This unit consists of two multipliers with three pipeline stages and five adders. New instructions are introduced to perform different combinations of additions and multiplications. This unit can perform two butterfly operations in three cycles. The advantage of this unit is the flexibility to switch between real and complex operations. The computation of the FFT algorithm is about 40 % faster than DSPs using SIMD-MAC units. Two data paths without SIMD support are used in this design. Each datapath contains a multiplier and several adders, which are used simultaneously in parallel algorithms such as the FFT.

A comparable multiple MAC architecture is introduced in [117]. It does not support the additional operations for computing parts of the butterfly and is about 20 % slower in the case of a 1024-point FFT computation (7680 cycles). A processor with this MAC unit is about 90 % faster than the single MAC approach presented in [92].

The butterfly unit of the FFT processor presented in [132] consists of four multipliers and four adders. It computes a butterfly operation in one cycle. This butterfly unit has two complex inputs and generates two complex outputs. The twiddle factors are stored in lookup tables (LUTs) or calculated on-the-fly by a coordinate rotation digital computer (CORDIC) algorithm. The computation of a 256 point FFT is about 50 % faster than the multi-cycle approach presented in [126]. A complete custom function unit (CFU) for butterfly computation is proposed in [133]. Each of these custom functional units consists of four multipliers and eight adders. These units have two input ports and two output ports. Each port is 32-bit wide and contains 16-bit real and complex values. With twelve of these units, twelve butterfly operations can be performed simultaneously. The 1024 point FFT performance (2496 cycles) is about three times faster than the performance of the processor introduced in [126].

A chronological representation of the publication dates for these architectures is given in Figure 3.7.

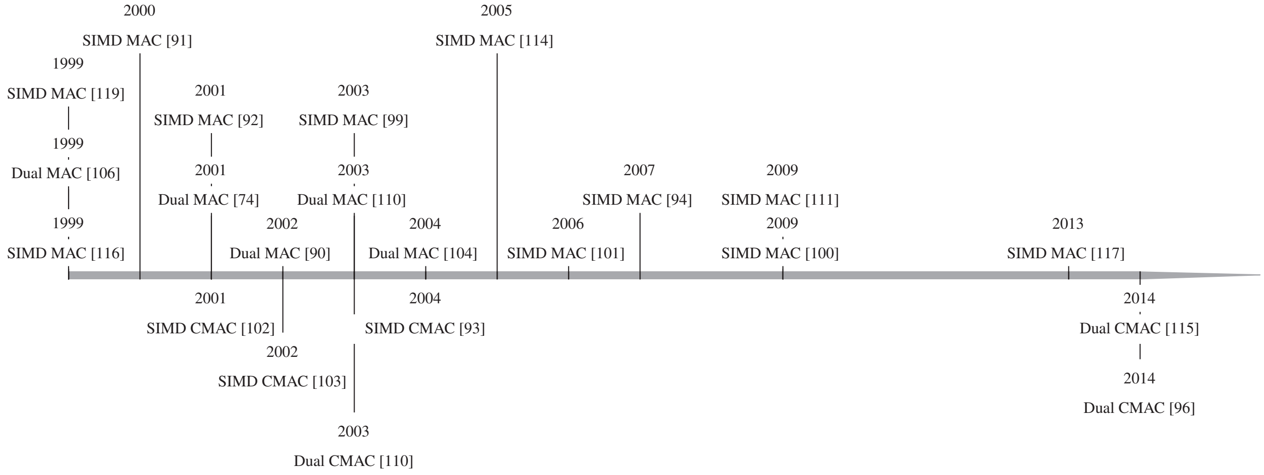


Figure 3.7: Related MAC architectures that have been published in recent years. Dual MAC architectures include at least two independent MAC units, SIMD MAC architectures include at least one MAC unit with SIMD capabilities and SIMD CMAC architectures include at least one CMAC unit with SIMD capabilities, that can additionally perform complex-valued operations.

Proposed area efficient real-and complex-valued multiply-accumulate SIMD unit

The proposed real- and complex-valued SIMD MAC unit, which is abbreviated as SIMD-CMAC unit, is described in detail in this section. First of all, the arithmetic operations are introduced and then the hardware implementation is presented. Table 3.2 shows the arithmetic operations implemented in the proposed SIMD-CMAC unit. These operations include real- and complex-valued multiply, multiply-accumulate operations, and a butterfly operation for the FFT acceleration. Since SIMD mechanisms are used, all operations can process several subwords simultaneously.

The generic multiply (MUL), the multiply-accumulate (MAC), and the multiply-accumulate with place zero (PLZ) (MACPLZ) operations are defined by Equation 3.1. The number of

Table 3.2: Real- and complex-valued operations: multiply, multiply-Accumulate and butterfly

Arithmetic Operation	**Real-valued**	**Complex-valued**
multiply	MUL	CMUL
multiply-accumulate	MAC	CMAC
multiply-accumulate-zero	MACPLZ	CMACPLZ
Butterfly	—	BUTTERFLY

SIMD subwords is hereinafter referred with the index variable s.

$$\begin{aligned} & a_s = a_s + b_s \odot c_s \\ & \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_s \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_s \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_s \end{pmatrix} \odot \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_s \end{pmatrix} = \begin{pmatrix} a_1 + b_1 \cdot c_1 \\ a_2 + b_2 \cdot c_2 \\ \vdots \\ a_s + b_s \cdot c_s \end{pmatrix} \end{aligned} \tag{3.1}$$

The real-valued multiplication factors b_s and c_s as well as the accumulator a_s are vectors containing s subwords. For the multiplication operation all subwords in b_s and c_s are multiplied element by element, while the accumulator a_s is not used (zero). The resulting subwords are stored in a_s. The multiply-accumulate-zero (PLZ) operation performs an element-wise multiplication and an addition with an accumulator a_s, which is set to zero. This operation is used to reset the accumulator and the compute a multiplication with full precision. In case of the multiply-accumulate operation the accumulator a_s is added to the multiplication. To perform a sequence of multiply-accumulate operations, the result vector a_s is the same as the accumulator vector a_s.

The same scheme can be defined for multiplication and multiply-accumulate operations with complex-valued numbers. In this case, the vectors a_s, b_s and c_s consist of s complex numbers with a real and an imaginary part. The complex-valued operations CMUL, CMAC and CMACPLZ are then given by Equation 3.2.

$$\begin{aligned}
a_s &= a_s + b_s \odot c_s \\
\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_s \end{pmatrix} &= \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_s \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_s \end{pmatrix} \odot \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_s \end{pmatrix} = \begin{pmatrix} a_1 + b_1 \cdot c_1 \\ a_2 + b_2 \cdot c_2 \\ \vdots \\ a_s + b_s \cdot c_s \end{pmatrix} \\
&= \begin{pmatrix} \Re(a_1) + \Re(b_1)\Re(c_1) - \Im(b_1)\Im(c_1) \\ \Re(a_2) + \Re(b_2)\Re(c_2) - \Im(b_2)\Im(c_2) \\ \vdots \\ \Re(a_s) + \Re(b_s)\Re(c_s) - \Im(b_s)\Im(c_s) \end{pmatrix} \\
&+ j \cdot \begin{pmatrix} \Im(a_1) + \Re(b_1)\Im(c_1) + \Im(b_1)\Re(c_1) \\ \Im(a_2) + \Re(b_2)\Im(c_2) + \Im(b_2)\Re(c_2) \\ \vdots \\ \Im(a_s) + \Re(b_s)\Im(c_s) + \Im(b_s)\Re(c_s) \end{pmatrix}
\end{aligned} \tag{3.2}$$

Complex-valued MAC operations were proposed by [93, 102, 110, 126, 127], but these architectures do not support SIMD operations or access to single subwords is restricted.

To speed up the computation of a radix-2 fast Fourier transform, a butterfly operation can be used. The butterfly operation consists of a complex-valued multiplication, one complex-valued addition and a subtraction. The equation for the computation of a SIMD butterfly operation is given in Equation 3.3 and Equation 3.4

$$\begin{aligned}
a_s &= a_s + b_s \odot c_s \\
\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_s \end{pmatrix} &= \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_s \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_s \end{pmatrix} \odot \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ w_s \end{pmatrix} = \begin{pmatrix} a_1 + b_1 \cdot c_1 \\ a_2 + b_2 \cdot c_2 \\ \vdots \\ a_s + b_s \cdot c_s \end{pmatrix}
\end{aligned} \tag{3.3}$$

$$\begin{aligned}
b_s &= a_s - b_s \odot c_s \\
\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_s \end{pmatrix} &= \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_s \end{pmatrix} - \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_s \end{pmatrix} \odot \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_s \end{pmatrix} = \begin{pmatrix} a_1 - b_1 \cdot c_1 \\ a_2 - b_2 \cdot c_2 \\ \vdots \\ a_s - b_s \cdot c_s \end{pmatrix}
\end{aligned} \tag{3.4}$$

where a and b represent the complex-valued signal subwords and the vector c contains the complex-valued twiddle factors.

Several functional units for computing a butterfly operation shown in Figure 3.8 have been presented in related work, but none of them supports SIMD operation [93, 126, 133].

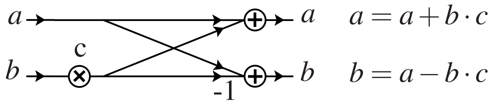


Figure 3.8: Radix-2 decimation in time butterfly.

In the following, it is described how the given SIMD operations defined by Equation 3.1, Equation 3.2, Equation 3.3, Equation 3.4 and Table 3.2 are implemented as a functional unit. The proposed SIMD-CMAC unit supports four subword modes. Each input port is 64-bit wide. The 64-bit are interpreted as subwords of equal length (s_0, s_1, $\dots$, s_n) as shown in Figure 3.9. Subwords can have either 64-bit, 32-bit, 16-bit, or 8-bit. Each subword represents an unsigned or signed data type, that stores integer or fixed-point numbers. For complex-valued operations, two consecutive subwords are interpreted as a complex number. The first subwords (s_1, s_3, $\dots$, s_n) represent the real part and the second subwords (s_0, s_2, $\dots$, s_{n-1}) represent the imaginary part of a complex-valued number. The imaginary parts are highlighted in gray in Figure 3.9.

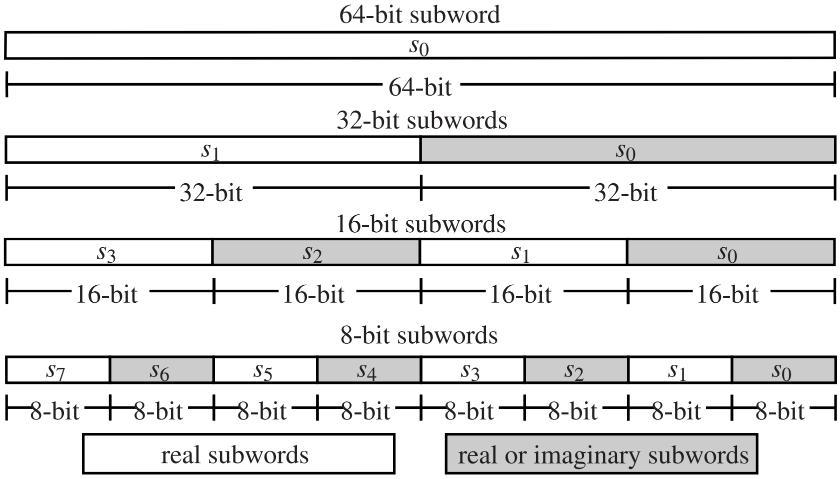


Figure 3.9: SIMD data format. Each word of 64-bit is composed of the subwords 64-bit, 32-bit, 16-bit or 8-bit. To compute real or complex-valued operations, the subwords must represent real or imaginary values. Real-valued subwords are colored white, and subwords representing real or imaginary values are highlighted with the color gray.

Figure 3.10 illustrates the architecture of the proposed SIMD-CMAC unit. Three input ports and one double-width output port are used for real and complex-valued multiply (MUL), multiply-accumulate (MAC), and butterfly operations. Compared to other MAC architectures [93, 102, 106, 117], the proposed architecture does not contain registers. Instead, the register file of the processor is used to replace the additional accumulator register. This design decision provides more flexibility in implementing the butterfly operation and reduces the number of registers used for an area-efficient implementation. Therefore, the accumulator output port ($a0$ and $a1$) is connected to the same registers of the processor as the accumulator input port ($a0$ and $a1$). The accumulator, using a register pair, is 128-bit wide. The input ports (b and c) represent the product factors. These ports are 64-bit each.

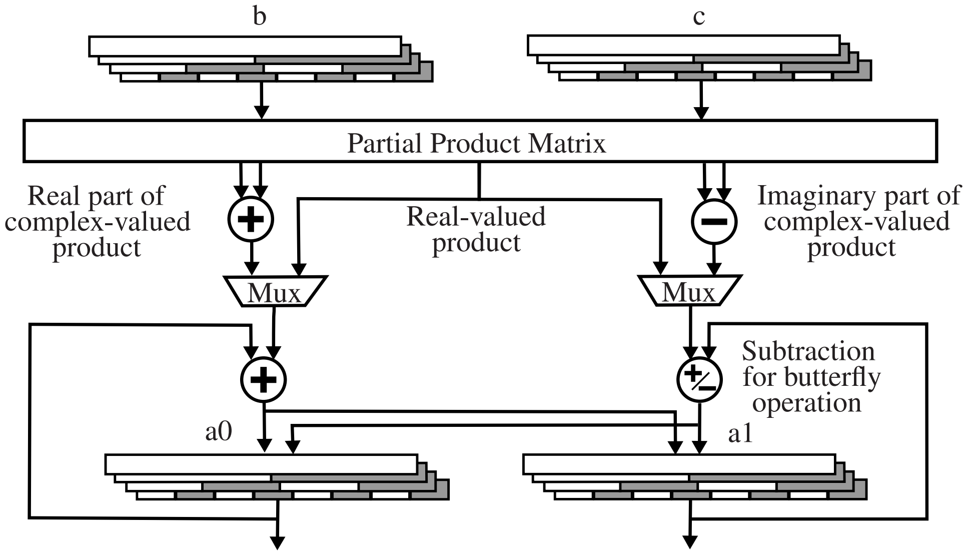

b
c
Partial Product Matrix
Real part of complex-valued product
Real-valued product
Imaginary part of complex-valued product
Mux
Mux
Subtraction for butterfly operation
a0
a1

Figure 3.10: Real- and complex-valued MUL/MAC/CMAC/butterfly SIMD architecture.

The number of multipliers required for the implementation depends on the operation itself and the SIMD mode. The partial product architectures presented in [99, 114, 120, 134] were extended in this thesis to perform both real- and complex-valued multiplications with nearly the same hardware requirements. A real-valued multiplication based on a single-stage partial product matrix is shown in Figure 3.11.

The multiplier b and the multiplicand c are divided into subwords of equal size. The corresponding subwords are multiplied and then added together to obtain the final product. This product scheme can be applied to different subword modes and word lengths. The advantage of this product scheme is that the same multiplier stages can be used for different subword modes, reducing hardware costs as opposed to multiple parallel high-precision multipliers. This scheme has been extended in this thesis for complex-valued multiplications. Figure 3.12 shows that the same multipliers can execute all products required for real- and complex-valued multiplication operations. In this case, the subwords b_1 and c_1 represent the real-part and b_0 and c_0 represent the imaginary part of the complex-valued words b and c.

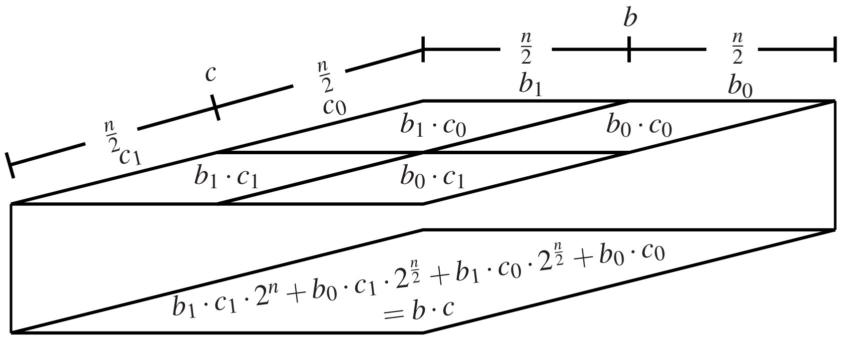

b
n/2
n/2
b1
b0
c
n/2
c0
n/2
c1
b1 · c0
b0 · c0
b1 · c1
b0 · c1
b1 · c1 · 2^n + b0 · c1 · 2^(n/2) + b1 · c0 · 2^(n/2) + b0 · c0
= b · c

Figure 3.11: Real-valued partial products. The product $b \cdot c$ is formed by multiplying and adding the subwords b_0, b_1 and c_0, c_1 according to their bit significance. b and c are of word length n.

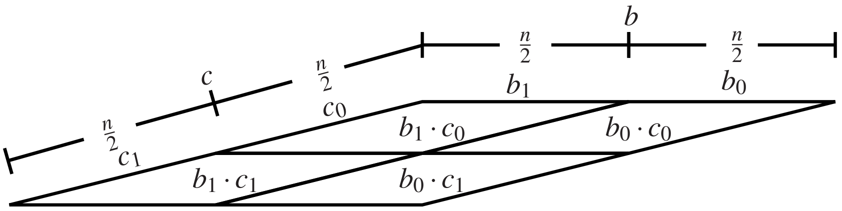


Figure 3.12: Complex-valued partial products. Complex-valued multiplication: $b \cdot c = (b_1 \cdot c_1 - b_0 \cdot c_0) + i(b_1 \cdot c_0 + b_0 \cdot c_1)$. The complex-valued product $b \cdot c$ is formed by multiplying real subwords b_1 and c_1 and the imaginary subwords b_0 and c_0.

This multiplication scheme is depicted in Figure 3.13 for subwords of 8-bit, 16-bit or 32-bit width. All 8-bit products are computed within the first stage of the product matrix. The white colored cells, which are outlined bold on the diagonal axis, either contain the real-valued products, parts of the complex-valued results or are used for partial products. The gray hatched cells contain either parts of the complex-valued products or partial products. All other cells are not used to generate partial products or results. Products that are computed by partial products are marked with a plus sign. All other products are computed directly by dedicated multipliers. These are marked with a multiplication sign. This implementation scheme reduces hardware costs.

Before the products are fed to the accumulator stage, the complex-valued multiplication results must be computed. Equation 3.5 indicates which products of the product matrix shown in Figure 3.12 must be added and subtracted to compute the complex-valued multiplication.

$$b \cdot c = (b_1 \cdot c_1 - b_0 \cdot c_0) + i \cdot (b_1 \cdot c_0 + b_0 \cdot c_1) \tag{3.5}$$

By using an additional SIMD adder and subtractor, the complex-valued multiplication can be computed with the same multipliers used for real-valued multiplications. This is shown in Figure 3.10. For the additional adder and subtractor stage, a ripple carry adder architecture is used as shown in Figure 3.14. Eight 8-bit adders connected by a carry chain can compute eight 8-bit additions, four 16-bit additions, two 32-bit additions or one 64-bit addition by controlling the propagation of the carry bit.

The last accumulator stage in Figure 3.10 adds the outputs from the product matrix or the SIMD adder/subtractor to the accumulator inputs $a0$ and $a1$. The outputs of the partial product matrix can be either real- or complex-valued and are multiplexed for each operation listed in Table 3.2. The accumulator stage uses the same ripple carry SIMD adder architecture shown in Figure 3.14. One of the adders can be configured as a subtractor to compute butterfly operations according to Equation 3.3 and Equation 3.4.

Case Study: FFT

The fast Fourier transform (FFT) algorithm is used as a benchmark to evaluate the performance of proposed real and complex SIMD-CMAC unit and other MAC architectures presented in related works. The fast Fourier transform is used to compute the discrete Fourier transform defined by Equation 3.6.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-2i\pi \frac{kn}{N}} \quad k = 0, \ldots, N-1 \tag{3.6}$$

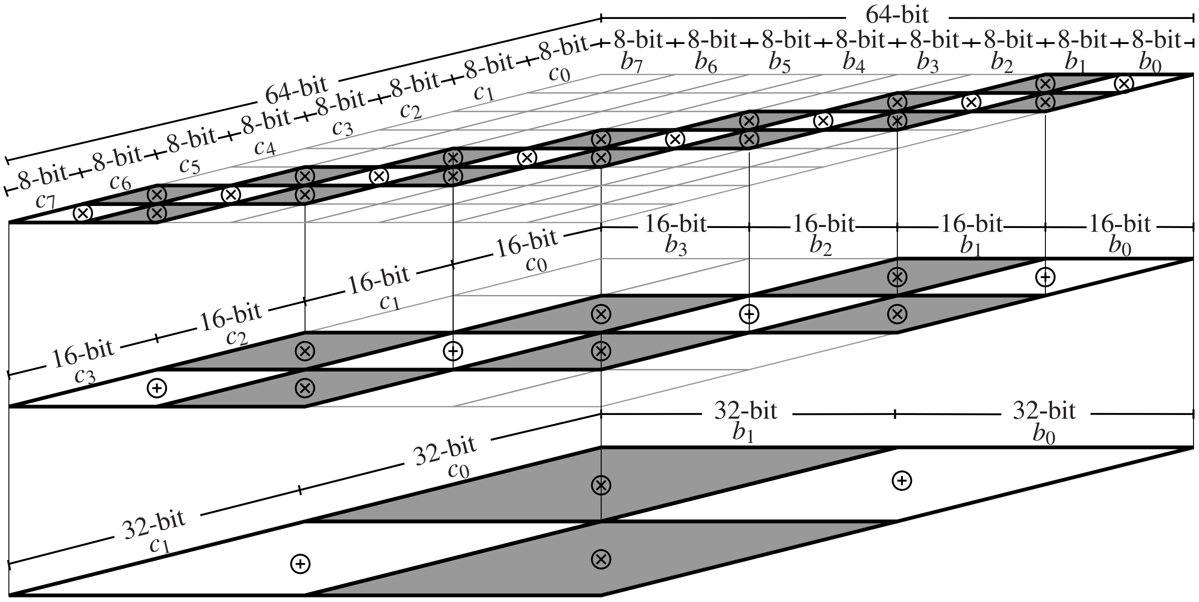


Figure 3.13: Partial product matrix. The product matrix consists of three stages. The products are generated from the products of the previous stage. A product sign indicates results generated by direct multiplication, while a plus sign indicates products generated by addition of partial products. The results are used for real (white cells) and complex-valued (white or gray cells) multiplications.

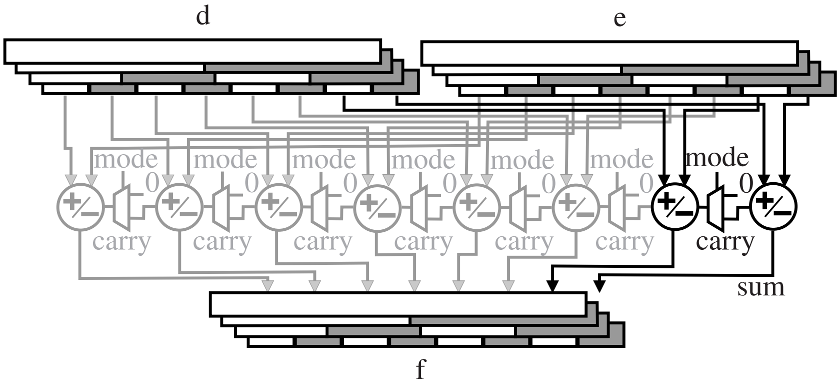


Figure 3.14: Ripple carry SIMD adder architecture. Different subword modes are depicted with either white or gray subwords.

The transformation is computationally intensive. The Cooley-Tukey FFT [135] algorithm is usually used to reduce the number of complex multiplications and additions. A decimation in time (DIT) radix-2 FFT is shown in Figure 3.15.

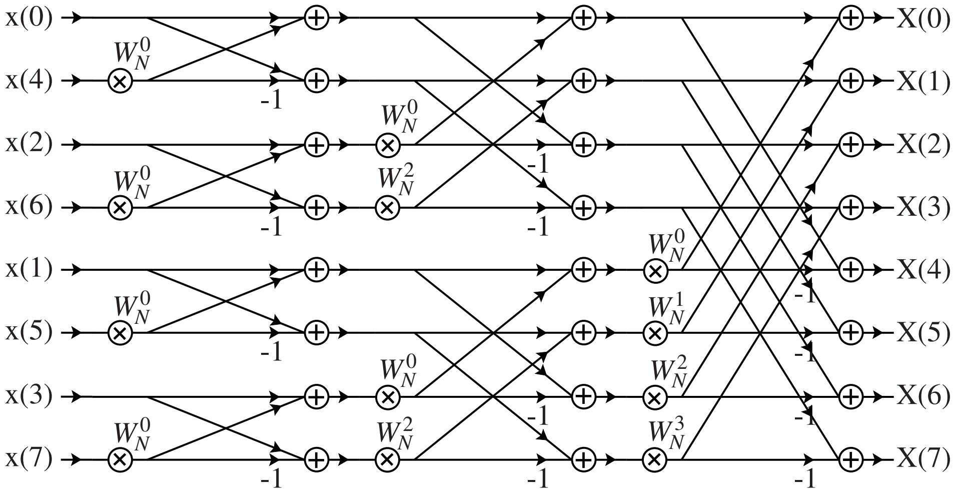


Figure 3.15: Radix-2 decimation in time FFT. The input samples x are transformed by size-2 DFTs (butterflies) to the output signal X. The constant multiplication factors W are twiddle factors.

In the literature, various complex-valued radix-2/4 Cooley-Tukey FFT algorithms have been implemented on a variety of different DSPs with different MAC architectures [92, 97, 100, 119, 123, 126, 129, 131–133, 136]. The processing performance of the proposed SIMD-CMAC unit is determined by integrating the unit into the execution pipeline stage of the *KAVUAKA* VLIW-SIMD processor. The SIMD-CMAC unit computes butterflies of a radix-2 decimation in time FFT algorithm. The FFT processing performance for differently sized FFTs is listed in Table 3.3.

Those architectures marked as *programmable* are flexible. Their MAC unit processes real- and complex-valued operations and can therefore be used for algorithms other than the FFT. The proposed SIMD-CMAC architecture requires fewer cycles to compute the FFT compared to other programmable MAC architectures. The reason for this is the full SIMD support for butterfly operations. In addition to the parallel execution of butterflies, no permutation and alignment operations are required, compared an implementation with a real-valued MAC. The performance of the FFT algorithm also depends on local memory bandwidth. In the case of the *KAVUAKA* processor, the acceleration using the SIMD-CMAC instead of a standard SIMD-MAC is about 3.7 for the same local memory interface. These architectures, marked *dedicated*, are equipped with dedicated hardware architectures, such as dedicated butterfly units and local memory banks for twiddle and sample data. These hardware mechanisms provide higher performance compared to the more flexible *programmable* architectures.

Table 3.3: Radix 2/4 Complex FFT Performance of proposed and related Architectures

	MAC Arch.	**Name**	**Clock frequency [MHz]**	**Number of MAC** Units	**Word/Subword [bit]**	**FFT Points**					
						32	**64**	**128**	**256**	**512**	**1024**
						cycles and percentage deviation compared to the *KAVUAKA* processor					
programmable	**Single MAC**	CFX [129]	300	2	24/24	997 (697%)	1657 (496%)	4129 (503%)	8393 (438%)	18654 (424%)	41385 (416%)
		C55x [97]	200	2	16/16	—	—	—	4786 (250%)	—	—
		Hinrichs [92]	40	4	16/16	—	—	—	—	—	14440 (145%)
		ADSP-21161N [136]	100	2	32fl	—	1156 (346%)	2158 (263%)	4316 (225%)	8770 (199%)	18288 (184%)
	SIMD MAC	Arm [100]	50	2	32/32	—	—	2700 (329%)	—	—	—
		C674x [123]	456	2	32/32	258 (191%)	545 (163%)	953 (116%)	2216 (116%)	4664 (106%)	10055 (101%)
		Nadehara [119]	200	1	64/16	— —	839 (251%)	— —	4093 (214%)	— —	19257 (193%)
		SC3850 [131]	1000	4	64/16	212 (157%)	525 (157%)	1273 (155%)	2587 (135%)	5854 (133%)	11898 (119%)
	SIMD CMAC	**KAVUAKA**	**50**	**1**	**64/32**	**135** **(100%)**	**334** **(100%)**	**821** **(100%)**	**1915** **(100%)**	**4397** **(100%)**	**9959** **(100%)**
dedicated	**Specialized CMAC**	Al [132]	-	1	16/16	—	—	—	1024 (53%)	—	—
		Lee [126]	144	1	-/-	—	—	—	1536 (80%)	—	7680 (77%)
		Liu [133]	320	2	32/16	—	—	284 (35%)	568 (30%)	1188 (27%)	2496 (25%)

The cell area and power consumption overhead for implementing a complex-valued SIMD-CMAC unit is shown in Table 3.4. Three different variants of the proposed SIMD-CMAC unit are synthesized using a 40 nm low-power technology library of TSMC [137]. No pipelining is used, the maximum clock frequency is set to 50 MHz and the operating voltage is 1 V. This configuration is based on a low-power design flow. The total power consumption is an estimate based on a switching factor of 0.5. The implementation *Imp1* is synthesized without complex-valued operations and represents a standard SIMD-MAC unit. This implementation is used as a reference. If complex-valued multiply and multiply-accumulate operations are additionally synthesized, the area overhead is about 70 %. This overhead is 30 % smaller compared to the duplication of MAC units as proposed in [96]. The SIMD butterfly operation, implemented in the *Imp3* variant, requires only 4 % additional cell area.

Table 3.4: Cell count, cell area and total average power of different implementations of the proposed SIMD-CMAC architecture. Results from an ASIC synthesis with the TSMC 40 nm HVT low-power technology at 50 MHz.

Implementation	Cell count	Cell Area [μm^2]	Est. Total Power [mW]
Imp1	7417 (100%)	12515(100%)	0.21 (100%)
Imp2	12434 (168%)	21263(170%)	0.38 (181%)
Imp3	12838 (173%)	21730 (174%)	0.39 (186%)

Imp1: Real-valued MUL and MAC operations implemented.
Imp2: Real-valued and complex-valued MUL and MAC operations implemented.
Imp3: Real-valued and complex-valued MUL and MAC operations and butterfly operation implemented.

3.2.2 Efficient Emulation of Floating-Point Arithmetic

Rational numbers are represented as fixed-point and floating-point numbers in embedded digital signal processing systems. The hardware architecture of these systems defines whether the floating-point number format is supported or not. It depends on many design criteria whether floating-point hardware support is required or better suited for a specific system and application compared to a pure fixed-point implementation. Typical design criteria include computational accuracy, time to market, ease of use, hardware cost, power consumption, and processing performance.

The hardware complexity of fixed-point processors is lower compared to floating-point processors. However, the development time of fixed-point applications is considerably higher [138].

Regardless of the programming language, whether it is a high-level programming language or low-level assembler code, the design of fixed-point code requires additional development time. This additional development time required for a floating-point to fixed-point conversion takes up to 30 % of the total development time [139]. Writing fixed-point code is a challenge [138–140] because fixed-point code is very susceptible to quantization effects. This development requires a fixed-point analysis/simulation and rewriting of the code. Fixed-point analysis is a procedure to determine the number of fractional bits and thus the range for each fixed-point variable used in the code. These analyses, which use an analytical or simulation-based approach, require additional development time to analytically determine the fixed-point formats or to set up and use the simulation environment [141]. In a simulation-based approach, variable ranges are determined by creating a histogram for each variable based on a given input to the algorithm.

There are cases where the analysis based on the fix-point simulation cannot provide reliable results [141]. This is the case when the worst-case scenario for the input is unknown or not representative. This leads to a faulty fixed-point implementation. Errors caused by rounding or overflow errors can lead to unpredictable system behavior.

Therefore, in some applications or parts of them it is desirable to shorten the development time by using floating-point emulation on low-cost fixed-point processors [142, 143]. Although floating-point emulation on fixed-point processors introduces run-time penalties [144], it is still applicable if the required processing performance and power consumption are maintained, which is only the case if floating-point emulation is efficient and fast. For those parts, that use floating-point arithmetic, fixed-point analysis is not required. Another advantage of the floating-point implementation is the reduced code size for frequently used subroutines. Since the fixed-point code is written for a specific input range, it contains shift instructions for scaling fixed-point values. A fixed-point subroutine must be copied and adapted for each application and input range. In contrast, the floating-point implementation of the same subroutine can be used for almost any input range.

This section introduces a software library for floating-point emulation for fixed-point processors with SIMD instructions. This library uses the available fixed-point SIMD commands and data types to increase emulation speed. The main feature of this library is that the significand and exponent of the floating-point format can be stored in different subwords of unequal size of the same register word. These subwords can then be processed independently with the available multi-precision fixed-point instructions SIMD. It is not necessary to unpack or pack the floating-point numbers before or after processing, as is done in the corresponding libraries. The presented floating-point emulation is therefore comparatively fast. No additional hardware overhead is required, since only hardware features of the fixed-point microSIMD architecture are used.

The related floating-point emulation architectures are described in Section 3.2.2, and the proposed architecture is presented in Section 3.2.2. Two case studies for the proposed architecture are presented in Section 3.2.2 and Section 3.2.2.

Related Hardware and Software Floating-Point Emulation

A software floating-point implementation, called Berkeley SoftFloat, is presented in [145]. This implementation supports binary floating-point numbers that conform to the IEEE 754 standard for floating-point arithmetic for 16-bit half-precision, 32-bit single-precision and 64-bit double-precision, 80-bit double-extended-precision up to 128-bit quadruple-precision. Proposed floating-point functions include addition, subtraction, multiplication, division, and square root.

A floating-point emulation library for integer processors is described in [146]. This library is derived from the SoftFloat [145] library. For an optimized implementation on a VLIW processor, some of the rounding modes and the below-average numbers have been removed from the standard. Other optimizations include intrinsic functions, like a count leading zeros (CLZ) function. Compared to the SoftFloat library, an acceleration of about 2.54 is achieved.

A custom software floating-point format for RISC processors that do not have dedicated floating-point units is described in [147]. This custom floating-point format is based on the IEEE 754 floating-point standard. The target applications are automotive control systems. The performance requirements for these applications are high. Features of the IEEE 754 standard are removed or simplified in this work for performance reasons. Examples for such simplifications are the unification of the positive and negative zero representation and the use of an explicit leading bit for the significand. The average acceleration achieved by this custom floating-point format over the IEEE 754 floating-point standard is 2.1. Due to the lower complexity of the custom format, the code size is reduced by 55 %.

The Blackfin processor platform [143] is a low-power fixed-point processor. Two libraries are available for floating-point emulation, one of which is compatible with the IEEE 754 floating-point standard. The second one is optimized for speed by reducing the complexity of the required computations. An essential feature of this library is the data format for handling floating-point types. The significand and the exponent are stored in different words. This format is called two-word format. Compared to other libraries like the in [146] the computational complexity is reduced, because no unpacking or packing operations have to be performed to extract or merge the exponent and the significand.

In [148], numerical linear algebra algorithms are implemented on fixed-point DSPs due to lower hardware costs. Processing speed is increased by using all available fixed-point units and by applying application-specific code optimizations and conversion techniques for the *TMS320C6000* DSP family [149].

The HiFi4 DSP family [121] is designed for audio processing. The processor contains four VLIW slots and can execute four 32-bit fixed-point MACs per cycle. An application-specific floating-point library is optimized for low processing times and also supports optimized vector modes. Special numbers such as zero, positive, and negative infinity are represented in a custom format.

The floating-point emulation software library introduced in [146] is optimized in [150] to accelerate floating-point emulation. The target platform is a VLIW processor. Application-specific numerical blocks are identified to define new non-generic floating-point operators in the software. These operators include merged operators that combine two floating-point operations into a single operation. Paired operators compute two operations on the same input in parallel. These non-generic floating-point operations are then verified with a VLIW processor. For a benchmark suite with typical DSP code an acceleration of 1.59 is achieved compared to [146].

So-called hybrid floating-point hardware modules (HFPM) are introduced in [151]. These modules speed up the computation of software floating-point emulation operations. The application software is written in fixed-point format, while the HFPMs add custom floating-point instructions to the available instruction-set. The combination of fixed-point software and hybrid floating-point hardware modules increase the performance of floating-point emulation. Compared to standard hardware floating-point units, these HFPMs reduce the area by 1.08 to 12.5. The throughput of the floating-point emulation per area is increased by 1.05 to 8.5.

In [152] and [153] so-called virtual floating-point hardware units are introduced. These virtual hardware modules are based on several existing fixed-point arithmetic units, which are combined to a virtual floating-point unit. Only a few hardware extensions are required to process floating-point numbers. This approach offers a higher floating-point emulation performance than the pure software approach with fixed-point arithmetic. With these hardware extensions, bit-level manipulations can be performed more efficiently. A block floating-point implementation (BFP) is used to reduce the latency of virtual floating-point units. The area of these units is about 24 % smaller than full floating-point units. The power is reduced by 30 %.

In [154] custom floating-point hardware modules are presented. These modules support various floating-point formats that differ in bit-width. The bit-width ranges from 8-bit to 64-bit. The IEEE 754 single (32-bit) and double (64-bit) precision floating-point formats are also supported. The variable bit-width allows range, latency and throughput to be optimized for a target application. Fixed-point hardware modules are also presented. A hybrid design with fixed-point and floating-point modules in one architecture is proposed. The conversion between fixed-point and floating-point format is performed by additional conversion hardware modules. This can increase the overall efficiency of the processing system.

An architecture for a floating-point adder is presented in [155]. This adder can either perform two single-precision additions in parallel (SIMD) or a double-precision floating-point addition. Hardware resources such as comparator units, shift units and a leading single detector (LOD) are shared to efficiently implement both floating-point precision modes. Based on the results of the ASIC synthesis, a 35 % reduction of the silicon area is achieved compared to an architecture with one double-precision adder and two single-precision adders.

The work presented in [156] aims at reducing the execution latency of floating-point hardware units. The proposed high-throughput floating-point unit and instruction scheduling techniques allow early execution of a floating-point addition operation that depends on other floating-point addition or multiplication operations. The latency of floating-point operations can thus be reduced. The performance of a benchmark suite is increased by an average of 7.6 %.

Proposed SIMD-Based Software Floating-Point Format

The software floating-point emulation arithmetic for SIMD-based architectures is described in this chapter. In these SIMD-based architectures, a register can contain subwords of different word lengths. These architectures are described in [157]. The subword formats used in this thesis are shown in Figure 3.16. The register bit-width is 64-bit. Therefore, either one 64-bit subword, two 32-bit subwords, four 16-bit subwords, or eight 8-bit subwords fit into one register. The subwords are processed in parallel by the functional units of the processor. The subword execution mode, which defines the number of subwords per register, is controlled by the processor's instruction-set. The instructions for controlling the functional units can be defined for each subword mode. For example, there is an ADD_32 instruction to add 32-bit subwords and an ADD_8 instruction to add 8-bit subwords. The flexibility of these microSIMD processors is used here for efficient software floating-point emulation. The significand, exponent and sign bit of a floating-point number are stored in one register and processed independently with different subword modes.

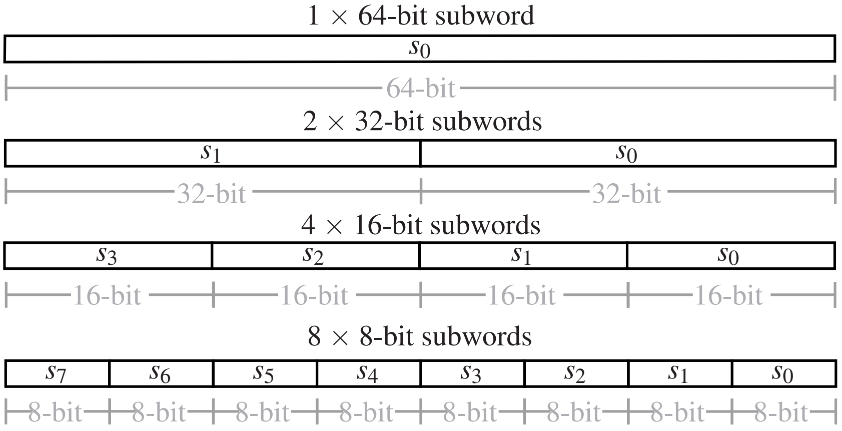


Figure 3.16: microSIMD data format. Each 64-bit word consists of either 64-bit, 32-bit, 16-bit or 8-bit subwords.

The IEEE 754 [158] single-precision floating-point format for 32-bit data is shown in Figure 3.17. It consists of a sign-bit, a biased exponent (8-bit) and a trailing significand field

(23-bit). The floating-point value for the base of 2 can be computed with Equation 3.7:

$$f = (-1)^S \cdot 2^{E-bias} \cdot (1 + 2^{1-24} \cdot T) \qquad \text{if } 1 \leq E \leq 2^8 - 2 \tag{3.7}$$

where S represents the sign, T is the significand and E is the exponent.

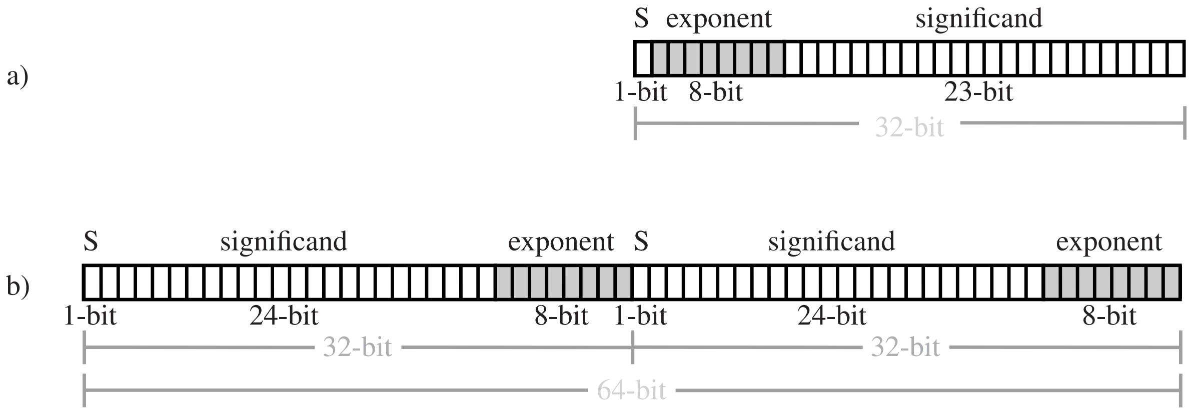


Figure 3.17: (a) IEEE 754 single-precision floating-point [158] format and (b) proposed floating-point format for fixed-point microSIMD processors.

In order to enable efficient emulation of floating-point signal processing in terms of processing performance, the 754 [158] single-precision floating-point format is simplified. The proposed format, shown in Figure 3.17, is similar to the formats presented in the related work Section 3.2.2. Subnormal numbers were removed. Rounding to zero and infinity is possible. The significand and the exponent are stored in the two's complement number representation. The sign-bit is part of the significand. The 8-bit representing the exponent is shifted to the least significant bits, while the significand is shifted to the most significant bits. With this format, the floating-point value can be computed with Equation 3.8:

$$f = T \cdot 2^E \tag{3.8}$$

where T is the significand and E is the exponent.

This floating-point format is used because the significand and exponent of the floating-point numbers are aligned with the subwords of the microSIMD register format. Therefore, the available microSIMD instructions can be used to process floating-point numbers efficiently without packing and unpacking the significand and exponent before and after processing, as suggested in [146]. Two floating-point numbers are stored in a 64-bit register in the case of the *KAVUAKA* processor architecture to process the floating-point numbers with SIMD and reduce the number of registers required. In contrast, two 32-bit registers are used to store a floating-point value in [143].

The floating-point emulation operations are described below. These operations include multiplication, addition, and subtraction. The intermediate computations required for a floating-point multiplication are shown in Figure 3.18. This figure shows the computations for only one of the two floating-point numbers that were processed in parallel (SIMD) during a *FP_MUL_-32* operation. To compute this floating-point multiplication, the significands are multiplied together while the exponents are added. The resulting significand is then normalized and the exponent is updated based on this normalization to obtain the correct multiplication result. During normalization, rounding to zero or infinity is possible.

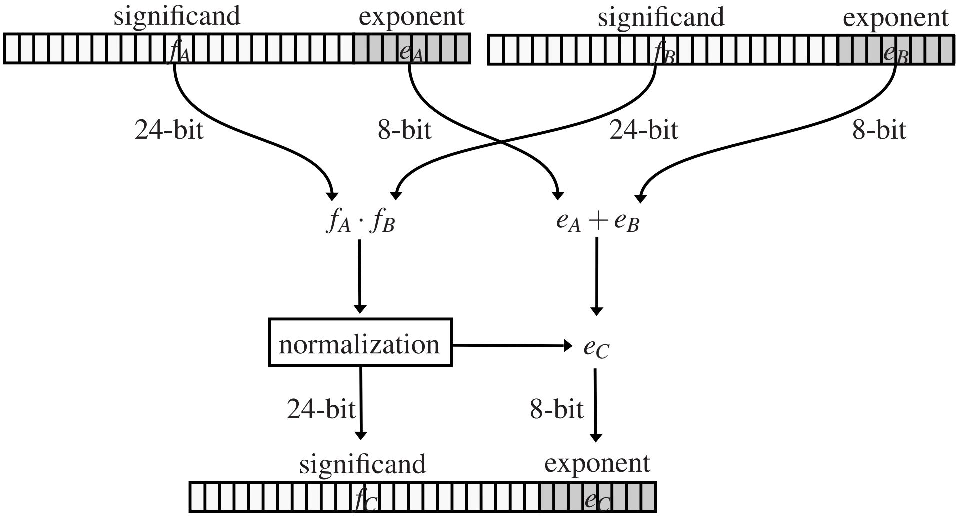


Figure 3.18: Emulated floating-point multiplication for microSIMD processors.

Assembler instructions for emulating floating-point multiplication are listed in Figure 3.19. These instructions, combined as macro calls, compute the multiplication of two floating-point numbers in parallel. The exponents computed in two's complement are added together with a *ADD_8s* instruction in line 7. This instruction saturates the result in 8-bit subword mode. The significands are multiplied (32-bit) after the least significant eight bits have been set to zero by the *PERMREG* instructions (lines 10–11). For normalization a *CLX* instruction is used, which counts leading zeros or ones. The exponent is updated based on the normalization and moved to the target register (line 28).

The emulation of floating-point addition and subtraction operations is shown in Figure 3.20. For a floating-point addition or subtraction, the significands of both numbers must be transformed into the same fixed-point range. For this, the absolute difference of the exponents is computed. The significance of the smaller number is shifted to the right by the absolute difference of the exponents. The larger of the two exponents is used for the result. The significands are then added or subtracted from each other. After normalizing the significand, the exponent is updated and the floating-point addition or subtraction is computed.

```
// ****************************************************
// emulated floating-point multiplication (24:8 format)
// ****************************************************
MACRO FP_MUL_32 DST, OP1, OP2

  // add exponents
  ADD_8s      TEMP0, OP1, OP2

  // isolate and multiply significands
  PERMREG1_8  xOP1, OP1, R_ZERO
  PERMREG1_8  xOP2, OP2, R_ZERO
  MACPLZ_32   MAC_H, xOP1, xOP2

  // normalize resulting significand
  CLX_64    TEMP2, MAC_H, MAC_H
  SL_64     MAC_H, MAC_H, TEMP2

  CLX_64    TEMP3, MAC_L, MAC_L
  SL_64     MAC_L, MAC_L, TEMP3
  MIXL_32   TEMP1, MAC_H, MAC_L

  // compute new (resulting) exponent
  MIXR_32   TEMP2, TEMP2, TEMP3
  SUBI_32   TEMP2, TEMP2, #1
  SUB_8s    TEMP2, TEMP0, TEMP2

  // recreate original format
  PERMREG1_8  DST, TEMP1, TEMP2

ENDMACRO
```

Figure 3.19: Assembler instructions for the emulation of a floating-point multiplication using SIMD instructions.

The assembler instructions for emulating floating-point additions are shown in Figure 3.21. Here the exponents are subtracted from each other (line 9). Depending on the result of this subtraction, one of the significands is selected for the shift operation before adding the significands (line 19). The normalization is done by counting the leading zeros or ones in line 30. The resulting register is then rebuilt from the updated exponents and the normalized significand.

The presented software floating-point operations are part of a floating-point emulation library. This library consists of floating-point operations and conversion macros to incorporate floating-point operations into fixed-point algorithms. To evaluate the efficiency and performance of the proposed library, it is compared to related libraries. The library is implemented on the *KAVUAKA* processor VLIW-SIMD. The *KAVUAKA* processor executes two instructions per cycle. The data width of *KAVUAKA* is 64-bit.

For comparison the performance results shown in [121, 143, 145, 146, 148, 150] are used. The *ST231* VLIW processor is used in [150] to evaluate the libraries presented in [145] and [146]. The *ST231* VLIW processor does not support hardware floating-point operations. The *ST231*

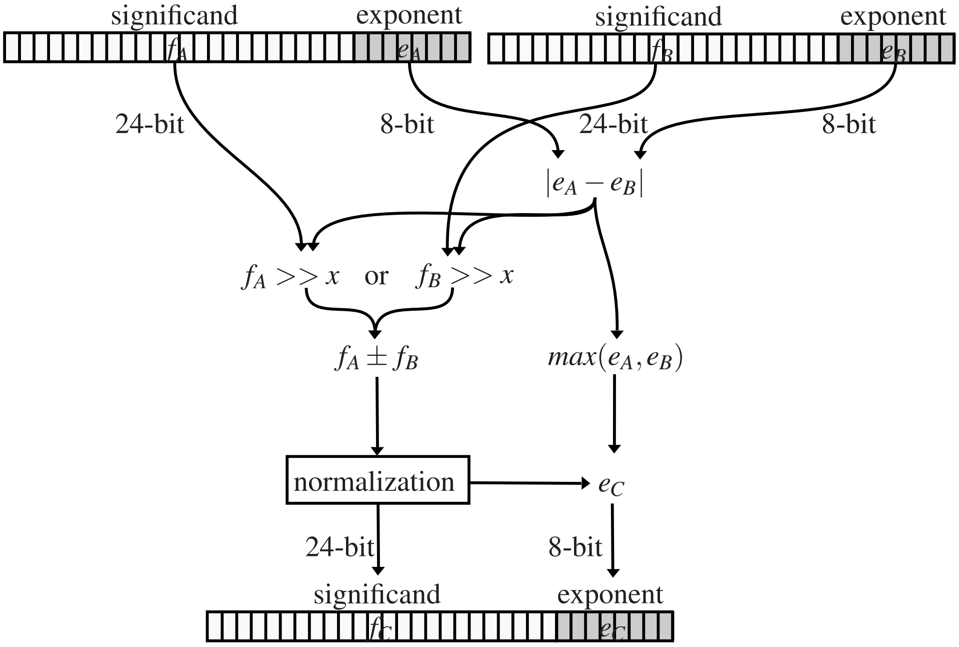


Figure 3.20: Emulated floating-point addition and subtraction for microSIMD processors.

can execute up to four instructions per cycle, and the data width of the *ST231* processor is 32 bits. The *Blackfin* processor [143] consists of a 16-bit fixed-point DSP and a 32-bit RISC processor. The processor supports SIMD instructions. There are two libraries for emulating floating-point operations. A high-performance library and a library that conforms to the IEEE 754 floating-point standard. In [148] the authors use the *TMS320C6000* DSP family. The authors developed a target-optimized DSP floating-point emulation library for the *TMS320C64x+* fixed-point processor. With VLIW, up to eight 32-bit instruction are executed in parallel. SIMD is supported and the data width is 64-bit.

Table 3.5 shows the cycle counts for addition, subtraction, multiplication, and addition/subtraction software floating-point operations of the above software libraries and processors. As can be seen from the table, the proposed software floating-point library requires fewer cycles to compute the operations compared to the corresponding libraries. When using the proposed microSIMD operations, the number of cycles per operation is halved. For example, two floating-point additions are computed within 13 cycles.

Case Study I: Digital Filter

Two common algorithms from the field of audio processing were implemented with the proposed software floating-point library as case studies. The first algorithm is a finite impulse

```
// ****************************************************
// emulated floating-point addition (24:8 format)
// ****************************************************
MACRO FP_ADD_32 DST, OP1, OP2

  // compare exponents
  PERMREG0_8  xOP1, OP1,  OP2  // E0|E0|E2|E2|E1|E1|E3|E3
  PERMREG0_8  xOP2, OP2,  OP1  // E2|E2|E0|E0|E3|E3|E1|E1
  SUBCS_8s    TEMP0, xOP1, xOP2 // exponent difference

  // swap significands according to magnitude of exponents
  // (choose number with smaller exponent)
  MV          TEMP1, OP1
  MV          TEMP4, OP2
  MVCR_32     TEMP1, OP2
  MVCR_32     TEMP4, OP1

  // remove exponent (for significand computation)
  SRI_32      TEMP5, TEMP1, #8

  // right shift of secondary significand operand
  ABSADD_8    TEMP0, R_8, TEMP0
  CLIPI_U8    TEMP0, TEMP0, #0b11111
  SR_32       TEMP0, TEMP4, TEMP0

  // add significands
  ADD_32      TEMP5, TEMP5, TEMP0

  // normalize resulting significand
  CLX_32      TEMP0, TEMP5, TEMP5
  SL_32       TEMP5, TEMP5, TEMP0

  // compute new exponent
  SUB_8s      TEMP0, R_8, TEMP0
  ADD_8s      TEMP0, TEMP1, TEMP0

  // recreate original format
  PERMREG1_8  DST, TEMP5, TEMP0

ENDMACRO
```

Figure 3.21: Assembler instructions for emulating the floating-point addition using SIMD instructions.

Table 3.5: Cycle count comparison of different software emulated floating-point operations. The number within the brackets represents the number of cycles if SIMD is not used.

Cycles	addition	subtraction	multiplication	addsub
ST231/SoftFloat [145, 150]	48	49	31	—
ST231/FLIP [146, 150]	26	26	21	—
ST231/Jean. [150, 159]	26	26	21	26
Blackfin/fast-fp [143]	76	76	71	—
C64x+ [148]	66	66	69	—
HiFi4 [121]	23	23	13	—
KAVUAKA	6.5 (13)	7.5 (15)	5 (10)	8.5 (17)

response (FIR) or infinite impulse response (IIR) filter [63] and the second algorithm is the fast Fourier transform (FFT) [63, 135]. Both algorithms are evaluated for performance and precision.

Digital filters are systems that perform mathematical operations on signals. Common filter types are low-pass, band-pass, high-pass, or all-pass. The operations can be expressed by the difference Equation 3.9:

$$y(n) = \sum_{i=0}^{M} b_i \cdot x(n-i) - \sum_{j=1}^{N} a_j \cdot y(n-j) \tag{3.9}$$

where x is the input signal and y is the output signal. a and b are the filter coefficients for implementing filters with a finite impulse response (FIR) and a infinite impulse response (IIR). A direct implementation of these filters in form II is shown in Figure 3.22. These filters are sensitive to quantization errors and are computationally intensive, because they repeatedly work with the same data values [140].

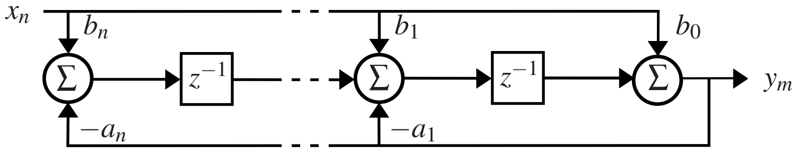


Figure 3.22: Direct form II transposed implementation of a digital filter.

This digital filter was implemented on the *KAVUAKA* processor in fixed-point and emulated floating-point. The number of cycles, code size, and instructions per cycle (IPC) of the same IIR filter with fixed-point arithmetic and software floating-point emulation are listed in Table 3.6. The number of cycles is increased by a factor of 3.9 × for the floating-point implementation. Although the code size of the floating-point implementation is larger, the floating-point code can be used as a subroutine for several different filter implementations. The fixed-point implementation is only suitable for exactly one filter implementation, because fixed-point arithmetic, e.g., including shifts, is fixed for a given filter. Therefore, the code size doubles for each filter instance, unless variable shifting operations are implemented, which reduce processing performance.

Compared to the reference implementation with double-precision floating-point (IEEE), quantization errors occur when the filter is computed with single-precision floating-point (IEEE), emulated floating-point, or fixed-point. The absolute errors compared to the double-precision floating-point implementation are shown in Figure 3.23 for 100 random input vectors of length 1000. The mean and maximum absolute errors for the different data types are listed in Table 3.7. The 32-bit fixed-point IIR filter implementation has a smaller quantization error than the single-precision floating-point implementation. This is because the fixed-point data type has more fractional bits and the range for each variable was determined by a fixed-point analysis.

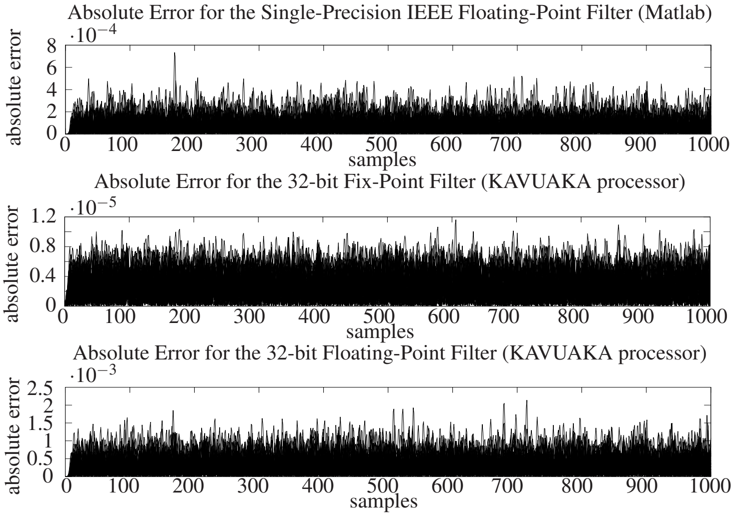
Table 3.6: Cycle count, code size and instructions per cycle (IPC) for a fixed-point and a floating-point filter

Format	**Cycle Count**	**IPC**	**Code Size**
32-bit fixed-point	141	1.52	1072 byte
32-bit floating-point	550	1.69	1984 byte

Table 3.7: Precision comparison between single and double-precision IEEE floating-point, fixed-point and emulated floating-point digital filter implementations

Format	**Mean Absolute Error**	**Maximum Absolute Error**
32-bit single-precision floating-point (*Matlab*)	7.3141e-05	7.3491e-04
32-bit fixed-point (KAVUAKA)	2.8524e-06	1.1622e-05
32-bit floating-point (KAVUAKA)	3.1342e-04	2.1398e-03

The dynamic power consumption for a 17-tap FIR filter using emulated floating-point at 13 MHz is 0.8135 mW.



Absolute Error for the Single-Precision IEEE Floating-Point Filter (Matlab)

Absolute Error for the 32-bit Fix-Point Filter (KAVUAKA processor)

Absolute Error for the 32-bit Floating-Point Filter (KAVUAKA processor)

Figure 3.23: Absolute errors for the digital IIR filter implementation with single-precision IEEE floating-point, fixed-point and emulated floating-point compared to double-precision IEEE floating-point.

Case Study II: Fast Fourier Transform

The fast Fourier transform is used to compute the discrete Fourier transform [135] defined by Equation 3.10.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-2i\pi\frac{kn}{N}} \quad k = 0,\ldots,N-1 \tag{3.10}$$

The algorithm transforms the input signal x to the frequency domain X by computing a series of complex-valued multiplications and additions defined in Equation 3.10.

The number of cycles, code size, and instructions per cycle (IPC) for the FFTs with fixed-point arithmetic and user-defined floating-point emulation are listed in Table 3.8. The number of cycles increases by a factor of 8.3 × for the proposed floating-point version. The instructions per cycle (IPC) is high for both implementations, because the maximum IPC value for the *KAVUAKA* processor is 2. The instruction parallelism of the processor can therefore be used efficiently in these implementations. The code size of the floating-point emulation implementation increases by a factor of 7.5 × .

The mean and maximum absolute errors compared to a IEEE floating-point implementations with double-precision in *Matlab* are listed in Table 3.9 and shown in Figure 3.24. With an optimized fixed-point implementation and shifts after each stage of the FFT for scaling, the total quantization error is smaller for the fixed-point implementation.

Table 3.8: Cycle count, code size and instructions per cycle (IPC) for fixed-point and floating-point FFTs

Format	**Cycle Count**	**IPC**	**Code Size**
32-bit fixed-point	451	1.9	4056 byte
32-bit floating-point	3765	1.7	30592 byte

Table 3.9: Precision comparison between double-precision IEEE floating-point, fixed-point and emulated floating-point FFT implementations

Format	**Mean Absolute Error**	**Maximum Absolute Error**
32-bit single-precision floating-point (*Matlab*)	1.8709e-07	6.6534e-07
32-bit fixed-point (KAVUAKA)	4.0799e-08	1.3907e-07
32-bit floating-point (KAVUAKA)	7.0870e-07	4.1363e-06

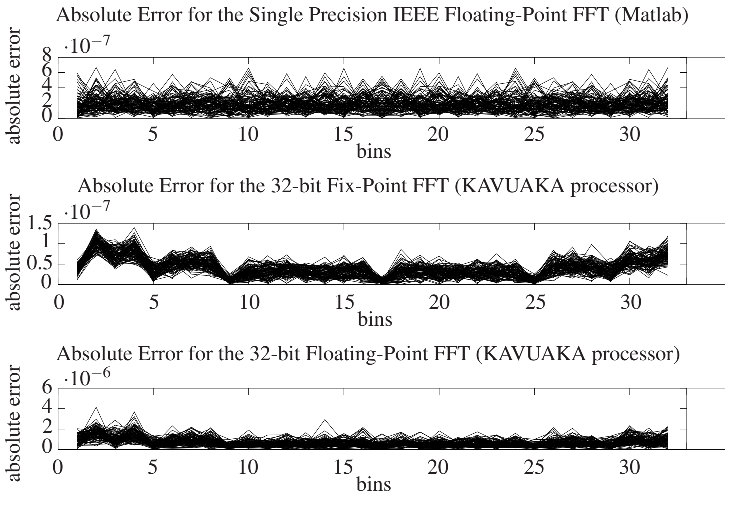


Figure 3.24: Absolute errors for the FFT implementation with single-precision IEEE floating-point, fixed-point and emulated floating-point compared to double-precision IEEE floating-point.

3.2.3 Tightly Coupled Co-Processors

Complex arithmetic functions, including trigonometric or hyperbolic functions such as the computation of sine or cosine, require a considerable number of computing cycles on instruction-driven von Neumann central processing units [160]. A common approach is therefore to integrate data-driven hardware accelerators in addition to the CPU.

For hardware integration, the coordinate rotation digital computer (CORDIC) [161] algorithm and a non-restoring division algorithm are chosen. A variety of elementary functions, that are part of the hearing aid applications, including hyperbolic and trigonometric arithmetic functions, can be computed with these hardware accelerators. These functions include division, sine, cosine, tangent, arctangent, hyperbolic sine, hyperbolic cosine, hyperbolic tangent, hyperbolic arctangent, natural logarithm, exponential, and square root. A fast and accurate calculation compared to other approximation algorithms is possible with reduced memory requirements compared to lookup table (LUT) interpolation algorithms [161]. Algorithms with high memory requirements are excluded due to limited on-chip memory for hearing aid processors. In comparison to related iterative algorithms [162], such as Newton's method, which generally do not end after a given number of iterations, which is undesirable in a real-time system, the CORDIC algorithm ends with a predictable run-time [163], where the maximum number of iterations depends on the size of the input.

A radix-2 CORDIC [161], based on the architecture described in [164, 165], and a non-restoring division hardware accelerator based on the architecture described in [86, 87] are coupled as co-processors to *KAVUAKA*.

This chapter examines the architecture, processing performance, and silicon area requirements of the co-processors with special emphasis on the coupling of the co-processor to *KAVUAKA*, the main processor (CPU). Therefore, both algorithms are implemented as hardware accelerators and as a pure software library on the *KAVUAKA* processor. Taking into account the data transfer between the processor and the co-processor, the degree of acceleration using the hardware co-processors is investigated. Furthermore, the silicon area and the power consumption of the hardware co-processor are considered.

CORDIC Co-Processor Architecture

The co-processor consists of a CORDIC module, an iteration controller, an output register, and an angle table as shown in Figure 3.25. The CORDIC module iteratively computes Equation 3.11:

$$\begin{aligned} x_{i+1} &= x_i - \sigma_i \cdot 2^{-i} \cdot y_i \\ y_{i+1} &= y_i + \sigma_i \cdot 2^{-i} \cdot x_i \\ z_{i+1} &= z_i - \sigma_i \cdot \arctan \cdot (2^{-i}) \end{aligned} \tag{3.11}$$

which describes the iterative rotation of the vector (x_i, y_i). The rotation around the angle Θ is represented as a linear combination of partial rotations around the selected partial angle α_i (Equation 3.12):

$$\Theta = \sum_i \sigma_i \cdot \alpha_i \qquad \sigma_i \in -1, 1 \tag{3.12}$$

where i is the iteration index and σ_i determines the sign change and the multiplication form the rotation matrix is replaced by iteratively adding or subtracting values stored in a lookup table (LUT) in each iteration (Equation 3.13):

$$\alpha = \arctan 2^{-i} \tag{3.13}$$

The CORDIC module compute the matrix multiplications using additions and shifts. Compared to [164, 165] the pre- and post-scaling of the CORDIC [161] are not part of the hardware architecture. Instead, the pre- and post-scaling is computed by the *KAVUAKA* processor. The co-processor computes the transformation equations. The angle table contains 32 angles, which are written by the *KAVUAKA* processor before the computation. The values depend on the fixed-point input and output format, and the coordinate system mode. The coordinate system can be circular, linear or hyperbolic. The iteration controller selects the σ_i and α_i values based on the iteration i. The output and input data of the co-processor are stored in registers.

This co-processor is coupled to the *KAVUAKA* processor as shown in Figure 3.75. The *KAVUAKA* processor uses *LOAD* and *STORE* instructions to access and control the external bus interface, which is mapped to the global address space. The interface routing for accessing the co-processor is specified in Figure 3.26. For the x_i, y_i and z_i values three *STORE* instructions are required and one *STORE* sets the coordinate system. The co-processor starts the computation iterations when the x_i value is written. The *KAVUAKA* processor retrieves the result registers with *LOAD* instructions after a certain number of *CYCLES*, which depends on the datapath width and the required computational accuracy. This number can be reduced by computing more than one iteration per cycle by connecting several CORDIC modules in series. To increase the throughput, SIMD co-processor configurations are designed where multiple CORDIC modules process multiple data values in parallel.

A total of ten co-processor configurations are evaluated in this chapter. These co-processors are part of the co-processor cluster, which is shown in Figure 3.76 and listed in Table 3.10.

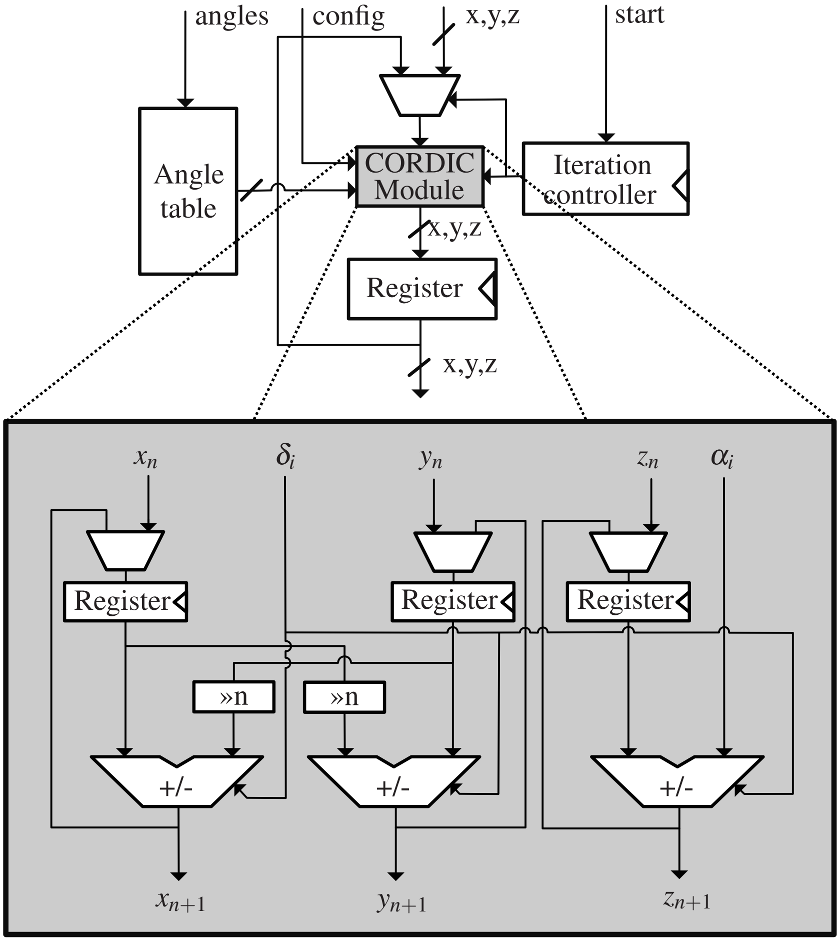


Figure 3.25: Architecture of the CORDIC co-processor and the CORDIC module.

```
STORERCU0 HW_ADDRESS_RCU0_1 ,  REG_Y_IN , #1
STORERCU0 HW_ADDRESS_RCU0_2 ,  REG_Z_IN , #1
STORERCU0 HW_ADDRESS_RCU0_4 ,  MODE ,     #1
STORERCU0 HW_ADDRESS_RCU0_0 ,  REG_X_IN , #{CYCLES + 1}

LOADRCU0  REG_X_OUT , HW_ADDRESS_RCU0_0
LOADRCU0  REG_Y_OUT , HW_ADDRESS_RCU0_1
LOADRCU0  REG_Z_OUT , HW_ADDRESS_RCU0_2
```

Figure 3.26: Assembler instructions using the CORDIC co-processor. The *STORE* instructions transfer x_i, y_i and z_i. The *MODE* determines the coordinate system: circular, linear of hyperbolic. The write to the address of x_i starts the iteration controller of the co-processor. The number of *CYCLES* determines the latency of the *LOAD* instructions to retrieve the results from the output registers.

The least complex CORDIC co-processor is called CORDIC M1, which has a single CORDIC module and no SIMD. The number of chained CORDIC modules is increased for the co-processors CORDIC M2 and CORDIC M4. The option for SIMD support doubles the number of parallel processing CORDIC modules, each iterating on a subword, for the co-processors CORDIC M1 SIMD, CORDIC M2 SIMD and CORDIC M4 SIMD.

Table 3.10: CORDIC co-processor configurations. The number of chained (M2 and M4) and parallel CORDIC modules (SIMD) is varied.

Co-Processor Name	Number of CORDIC modules (M)	SIMD
CORDIC M1	1	no (1 word)
CORDIC M2	2	no (1 word)
CORDIC M1 SIMD	1	2 words
CORDIC M2 SIMD	2	2 words
CORDIC M4	4	no (1 word)
CORDIC M4 SIMD	4	2 words

Division Co-Processor Architecture

The architecture of the generic, non-restoring signed divider co-processor [86–88] consists of an array of controlled adder or subtractor cells (CAS). These cells add or subtract iteratively shifted divisor bits to the dividend. The results represent the partial remainders, and the sign of the result determines one bit of the quotient. The data width (DW) of the CAS cells corresponds to the subword length of 32-bit. The rows in the CAS array are referred to as divider levels (DL). The number of divider levels defines the processing throughput and the required hardware resources. In this study, the divider level is varied between 1 to 8.

Processing Performance

To determine the processing performance and throughput, the number of cycles required to compute hyperbolic and trigonometric arithmetic functions is measured. As a reference, the number of cycles for the software implementation of the CORDIC and the non-restoring algorithm on the *KAVUAKA* processor are given in Figure 3.27. Both implementations make use of the VLIW and SIMD processor mechanisms. The IPC is above 1.5 and two subwords are processed in parallel. Based on the type coordinate system, circular, hyperbolic or linear, the software arithmetic of the innermost transformation computation differs. Additionally, the pre- and post-scaling depend on the function type, the data and the fixed-point types. The number of cycles is the average for 100 randomly generated data values. The non-restoring division algorithm requires 62 % of cycles compared to the CORDIC algorithm for the division function. The exponentiation (power) is computed based on the Equation 3.14:

$$x^y = e^{y \cdot \ln x} \tag{3.14}$$

which requires two sequential calculations of the CORDIC algorithm for the exponential and natural logarithm functions.

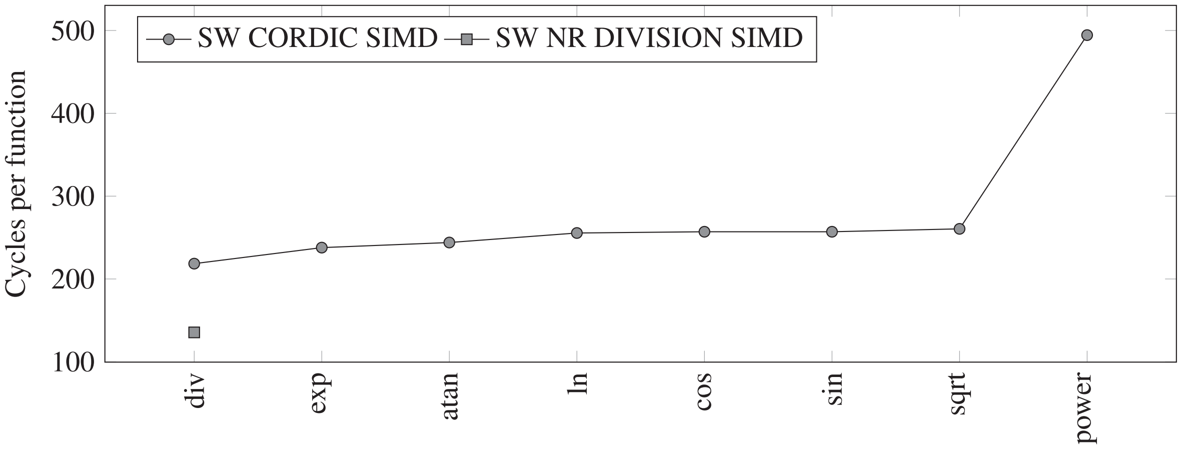


Figure 3.27: Number of cycles for the computation of hyperbolic and trigonometric arithmetic functions using the software implementation of the CORDIC and non-restoring division algorithm on the *KAVUAKA* processor.

The number of cycles required for hardware accelerator configurations for the same functions are specified in Figure 3.28. Increasing the number of modules reduces the number of cycles required, but the constant cycles of pre- and post-processing are maintained. Compared to the software implementation Figure 3.27, the order of the functions varies according to the number of cycles, because the hardware accelerators always require the same number of cycles. The non-restoring division accelerator requires fewer cycles compared to CORDIC accelerator with the same degree of parallelism, since no pre- and post-processing is required. The required number of cycles for the computation is linearly dependent on the divider level (Equation 3.15):

$$N \sim \frac{\mathrm{DW}}{\mathrm{DL}} \tag{3.15}$$

Table 3.11 presents the number of cycles, subdivided according to the different processing steps. The cycles for initializing the angle table are not required if the same coordinate system

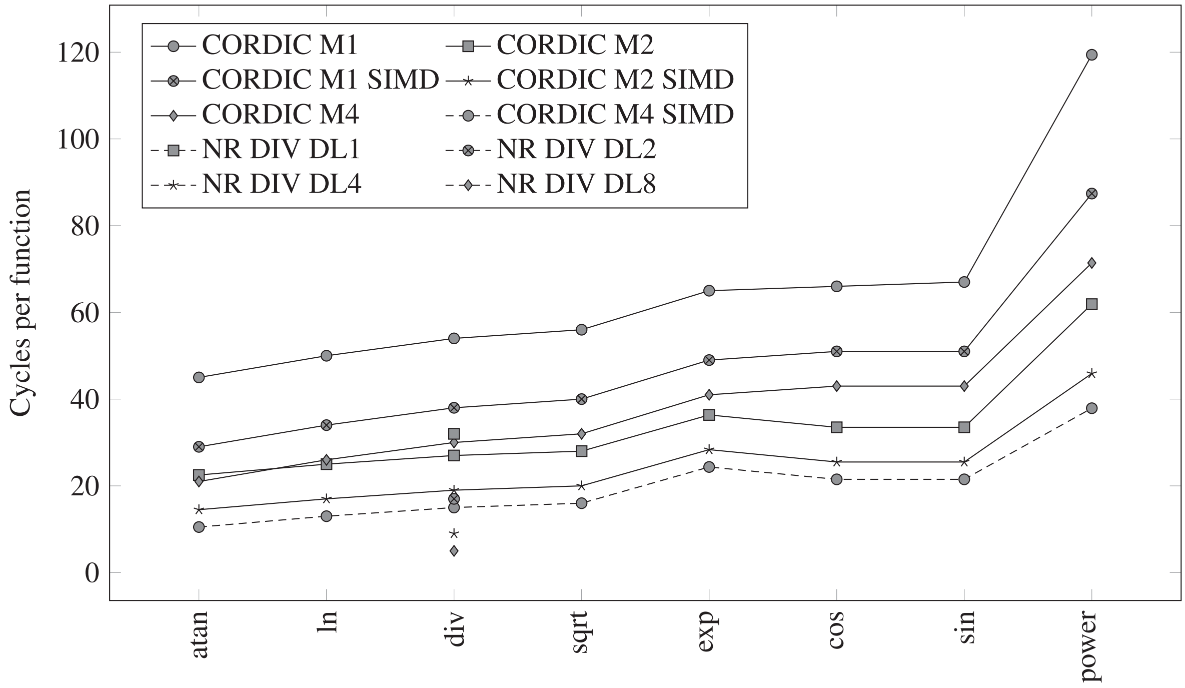


Figure 3.28: Number of cycles for the computation of hyperbolic and trigonometric arithmetic functions using the hardware CORDIC accelerators.

and fixed-point format is used by successive functions. The pre- and postscaling steps depend on the function type. The number of CORDIC iterations is fixed to the maximum value of 32 iterations. In this case, the highest achievable precision is achieved with the CORDIC M1 accelerator (Section 3.2.3).

Compared to the related processors and implementations [166–168] that do not use a look-up table approach, the number of cycles is lower for the *KAVUAKA* processor with a coupled CORDIC M4 co-processor for all hyperbolic and trigonometric functions. The compared processors are multi-issue (VLIW) SIMD instruction-set architectures (Table 3.12). The *C67x* executes up to eight and the *ADSP-BF533* up to three instructions per cycle. To take advantage of the parallelism offered by the processor architectures, suitable algorithms are selected and optimized. In [166], the authors present a method to implement mathematical functions on the *TMS320C67X* using polynomial approximation and data dependency optimizations to take full advantage of the multiple parallel execution units. With this method they achieve up to 70.2 % performance improvement over the standard implementation. Polynomial approximation algorithms are also used in [168]. In addition, a fast floating-point emulation format is used to increase performance for fixed-point processor execution. The maximum reduction of the number of cycles compared to the standard library is 85 %. Although the number of

Table 3.11: Number of cycles for hyperbolic and trigonometric functions for the *KAVUAKA* processor with a coupled CORDIC M1 co-processor. Cycles subdivided by: Initialization of the angle table, pre-scaling, CORDIC iterations and postscaling.

Hyperbolic and trigonometric functions	Angle table initialization	Prescaling	CORDIC iterations	Postscaling	Total
Sine	62	18	32	16	128
Cosine	62	18	32	16	128
Arc tangent	62	8	32	5	107
Divide	62	16	32	8	118
Exponential	62	29	32	9	132
Natural logarithm	62	15	32	6	115
Square root	62	21	32	8	123

issue-slots of the related processor architectures is comparatively high, the number of cycles for computing hyperbolic and trigonometric functions is higher than on a processor with fewer issue-slots and a co-processor. An advantage of the co-processor is that the issue-slots can be used for other calculations. However, the silicon area, including the area of the co-processor and the processor (Section 3.2.3), programming flexibility, and power consumption must be considered.

Table 3.12: Comparison of the number of cycles for hyperbolic and trigonometric functions for the *KAVUAKA* processor with a coupled CORDIC M4 co-processor and the *TI TMS320C6000* DSP.

Function	KAVUAKA and CORDIC M4	TI Lib. [166, 167] C67x (SP)	Opt. [166] C67x (SP)	Opt. [168] Blackfin ADSP-BF533
sin	43	173	69	412
cos	43	183	73	—
atan	21	265	79	—
div	30	120	—	—
exp	41	213	74	331
ln	26	136	44	569
sqrt	32	112	—	—

Precision Evaluation

The precision analysis for the hyperbolic and trigonometric functions for different co-processor configurations is presented in Figure 3.29, Figure 3.30 and Figure 3.31. The precision is measured as the maximum absolute error compared to the reference double-precision floating-point result. The precision decreases exponentially with the number of algorithm iterations for both, the CORDIC and the non-restoring division algorithm, since one or more fractional bits are computed per iteration. By increasing the number of CORDIC modules, the absolute error decreases more rapidly. The comparison between the functions shows slight differences. The precision of the square root function is higher and increases faster than the others, because the input values are limited to positive values and the output dynamic range is comparatively small. Both algorithms can stop the computation at the required precision requirements to keep the processing time as low as possible.

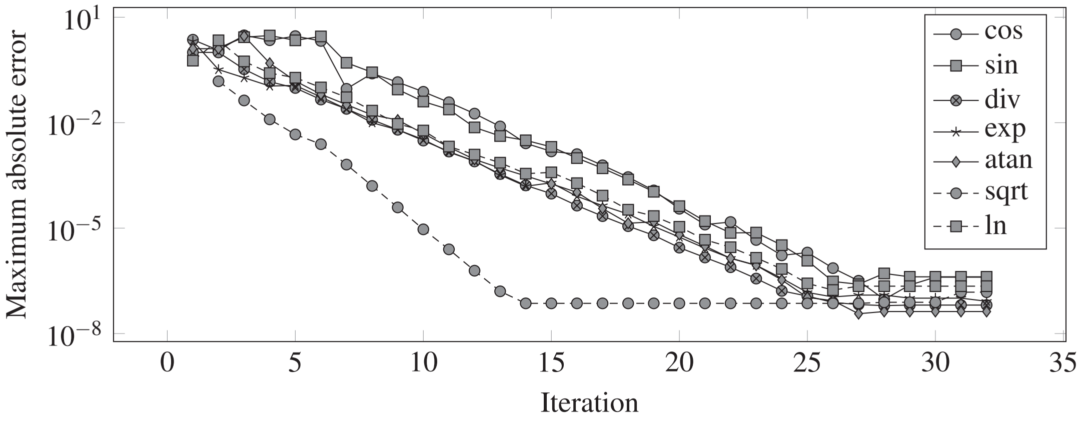


Figure 3.29: Maximum absolute error for each CORDIC iteration with one CORDIC module. 32 bit fixed-point values are used.

Area Evaluation

The silicon area of the CORDIC and the non-restoring division co-processor synthesized with the TSMC 40 nm HVT low-power ASIC technology for a clock frequency of 50 MHz is shown in Figure 3.32 and Figure 3.33. CORDIC accelerators with SIMD or more than one module require more combinational logic, while the non-combinational area (sequential logic area) remains almost constant. For the SIMD configurations, there is a small increase due to the increased number of output buffers. Increasing the number of parallel CORDIC modules for SIMD support requires less additional combinational logic per module than chaining the same number of CORDIC modules (M2 and M4 configurations). The reason for this is to share

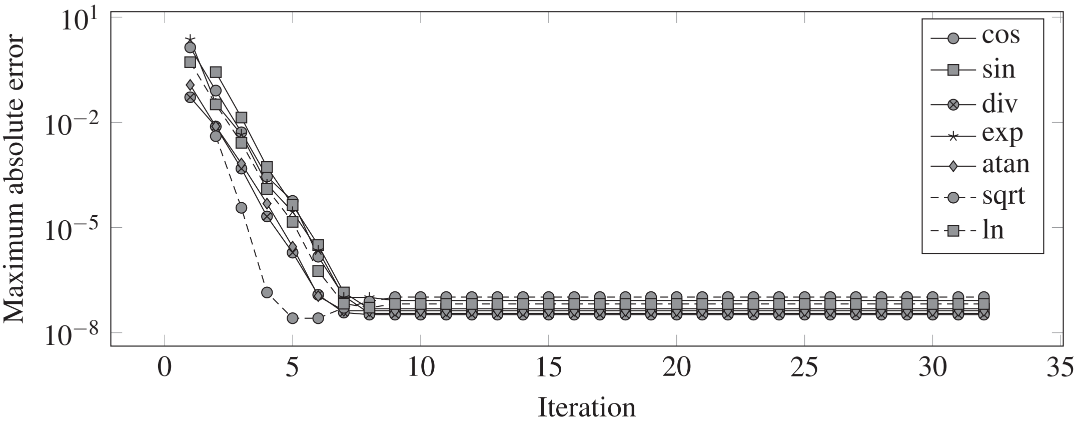


Figure 3.30: Maximum absolute error for each CORDIC iteration with four CORDIC modules. 32 bit fixed-point values are used.

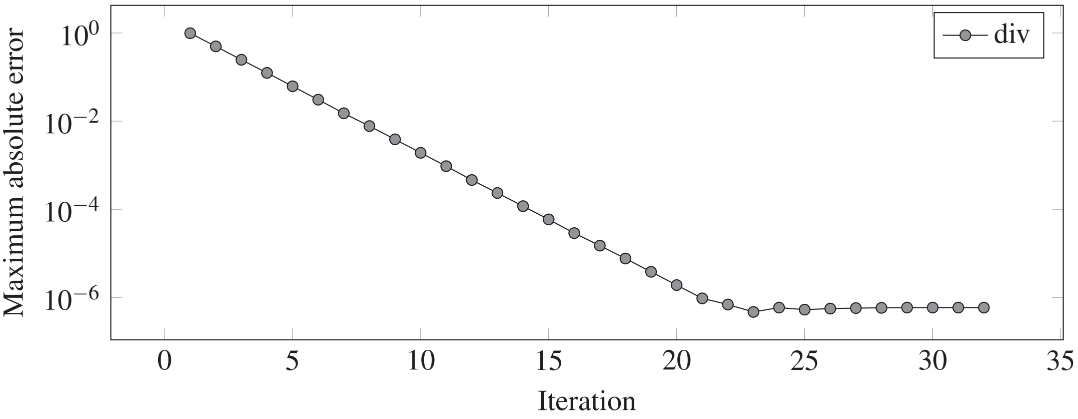


Figure 3.31: Maximum absolute error for the non-restoring division co-processor. 32 bit fixed-point values are used.

resources for SIMD CORDIC modules. However, due to the aforementioned increase in non-combinational logic, the SIMD configurations require more total area.

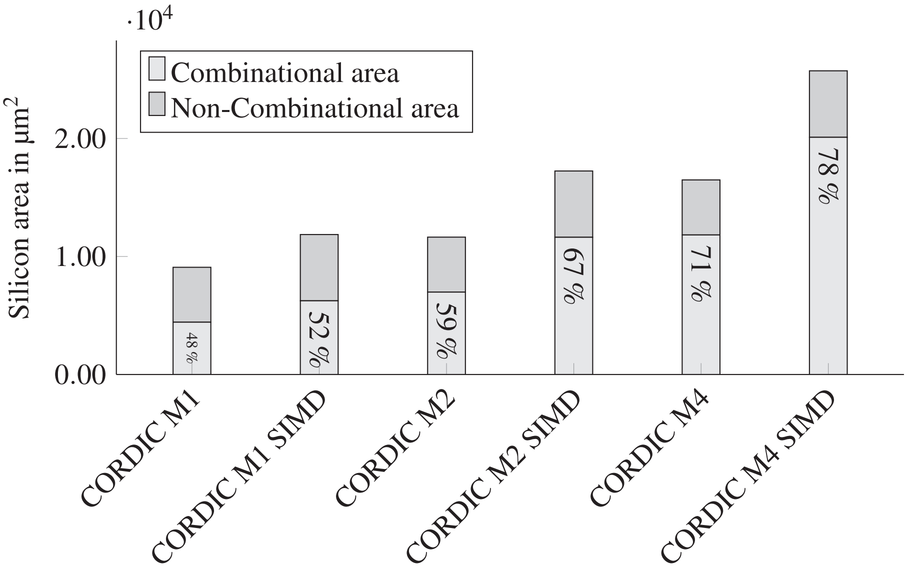


Figure 3.32: Silicon area for different CORDIC configurations. Results obtained from an ASIC synthesis with the TSMC 40 nm HVT low-power ASIC technology at 50 MHz.

The silicon area for the non-restoring division accelerator with different numbers of CAS rows (divider levels (DL)) is shown in Figure 3.33. The combination area increases by about 71 % per divider level. The non-combinational area decreases negligibly due to the smaller iteration counter. The non-restoring division co-processor is about 33 % smaller compared to the equivalent CORDIC co-processor. Both, the combinational and non-combinational area is more than half as large.

The silicon area of the CORDIC co-processor compared to the *KAVUAKA* processor with different datapath width is shown in Figure 3.34. The area requirement for the CORDIC co-processors is 0.0068 mm^2 for a 24-bit datapath width/resolution and 0.0090 mm^2 for a 32-bit datapath width. The silicon area overhead caused by attaching this co-processor to the 48-bit and 64-bit *KAVUAKA* is smaller, since the processor and co-processor is implemented without SIMD support in this case (Section 3.3.1). The co-processor area overhead is approximately 14 % for the 24-bit and 32-bit processor. Compared to the 48-bit and 64-bit processor, the area of the co-processor is approximately 8 %.

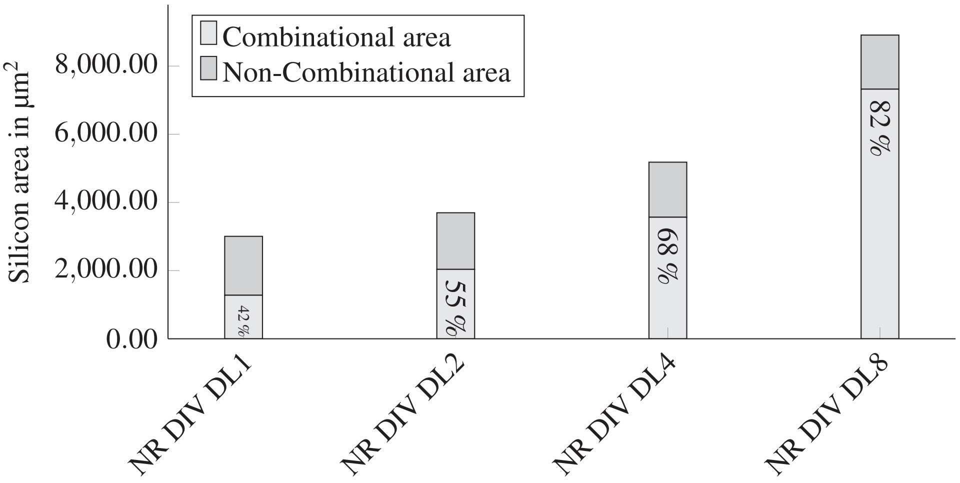


Figure 3.33: Silicon area for different non-restoring division co-processor configurations. Results obtained from an ASIC synthesis with the TSMC 40 nm HVT low-power ASIC technology at 50 MHz.

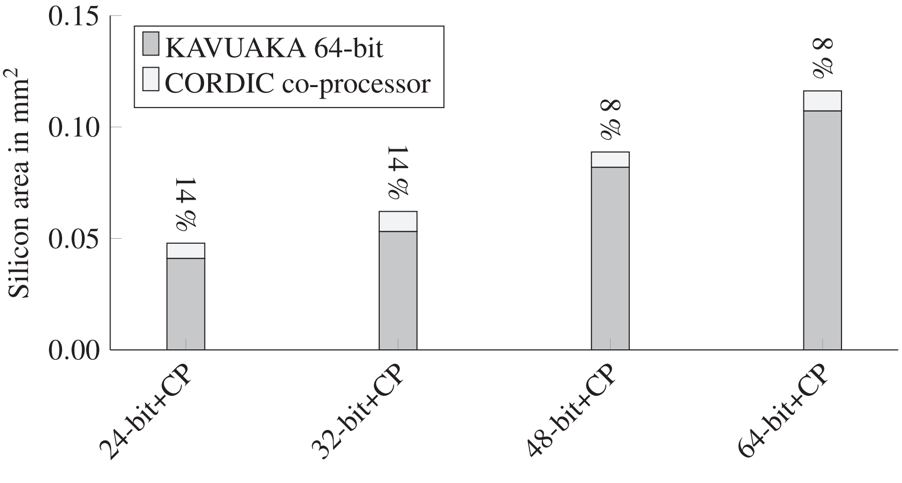


Figure 3.34: CORDIC (M1, no SIMD) co-processor silicon area compared to the silicon area of the *KAVUAKA* 64-bit processor. The data width is changed for the processor and the co-processor. Results obtained from an ASIC synthesis with the TSMC 40 nm HVT low-power ASIC technology at 50 MHz.

3.2.4 Operation Merging Extensions

The baseline VLIW-SIMD architecture (Section 3.1) was introduced in [169] for video processing applications. A mechanism for merging operations, called X2-mode, is part of this architecture. This mechanism encodes several operations with the same command and command modifier, i.e., signed/unsigned, overflow, condition set and condition read, into one micro-operation (MO). A merged MO differs only in one bit (X2 encoding bit) in the binary instruction encoding compared to a single MO. In particular, the length of the encoding is the same. The register addresses in a pair (e.g., the two target registers) must be consecutive, starting with an even address. Thus a single bit is sufficient to signal, when addressing a pair of registers, instead of encoding both register addresses. This one operation is decoded in the instruction decode (DE) stage, but several micro-operations are executed on duplicated functional units within the processor. Encoding two MOs of the same type into a single MO doubles the maximum possible number of MOs per cycle [84]. The functional units must be implemented in the processor as often as they are used in parallel, i.e., in this example two arithmetic logic units (ALUs), as shown in Figure 3.75. The X2 mode allows the parallel execution of up to two operations, resulting in an IPC over two on the dual-issue VLIW architecture.

In Figure 3.35, an assembler code is shown, which presents the application and advantages of the X2-mode. This example is part of a computation of a tapering function. The window weights and samples are stored in memory one after the other. With one X2 move (MV) operation, two values are loaded from memory and stored in a consecutive pair of registers (*V0R0+V0R1*). This is a special case for the X2 mode, which is called memory X2 mode. The X2-mode requires only two move (MV) operations instead of four, doubling the data rate to main memory. The indirect address pointers (*WeightPtr* and *SamplePtr*) to main memory are incremented twice, since a X2 operation loads two values at the same time. The MAC operations are not merged, because the MAC functional unit is not duplicated due to its comparatively high silicon area requirement.

The area increase for implementing the X2-mode in the processor architecture is negligible, but the algorithm execution is accelerated and the minimum clock frequency can be reduced. This results in a smaller area-time (AT) product for the processor implementation, i.e., higher efficiency. However, there are limitations when using the X2-mode. The register pair of a X2-operations must be an *even* and an *odd* register pair. The reason for this is the hardware implementation of the X2 mode, which modifies the last bit of the register address for the *odd* register when reading or writing the register file. Another restriction applies to the memory X2 operation mode. Since there are two VLIW issue slots, each with one load and store unit, there are two separate local memory interfaces. The memory is attached to these interfaces, which in turn is divided into two instances. The reason for this separation is to support the memory X2 operation, with one instance for the *even* and the other for the *odd* memory addresses. There is no possibility to perform successive data transfers to a memory block, when using

(a)

```
MV     VOR0,(WeightPtr)+
MV     VOR2,(SamplePtr)+
MAC_16 VOR4+VOR5,VOR0,VOR2

MV     VOR1,(WeightPtr)+
MV     VOR3,(SamplePtr)+
MAC_16 VOR6+VOR7,VOR1,VOR3
```

(b)

```
MV_X2 VOR0+VOR1,(WeightPtr)++
MV_X2 VOR2+VOR3,(SamplePtr)++

MAC_16 VOR4+VOR5,VOR0,VOR2
MAC_16 VOR6+VOR7,VOR1,VOR3
```

Figure 3.35: Assembler code example for the X2-mode. Data is loaded with indirect memory access (file indirect register (FIREG)) and the X2-mode. With the X2-mode (b) a dense execution is possible. Operations with the same opcode are merged to *_X2*-operations. By using MV_X2 instructions, two 64-bit registers are loaded with one instruction. Target and source registers of X2-operations have to have successive addresses. MAC-operations have two destination registers by default.

either load and store X2 operations.

This section introduces two hardware modifications that remove the restrictions under certain circumstances. The modifications are evaluated in terms of hardware overhead.

Special Register File Extension: X2 Offset Register

For the original X2-mode [16], the addressed registers must necessarily be an *even* and an *odd* register pair. This restriction leads to a neglectable hardware overhead for the register addressing logic, because only the least significant bit (LSB) of the register address needs to be modified to compute the second register address based on the given register, as shown in Figure 3.36. However, signal processing applications may require different addressing modes for an efficient processing scheme. An example is the radix-2 decimation in time FFT algorithm, shown in Figure 3.15. The data to be processed is stored in main memory sequentially. However, after each stage is processed, the data is rearranged based on the computing scheme of the butterfly operations. This reordering is very regular, and the difference between two adjacent data points can be described by an offset. This offset varies with the current FFT stage. A fast data transfer between the register file and main memory is important for the overall processing performance of the FFT. Therefore, a hardware mechanism with an offset addressing mode for the X2-operations is desirable for efficient processing.

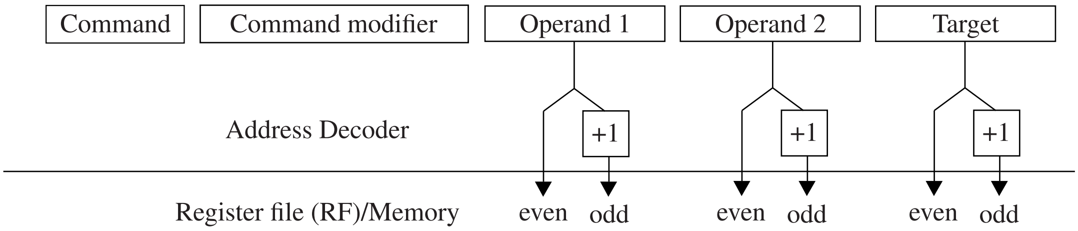


Figure 3.36: X2-mode address decoding [16]. The odd register address is computed by inverting the least significant bit (LSB) of the encoded operand or target register address.

To remove the restriction that the register pairs for the X2-mode must be an *even* and an *odd* register, a X2 offset register (Reg X2) is introduced. This register contains the offset for the register address, which is added to the register address encoded in the operation, as shown in Figure 3.37. The X2 offset register is located in the special register file of the processor. If there are 64 registers in total, the bit-width of the offset register is 7-bit for storing the signed offset value. This offset is added to the given register address. The processor can read or overwrite the content of this register. The default value is 1, which is the default offset for the register address. Not all combinations of values for the offset are allowed, because the maximum read or write ports of the register file can be exceeded. An example of using the X2 offset register is shown in Figure 3.38. The value of the offset register is set to 32 by the special move (*SMV*) operation.

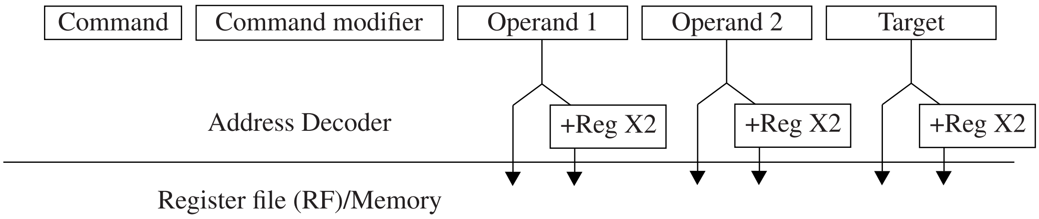


Figure 3.37: X2-mode address decoding with the proposed X2 offset register. The second register address is computed by adding the offset register value to the encoded operand or target register address.

The implementation of the X2 offset register including the required adder, which adds the offset to the given register address, increases the total area of the *KAVUAKA* 64-bit processor by 0.45 %.

```
STORE_STAGE_X2_FFT
SMVI_32 V0OFFSET_X2, #32 //set offset to 32
--scheduling-off
:0  MV_X2 4(FIR_AUDIO_1_FIRST),  V0R0+V1R0
:1  MV_X2 4(FIR_AUDIO_1_SECOND), V0R2+V1R2
:0  MV_X2 4(FIR_AUDIO_1_FIRST),  V0R4+V1R4
:1  MV_X2 4(FIR_AUDIO_1_SECOND), V0R6+V1R6
:0  MV_X2 4(FIR_AUDIO_1_FIRST),  V0R8+V1R8
:1  MV_X2 4(FIR_AUDIO_1_SECOND), V0R10+V1R10
:0  MV_X2 4(FIR_AUDIO_1_FIRST),  V0R12+V1R12
:1  MV_X2 4(FIR_AUDIO_1_SECOND), V0R14+V1R14

:0  MV_X2 4(FIR_AUDIO_1_FIRST),  V0R1+V1R1
:1  MV_X2 4(FIR_AUDIO_1_SECOND), V0R3+V1R3
:0  MV_X2 4(FIR_AUDIO_1_FIRST),  V0R5+V1R5
:1  MV_X2 4(FIR_AUDIO_1_SECOND), V0R7+V1R7
:0  MV_X2 4(FIR_AUDIO_1_FIRST),  V0R9+V1R9
:1  MV_X2 4(FIR_AUDIO_1_SECOND), V0R11+V1R11
:0  MV_X2 4(FIR_AUDIO_1_FIRST),  V0R13+V1R13
:1  MV_X2 4(FIR_AUDIO_1_SECOND), V0R15+V1R15
--scheduling-on
SMVI_32 V0OFFSET_X2, #1 //reset value to default one
ENDMACRO
```

Figure 3.38: Assembler code example for X2-mode with the X2 offset register addressing. This function is part of the FFT algorithm and stores the data after the butterfly operations to the main memory. The second register address is calculated by adding the offset register value to the address of the first register. The offset is 32 in this case. This subroutine stores four registers (e.g., 4 × 64-bit) per cycle to the main memory, without any alignment or permutation operations.

Interleaved X2 Memory Interface

In this section, an interleaved X2 memory interface is proposed. This memory interface allows successive data transfers to the main memory using two X2 operations per cycle. In the example shown in Figure 3.38, four registers are stored in local memory using two MV_X2 instructions per cycle. This is possible with X2-operation mode [169], which can also be used to access the main memory (memory X2-operation mode). The main memory instance is divided into two parts, a local memory instance for the *even* and *odd* positions/addresses. Only the last bit of the consecutive *even* and *odd* addresses is flipped. As a result of this architecture, the data to be stored is split and stored in the two memory instances. Half of the data (*even* operand) is stored in the first memory block and the other data (*odd* operand) in the second memory block. When sequentially reading or writing the data without memory X2 mode, the memory address gap between the two memory instances must be considered. In this case, two *file indirect registers (FIREGs)* are required. Special addressing modes such as bit reversed (Section 3.3.3) or circular indirect buffer addressing [83] are not possible because of this memory gap.

The proposed interleaved X2 memory interface prevents the memory address gap for the X2 operation mode. To support X2 and individual load and store accesses from each of the two load and store units, four memory modules are required, as proposed in [169]. However, all memory modules are interleaved, as shown in Figure 3.39. The last two address bits of each of the two address ports are used. For example, if the last two address bits are ”00”, the first memory module is activated. Two memory modules are activated for X2 memory accesses. One of the memory modules is determined by the address port. The second memory module is selected in a circular manner, so there is no restriction on the use of *even* memory addresses as in [169]. Structural hazards are prevented by a logic that detects and resolves the hazard. An example of a hazard is a write after read transfer at the same location. Temporary bypass registers are used to buffer the transfer until the memory port is free.

With the proposed architecture, both single and X2 memory accesses are possible without the limitations mentioned above. A single file indirect register (FIREG) can be used to access local memory. This memory hierarchy behaves like a single memory block for the programmer. Special addressing modes such as bit reverse or ring memory addressing (Section 3.3.3) are possible using the entire address space. The area required to implement the interleaved X2 memory interface logic is shown in Figure 3.51. Compared to the actual SRAM area, the logic accounts for 3 % up to 6 %.

3.2.5 Issue-Slot Based Predication Encoding Technique

Very long instruction word (VLIW) processors are commonly used for embedded high performance and low-power multimedia applications [170–177]. VLIW processors are designed

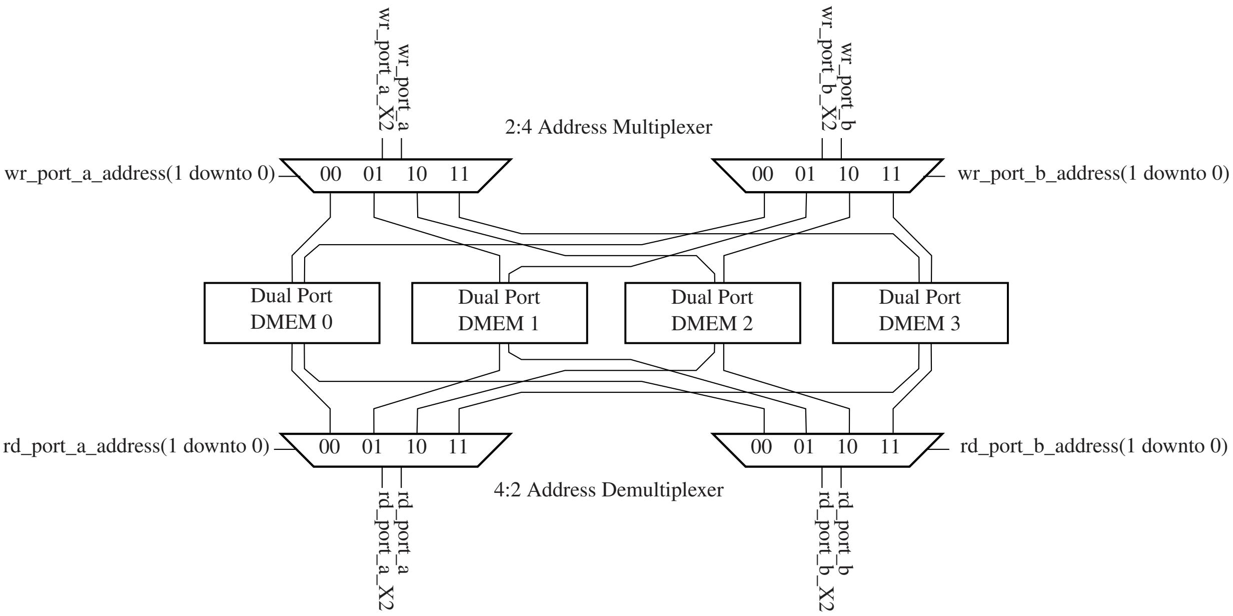


Figure 3.39: Interleaved data memory (DMEM) hierarchy. The data of the two read and write ports (port_a and port_b) and the additional data ports (port_a_X2 and port_b_-X2), which do not require any address, are multiplexed to one of the dual port SRAM modules based on the two LSBs of the port address.

for instruction level parallelism (ILP), executing multiple instructions with a fixed order in parallel. Since the order of the instructions is determined by the compiler, the hardware is less complex compared to superscalar architectures. Therefore, to fully utilize hardware resources, ILP compiler optimizations are required [170, 177].

One limitation for further compiler optimizations are branch instructions, which cause smaller basic blocks (straight line microcodes (SLMs)) in the code, restricting the scope of ILP optimizations [177]. These branch instructions, which are required for the control flow of an application, are expensive in terms of processing performance [176]. The condition of a branch cannot be evaluated at the beginning of the pipeline. Hence, the successive instructions already fetched and decoded must be discarded, while the pipeline is flushed, or are executed whether or not the conditional branch is taken. One alternative to conditional branches is predication [171, 177]. Predication is the conditional execution or guarded execution of instructions. The instructions of both instruction-sequences of the conditional code are executed, but only the instructions of one of the sequences change the state of the processor and memories, depending on the value (condition flag) stored in a predication or guard register [177]. The remaining instructions act as no operation (NOP) instructions. Branches may be replaced by conditionally executed instructions during the *if*-conversion [171, 172, 178]. The control dependencies are converted to data dependencies by converting multiple regions of a control flow graph into a basic block, composed of predicated (conditional) code [170].

The conditional executed code can be implemented in two common variants, named full and partial predication. Most instructions have an additional predicate register, which determines if the instruction should change the processor state. A conditional *ADD* instruction is shown Figure 3.13.

Table 3.13: Instruction extension for full predication: An ADD instruction with two source, one target operand and a predicate register (p0), holding a condition flag (*true* or *false*). This flag determines if the instruction modifies the processor state [171].

Instruction Opcode	Target Register	Source Register	Source Register	Predicate Register
ADD_CR	V0R0	V0R1	V0R2	(p0)

The second variant for conditional code implementation is partial predication. Here, a small set of instructions are extended to be conditional. In Figure 3.14, a conditional move and a conditional select instruction are depicted. The move instruction overwrites the target register based on the flag stored in the predicate register. The select instruction selects on of the two source operands based on the predicate register.

Table 3.14: Instruction extension for partial predication: A conditional/predicated move instruction (MV_CR) with one source, one target operand and a predicate register (p0), holding a condition flag. This flag determines if the content of the source register is copied to the target register. The select instruction (SEL_CR) copies the content of one of the two source register to the target register. Which source register is selected is based on the condition flag [171].

Instruction Opcode	Target Register	Source Register	Source Register	Predicate Register
MV_CR	V0R0	V0R1	—	(p0)
SEL_CR	V0R0	V0R1	V0R2	(p0)

However, the application of a predication technique requires an extension of the instruction set architecture (ISA) of the processor [174, 177]. Additional instruction encoding bits are needed in order to address one of the predicate register. But adding for example six instruction encoding bits to address 64 predicate register is prohibitively too expensive for embedded processors [177]. Studies for related embedded processors [179, 180] show the importance of the instruction memory size on power consumption. As shown in [179], the memory subsystem of *ARM* processor [181] accounts for 65.2 % of the total energy. For multi-processor *ARM* systems, this proportion is 45.9 %. The instruction memory cache alone accounts for 20.6 % in this system. The *ARM* processors with the reduced 16-bit *Thumb* instruction set architecture (ISA) are more energy efficient compared to the 32-bit *ARM* processors, however the processing performance is lower [179]. The instruction caches of the *TMS320C6000* VLIW processor family [182] account for up to 30 % of the total processor power [183, 184]. Besides the power consumption, the area requirements of the instruction memory are crucial for embedded processors. Each additional instruction bit increases the required SRAM area linearly, as depicted in Figure 3.40 for a 40 nm ASIC technology.

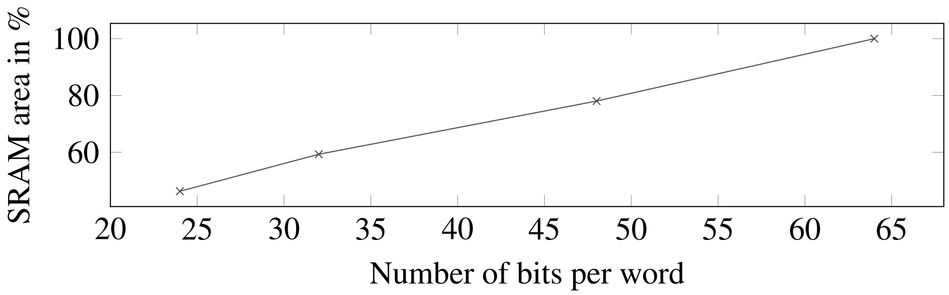


Figure 3.40: Area of SRAM macro blocks of a 40 nm ASIC technology with different number of bits per word. The area of the SRAM macro block with 64-bit per word is used as a reference.

Additionally, the number of read and write ports of the predicate register file and the instruction fetch and decode logic are important for the resulting hardware complexity [174, 180]. The instruction fetch and decode logic of a DSP can consume up to 40 % of the total processor logic [180, 185]. Therefore, the goal is to reduce the overhead caused by implementing the predication technique in these processors [174].

In this section, a new scalable and low overhead predication technique for VLIW processors is proposed. The corresponding compiler extension is described in Section 4.2.1. Instead of encoding the address of the predicate registers with additional instruction encoding bits, each issue-slot of the VLIW architecture includes a dedicated predicate register. With the help of compiler optimizations, which are also presented in this thesis, instructions are scheduled on the issue-slot with the corresponding predicate register. This technique scales with the number of issue-slots, requires no additional instruction encoding bits, and therefore decreases the hardware overhead for predication.

The related predication techniques are described in Section 3.2.5. The proposed technique is presented Section 3.2.5. The three case studies, which are presented in Section 3.2.5, Section 3.2.5 and, Section 3.2.5, show the applicability of the proposed technique.

Related Work

An overview of the related predication techniques is given in Table 3.15. The number of predicate registers and the required instruction encoding bits to address these registers are compared. The number of predicate registers determines the maximum number of conditional statements, which can be processed in parallel. The cause for these conditional statements are parallel or nested *if-else* constructs. A comparatively high number of up to 64 predicate registers is used by [171, 186–188]. Consequently, these architectures require most instruction encoding bits to address one of the predicate registers. The number of encoding bits is reduced in [188], by splitting the registers into two sets. A special instruction is used to switch between the two register sets. The *ARMv7-A* [181] architecture includes one *application program status register* (APSR), which holds four different conditions: negative, zero, carry and overflow. In order to select these conditions 4-bit are required for every instruction [177]. In [174, 182, 189] the predication technique of the *TMS320C6X* processor family is described. The condition flags are stored in a restricted number of registers of the general purpose register file. The limited number of six predicates for the *TMS320C6X* processor family results in a limited control-flow nesting [177]. As every *if-then-else* statement requires two predicates, only two levels of control-flow nesting are possible. The number of read and write ports of the general purpose processor register file needs to be increased and the access and usage patterns might influence overall performance [190]. However, no dedicated predicate register file is required and the predicate registers can be used otherwise if they are not needed. The authors of [174] propose to reduce the overhead of predication by restricting the number of predicate registers to one. Consequently, only one additional read port for the predicates is required. Four ports were previously required by the four issue-slots of the VLIW architecture to read four predicate registers per cycle. Instead of 3-bit for the predicate operand, only 1-bit is required. Nested and parallel *if-else* statements can not be fully predicated with only one predicate register [174].

A predication technique is presented in this thesis, which does not require any instruction bits to encode the predicate registers. The predicate registers are encoded by scheduling the conditional instructions on different issue-slots. The instruction bits saved compared to the related architectures are summarized in Table 3.15. Due to the issue-slot based predicate encoding, the proposed technique decreases the required instruction memory size, power consumption and the complexity of the instruction decoding stage compared to the related work. A compiler extension for this encoding technique is presented, which handles the predicate register allocation automatically, without the need to manually encode the predicate register within the instructions.

Table 3.15: Comparison of related predication techniques. The required instruction encoding bits are those that address one predicate register.

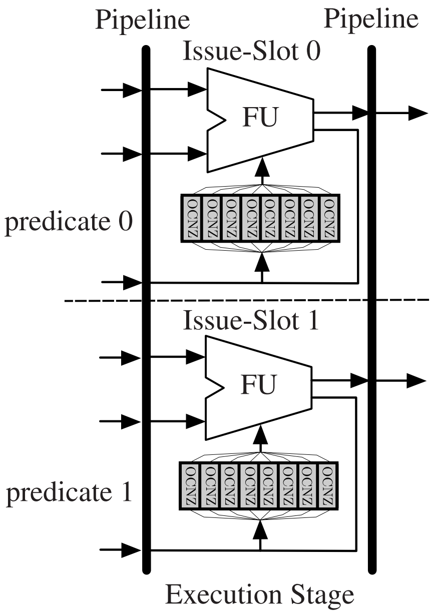
Architecture	Number of predicate registers	Required instruction encoding bits
Itanium IA-64 processor [186, 187]	64	6-bit
Generic ILP processor [171]	32	5-bit
ARMv7-A [181]	1	4-bit
TMS320C64x/C67x VLIW [182, 189]	5	3-bit
PLX [188]	128*	3-bit
HP VLIW ST231 ISA [174]	1	1-bit
KAVUAKA [this work]	16**	none

* (16 sets of 8) ** (8 per issue-slot, one for each SIMD subword)

Proposed Issue-Slot Based Predication Technique

A new predication technique is presented, which exploits the issue-slot based processing of VLIW processors. The architecture is depicted in Figure 3.41. A conditional instruction, which is defined by setting one instruction encoding bit to '1', receives or sets the condition flags of one predicate register. Instead of selecting a predicate register using additional instruction encoding bits, the predicate register is selected by the issue-slot, on which the conditional instruction is scheduled. Each issue-slot of the VLIW processor contains one dedicated predicate register. The predicate register is selected with the issue-slot, on which the instruction has been scheduled. No additional bits for addressing the predicate register are required in the instruction encoding. Every conditional instruction, which reads or writes to same predicate register, is scheduled on the same issue-slot. This technique requires that the issue-slots of the VLIW architecture are identical/symmetric in terms of functional and data movement units (FUs), so that the instruction scheduler can switch the instructions between the slots without constraints.

These predicate registers contain four condition flags (overflow (O), carry (C), negative (N) and zero (Z)) for each subword of one SIMD data word, as shown in Figure 3.41 and proposed by [177, 190]. On the one hand, this format can be used to process conditional SIMD instructions on subword-level, with up to eight subwords in parallel depending on the condition flags stored in the corresponding position in the predicate registers. On the other hand, each single subword with four condition flags can be used for nested *if-else* statements. This use case assumes that each subword holding the conditions can be selected and used for one of the conditional *if-else* statements. The total number of predicates is therefore 16, stored in two



Predicate register

OCNZ	OCNZ	…	OCNZ	OCNZ
SW #7	SW #6	…	SW #1	SW #0

Predicate register content: flags (32-bit) with four conditions per SIMD subword (SW): overflow (O), carry (C), negative (N) and zero (Z)

Figure 3.41: One dedicated predicate register is part of each issue-slot. Each functional unit (FU) reads the condition flags for each subword from the predicate register of the same issue-slot.

predicate registers with eight subwords holding four condition flags each. Using conditional X2 operations, as presented in [84], is also possible when duplicating the flag register for the *even* and *odd* functional units.

The number of predicates scales with the number of issue-slots and available SIMD subwords of the processor. The number of read and write ports of the predicate register file is one. The area overhead for implementing one additional predicate register per issue-slot with one read and write port is around 1 % of the total core cell area of the *KAVUAKA* processor for an application-specific integrated circuit (ASIC) synthesis with a 40 nm TSMC low-power technology [56].

Case Study: Floating-Point Emulation

In this case study, floating-point emulation code with a high number of conditionally and independently executed operations, like overflow and sign checks, normalization operations and bitwise comparisons, is used. The underlying floating-point emulation library is described in detail in [191]. The computation of the addition and subtraction operations of this library is shown in Figure 3.42. For a floating-point addition or subtraction, the exponents of both numbers have to be adjusted to the same value. Therefore, the absolute difference of the exponents is computed. Based on the difference, the significand of the smaller number is shifted right by the absolute difference of the exponent and the larger exponent is selected. The significands are then added or subtracted from each other. After normalization of the significand, the exponent is updated and the floating-point addition or subtraction is computed.

The floating-point emulation addition assembler code macro is shown in Figure 3.43. Here the exponents are subtracted from each other. Depending on the result of this subtraction, one of the significands is chosen for shifting before adding the significands. The normalization is performed by counting leading zeros or ones. The resulting register is then rebuilt from the updated exponents and the normalized significands.

In the related work, presented in [146] and [192], processor architectures with multiple conditional registers and predicated execution features are selected for floating-point emulation. In this case study, the processing performance is evaluated based on the number of available predicate registers. Digital filter implementations with the difference Equation 3.16 are used as floating-point algorithms:

$$y(n) = \sum_{i=0}^{M} b_i x(n-i) - \sum_{j=1}^{N} a_j y(n-j) \tag{3.16}$$

where x is the input data and y is the output data. a and b are the filter coefficients for finite impulse response (FIR) and infinite impulse response (IIR) filters. In this case study, a FIR and

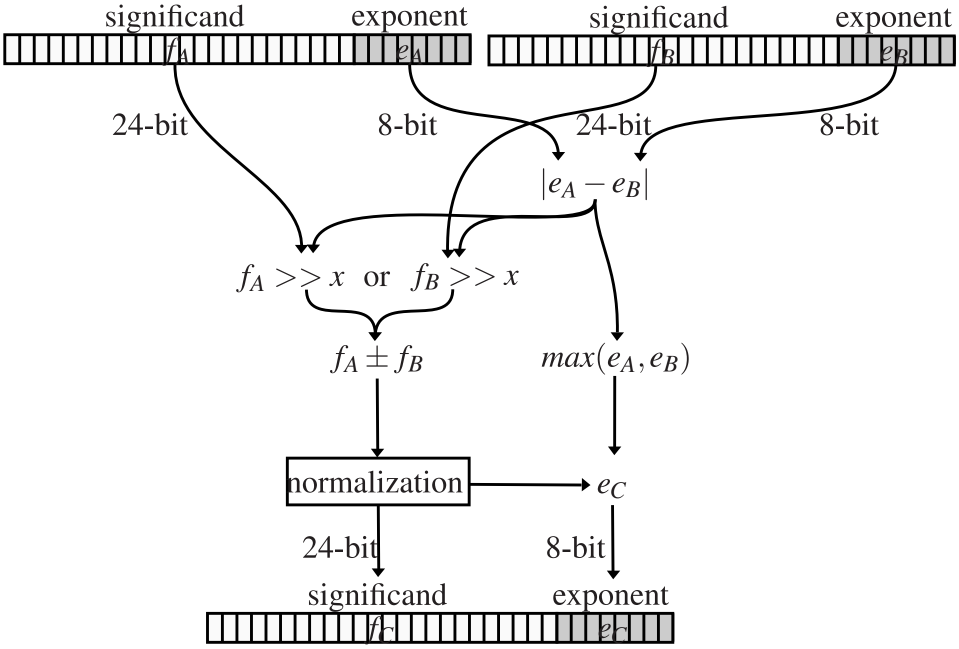


Figure 3.42: Optimized floating-point addition for SIMD processors.

IIR filter with an order of 17 and 7 are computed using floating-point emulation macros. Table 3.16 lists the floating-point point emulation macros and how often these macros are called during the computation. These macros contain up to 26 % conditional operations. The number of predicate registers determines how many operations of these floating-point emulation macros can be scheduled in parallel.

Table 3.16: Floating-point emulation macros

Floating-Point (FP) macro	Number of conditional operations	Number of macro calls
FP_ADD	5 (26 %)	8
FP_SUB	5 (26 %)	8
FP_MUL	2 (14 %)	16

The required number of processing cycles for computing one sample with one or two predicate registers is shown in Table 3.17. The sampling frequency of the input audio signal is 16 kHz in this case. When two predicate registers can be accessed in parallel, the number of required processing cycles decreases by 4.4 % and the instructions per cycle (IPC) increases from 1.82 to 1.91 for the *KAVUAKA* processor with a maximum IPC of 2, when no operation merging is used.

```
// ***************************************************
// emulated floating-point addition macro (Q 24.8 format)
// ***************************************************
MACRO FP_ADD_32 DST, OP1, OP2
// compare exponents
PERMREG0_8  xOP1, OP1,  OP2  // E0|E0|E2|E2|E1|E1|E3|E3
PERMREG0_8  xOP2, OP2,  OP1  // E2|E2|E0|E0|E3|E3|E1|E1
SUBCS_8s    TEMP, xOP1, xOP2 // exponent difference
// swap mantissas according to magnitude of exponents
// (choose number with smaller exponent)
MV          TEMP1, OP1
MV          TEMP4, OP2
MVCR_32     TEMP1, OP2
MVCR_32     TEMP4, OP1
// remove exponent (for significand computation)
SRI_32      TEMP5, TEMP1, #8
// right shift of secondary significand operand
ABSADD_8    TEMP, R_8, TEMP
CLIPI_U8    TEMP, TEMP, #0b11111
SR_32       TEMP, TEMP4, TEMP
// add mantissas
ADD_32      TEMP5, TEMP5, TEMP
// normalize resulting significand
CLX_32      TEMP, TEMP5, TEMP5
SMVI        V0CONDSEL, #0b0010
SLCS_32     TEMP5, TEMP5, TEMP
// compute new (resulting) exponent
SUB_8s      TEMP, R_8, TEMP
ADD_8s      TEMP, TEMP1, TEMP
// recreate original format
PERMREG1_8  DST, TEMP5, TEMP
MVCR_32 DST, REG_ZERO
ENDMACRO
```

Figure 3.43: Assembler code macro for emulating the floating-point addition using SIMD instructions.

Table 3.17: Processing performance in number of cycles

	one predicate register	two predicate register
17-tap FIR	740	707 (−4.4 %)
7-tap IIR	300	287 (−4.3 %)

Case Study: Loop Unrolling and Operation Merging

An example for an *if*-conversion based on partial predication is shown in Figure 3.44, Figure 3.45 and Figure 3.46. A reference code for bit-reversal permutation is given, which is required for the radix-2 Cooley-Tukey FFT algorithms [135]. The *if* construct is evaluated within the for loop for every index of the input vector.

```
function x = bit_reversal(x)
    number_of_elements = length(x);
  for i = 1:length(x)
      bit_reversed_index = bit_reversed_index(i-1, number_of_elements);
      if(0<(bit_reversed_index-(i-1)))
          temp = x(bit_reversed_index+1);
          x(bit_reversed_index+1) = x(i);
          x(i)=temp;
      end
  end
end
```

Figure 3.44: Reference *Matlab* code with a conditional *if* construct. The example is a bit-reversal permutation of an input vector x. The *if* condition is needed for swapping elements based on their position (index) within the vector. This code is required for the Cooley-Tukey FFT algorithm.

Taking the bit-reversal permutation code in Figure 3.44 as an example, the inner loop can be unrolled to increase the size of the basic block. This loop including the *if* condition is implemented on the *KAVUAKA* processor with conditional branches or predication with one and two predicate registers. Different levels of loop unrolling are evaluated, measuring the dynamic IPC. All ILP compiler optimizations are turned on, which includes automatic operation merging [193]. The results are depicted in Figure 3.47. The conditional branch (BR) implementation requires fewer cycles than the predicated version (CE1_PR), although the branch requires a branch delay slot with a NOP instruction. The reason for this is that *if-else* statement is unbalanced [194]. The *else* path does not contain any instructions in this case. Increasing the predicate registers to two (CE_2PR) does not improve performance, since only one condition is used per loop iteration. When the loop unrolling is applied, the size of the of the basic block and the number conditions, that can be processed in parallel, is increased. Additionally, more operations can be merged. The achieved IPC increases from 1.53 (CE_1PR) to 2.88 (CE_2PR_2LU), when using two predicate registers, loop unrolling and operation merging. Loop unrolling and operation merging are not as effective when using conditional branches (BR_1LU, BR_2LU) because the basic blocks are smaller and prevent further compiler optimizations.

Conditional branch implementation

```
SUBCS_64 REG_Z,REVERSED_INDEX,INDEX
//Conditional branch (greater than)
BSR_AND NO_SWAP, #0b1001, #0b10000000
//Load two elements
MV_64 REG_INDEX_plus_1, (FIR_INPUT_ELEMENT_ADDRESS)
MV_64 REG_i, (FIR_OUTPUT_ELEMENT_ADDRESS)
//Store elements in memory
MV_64 (FIR_INPUT_ELEMENT_ADDRESS), REG_i
MV_64 (FIR_OUTPUT_ELEMENT_ADDRESS), REG_INDEX_plus_1
:L_NO_SWAP
```

Scheduled code: conditional branch implementation

```
SUBCS_64     V1R30   V0R2    V0R0;   NOP
BSR_AND      NO_SWAP 0x9     0x80;   NOP
NOP                              ;   NOP
MV_64        V1R30   FIR_IND1    ;   NOP
MV_64        FIR_IND0    V1R30   ;   MV_64   V1R30   FIR_IND0
MV_64        FIR_IND1    V1R30   ;   NOP
:L_NO_SWAP
```

Figure 3.45: Assembler implementation and scheduled code using conditional branching. Reference code is given in Figure 3.44.

Conditional execution implementation

```
//Set condition (greater than)
SMVI_64 V0CONDSEL, #0b0101
SUBCS_64    REG_Z,REVERSED_INDEX,INDEX
//Load two elements
MV_64 REG_INDEX_plus_1, (FIR_INPUT_ELEMENT_ADDRESS)
MV_64 REG_i, (FIR_OUTPUT_ELEMENT_ADDRESS)
//Conditionally swap to elements
MV_64   REG_TEMP, REG_INDEX_plus_1
MVCR_64 REG_INDEX_plus_1, REG_i
MVCR_64 REG_i, REG_TEMP
//Store elements in memory
MV_64 (FIR_INPUT_ELEMENT_ADDRESS), REG_i
MV_64 (FIR_OUTPUT_ELEMENT_ADDRESS), REG_INDEX_plus_1
```

Scheduled code: conditional execution implementation

```
SMVI_64 V0CONDSEL, #0b0101      ;   NOP
MV_64   V1R0    FIR_IND1        ;   SUBCS_64    V1R30   V1R1    V1R30
MV_64   V1R1    FIR_IND0        ;   NOP
MV_64   V1R30   V1R1            ;   MVCR_64     V1R1    V1R0
MV_64   FIR_IND1    V1R1        ;   MVCR_64     V1R0    V1R30
MV_64   FIR_IND0    V1R0        ;   NOP
```

Figure 3.46: Assembler implementation and scheduled code using conditional execution. Reference code is given in Figure 3.44.

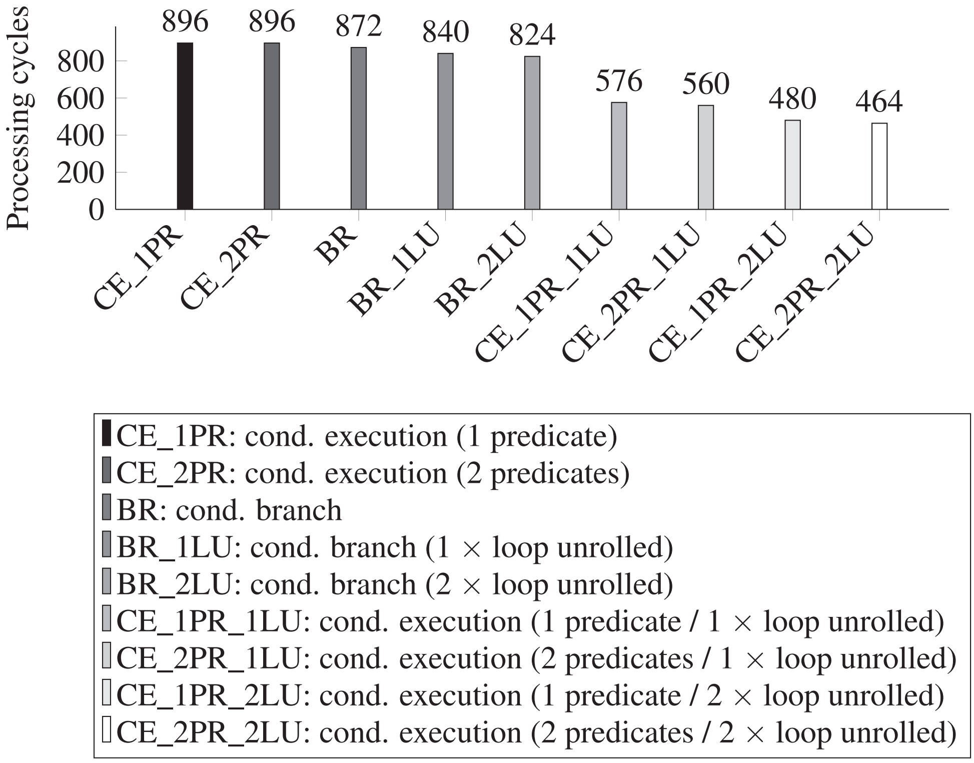


Figure 3.47: Processing performance in number of cycles for the bit-reversal permutation (Figure 3.44) of 32 elements with conditional branches, predication with one and two predicate registers with different levels of loop unrolling. Automatic operation merging is activated.

Case Study: Co-Processor Interface

This case study addresses the co-processor interface and algorithm, which require to store the predicate for multiple cycles. The conditional code example, shown in the block diagram in Figure 3.48, is based on the CORDIC algorithm. The quadrant correction determines quadrant of the input value and scales the input value for the computational demanding CORDIC iterations. When the iterations are done, the result is scaled and the quadrant is mapped based on the results from the input quadrant correction. This conditional output negation is based on condition flags. If only one predicate register is available, it is blocked for the CORDIC iterations. Multiple CORDIC computations in parallel are not possible. In this case 105 cycles are required for to cosine computations, using two hardware CORDIC accelerators for the CORDIC iterations and the *KAVUAKA* for the quadrant correction and mapping. If more than one predicate register is available, the cordic computations can be parallelized. Only 66 cycles are required, using two CORDIC accelerators in parallel.

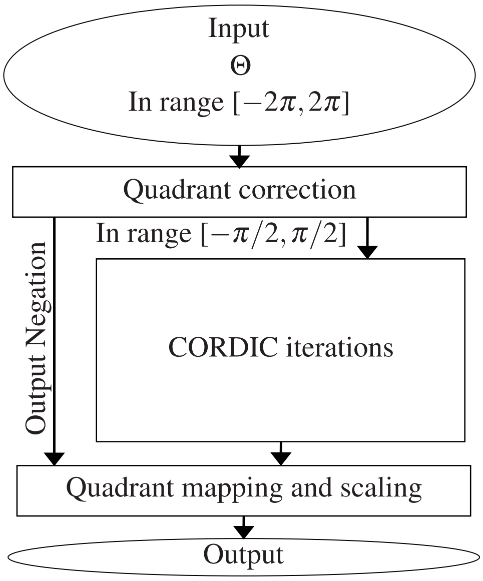


Figure 3.48: Block diagram of the CORDIC algorithm for computing a cosine function [195].

3.3 Specialization Towards Low-Power

Hard- and software specialization towards low-power is required during the design of hearing aid processors in order to meet battery life targets. In this section, new hard- and software architecture proposals towards low-power for application-specific hearing processors are presented.

3.3.1 Configurable Datapath Width

The data types for multimedia applications are derived from the sampling of continuous analog signals in the time domain [196]. The analog-to-digital converters of current hearing aids typically have 16-bit or 24-bit audio bit depth per sample [10, 11, 63, 197]. Due to limited power consumption and hardware resources, fixed-point arithmetic is used in hearing aids for digital signal processing [4, 68, 101, 104, 130]. For this reason, hearing aid algorithms are implemented on fixed-point hardware with fixed-point data types, while the fixed-point word length determines the rounding error, area, power, and power consumption of the target hearing aid system [139–141, 191, 198–200]. Therefore, the optimization of the word length is crucial for an efficient hearing aid system [196, 200].

This section introduces a generic and configurable datapath implementation for fixed-point processors. The goal of this implementation is the application-specific customization of the datapath of the hearing aid processor architecture. This customization includes a parameterizable bit-width of the datapath. The trade-off between computing power, hardware resources and power consumption is studied with this variable datapath width. In addition to the datapath width, the data-level parallelism (DLP) of the architecture is designed to be configurable. The datapath can be subdivided into several SIMD subwords. This enables the parallel processing of multiple configurable number of subwords per word. Multiple subwords can represent more complex data structures, such as the real and imaginary parts of complex numbers. With this option, the performance provided by the DLP of the architecture can be adapted to the requirements of the audio algorithms. If certain subword modes are not used in an application, the unused logic can be removed to reduce design complexity and power consumption. This includes datapath width, flag generation, result selection, or operand masking in the functional units.

With the generic datapath implementation, the effects of the datapath configuration on area, performance, power consumption and rounding errors are studied.

Related Datapath Width Optimizations

A system-level datapath width optimization is presented in [201]. The goal of the presented optimization is the minimization of the energy consumption of embedded systems by minimizing the number of redundant bits for each variable of a given application. The effective bit-width for each variable is determined by a variable size analysis. Based on the results of this analysis, the target application is rewritten using a Valen-C language to specify the word length of each variable in the application. The datapath width, the number of registers and the instruction-set of the target soft-core processor is modified based on this application. As a case study a MPEG-2 video decoder application is used and optimized with the proposed datapath optimization, resulting in a reduced energy consumption between 10.8 % and 48.3 % without any performance penalty.

In [202], a coarse-grained power gating mechanism for integer arithmetic circuits is presented, which is data-width-driven. The power gating mechanism is implemented in two different extends. The unused logic of an ALU unit for narrow-data-width computations or the complete unit is power gated in idle times. The considered units are adders and multipliers. Different architectures and power gating switch sizes are studied with an automatic design framework. For a 32-bit multiplier a leakage reduction of 11.6 × is achieved for 8 × 8-bit operations.

In order to reduce the register file pressure, the authors of [203] propose to pack multiple narrow-width data types in one sizable register of a superscalar processor architecture. Several deterministic and predictive micro-architectural techniques for the packing of multiple narrow-width values in one register are presented and the performance impact is studied. The applied techniques are based on the high predictability of the data width. Two register assignment schemes are evaluated, a conservative and a predictive one. The predictive scheme achieves 15 % IPC improvement for simulated benchmarks.

In [204], a so called *Multi-Bit-Width* microarchitecture is proposed, which takes the operands of 64-bit instructions and reuses these for multiple instructions with narrow-width operands instead. This re-partitioning of the datapath is designed for multiple instruction, multiple data (MIMD) processing. Since the operands of the existing instructions are used, the additional hardware resources are small. A speedup of 7.1 % for the simulated benchmarks is achieved.

Implementation of a Generic Datapath

For the implementation of the generic datapath, a set of parameters are introduced. These parameters define the bit-width (*op_data_w_c*) of the datapath as well as the DLP of the architecture (*op_sub32_active_c, . . .*). The complete parameter set is listed in Figure 3.49.

```
constant op_data_w_c         : natural := <OP_DATA_W>; -- datapath width
constant op_sub32_w_c        : natural := op_data_w_c/2;
constant op_sub16_w_c        : natural := op_data_w_c/4;
constant op_sub8_w_c         : natural := op_data_w_c/8;

constant op_sub32_active_c   : boolean := <OP_SUB32_ACTIVE>; -- true or false
constant op_sub16_active_c   : boolean := <OP_SUB16_ACTIVE>; -- true or false
constant op_sub8_active_c    : boolean := <OP_SUB8_ACTIVE>; -- true or false
```

Figure 3.49: Configuration parameters for defining the data bit-width (*op_data_w_c*) and available SIMD subword modes (*op_sub32_active_c, . . .*). [205]

With these set of parameters a similar study, as described in [201], can be performed to explore algorithms for digital hearing aid processors. Compared to [201, 203, 204], a datapath with the parallel processing mechanism single instruction, multiple data is investigated, since the SIMD format is a common and approved processing technique for hearing aid processing.

Table 3.18: Comparison of silicon area requirements depending on activated subword modes. An exemplary datapath width of 48-bit was selected. [205]. Results from an ASIC synthesis with the TSMC 40 nm HVT low-power ASIC technology at 50 MHz.

Subword modes	Area in mm^2	Combinational	Non-Combinational
_8 _16 _32 _64	(100 %) 0.0805	0.0650	0.0155
_16 _32 _64	(94 %) 0.0756	0.0601	0.0155
_32 _64	(90 %) 0.0722	0.0566	0.0155
_64	(85 %) 0.0685	0.0530	0.0155

Furthermore, it is used in this work also for handling complex-valued numbers in Section 3.2.1 and floating-point numbers in Section 3.2.2.

Different datapath configurations are chosen for evaluation. These are shown in Figure 3.50. The bit-width of the datapath is varied between 24-bit, 32-bit, 48-bit, and 64-bit. SIMD is implemented depending on the datapath width. SIMD instructions with two, four or eight subwords are enabled for the 64-bit, 48-bit and 32-bit configurations. SIMD is not enabled for the 24-bit configuration, due to the short datapath. The width of the subwords of the SIMD modes is half, quarter, and eighth times as long as the width of the entire datapath. All processors are also implemented without SIMD. These processor configurations execute scalar instructions only.

Datapath Area Evaluation

The silicon area requirement for the 48-bit processor implemented in the TSMC 40 nm HVT low-power ASIC technology is depicted in Table 3.18. Four SIMD subword modes are evaluated by activating one mode at a time. It can be seen that the processor configuration with deactivated subword modes is 12.5 % smaller than that with all subword modes activated. The part of combinational logic decreases when fewer subword modes are implemented, while non-combinational components (registers) remain unchanged. The largest increase in area of 6.8 % is due to the eight subwords mode, another 3.9 % due to the four subwords mode, and 1.8 % due to the two subwords mode. This can be explained by the fact that the separation of the datapath into subwords generally requires combinational logic for flag generation, result selection, and operand masking. The complexity of the datapath increases with finer separation.

The resulting silicon area for the processors with different datapath configurations is shown in Figure 3.51. The silicon area scales with the datapath width. The smallest configuration is the 24-bit configuration without SIMD support. Increasing the datapath to 32-bit leads to 29 % additional silicon area. The 48-bit configuration is 84 % larger then the 24-bit configuration

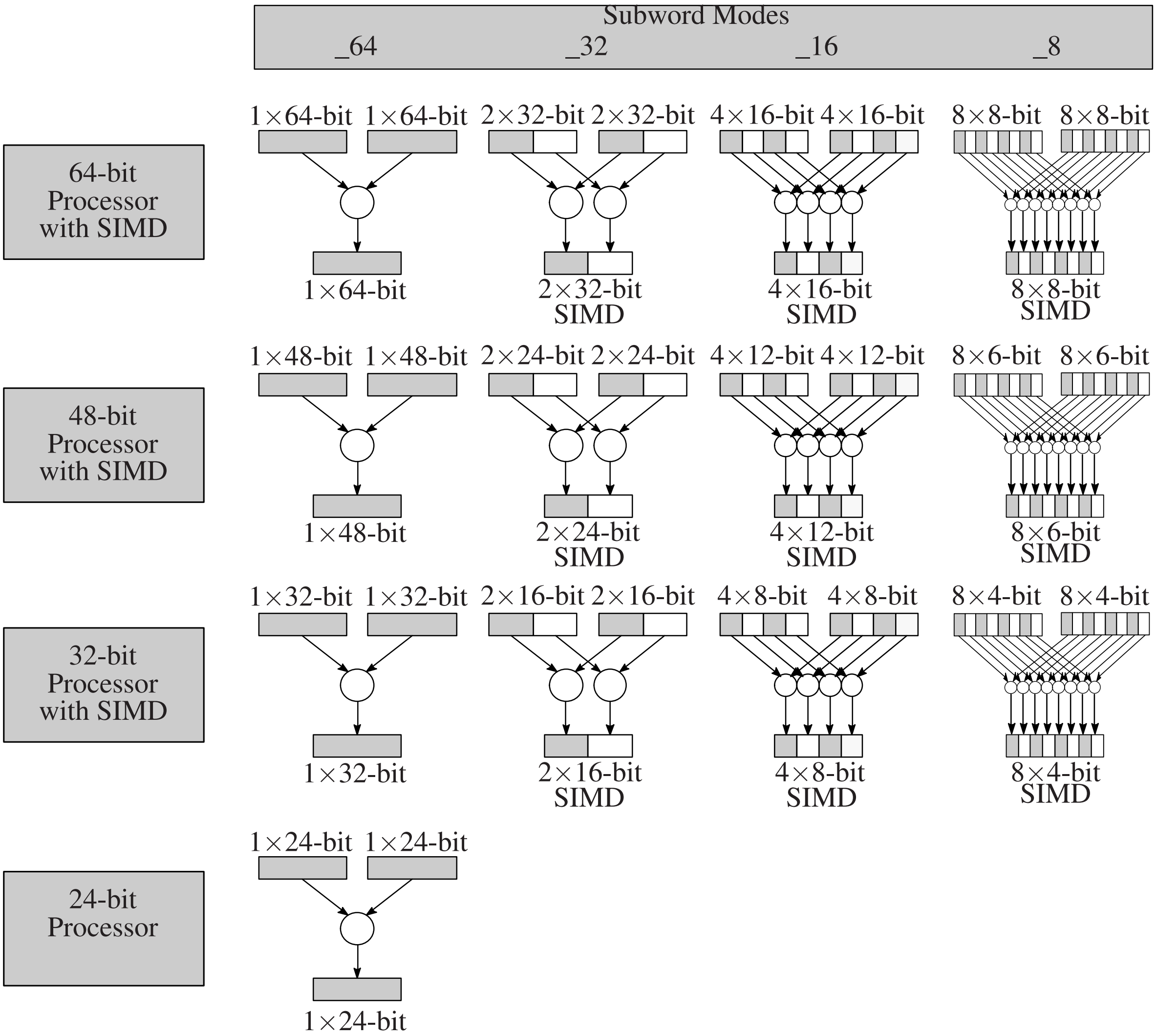


Figure 3.50: All processor configurations with a 24-bit, 32-bit, 48-bit, or 64-bit datapath bit-width. SIMD is not enabled for the 24-bit configuration.

and the 64-bit configuration is 72 % larger than the 32-bit configuration. The percentage area increase relative to the datapath width becomes smaller for larger datapath widths because the proportion of logic that is independent of the datapath width is almost unchanged. This logic includes, for example, the instruction decoder of the processor. By activating all SIMD subword modes the area increases by about 10 %, 17 % and 17 % for the 32-bit, 48-bit and 64-bit configurations. The activation of SIMD primarily increases the combinational logic.

The SRAM area and the required logic is shown in Figure 3.51. The number of addressable words is the same for all configurations, which is 4096. If SIMD is used, the number of words increases and the word length decreases accordingly. The area of the SRAM macros scales linearly with the bit-width.

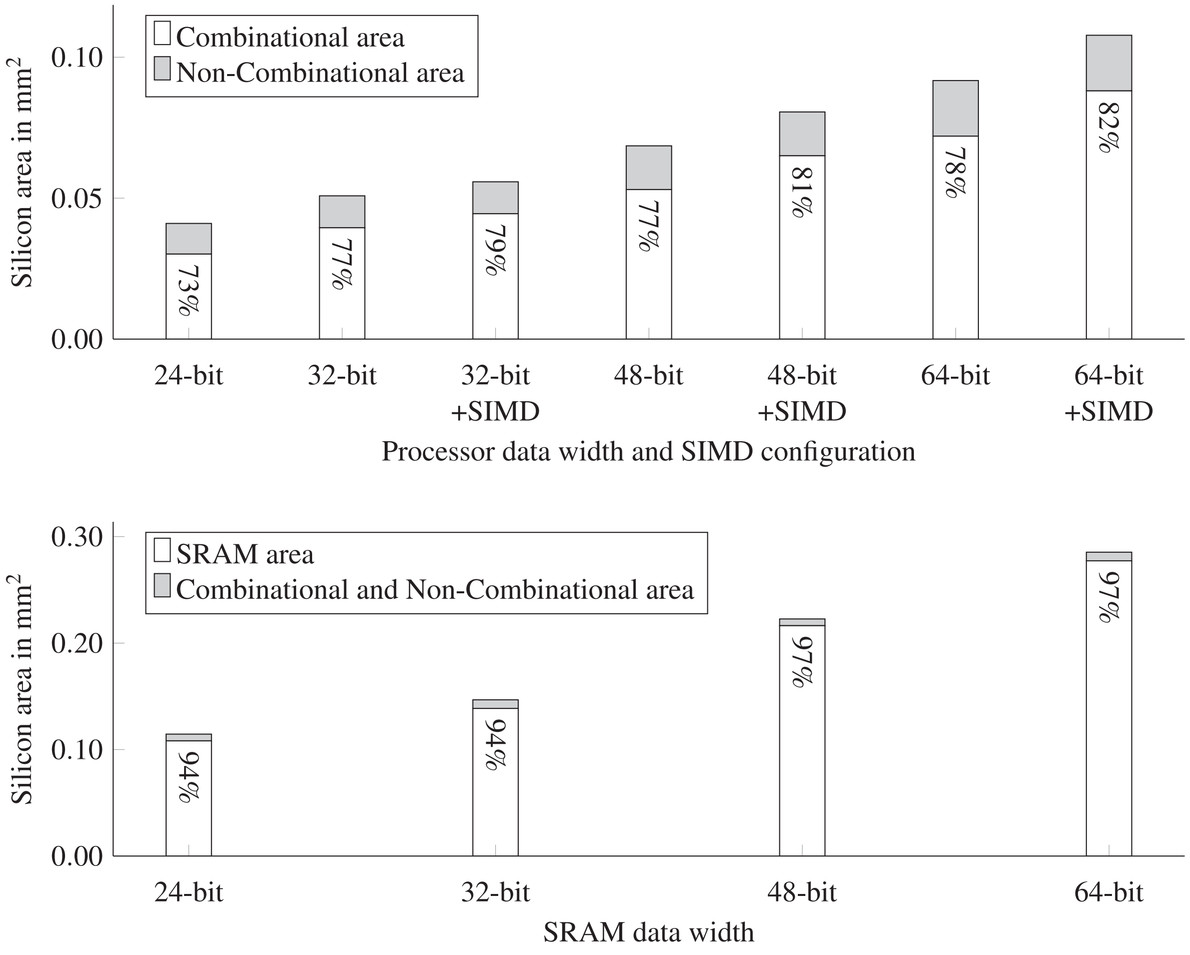


Figure 3.51: Combinational, non-combinational and SRAM silicon area for different datapath width and SIMD configurations. Results from an ASIC synthesis with the TSMC 40 nm HVT low-power ASIC technology at 50 MHz.

Datapath Performance Evaluation

The performance of different algorithm implementations on the different processor configurations is shown in Table 3.19. The number of cycles for implementations with SIMD and without SIMD are compared. For all FFT [128] variations of different sizes, the SIMD implementations need on average only 58 % of the cycles. The IPC values are over 1.8 for both implementations. The SIMD implementations of the FFT offer no advantage for the memory accesses. The required memory accesses are not aligned. The data is accessed in bit reversed or sequential order and the twiddle factors are accessed with varying offsets. As in [206], the computation of digital filters are parallelized by computing multiple filter instances at once with SIMD instructions. In this case two filters are computed concurrently using the two subwords (_32). The required number of cycles for the SIMD implementations is around 36 % smaller. No speed up is achieved for the beamforming algorithms. The beamforming algorithms [207] consist mainly of filter operations that are not accelerated by SIMD instructions. One exception for the adaptive beamformers are the *Lagrange* fractional delay filter for both microphone signals, which can be parallelized. The implementation of the *Lagrange* fractional delay filter with and without SIMD is depicted in Figure 3.52. Therefore, the total number of required cycles is negligibly smaller for the adaptive beamformers.

Table 3.19: Number of cycles and IPCs for the *KAVUAKA* processors with and without SIMD [128, 191, 205, 207].

	with SIMD (_32) (48-bit and 64-bit)		without SIMD (24-bit and 32-bit)	
	Cycles	IPC	Cycles	IPC
32-point-FFT [128]	615	1.8016	1084	1.8247
64-point-FFT [128]	1544	1.8187	2691	1.8339
128-point-FFT [128]	3752	1.8166	6442	1.8378
256-point-FFT [128]	8724	1.8261	14960	1.839
512-point-FFT [128]	19880	1.8339	33998	1.8438
1024-point-FFT [128]	44628	1.8401	75772	1.8571
17-tap FIR-Filter (2 ch.) [191]	136	1.8608	202	1.901
7-tap IIR-Filter (2 ch.) [191]	56	1.8407	96	1.8958
Fixed Beamforming [207]	22	1.8605	22	1.8182
Adaptive Gain Beamforming [207]	92	1.1813	103	1.3168
Adaptive Filter Beamforming [207]	148	2.000	149	1.9252
Adaptive GSC Beamforming [207]	122	1.7902	123	1.8843

```
// Filter computation with SIMD, each register contains two subwords
// Front and rear queue for processed in parallel (subwords)
MV_X2 REG_FREE_0+REG_FREE_1, 4(FIR_LAGRANGE_FILTER_COEFFICIENTS_DMEM_ADDR)

MV REG_FREE_2, (FIR_F1QUEUE_F_DMEM_ADDR)+ // front + rear queue in separate subwords
MV REG_FREE_3, (FIR_F1QUEUE_F_DMEM_ADDR)+

MAC_32 REG_FILTER_0_L+REG_FILTER_0_H, REG_FREE_0, REG_FREE_2
MAC_32 REG_FILTER_0_L+REG_FILTER_0_H, REG_FREE_1, REG_FREE_3
```

```
// Filter computation without SIMD, each register contains one data word
// Front and rear queue for processed sequentially
MV_X2 REG_FREE_0+REG_FREE_1, 4(FIR_LAGRANGE_FILTER_COEFFICIENTS_DMEM_ADDR)

MV REG_FREE_2, (FIR_F1QUEUE_F_DMEM_ADDR)+ // front queue
MV REG_FREE_3, (FIR_F1QUEUE_F_DMEM_ADDR)+

MAC_64 REG_FILTER_0_L+REG_FILTER_0_H, REG_FREE_0, REG_FREE_2
MAC_64 REG_FILTER_0_L+REG_FILTER_0_H, REG_FREE_1, REG_FREE_3

MV REG_FREE_2, (FIR_R1QUEUE_F_DMEM_ADDR)+ // rear queue
MV REG_FREE_3, (FIR_R1QUEUE_F_DMEM_ADDR)+

MAC_64 REG_FILTER_1_L+REG_FILTER_1_H, REG_FREE_0, REG_FREE_2
MAC_64 REG_FILTER_1_L+REG_FILTER_1_H, REG_FREE_1, REG_FREE_3
```

Figure 3.52: Assembler code excerpts from subword-based and subword-less filter implementations for beamforming algorithms [207]. The *Lagrange* filter coefficients must be multiplied and accumulated with audio data from a front queue and a rear queue. While in the example above the front and rear data can be processed parallel to each other, this is not possible with a configurations without SIMD (lower code block). Sequential scalar instructions are necessary, which entails a doubling of the code, the file indirect registers (FIREGs) and the necessary temporary and result registers. [205]

Data Path Precision Evaluation

The available data types and formats for an algorithm implementation depend on the datapath configuration of the target processor architecture. Therefore, the overall computational precision and the final accuracy of the algorithm implementation depends on the datapath. To evaluate this precision, different algorithms with different datapath configurations are implemented.

An assembler code excerpt of data movement and addition is shown in Figure 3.53 for a 48-bit and 64-bit datapath configuration. In both cases, subwords are used (_32). These subwords have half the word length. Compared to the 64-bit datapath, the 48-bit datapath provides fewer most significant bits (MSBs) for integer representation and fewer least significant bits (LSBs) for fixed-point representation.

```
// 48 bit datapath width
MVIL_32  V0R0, #0x0f, #0x1234
MVIL_32  V0R0, #0xf0, #0x5678   // V0R0 <- 0x005678001234
ADDIL_32 V0R1, V0R0,  #0x90000  // V0R1 <- 0x095678091234
```

```
// 64 bit datapath width
MVIL_32  V0R0, #0x0f, #0x1234
MVIL_32  V0R0, #0xf0, #0x5678   // V0R0 <- 0x0000567800001234
ADDIL_32 V0R1, V0R0,  #0x90000  // V0R1 <- 0x0009567800091234
```

Figure 3.53: Assembler code excerpt showing operations on 64 bit and 48 bit wide data registers. The SIMD operation mode (_32) uses two subwords of half word length, 24 bit and 32 bit. [205]

The precision evaluation for different datapath configurations and different algorithms is listed in Table 3.20. The input signal is speech in noise with a word length equal to the datapath width. The evaluated algorithms are either feedforward or feedback systems [208]. The feedforward algorithms that do not include a feedback loop on the data are the FFT, FIR and the fixed beamforming algorithms. The FFT size defines the number of FFT stages, which determine the number of operations performed on the data. Larger FFT sizes therefore decrease the overall precision [209]. Since the implementation of the FFT algorithm performs a scaling procedure after each FFT stage by shifting the data by one in order to prevent overflows, the overall computing precision increases with the datapath width. However, the precision of algorithms with a feedback loop is significantly smaller for smaller datapath widths. Finite-word-length effects are the cause for this behavior [140].

Current and future ADCs of hearing aids convert the analog microphone signal into up to 24-bit digital audio data [210]. In this thesis, this source audio bit-width and its effect on the datapath configuration and the algorithms is examined in the following. In order to evaluate, which datapath is suitable for which algorithm, the word length, which is equivalent to the

Table 3.20: Maximum and average absolute error of different fixed-point hearing aid algorithms compared to the reference double-precision floating-point implementation. The input signal for this evaluation is speech in noise. [128, 191, 205, 207]

	Fixed-Point Word Length					
	24 bit		32-bit		48-bit	
	max. abs. error	mean abs. error	max. abs. error	mean abs. error	max. abs. error	mean abs. error
32-point-FFT [128]	2.5614e-06	1.1859e-06	1.0143e-08	5.0515e-09	8.2634e-13	3.3303e-13
64-point-FFT [128]	3.5559e-06	1.6208e-06	1.5221e-08	6.352e-09	1.7416e-12	7.1892e-13
128-point-FFT [128]	6.1775e-06	2.2241e-06	2.5002e-08	8.7475e-09	3.4409e-12	1.449e-12
256-point-FFT [128]	8.8791e-06	3.2767e-06	4.1998e-08	1.2614e-08	8.3771e-12	3.1056e-12
512-point-FFT [128]	1.3608e-05	4.63e-06	5.1648e-08	1.8475e-08	1.7352e-11	6.2829e-12
1024-point-FFT [128]	6.5191e-04	2.1287e-04	2.3142e-06	8.3655e-07	3.4424e-11	1.2671e-11
Fixed Beamforming [207]	2.716e-07	7.171e-08	7.015e-10	2.495e-10	7.015e-10	2.495e-10
Adaptive Gain Beamforming [207]	1.020e-02	2.395e-05	2.528e-04	3.756e-07	2.711e-04	3.859e-07
Adaptive Filter Beamforming [207]	2.626e-04	1.335e-05	9.508e-07	5.265e-08	7.603e-10	2.356e-10
Adaptive GSC Beamforming [207]	1.491e-04	7.617e-06	5.705e-07	3.053e-08	1.075e-09	4.676e-10
17-tap FIR-Filter [191]	1.499e-06	9.913e-07	5.416e-09	3.871e-09	5.958e-10	2.335e-10
7-tap IIR-Filter [191]	1.995e-03	5.071e-04	8.452e-06	2.675e-06	9.526e-07	2.934e-07
DNN [24]	0.00354	0.00305	1.3828e-05	1.1921e-05	2.0373e-10	1.819e-10

datapath width if no SIMD is used, is varied between 4-bit and 40-bit. The word length of the input signals to the algorithm are limited to 24-bit. The range of the input signals is scaled to -1 to 1. The maximum absolute error, compared to the double-precision floating-point implementation, is measured at the algorithm output stage. The bit-width of the datapath is varied.

In Figure 3.54 the influence of the word length on the FFT computing precision is shown. The FFT size is varied between 32- and 1024-points. The maximum absolute error is higher for larger FFT sizes, due to increased number of stages and butterfly operation and error propagation. The absolute error deceases exponentially with larger datapath widths. With a word length of larger than 36-bit, the error only decreases insignificantly. Therefore, datapath configurations with word lengths larger than 36-bit do not increase the computing precision for this FFT implementation, when the input signal is 24-bit wide.

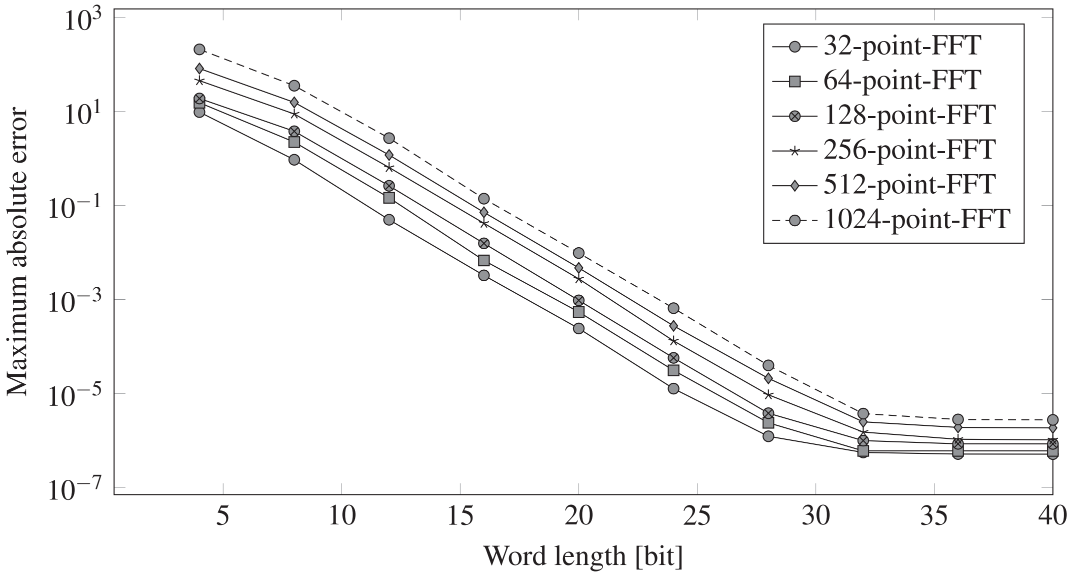


Figure 3.54: Maximum absolute error for the FFT algorithm with different datapath widths. FFT sizes from 32 to 1024 are shown. The word length of the input signal is 24-bit.

The precision analysis for digital filters is shown in Figure 3.55. Although the tap size of the FIR filter is higher, the overall absolute error is lower for the same word length, approximating the same frequency response. One of the reasons is the feedback loop in the IIR filter. Word lengths larger than 32-bit do not increase the overall precision significantly.

The maximum absolute error for the beamforming algorithms for different word lengths is shown in Figure 3.56. Due to the lowest computing complexity and no feedback loop, the fixed beamformer offers the lowest error compared the double-precision floating-point im-

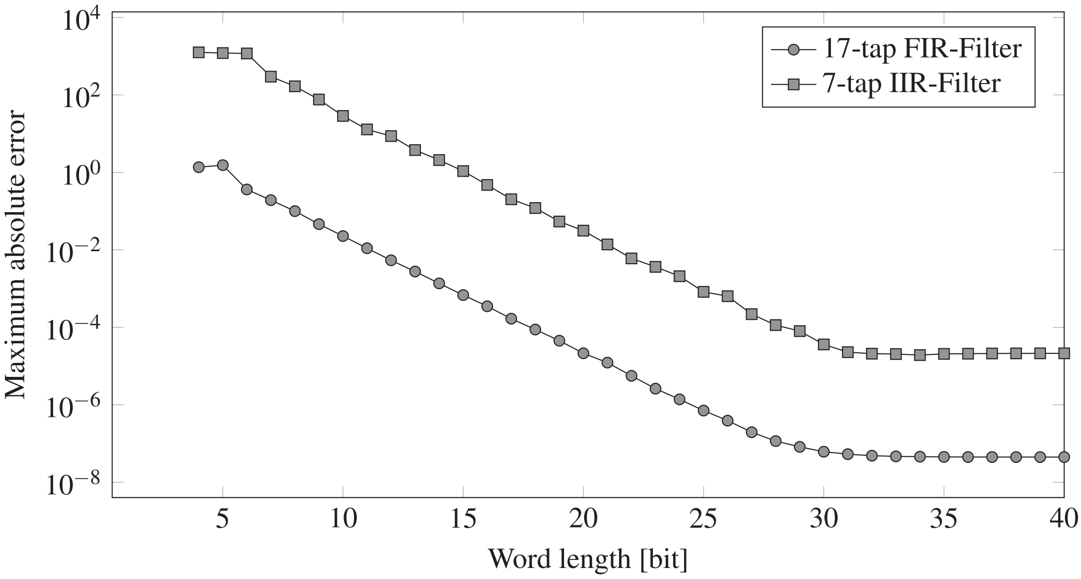


Figure 3.55: Maximum absolute error for digital filter algorithms, implemented with different word lengths. The word length of the input signal is 24-bit.

plementation. The almost lowest maximum absolute error is achieved with a word length of around 28-bit. The adaptive filter and the adaptive GSC beamformer have a similar error characteristic. Due to the fixed-point division operation in the feedback loop, the error of the adaptive gain beamformer is comparatively high and fluctuates. The adaptive beamforming algorithms require around 36-bit word length in order to achieve the highest precision for a 24-bit input signal. The influence of the error induced by changing the word length is further studied in Section 5.1.1. This chapter examines the influence of word length on speech intelligibility, among other things.

3.3.2 Dummy Register and Register Address Isolation

The register file (RF) of the *KAVUAKA* processor contains two separate register files, each with 32 register. These are accessed by four read and two write ports [16, 83]. The register size depends on the datapath width, which is varied between 24-bit to 64-bit, as described in Section 3.3.1. The silicon area of the register file with 64-bit accounts for 40 % the total silicon area of the *KAVUAKA* processor and it accounts on average for about 45% of the *KAVUAKA* processor's power consumption. A standard cell placement of the *KAVUAKA* processor including the register file is shown in Figure 3.57a. The area of the register file is composed of combinational/logic and non-combinational/register area. The area ratio for this register file implementation is given in Figure 3.57b.

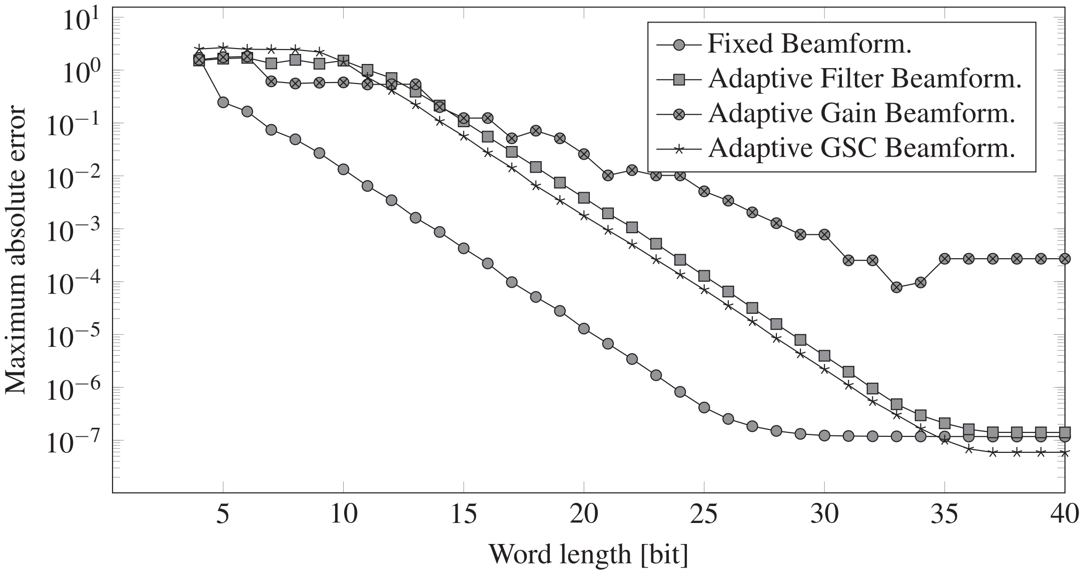
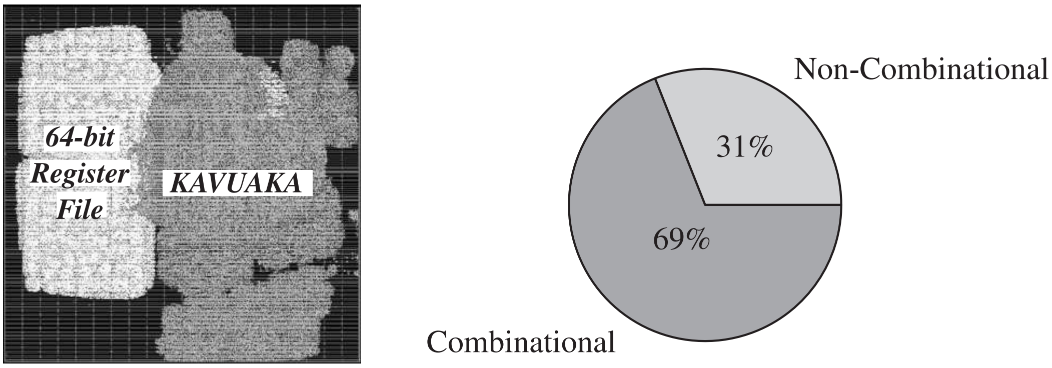


Figure 3.56: Maximum absolute error for beamforming algorithms, implemented with different word lengths. The word length of the input signal is 24-bit.



(a) Layout view of the register file placement [211]

(b) Area distribution of the register file

Figure 3.57: (a) Area of the register file compared to the area of the remaining *KAVUAKA* processor for an unconstrained placement using the TSMC 40 nm HVT low-power ASIC technology. [211]. (b) Combinational and non-combinational (sequential) area of the register file. [207]

Due to the high silicon area requirements of the register file compared to the other components of the processor (Figure 3.57), the power consumption optimization of the register file is crucial. A common optimization approach is the handling of temporary or so called short-lived values [16, 212–215]. Transferring these temporary values within the processor, e.g., between the register file and the functional units, causes additional processing time, which results in additional power consumption and silicon area requirements [215]. In order to prevent these issues, register file bypass and isolation techniques are presented in the related work [16, 85, 212–217].

The authors of [213] propose a multi-stage bypass pipeline for VLIW processors. This approach is compiler driven. Whether an instruction reads from the register file or from one of the bypass register is determined by the compiler and encoded in the register operand bits.

The register file activity is reduced in [214] by avoiding writing the short-lived variables to the register file and using the forwarding paths of the VLIW pipeline architecture. The compiler encodes the writeback of those registers, which need to be stored in the register file, within the instructions.

A bypassing method for datapath architectures is presented in [215]. A software bypassing is proposed, where the compiler schedules data transfers between the execution units of the datapath architecture. The results show, that software bypassing decreases the number of required read or write accesses to the register file, which results in a reduction in the processor energy consumption.

In [216] the short-lived variables are stored in a smaller and dedicated register file to isolate these from the remaining variables stored in the larger multi-ported register file. The achieved energy saving is about 20 % to 25 % in the large register file, which results in a total reduction of 5 % for the considered processor. A similar approach is used in [217]. Here the power of the register file accesses can be reduced by 30 %.

The approach to decrease the power consumption within the register file, which is presented in this section, is based on the dummy register mechanism [16, 85, 212] in combination with an multi-port address isolation technique. The architecture of the dummy register mechanism is shown in Figure 3.58. Dummy registers are registers in the register file, whose read and write accesses may be bypassed. This bypass is activated for temporal short-lived variables, using the dummy register addresses.

The compiler determines which register can be dummy registers by detecting the use of short-lived variables in the application. During optimization, the compiler tries to maximize the allocation of dummy registers instead of physical registers, decreasing the dynamic power consumption of the register file.

Hearing aid algorithms of single straight line microcodes (SLMs) differ with regard to the number of short-lived variables, which can be allocated. Therefore, the number of dummy registers and bypass options is configurable during runtime. The maximum number is four, if two X2 VLIW instructions write two target registers each. However, a constant bypassing of four registers is not beneficial, if many registers are needed to store long-lived variables. Therefore, a programmable configuration register is introduced, which defines the number of dummy registers during runtime as shown in Figure 3.58.

The second proposed approach to decrease the power consumption within the register file is the address isolation of the register file. Like other register files of current digital signal processors, the register file of the *KAVUAKA* processor has many read and write ports to feed all parallel working functional units (FUs). In this case, four read and two write ports are used to address 32 registers in each of the two partitioned register banks. These ports account for around 14 % of the register file silicon area. To decrease the switching activity and therefore reduce the power consumption, the last read and write address of each port is held by an additional register, if this port is temporarily not used. This is case if no read or write access is performed or the addressed register is a dummy register. The address isolation mechanism is shown in Figure 3.58.

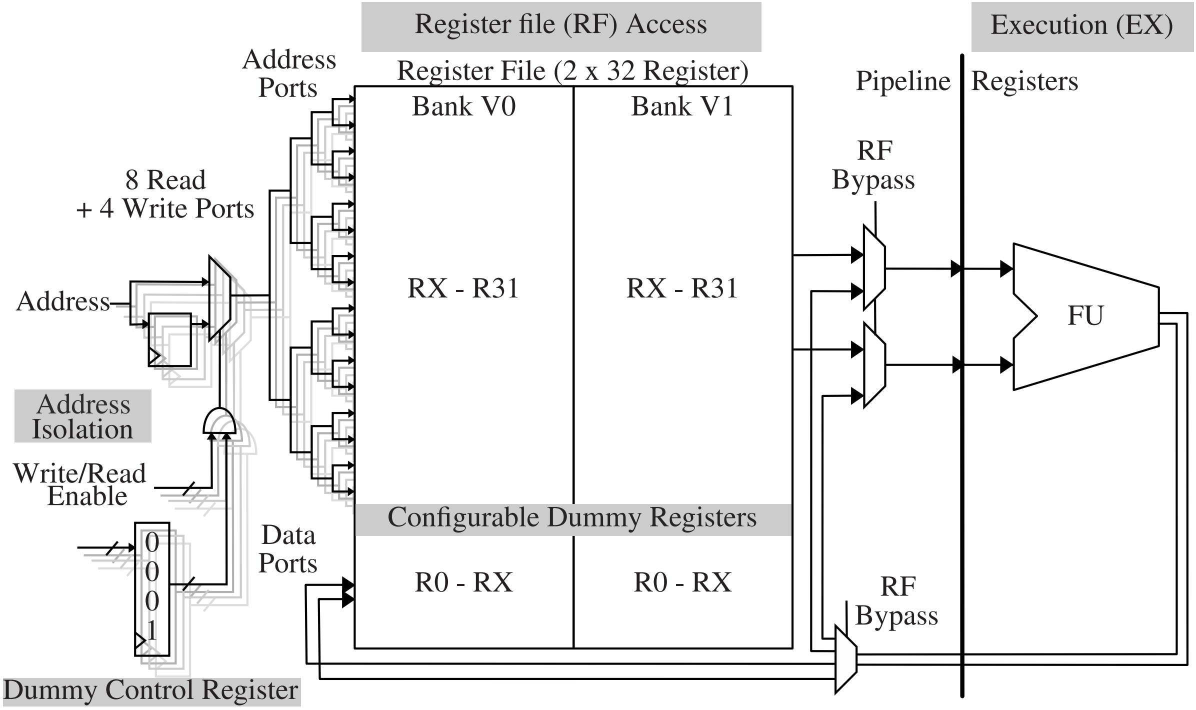


Figure 3.58: Register file (RF) implementation with address isolation and dummy registers [207].

The silicon area overhead for implementing the dummy registers and the address isolation mechanism in addition to the register file is depicted in Figure 3.59. Compared to the unoptimized reference register file, the silicon area increases by 2.3 % to 3.6 % for the isolation hardware. If the dummy registers are additionally added, this overhead rises to 2.4 % to 4.4 %. The total silicon area overhead for the *KAVUAKA* processor is shown in Figure 3.60.

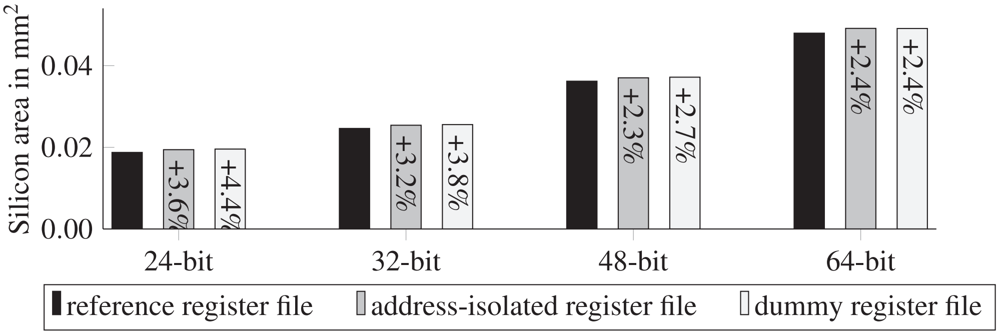


Figure 3.59: Register file (RF) silicon area overhead for the unoptimized reference register file, the address-isolated register file and the dummy register file.

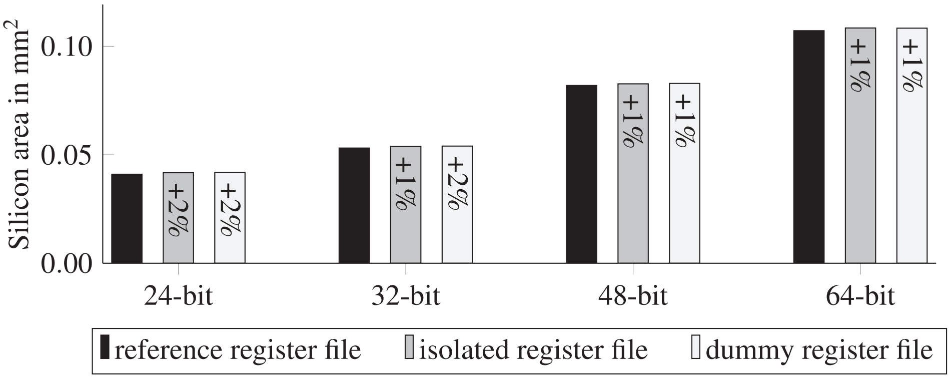


Figure 3.60: Total silicon area overhead for the unoptimized reference register file, the isolated register file and the dummy register file.

The power consumption for these three register file configurations, including the optimizations, is evaluated with beamforming algorithms. The static and dynamic power consumption comparison to the unoptimized reference register file is shown in Figure 3.61. The power consumption decreases by 4 % to 10 % for the address-isolated register file. With dummy registers, the power consumption in the register file can be decreased by 6 % to 17 %. The power consumption saving by the dummy register optimization depends on the utilization of dummy registers for a given application.

The dynamic utilization is given in Table 3.21. These values are based on the maximum number of addressable register read and write ports per cycle, which is 12 in case of the eight read and four write ports. If neither a register or a dummy register is read or written by one

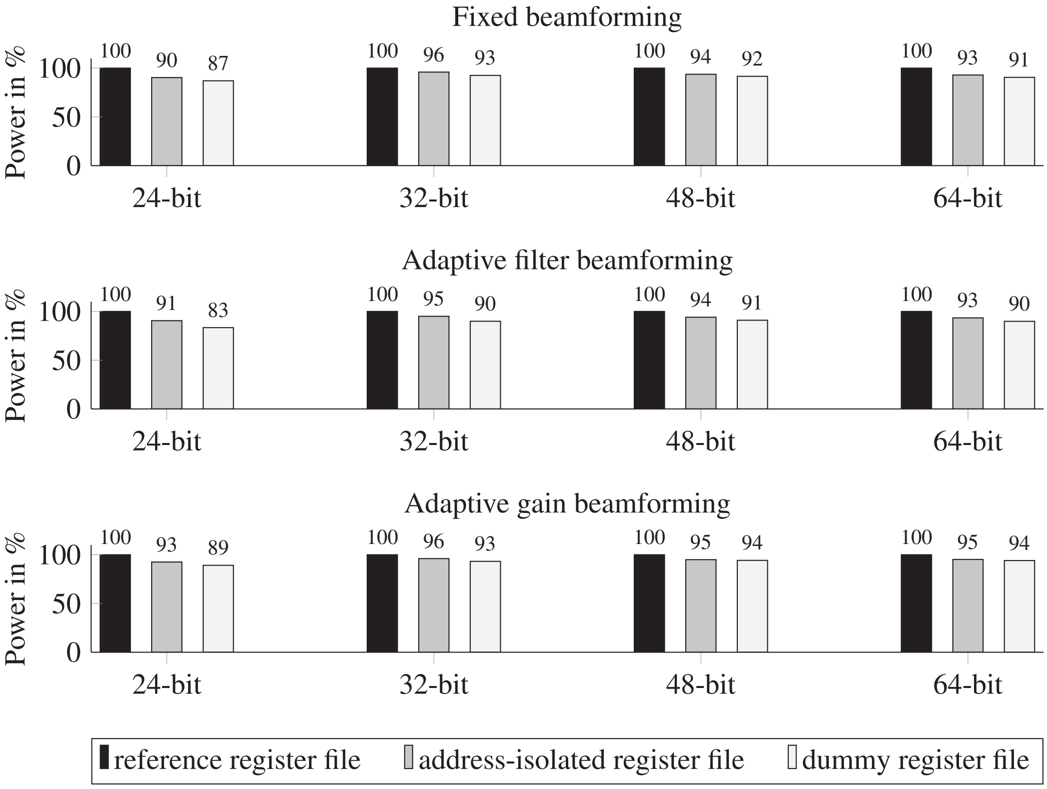


Figure 3.61: Power consumption of the address-isolated register file and dummy register file compared to unoptimized register file for the fixed beamformer, the adaptive filter beamformer and the adaptive gain beamformer.

port, this port stays unused. If the dummy usage is high compared to the standard register usage, which is noticeably the case for the 24-bit and 32-bit adaptive filter beamformer, the power consumption decreases.

Table 3.21: Register and Dummy Usage for Beamforming Algorithms.

Fixed beamformer				
Bit width	24-bit	32-bit	48-bit	64-bit
Register usage	24.19 %	21.22 %	23.99 %	27.03 %
Dummy usage	5.24 %	8.22 %	7.74 %	4.71 %
No usage	70.57 %	70.56 %	68.27 %	68.26 %

Adaptive filter beamformer				
Bit width	24-bit	32-bit	48-bit	64-bit
Register usage	20.91 %	20.74 %	25.0 %	25.97 %
Dummy usage	17.04 %	17.37 %	11.82 %	11.52 %
No usage	62.05 %	61.89 %	63.14 %	62.51 %

Adaptive gain beamformer				
Bit width	24-bit	32-bit	48-bit	64-bit
Register usage	21.31 %	20.95 %	17.58 %	16.49 %
Dummy usage	7.12 %	7.22 %	8.76 %	9.01 %
No usage	71.52 %	71.83 %	73.66 %	74.50 %

3.3.3 Low-Level Low-Power Optimization Techniques

Operand Isolation

In order to reduce the dynamic power consumption in the functional units (FUs), the commonly used operand isolation technique presented in [218, 219] is used. Operand isolation reduces the dynamic power consumption of combinational logic circuits by preventing the propagation of switching activity into the logic, when the logic is not used. The inputs of all functional units of the *KAVUAKA* processor are selected for this technique. Among the FUs are two move units, two bit logic units, two arithmetic units, two shift round units, two permutation units, two min/max units and a complex-valued multiply-accumulate (CMAC) unit. The FUs account for 60 % of the silicon area of one *KAVUAKA* processor with the X2 mode. The isolation technique used in this case is based on AND gates isolation. This technique is used instead of the latch based technique due its simplicity in terms of design constraints and the usage statistics of the FUs, which show that most of the FUs are not used for multiple cycles regularly.

The internal, switching and leakage power estimation based on operand isolation of all inputs of all FUs of *KAVUAKA* processors with different data paths is shown in Table 3.22. Four different datapath width, ranging from 24-bit to 64-bit, of the *KAVUAKA* processor are evaluated. The operand isolation technique decreases the total power consumption by up to 44 %. Less power reduction is achieved for the smaller datapath widths since the ratio between the isolated logic compared to remaining processor components becomes smaller. The leakage power is not affected by operand isolation and does not affect the total power.

Table 3.22: Internal, switching and leakage power estimation based on operand isolation of all inputs of the FUs of *KAVUAKA* processors with different data paths. Results from an ASIC synthesis with the TSMC 40 nm HVT low-power ASIC technology at 50 MHz.

	Datapath	Internal Power	Switching Power	Leakage Power	Total Power
64-bit	without isolation	2.5072 mW	1.6560 mW	0.0012 mW	4.1645 mW
	AND-isolation	1.6577 mW	0.8085 mW	0.0012 mW	2.4674 mW
		(-44 %)	(-51 %)	(-0 %)	(-41 %)
48-bit	without isolation	1.7777 mW	1.0923 mW	0.0009 mW	2.8709 mW
	AND-isolation	0.9364 mW	0.6725 mW	0.0009 mW	1.6099 mW
		(-48 %)	(-39 %)	(-0 %)	(-44 %)
32-bit	without isolation	1.0831 mW	0.4844 mW	0.0005 mW	1.5680 mW
	AND-isolation	0.6126 mW	0.2960 mW	0.0005 mW	0.9091 mW
		(-44 %)	(-39 %)	(-0 %)	(-43 %)
24-bit	without isolation	0.6990 mW	0.2418 mW	0.0002 mW	0.9411 mW
	AND-isolation	0.6183 mW	0.1802 mW	0.0002 mW	0.7988 mW
		(-12 %)	(-26 %)	(-0 %)	(-16 %)

The additional area requirements for the operand isolation implementation based on AND gates is shown in Table 3.23.

Table 3.23: Total area based on operand isolation of all inputs of all FUs of *KAVUAKA* processors with different data paths. Results from an ASIC synthesis with the TSMC 40 nm HVT low-power ASIC technology at 50 MHz.

	Datapath	Total Area
64-bit	without isolation	135,750.20 µm^2
	AND-isolation	135,913.37 µm^2
		(+0.12 %)
48-bit	without isolation	99,550.81 µm^2
	AND-isolation	101,008.40 µm^2
		(+1.46 %)

	Datapath	Total Area
32-bit	without isolation	59,339.90 µm^2
	AND-isolation	59,515.77 µm^2
		(+0.29 %)
24-bit	without isolation	36,099.73 µm^2
	AND-isolation	36,255.84 µm^2
		(+0.43 %)

Pipeline Architecture

The number of required pipeline stages of an ASIP architecture depends on several requirements and conditions. The ability to customize the pipeline architecture of the processor allows adaptation to the requirements of the target applications. The maximum number of pipeline stages considered in this work are six pipeline stages, which were originally proposed in [16]. These pipeline stages are instruction fetch (IF), instruction decode (DE), register access (RA), execution (EX) and write back (WB), whereas the EX stage is subdivided into two stages. The pipeline architecture of the *KAVUAKA* processor with the five and the minimum number of two of pipeline stages is depicted in Figure 3.62. The EX stage may be divided into two pipeline stages for performance purposes, since the critical path is located in the complex functional units (FUs). For the two pipeline stage configuration, the IF, DE and the RA pipeline stages and the EX and the WB pipeline stages are merged into one. The pipeline registers, holding the decoded instructions, control signals or the processed data, are removed. No forwarding paths in the EX stage are required, since the register file can be bypassed using the WB paths.

In order to analyze the area requirements for the pipeline registers of the *KAVUAKA* processor, the pipeline stages are synthesized with and without the pipeline registers. The synthesis results are shown in Figure 3.63. The ASIC synthesis are performed with the TSMC 40 nm HVT low-power ASIC technology at the target clock frequency of 50 MHz. When pipeline registers are used, the silicon area requirements are increased for all pipeline stages. The increase is caused by additional non-combinational (sequential/register) area. The combinational and buffer/inverter area is almost equal, which indicates that the synthesis tool did not insert additional buffers of logic in order to meet the timing constraints. Based on the ratio between combinational and sequential logic of the underlying pipeline architecture, the differ-

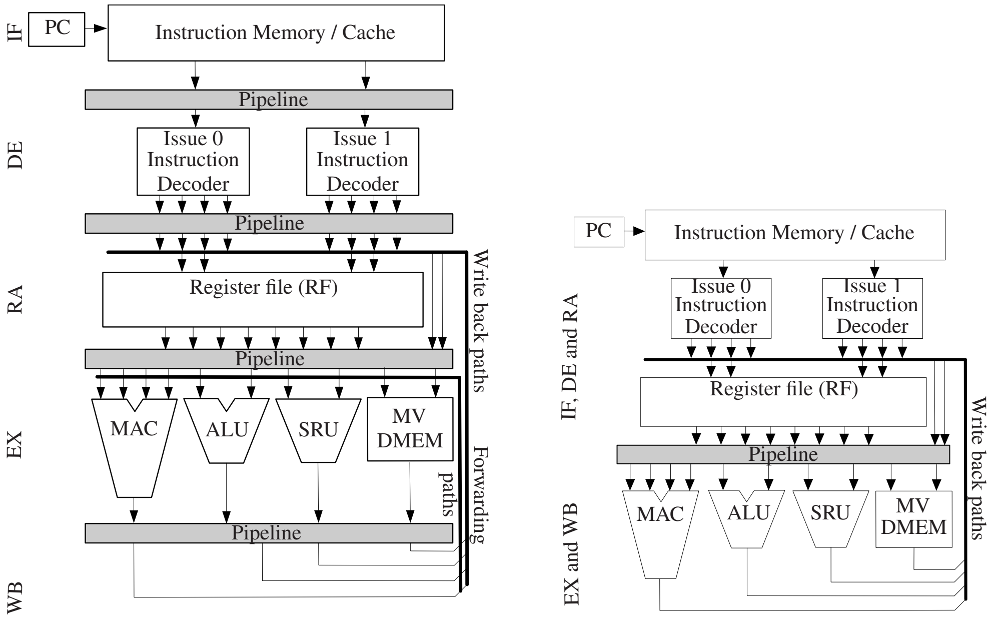


Figure 3.62: *KAVUAKA* processor with two and five pipeline stages. The EX stage can be subdivided into two pipeline stages, in order to meet timing constraints.

ence in the total area between the pipelined and non-pipelined version ranges between 3.6 % to 31.6 %.

The estimated total power consumption for the pipeline stages with and without pipeline registers is shown in Figure 3.64. The power is estimated after ASIC synthesis using a static probability for the signal values and a constant toggle rate for a given period. The total power consumption is reduced for pipeline stages without pipeline registers. The deactivation of the pipeline stages has the biggest influence on the internal power dissipation. The switching power stays the same. The leakage power is negligible with 0.03 % of the total power. The total reduction ranges between 26.5 % to 47.5 %.

The total silicon area and estimated total power consumption for the *KAVUAKA* 64-bit processor with different pipeline register configurations and different target clock frequencies is shown in Figure 3.65. The target clock frequencies is set to 40 MHz, 50 MHz or 60 MHz.

In the case of low-power hearing aid applications, the processing performance requirements are less strict, compared to multimedia applications [16], and the power and area constraints are of higher importance [83,198]. The processing performance requirements are derived from the real-time processing constraints of the digital audio streams, generated by the analog-to-digital converters. Commonly, two channels sampled at 16 kHz are processed per hearing aid device [63]. Two audio stream processing schemes are present, sample/sequential- and frame/block-based processing [220]. The minimum clock frequency for a given processor

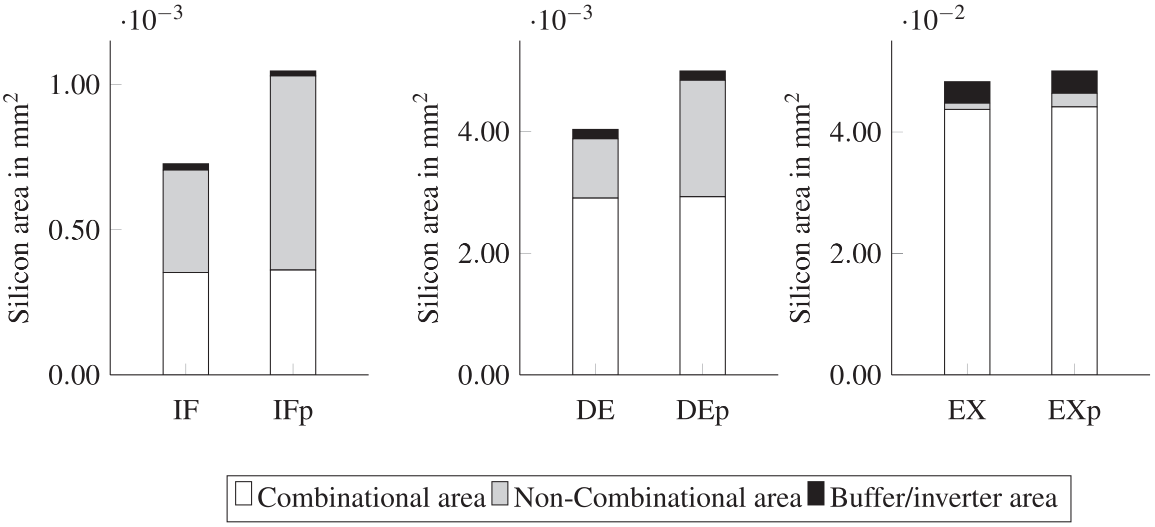


Figure 3.63: Combinational, non-combinational and buffer/inverter silicon area for all pipeline stages with (IFp, DEp and EXp) and without (IF, DE and EX) pipeline registers. Results from an ASIC synthesis with the TSMC 40 nm HVT low-power ASIC technology at 50 MHz.

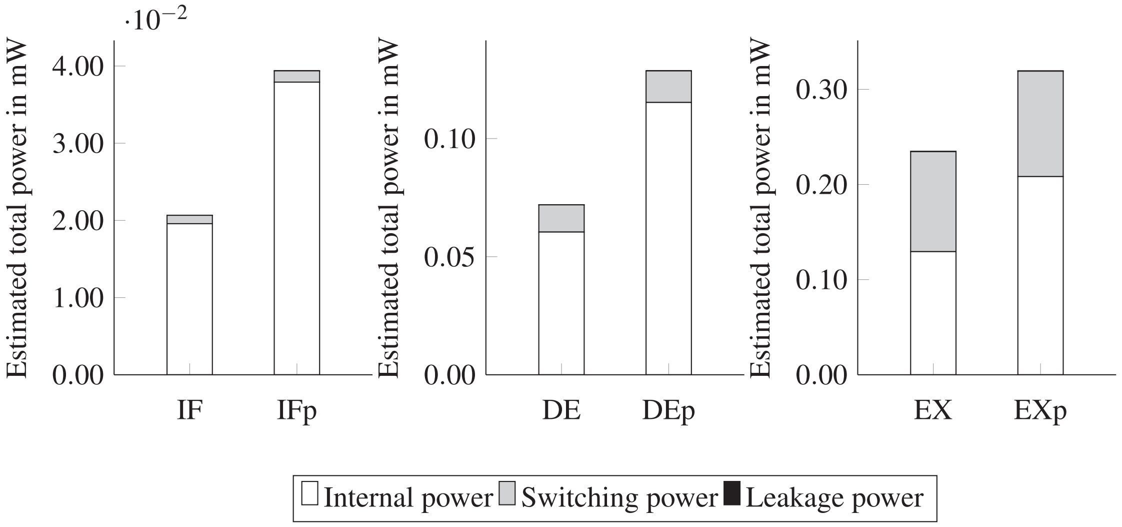


Figure 3.64: Internal, switching and leakage power estimation for all pipeline stages with (IFp, DEp and EXp) and without (IF, DE and EX) pipeline registers. Results from an ASIC synthesis with the TSMC 40 nm HVT low-power ASIC technology at 50 MHz.

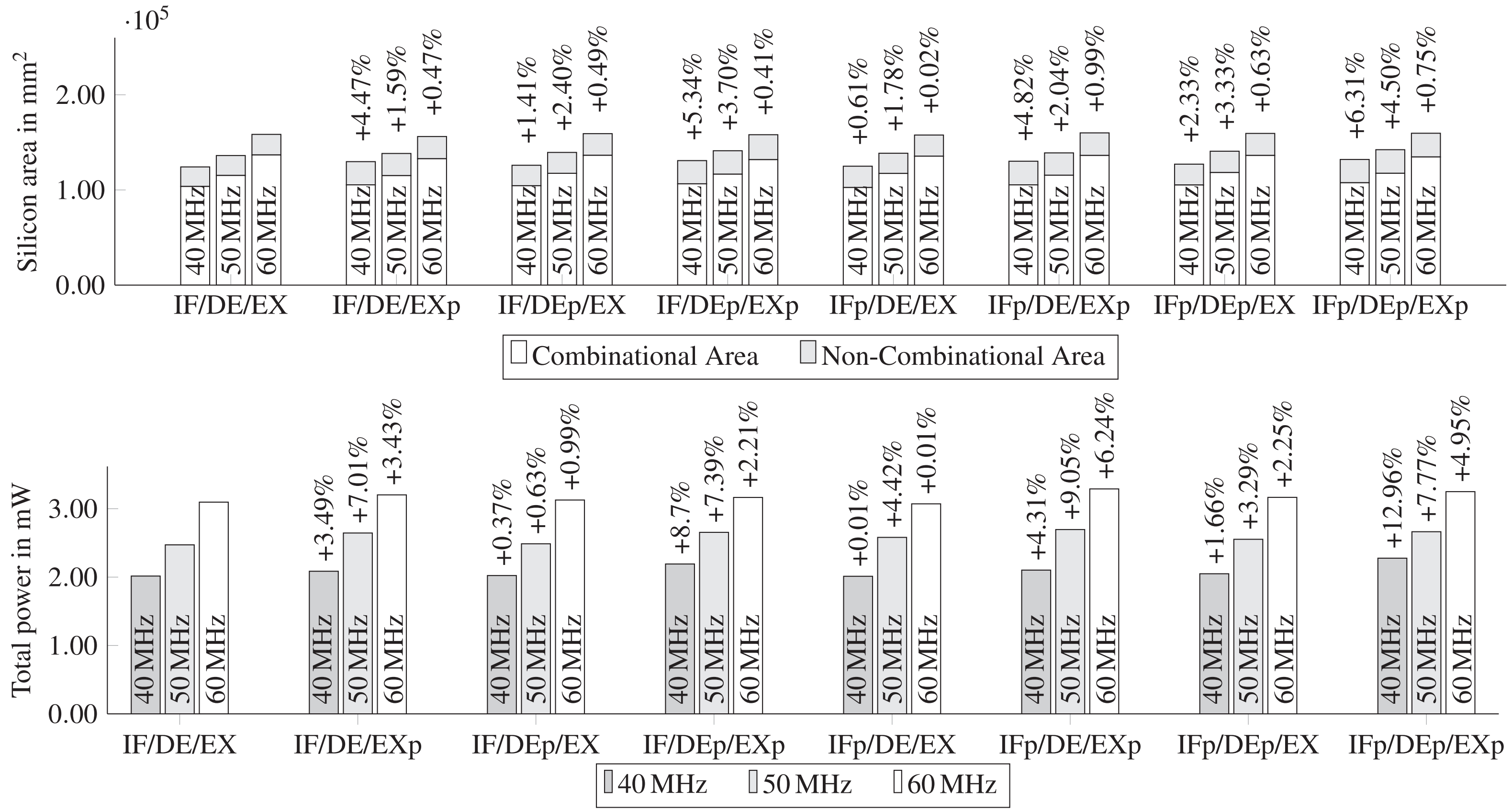


Figure 3.65: Total silicon area and estimated total power consumption for different pipeline register configurations. The pipeline stages are configured with (IFp, DEp and EXp) and without (IF, DE and EX) pipeline registers. The relative percentage deviation is given compared to non-pipelined version IF/DE/EX. Results from an ASIC synthesis with the TSMC 40 nm HVT low-power ASIC technology at 40 MHz, 50 MHz and 60 MHz.

and audio algorithm can be calculated based on the number of required processing cycles, the audio sampling frequency and the frame/block size as follows:

$$F_{minimum,Processor} = \frac{\#Cycles_{Frame}}{\#Samples_{Frame}} \cdot F_{Sampling} \tag{3.17}$$

The minimum clock frequencies for the *KAVUAKA* processor are depicted in Figure 3.66. The number of required processing cycles is determined by the computing complexity of hearing aid algorithm. The range of the clock frequencies is from 0.05 MHz to 75.125 MHz. The maximal value of 75.125 MHz is required spectral-subtractive noise reduction algorithm, which is computed using the SW CORDIC approximation for hyperbolic and trigonometric functions. When the HW CORDIC is used, a lower clock frequency is sufficient. Based on these minimum required clock frequencies the target maximum clock frequency for the *KAVUAKA* SoC, described in Section 3.5, is set to 50 MHz, which is enough for most of chained algorithm combinations and represents an acceptable trade-off between performance and power consumption.

Due to the comparatively low maximum clock frequency of 50 MHz, the number of pipeline stages of the processor can be reduced to a minimum. Another advantage of reducing the number of pipeline stages, besides the reduction of silicon area and power consumption, is an decrease of the minimum required clock frequency. A low number of pipeline stages decreases the number of required branch delay slots. In case of the two pipeline stage processor configuration, only one branch delay slot is required. The minimum required clock frequencies for the two and five pipeline stage processor is shown in Figure 3.66. The algorithms evaluated are: mel-frequency cepstrum [24], gammatone filter [221], beamforming algorithms [207], localization algorithm [6], DNN [24], spectral-subtractive noise reduction [60], binaural coherence filter [222], overlapadd [222], floating-point FFT and filters [191]. The minimum required clock frequencies of the two pipeline stage configuration are always lower. The difference between the required frequencies is based on the number of branch instructions of the algorithms and whether the branch delay slots can be filled with other instructions than NOPs.

Hardware Bit Reversed Addressing Mode

In order to speed up the computation of the FFT algorithm, a hardware bit reserved addressing mode is added to the file indirect register (FIREG). Once a buffer size is configured during runtime, one sample per cycle can be loaded or stored to the data memory in the bit reserved mode. The speedup of the FFT computation using this hardware over the software bit reserved addressing mode is around 1.48.

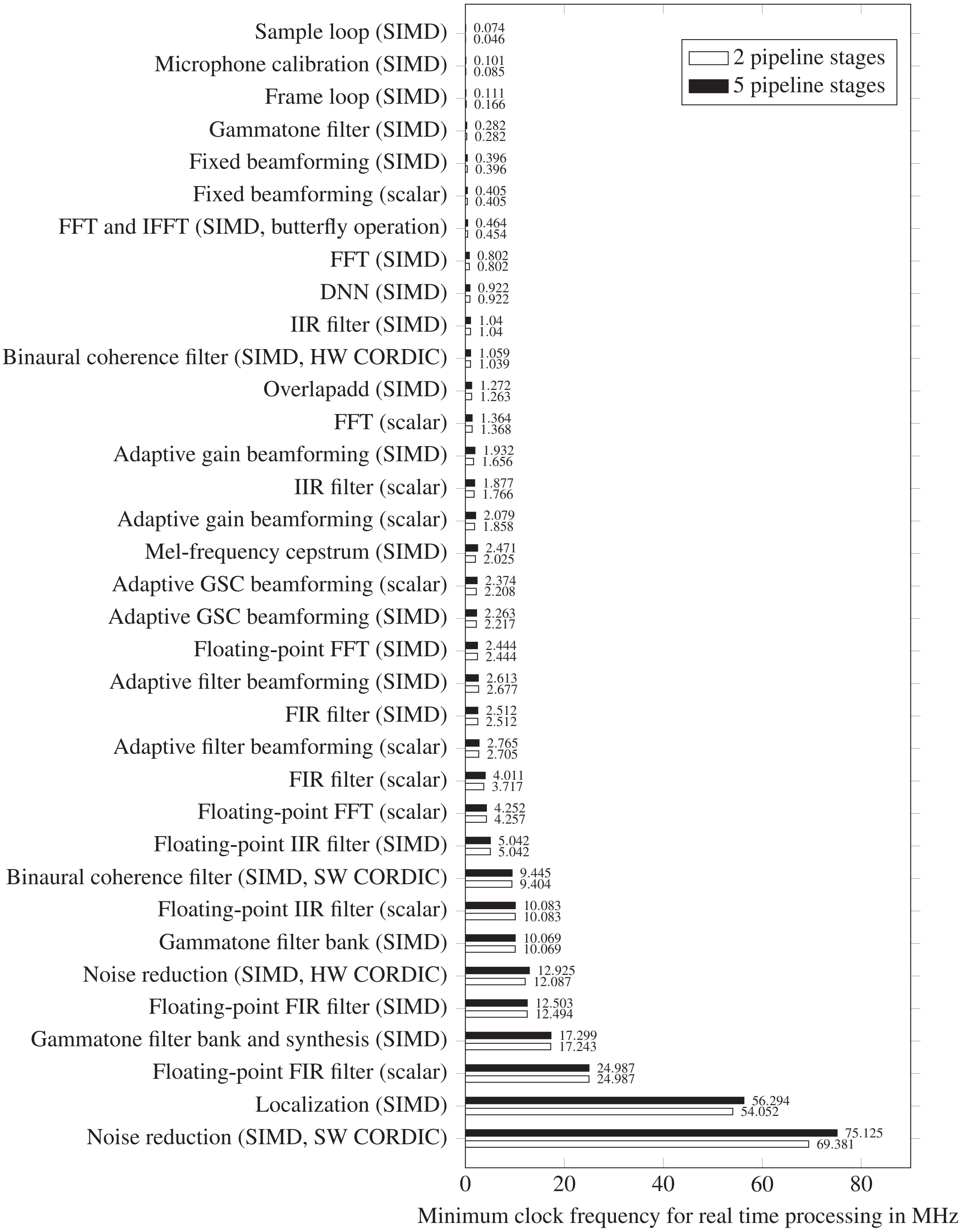


Figure 3.66: Minimum clock frequencies based on real time constrained processing (3.17) for the *KAVUAKA* processor with two and five pipeline stages.

3.4 Low-Power Interfaces and Connectivity

Hearing aid processors require external interfaces in order to communicate with other components and integrated circuits, which are part of the enclosing hearing aid system, the second hearing aid or other external electronic devices. Furthermore, communication interfaces are required for the interaction with the hearing aid user [5]. The range of functions and the related technical requirements for these interfaces are manifold and depend on the application and use cases.

The hearing aid user may want to control the volume or select the hearing aid program using hardware switches [4, 223]. A digital interface is required for the hearing aid fitting process [224]. The fitting rules for the hearing aid user are updated using and an external programming device by the audiologist. Firmware updates can be transferred via the same interface. An audio communication interface allows audio streaming from television, telephone or external microphones to the hearing aid or even between two hearing aids [225]. With the help of wireless interfaces the hearing aids exchange data with each other or other wireless communication devices like mobile phones or tablets [224, 226].

The latency of the communication interfaces is critical for the hearing aid application. Additionally, the limited power budget and the input/output (I/O) pad limitation are a major concern for the design and use of the interfaces.

This section introduces new audio and communication interfaces for digital hearing aid processors.

3.4.1 A Low Latency Multichannel Audio Interface

Hearing aid processors require a hardware audio interface for transferring audio data between the processing cores or other audio components or devices. Among these devices are off- or on-chip analog-to-digital converters (ADCs) and digital-to-analog converters (DACs), other processing devices of the embedded system [227, 228] or wireless communication interfaces like Bluetooth [20] or NFMI (near field magnetic induction). In cases where digital and analog circuits can not be implemented on the same die due to noise sensitivity of the analog front end circuity or due to limitations caused by the manufacturing process technology [4, 229], the hearing aid is composed of different chips. The audio stream is transferred between those chips using audio interfaces. The communication basis for these interfaces are common standardized serialized audio streams protocols like I^{2}S [226,230–232]. In the following important aspects for such an audio interface are listed:

Interrupt rate

One important aspect for the audio interface architecture is the coupling between the processing device and audio interface. The processing device, e.g., digital signal processors (DSPs) or application-specific instruction-set processors (ASIPs) for hearing aid devices, is highly optimized for the actual processing task, e.g., the hearing aid algorithms. If this processing is interrupted by the audio streaming task, the processing efficiency is decreased due the interrupt handling including the required context switch. However, the audio data is required for processing and is streamed with a high sampling frequency of at least 16 kHz in case of modern hearing aid devices [63]. This circumstance causes high interrupt rates. Even if the actual transfer is completely covered by a DMA, the processor has to execute the interrupt service routine (ISR) on every new DMA interrupt, which represents a performance degrading effect [233]. Therefore, for efficiency reasons, the aim is to minimize the interrupt frequency for the processor with the help of new audio interface architecture.

DMA utilization

If DMA are used for audio applications, a high DMA activity serving different DMA channels might be an issue. DMA channels are typically serviced in a sequential priority-based order. High activity on other high priority channels may therefore delay the transferring of audio samples and may therefore also delay the processing of these. The predictability of these events is difficult, which leads to larger buffer sizes or more DMAs.

Audio data format

Another issue for audio processing systems is the amount of data and and how the data formatted and aligned. Since audio data is continuously streamed, the processors need to perform a not inconsiderable amount of data movement and data permutation operations during processing. Although the related audio processors [95, 104, 113, 123, 227, 234] support hardware mechanisms like SIMD or VLIW to move and process multiple data words within one cycle, none of the coupled audio interfaces support suited SIMD or VLIW vector data formats. Therefore, permutation or alignment steps are required before and and after the efficient parallel processing [75, 83, 100, 101, 104] to convert between the different formats. As a result, the performance is decreased. This performance decrease is not negligible, since this task has to be performed repeatedly for every single sample with typical sampling frequencies of at least 16 kHz.

Audio latency

The architecture of the audio interface and its coupling to the processing device is also crucial for latency. Latency is introduced by additional audio buffers, which are required by the hardware architecture. The hardware related latency adds up to the software related latency. The hardware buffers are used, among other purposes, to implement double buffering [227, 232, 235] also called ping-pong buffering. Ping-pong buffering doubles the latency proportional to the size of the buffer. Additional latency occurs where minimum size of transfers is limited or FIFO buffers are used additionally to DMAs to increase the tolerance for DMA latencies. These effects have been studied in [236]. If the total audio delay of the hearing aid delay exceeds a value of 10 ms the perception of the hearing aid user's own voice gets affected and audio and visual information inputs get out of synchronization [220]. Therefore, the latency of the audio interface needs to be minimized.

Switching noise

Noise caused by periodic peak to peak variations of a dynamic current in mixed-signal hearing aid devices is an issue [237]. In order to reduce the periodic variations of the current, the activity of all components of the system should be kept at a constant level. The typical block based processing in hearing aids, where the processor load rises rapidly with the processing of a new block, may cause a tonal noise at the analog output. An audio interface, which provides the first samples of not yet complete block, enables the processing core to start the processing earlier. This concept spreads the load and reduces the generated noise.

Low-power mechanisms

Audio interfaces are suitable for the implementation of low-power mechanisms. The audio interface may not only trigger transfers or processing of audio samples but also trigger a low-power states in the case when not enough samples are present to be transferred or processed.

Based on the before mentioned considerations, a new architecture for a multi channel audio interface is presented in this section. Compared to related audio interfaces, the presented audio interface does support the SIMD suited vector data format for audio data. No permutation or alignment operations are required by the coupled SIMD processor. The FIFO-based mechanism of this design does not need additional DMAs, buffers or interrupt controllers. The audio latency and the load for the processor can therefore be decreased, because less audio buffers are used and no interrupt routines have to be processed. The presented FIFO-based mechanism is also used to decrease the noise and the power consumption of the processor by activating a sleep mode, if no samples are ready to be processed. Simulations and measurement case studies validate, evaluate and compare the proposed architecture to existing hardware interfaces.

This chapter is structured as follows. The related audio processor interfaces for DSPs are presented in Section 3.4.1.1. The proposed low latency multi channel audio interface for low-power SIMD digital signal processors is described in Section 3.4.1.2. The audio streaming latency is evaluated in a case study in Section 3.4.1.3.

3.4.1.1 Related Audio Processor Interfaces for DSPs

This section discusses the existing architectures of related audio interfaces and their integration into the system. The system consists of one or more processing core and an audio interface, which is coupled to the processing core.

A multi channel audio serial port (McASP) is presented in [232, 238]. This interface is used for the TMS320C6000 processor family, including the TMS320C6747 [95] and the TMS320C6748 [123], which are fixed and floating-point digital signal processors optimized for low-power applications. The interface consists of shift registers for serialization (XRSR) of the multi-channel audio streams. When a sample is completely transferred and stored in a XRSR register, it is copied to an additional buffer register (XRBUF). This register can than either be accessed directly by the processor or by the DMA unit. Additional FIFO buffers can be activated when using the DMA to increase the tolerance of DMA latencies [95]. The synchronization takes place via interrupts or polling.

An Enhanced Serial Audio Interface (ESAI) [226] is part of the Symphony DSP56300 family [112], which contains high performance multi-core digital signal processors. For serialization of the audio data, this interface uses a transmit and a receive ESAI shift registers. If audio sample is completely serialized, the data of these shift registers is transferred to one of the ESAI Data Registers (RX0, RX1, RX2, RX3). The audio data, stored in the data registers, is written and read by the processor or DMA using different interrupts.

The audio interface of the ultra-low-power CoolFlux DSP [104] provides shared FIFO memories, which have a reserved address range in the main memory. The access for all components of the system to these FIFOs is managed by the FIFO controller. The synchronization takes place via interrupts.

The Blackfin embedded processor family [239] is equipped with buffered serial ports (SPORT) and DMAs for the transmission of audio data. Two DMA channels are used to send and receive data and interrupts are generated each time a transmission is completed [240].

The Kinetis processor family [231, 234, 241] and the ADSP-21161 SIMD Sharc DSP [227] use DMAs and interrupts for transferring audio data to and from the audio interface and the local memory.

In [233] an environment for low latency audio processing with a *BeagleBone Black* board is presented. On this *BeagleBone Black* board is a *ARM Cortex-A8* processor with programmable *Realtime Units (PRUs)* for the timing-sensitive audio tasks. The authors of [233] developed a custom driver for the audio codecs and used the *RFUs* as DMA controllers for the audio data transfer.

3.4.1.2 A Low Latency multi channel Audio Interface for Low Power SIMD Digital Signal Processors

In this section, a low latency multi channel audio interface for low-power SIMD digital signal processors is presented. This interface transfers audio data between the serial inter-IC audio bus and the intra-IC parallel bus interface, which is connected to the processing device.

On the inter-IC audio bus side, audio data is transferred serially. The data formats for external inter-IC communication may be I^2S, TDM or PDM [229]. Analog-to-digital converters (ADCs) convert the analog audio signal and digital-to-analog converter (DAC) convert the digital audio signal sample by sample. These samples have to be buffered for a frame-based signal processing. This is the task for the audio interface, which transfers and buffers the audio data for the audio between the processing device and the converters. Two possible hearing aid configurations with audio interfaces are shown in Figure 3.67. Hearing aid configuration (A) comprises a chip containing both the processor and the DACs and ADCs. In configuration (B) the DACs and ADCs are integrated in an external chip, like proposed in [7].

Audio Data Format

The processor bus interface width varies between 24-bit, 32-bit, 48-bit and 64-bit. The underlying data is formatted for SIMD vector data as shown in Figure 3.68.

In case of this 64-bit bus width, each word (64-bit) is either composed of two 32-bit or four 16-bit subwords. Every subword represents one audio sample for one time slot in the time-division multiplexing format [229]. Therefore, multiple samples can be read and written simultaneously by the processor within one clock cycle. This architecture reduces the number of bus accesses needed to transfer the audio data, compared to single sample transfers. Since the audio data is already processed in a SIMD suited format by the processor, no additional permutations have to be performed. Samples stored within the subword of one word are of the same channel and sequentially ordered. No additional DMA transfers are needed since the address range of the audio interface is mapped into the address range of the processor. The processor loads and stores audio samples from the audio interface instead from the main memory. Different audio channels can be read and written using different addresses. Compared to other existing architectures [95,104,113,123,227,234], the presented audio interface supports the SIMD vector data formats and interface data width (24-bit, 32-bit, 48-bit and 64-bit).

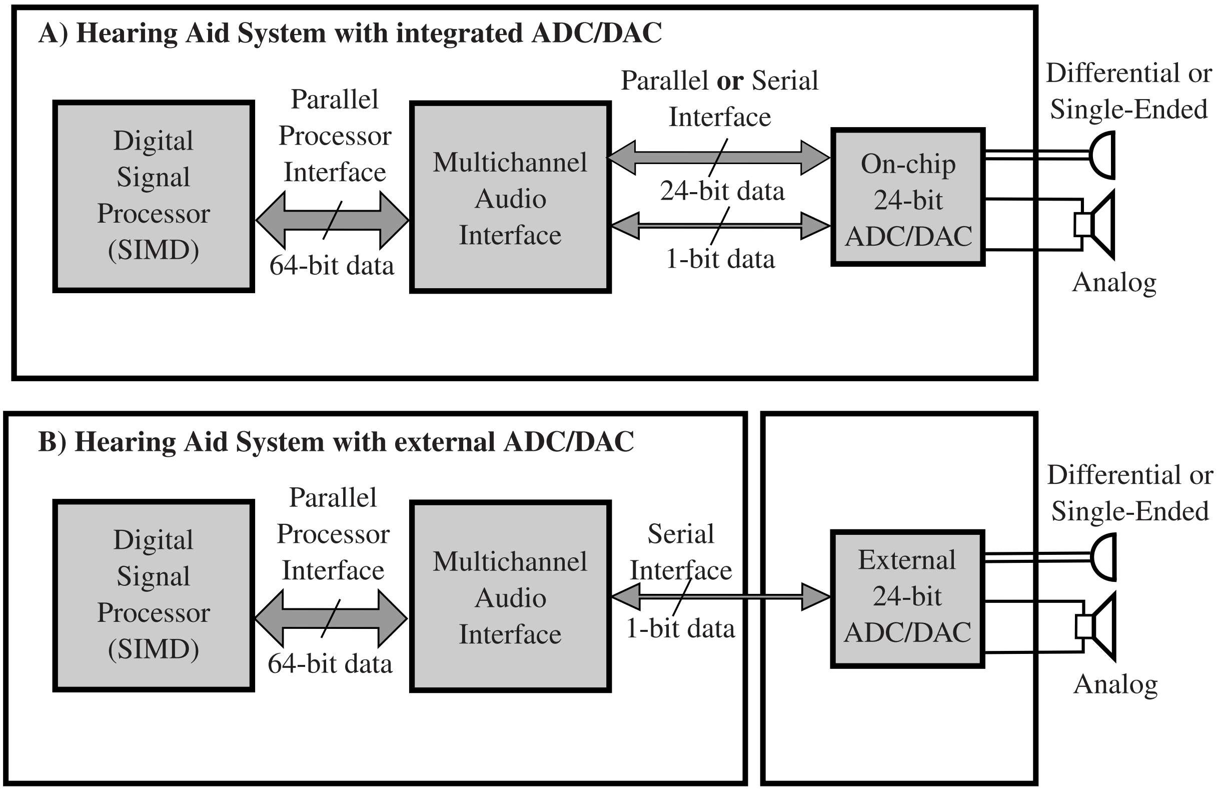

A) Hearing Aid System with integrated ADC/DAC
Digital Signal Processor (SIMD)
Parallel Processor Interface
64-bit data
Multichannel Audio Interface
Parallel **or** Serial Interface
24-bit data
1-bit data
On-chip 24-bit ADC/DAC
Differential or Single-Ended
Analog
B) Hearing Aid System with external ADC/DAC
Digital Signal Processor (SIMD)
Parallel Processor Interface
64-bit data
Multichannel Audio Interface
Serial Interface
1-bit data
External 24-bit ADC/DAC
Differential or Single-Ended
Analog

Figure 3.67: Block diagram of hearing aid systems with a multi channel audio interface. (A) The audio interface transfers audio data between the processor and on-chip DACs and ADCs. (B) The DACs and ADCs are integrated on an external chip.

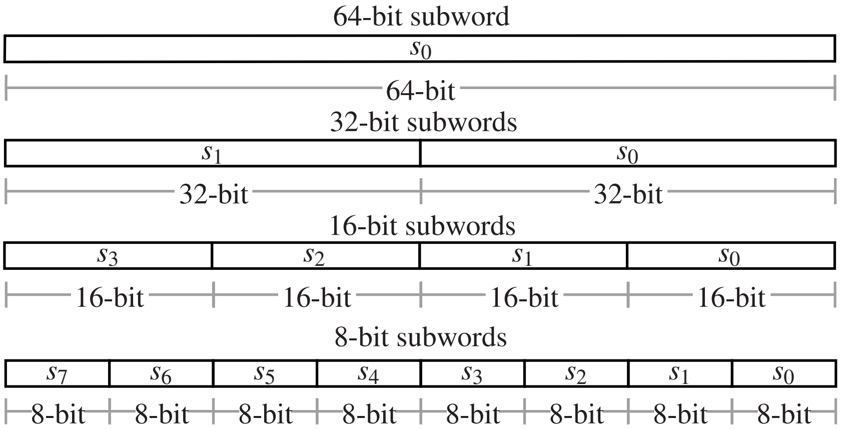


Figure 3.68: SIMD vector data format. Each word of 64-bit is either composed of 32-bit or 16-bit subwords. Every subword s represents one audio sample or time slot.

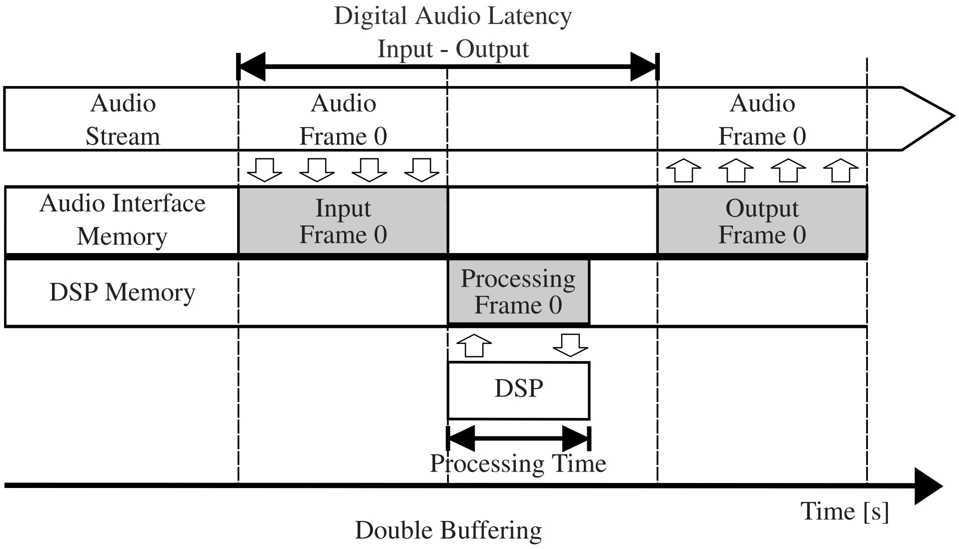
FIFO-based Architecture

FIFOs are used in this work to buffer the incoming and outgoing audio samples, since the samples must be transferred sequentially. The FIFO based memory access is preferred to a random memory access, as shorter access time latencies and simpler interfaces can be realized [104, 123]. The comparison of the audio latency, induced by the hardware architecture of the audio interface, is shown in Figure 3.69. In case of the double buffering technique, which is commonly used [112, 226, 227, 231, 232, 235], two buffers are used. The transfer of the processed sample buffer starts, when the input buffer is fully filled. Therefore, the latency depends on the frame size. In case of a FIFO buffering technique, the processed output samples are transferred as they are written to the output FIFO. The audio latency depends mainly on the processing time.

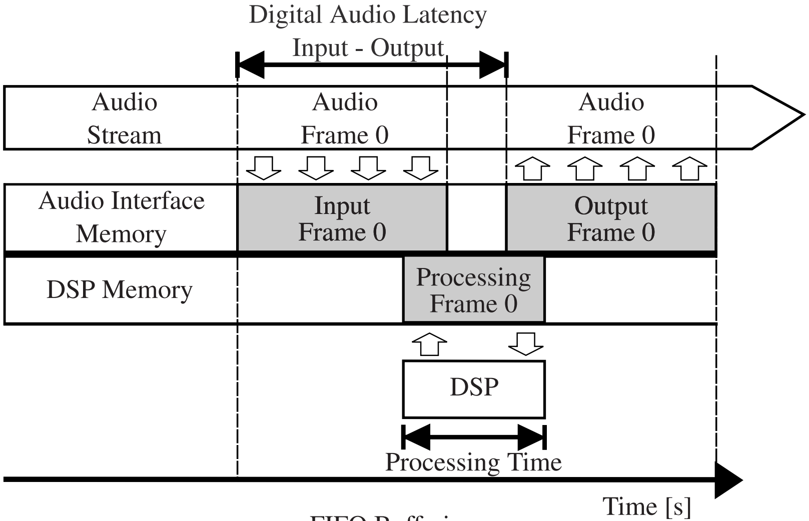
FIFOs, which are arranged and organized in an array as shown in Figure 3.70, can be used to support 64-bit read and writes as well as sequential accesses to single audio samples.

In the left part of Figure 3.70, a SIMD processor is shown, which accesses the multi channel audio interface. 64-bit SIMD vector audio data can be written and read every cycle, whereas each subword represents an audio sample. Multiple audio channels are supported. The channel is determined by the bus address. One row of the input or output FIFO array represents one audio channel. Depending on the bus address, all FIFOs of one of these rows are addressed. In case of a write access by the processor, 64-bit are written to four FIFOs of 16-bit data width each. The data width of the FIFO is 16-bit, because this is the minimum supported width for an audio sample.

The serialized data output and input of the audio interface are 1-bit signals. These are connected to external devices and data is transmitted according to the I^2S or TDM standard. Synchronized with the bit clock (BCLK) and the left-right clock (LRCLK) of the I^2S standard, these 1-bit data line signals are driven by the serialized audio data stored in the FIFOs. A output control logic generates the FIFO address pointers and selects a single bit of the correct channel and sample within the FIFO data. Therefore, three multiplexer layers exist. The first multiplexer selects the channel in a time-division manner. For each channel the samples or time slots are sequentially select by the slot multiplexer from each FIFO in one row. The multiplexer serializes the sample and drives directly the output data line. The same control logic is used for the serial audio input coming from external devices. A shift register is used to parallelize each sample or time slot. Depending on the channel and the time slot the audio data is then stored into one of the input FIFOs. The input FIFOs can then be read by the processor in the SIMD suited vector data format. This architecture can also be used as an interface for on-chip DAC/ADC. In this case either the serial interface or parallel sample-based interface can be established.



Double Buffering



FIFO Buffering

Figure 3.69: Hardware induced audio latency comparison between double buffering and FIFO buffering techniques.

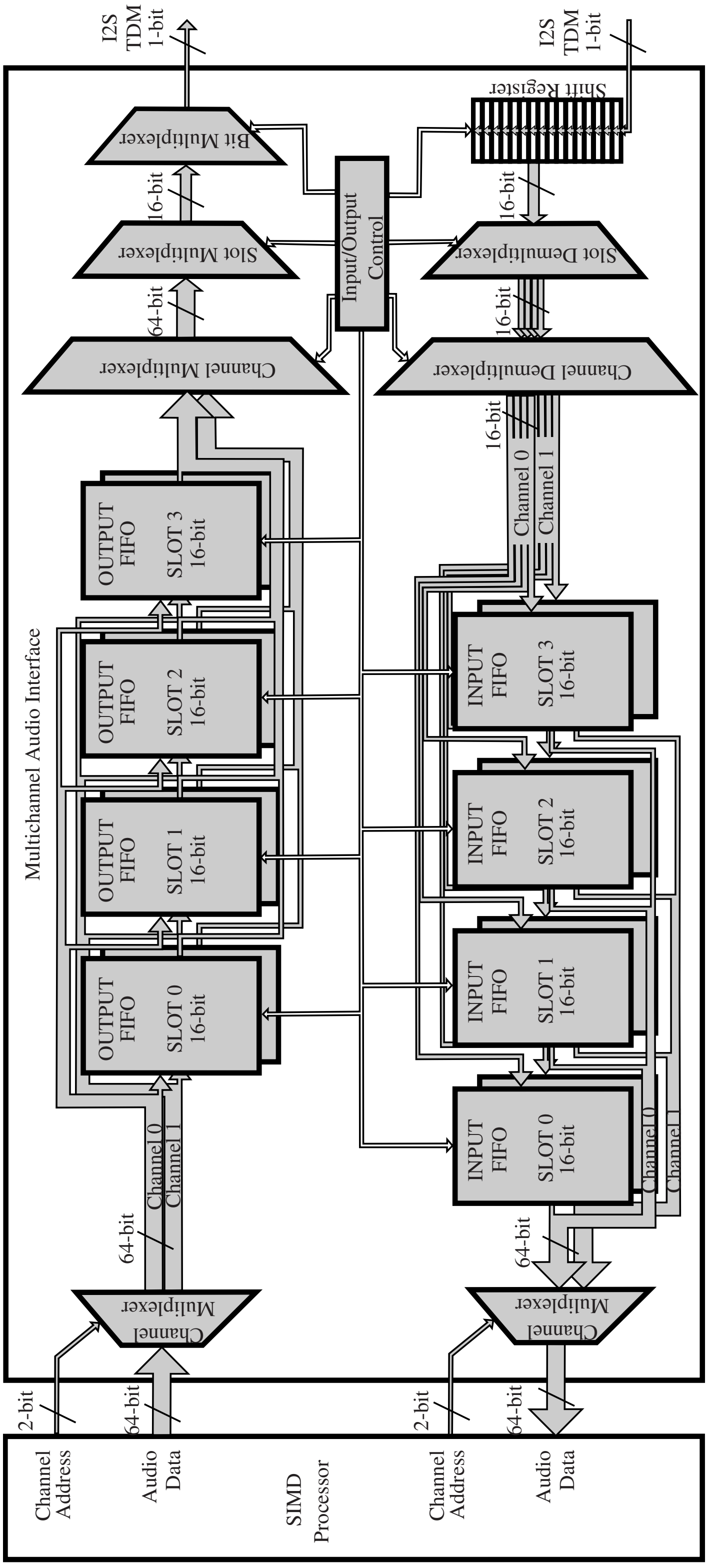


Figure 3.70: Architecture of the FIFO based multi channel audio interface. Parallel to serial conversion is realized with an array of FIFOs. On the left side the SIMD processor interface for reading and writing parallel audio data is shown. The audio interface with the serial audio bus interface is shown on the right side. Two audio channels are implemented in this case, resulting in two rows of the FIFO array. The minimum supported audio sample or time slot length is 16-bit. Therefore, the FIFO array consists of four columns of 16-bit FIFOs, which together form the 64-bit SIMD suited data word for the SIMD processor.

Low-Power Mechanism

In order to implement a low-power mechanism, the available FIFO address pointers of the FIFO-based audio interface architecture are used to check how many samples are available in the input FIFOs to be processed. If there are none and the processor tries to read audio samples from the audio interface, the processor is set to a low-power mode to decrease power consumption. This mechanism is used for synchronization at the same time. This approach was chosen in place of the interrupt approach of the related architectures [112, 226, 227, 231, 232]. The goal is to avoid high interrupt rates, which are caused by a continuous audio stream transfer. Interrupts caused by other events are still possible depending on the underlying processor architecture. Another goal is to reduce the time the processor spends transferring audio data. In case of the interrupt-approach, the interrupt service routine has to be processed including multiple branches.

The architecture of the low-power mechanism is shown in Figure 3.71. An idle flag signal is generated by the audio interface, which indicates that no audio samples are available in the input FIFOs. This flag is evaluated in the instruction fetch stage of the processor, if an *IDLE* instruction is used. This instruction is used before reading from the audio interface. If the idle flag is active and an *IDLE* operation is used, no operations (NOPs) are feed into the pipeline of the processor. By this approach, interrupts caused by other events are still possible and the power consumption of the processor is reduced, till the switching activity within the processor is minimized.

A clock gating mechanism is an alternative mechanism to set the processor to a low-power state using the idle flag. This option is not used, due to expected high silicon area requirement for routing the clock enable signal to every storage element, e.g., flip-flop, of the processor. However, only a small amount of additional controlling logic is required with the proposed implementation of the low-power mechanism for audio processing systems.

3.4.1.3 Case Study: Audio Streaming Latency

The main goal of this case study is the comparison of different audio interface architectures in terms of latency and the performance impact on the processor. The latencies are determined by simulations and measurements using an audio analyzer. The measurement setup is shown in Figure 3.72.

The digital-to-digital latencies are determined by simulations and measurements. For the simulations on RTL level, a hardware simulator is used. The SIMD processor, the audio interface and a virtual audio codec are part of the simulation. For the measurements an audio analyzer is used, which is connected to the inputs and outputs of the digital serial audio interface or the analog inputs and outputs of the audio codec chip.

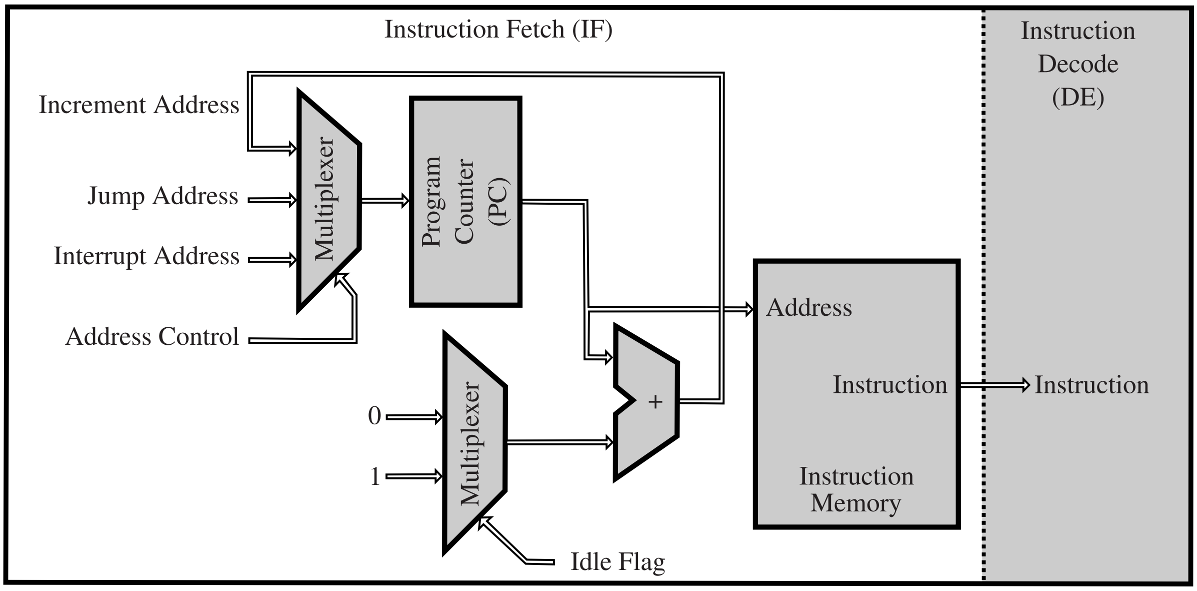


Figure 3.71: A low-power mechanism for pipelined audio processor architectures. The idle flag is set by the audio interface, if no audio samples are available in the input FIFOs. If an *IDLE* instruction is used, which can be used before read audio samples from the audio interface, and the idle flag is active, the program counter is stalled and no operations (NOPs) are feed into the pipeline of the processor.

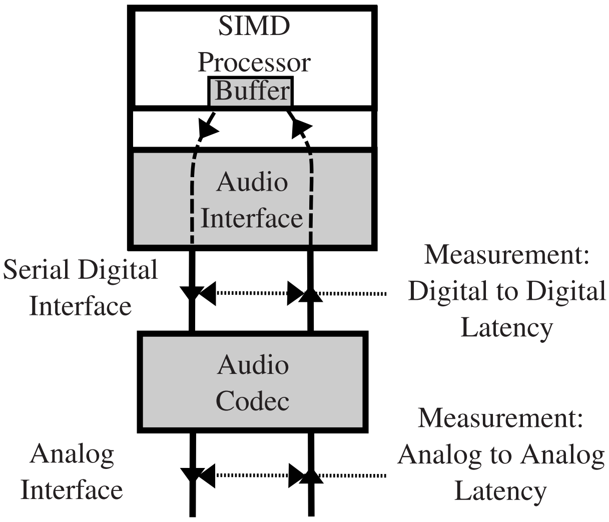


Figure 3.72: Measurement setup for digital to digital and analog to analog latencies. The SIMD processor is connected by the audio interface to an external audio codec.

To determine and compare the latencies and performance influences of different audio interface architectures, an audio loop application is used for each test setup. This audio loop application transfers audio with the processor from the audio input to the audio output of the audio interface directly without interruption. No audio processing is performed to avoid the effects of different processor performance. The same measurement setup is used in [233]. A comparison of the digital audio latencies of a DMA-based [233] and FIFO-based audio transfers at a sampling frequency of 44.1 kHz is shown in Table 3.24. For the DMA-based audio interface architecture the latency is given by twice the buffer size [233]. For the FIFO-based audio interface the latency is one time the buffer size plus the latency of one or two samples, which depends on when the sample is written to the output FIFO (Figure 3.69).

Table 3.24: Comparison of the digital audio latencies of a DMA-based and FIFO-based audio transfers at a sampling frequency of 44.1 kHz.

Buffer size	DMA-based [233] BeagleRT	FIFO-based *KAVUAKA*
64	2.90 ms	1.49 ms
32	1.45 ms	0.77 ms
16	0.73 ms	0.41 ms
8	0.36 ms	0.23 ms
4	0.18 ms	0.14 ms
2	0.09 ms	0.09 ms

The processor loads and latencies for the transmission of audio data are listed in Table 3.25 for the related audio interface architectures and the proposed FIFO-based architecture.

The related audio interface architectures are based on DMAs in combination with FIFOs [95, 123], only DMAs [104, 112, 113, 231, 234, 241] or a direct processor bus interface to the registers of the audio interface. Direct register access is supported by all related architectures when the DMAs and FIFOs are bypassed. All interface architectures are available within the *TI TMS320C6747* processor [95], which is used as reference architecture in this case study. These architectures are compared to the presented FIFO-based architecture, which is used by the *KAVUAKA* processor.

Various measurements are performed based on different audio signal processing scenarios. An important factor for a frame-based audio processing is the frame size, which determines the audio hardware buffer size. The buffer size is varied from 16 to 512 audio samples. The buffer size of 16 samples is the minimum supported size for DMA transfers [236]. The sampling frequency is also varied, since higher sampling frequencies cause smaller latencies and require a higher data throughput. The evaluated sampling frequencies range from 16 kHz to 96 kHz. This variation affects the load for the audio processor.

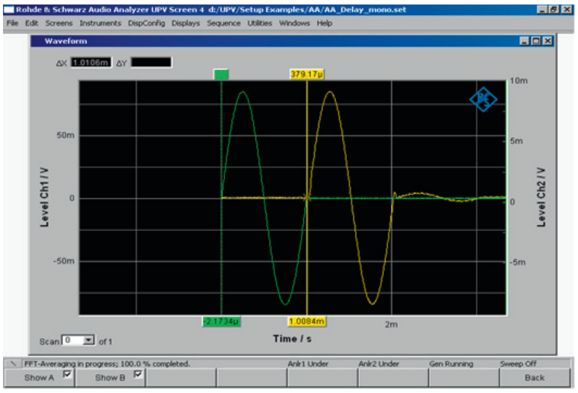
Table 3.25: Comparison of the digital audio latencies and processor loads for different audio interface architectures.

Audio Buffer Length	Processor	Audio Interface DMA	Audio Interface FIFO	Sampling Frequency 16 kHz Digital Latency ms	16 kHz Proc. Load %	32 kHz Digital Latency ms	32 kHz Proc. Load %	48 kHz Digital Latency ms	48 kHz Proc. Load %	96 kHz Digital Latency ms	96 kHz Proc. Load %
512	TI C6747	yes	yes	68.00	0.67	34.00	1.34	22.66	2.02	11.33	4.04
	TI C6747	yes	no	64.00	0.67	32.00	1.34	16.00	2.02	10.66	4.04
	TI C6747	no	no	32.00	2.12	16.00	4.24	10.66	6.36	5.33	12.07
	KAVUAKA	no	yes	**32.12**	**0.08**	**16.06**	**0.17**	**10.70**	**0.26**	**5.35**	**0.53**
256	TI C6747	yes	yes	36.00	0.89	18.00	1.78	12.00	2.67	6.00	5.35
	TI C6747	yes	no	32.00	0.89	16.00	1.78	10.66	2.67	5.33	5.35
	TI C6747	no	no	16.00	2.12	8.00	4.24	5.33	6.36	2.66	12.07
	KAVUAKA	no	yes	**16.12**	**0.08**	**8.06**	**0.17**	**5.37**	**0.26**	**2.68**	**0.53**
128	TI C6747	yes	yes	20.00	1.46	10.00	2.92	6.66	4.38	3.33	8.77
	TI C6747	yes	no	16.00	1.46	8.00	2.92	5.33	4.38	2.66	8.77
	TI C6747	no	no	8.00	2.12	4.00	4.24	2.66	6.36	1.33	12.07
	KAVUAKA	no	yes	**8.12**	**0.08**	**4.06**	**0.17**	**2.70**	**0.26**	**1.35**	**0.53**
64	TI C6747	yes	yes	12.00	3.14	6.00	6.28	4.00	9.42	2.00	18.85
	TI C6747	yes	no	8.00	3.14	4.00	6.28	2.66	9.42	1.33	18.85
	TI C6747	no	no	4.00	2.12	2.00	4.24	1.33	6.36	0.66	12.07
	KAVUAKA	no	yes	**4.12**	**0.08**	**2.06**	**0.17**	**1.37**	**0.26**	**0.68**	**0.53**
32	TI C6747	yes	yes	8.00	4.82	4.00	9.65	2.66	14.47	1.33	28.95
	TI C6747	yes	no	4.00	4.82	2.00	9.65	1.33	14.47	0.66	28.95
	TI C6747	no	no	2.00	2.12	1.00	4.24	0.66	6.36	0.33	12.07
	KAVUAKA	no	yes	**2.12**	**0.08**	**1.06**	**0.17**	**0.70**	**0.26**	**0.35**	**0.53**
16	TI C6747	yes	yes	6.00	9.17	3.00	18.35	2.00	27.53	1.00	55.07
	TI C6747	yes	no	2.00	9.17	1.00	18.35	0.66	27.53	0.33	55.07
	TI C6747	no	no	1.00	2.12	0.50	4.24	0.33	6.36	0.16	12.07
	KAVUAKA	no	yes	**1.12**	**0.08**	**0.56**	**0.17**	**0.37**	**0.26**	**0.18**	**0.53**

As shown in Table 3.25, an audio interface based on DMA and FIFO buffers leads to the highest latencies for all audio buffer sizes and sampling frequencies. Because the latencies caused by the double buffer mechanism for DMA transfers add up to the latency of the FIFO buffers. In case of the configuration in [236] each FIFO buffer stores 64 audio samples. These FIFO buffers do not affect processor load but increase tolerance to DMA latencies [95].

The processor load is defined here as the number of cycles required to transmit the audio samples at a time interval divided by the total number of cycles available at that time interval. The clock rates of the processors were normalized for comparison. There is no difference in processor utilization when using DMAs and FIFOs in combination, but the latencies increase by the latencies of the FIFO buffers. Latencies can be reduced by using only DMAs transfers or direct processor transfers instead of DMAs. The direct processor transfers cause higher processor loads for larger audio buffers, but offer lower latencies compared to DMA transfers. There are cases where the processor load is lower for direct transfers compared to using a DMA with double buffering. The higher load is due to the complexity of the DMA interrupt routine. In the case of DMA transfers, semaphores must be used to access the buffers, whereas this is not the case for direct processor access. Therefore, the generated processor load depends on DMA interrupt rate. The smallest processor load is caused by the proposed FIFO based audio interface. This is because neither interrupts nor double buffering are used.

The analog-to-analog latencies are measured with an audio analyzer connected to the analog inputs and outputs of the audio codec, as shown in Figure 3.72. A sine wave is generated at the input and the latency is determined by measuring the delay between the input and output of the audio codec. The measurement is displayed in Figure 3.73.



FFT-Averaging in progress; 100.0 % completed.

Figure 3.73: Measurement of the analog-to-analog audio latencies of audio processing systems. A audio analyzer connected to the analog in- and outputs of the external audio codec. The first sine wave is generated by the audio analyzer and is used as an input for the audio codec. The second sine wave is measured at the output of the audio codec. The latency is the time delay between these waves.

A comparison of audio latencies for three different audio processing systems is given in Table 3.26. The first system consists of the DSP *TMS320C6747* [123] and the audio codec *TLV320AIC3106* [242]. The audio samples are transmitted with the DMAs without FIFO buffering, the second system [233] is based on an *ARM Cortex-A8* processor on a *BeagleBone Black* board with a *TLV32OAIC3104* [243] audio codec and the third system consists of the *KAVUAKA* processor and the *ADAU1761* [244] audio codec. The sampling frequency for this measurement is set to the fixed value of 44.1 kHz because the latency of the audio codec varies with the change in sampling frequency. As the size of the audio buffer increases, most of the latency is caused by the digital audio interface. The system latency is comparatively lower for the FIFO-based architecture, since no double buffering is used.

Table 3.26: Comparison of system audio latencies of different processors equipped with different audio interfaces and connected to different audio codecs.

Audio System Delay in ms @44.1 kHz			
Audio Buffer Length	TI TMS320C6747 [245] + TVL320AIC3106	ARM Cortex-A8 [233] + TLV32OAIC3104	*KAVUAKA* + ADAU1761
512	24.16 ms	—	12.71 ms
256	12.51 ms	—	6.91 ms
128	6.74 ms	—	4.00 ms
64	3.83 ms	3.84 ms	2.55 ms
32	2.39 ms	2.38 ms	1.82 ms
16	1.65 ms	1.66 ms	1.46 ms
8	1.30 ms	1.30 ms	1.28 ms

3.4.2 A Serial Interface with Special DMA Capabilities

A hearing aid system, consisting of multiple components like the hearing aid processor, audio converters, power management ICs or wireless chips, requires a communication bus on the system level. Due to power consumption and silicon area constraints, the number of I/O pins of the hearing aid ASIC is limited. A parallel bus, connecting the hearing aid processor to external devices, like non-volatile memory devices, is not feasible. Commonly, a serial interface called inter-integrated circuit (I^2C) [246] is used [11, 20, 80, 112]. The I^2C-communication is based on a synchronous, multi-master, multi-slave and single-ended two wire serial bus. In the *fast-mode*, specified in [246], up to 400 kbit/s can be transferred between the master, which is in this case the hearing aid processor, and the slave, which is any external chip of the hearing aid system. At this interface speed, the hearing aid processor, which acts as the I^2C master and operates at a clock frequency of 50 MHz, needs to be stalled for 8000 cycles for transferring one word (64-bit) of data. Stalling the hearing aid processor for such an amount of cycles is not desirable during normal signal processing operations. Therefore, the I^2C interface commonly is controlled by a second processor and only used for debugging purposes [20, 80]. In [112] a serial host interface is introduced, which supports single-, double- or triple-word data transfers to reduce the DSP overhead. In case of receiving data, a ten-word FIFO buffers the data before generating an interrupt for the processor. Up to 18 interrupts are available for handling the DMA transfers. In [11, 20] the interrupt scheme is also used.

In this section, a I^2C DMA unit is introduced. This unit takes care of all data transfers in a typical hearing aid system. This unit is controlled by the hearing aid processor, but functions independently and handles the long-lasting serial transfers. The hearing aid processor starts the transfer using a register configuration interface. While the I^2C is on-going, controlled by the DMA unit, the hearing aid processor goes on with other tasks. The processor only has to wait when the data being transferred by the DMA is required for the next processing task. With the DMA functionality of the I^2C unit, the processing time of the hearing aid processor is increased. Another advantage of the proposed architecture is that it does not require any interrupts. A single configurable *IDLE* flag signal is used instead, which set the processor in a low-power state until the requested data by the processor becomes available after the transfer. Interrupts are not used in order to prevent the overhead of the interrupt service routines (ISRs).

An overview of the hearing aid system with all components, either equipped with a register or a memory interface, is shown in Figure 3.74. The DMA of the I^2C unit can access the whole memory address space of all components. The I^2C unit uses the parallel on-chip bus on the one hand and the I^2C bus on the other hand. The integrated DMA controller manages the actual data transfer between the components. The transfer is initiated by the hearing aid processor, which is not depicted in the figure. The hearing aid processor sets the source and target addresses, which include the internal on-chip bus addresses and the device and slave address in case of a I^2C transfer, using a register interface. Instead of interrupts, a simple *IDLE* flag approach is used. The processor receives the *IDLE* from the I^2C unit. The *IDLE* flag is set active as long the transfer is in progress. The processor is halted by the flag if an *IDLE* instruction is scheduled, which is configured to wait for the I^2C flag beforehand using a configurable register. The *IDLE* flag is evaluated before accessing the transferred data in order to maximize the processing time and keep the idle time short.

Figure 3.74 shows the different data formats and types, which need to be transferred within the hearing aid system. The I^2C DMA unit supports different data words, ranging form a single Byte (I^2C devices) up to complete memory blocks of different word lengths, and converts data between the different formats and types. The operating and I^2C speed is controlled by a configurable clock divider.

The proposed DMA functionality is used for the boot process of the hearing aid. During the power-on sequence, after the global reset is released, the DMA unit starts to transfer the binary program form the external non-volatile memory to the internal on-chip instruction SRAMs (IMEM). In addition, during normal hearing aid operation, while the hearing aid algorithms

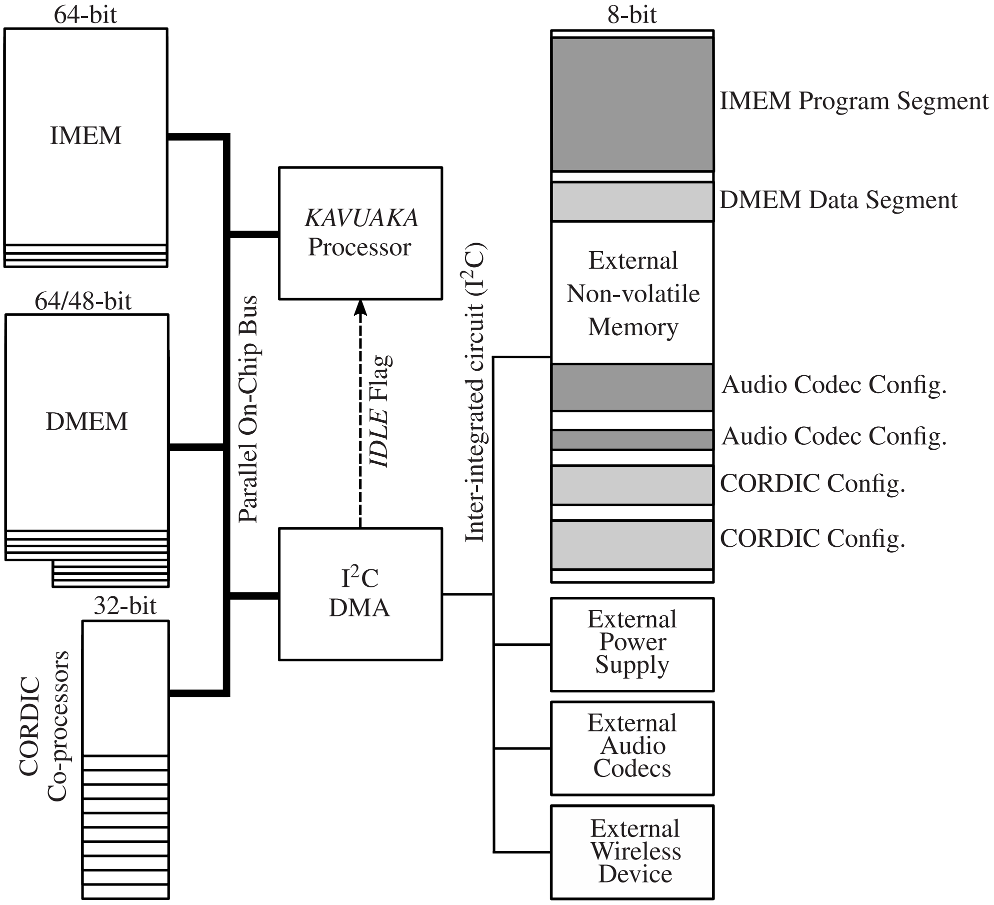


Figure 3.74: The I^{2}C DMA unit transfers data between the hearing aid components, which are the instruction memory (IMEM), data memory (DMEM), coordinate rotation digital computer (CORDIC), external audio codecs, power supplies and wireless devices as well as a non-volatile external memory.

are running, program code or complete algorithms can be replaced in the IMEM by the DMA. This requires that the processor is in idle state and does not use the global bus to access the IMEM. An alternative is the use of a dual port SRAMs for the IMEM in order to allow two accesses simultaneously, by the processor and the DMA. The purpose of replacing program code instructions during normal operation is to reduce the size requirements for the on-chip instruction memory. The whole program does not have to fit in the memory at once, when parts are being replaced. Applications are for example the replacement of user or algorithm parameters received by the wireless interface or the replacement of adaptive noise reduction algorithms, which are selected based on the detected noise sources [247].

In addition, the DMA unit can access the CORDIC co-processors (Section 3.2.3) in order to initialize their lookup tables with data stored in separate sections in the external non-volatile memory. The same applies for the external audio codecs, however the configuration is stored to byte-addressed registers in this case. The LUT size of the CORDIC co-processors is 32 words of 32-bit each. When using the processor for initializing one of these LUTs, 32 move instructions with 32-bit (*MVIL_32*) and 32 *STORE* instructions are required. These instructions are not required using the I^2C DMA. The required instruction memory size decreases from 1792 Byte to 1688 Byte, a reduction of 5.8 %, when computing a 32-bit cosine with one hardware CORDIC accelerator.

The complexity of the DMA architecture is low. Only 344 flip-flops are required for the configuration registers and the finite state machine. The complete I^2C unit is synthesized with the TSMC 40 nm HVT low-power ASIC technology at 50 MHz. The placed and routed architecture consists of 1882 standard cells and requires an area of 3109.9 μm^2, which is around 2.31 % of the area of the *KAVUAKA* 64-bit processor.

3.5 Hearing Aid System-on-Chip ASIC

In order to evaluate and compare multiple hearing aid processor and co-processor configurations on one chip, a research system-on-chip (SoC) is proposed. The system consists of four different configurations of the *KAVUAKA* processor (Section 3.1), ten different co-processors (Section 3.2.3), a multiple channel audio interface (Section 3.4.1) and a serial data interface (Section 3.4.2). The processors differ from each other by their computing performance, power consumption, chip area and computing precision. Figure 3.75 shows one vector unit of the *KAVUAKA* processor with two pipeline stages. This is the baseline configuration of the processor.

The main differences between the processors, which are part of the SoC, are the width of the datapaths, which is varied between 64-bit down to 24-bit, the MAC architecture [128] and the presence of the operation merging feature called X2 mode [83, 84, 193]. The overview of the proposed hearing aid system-on-chip is shown in Figure 3.76. The hearing aid system is

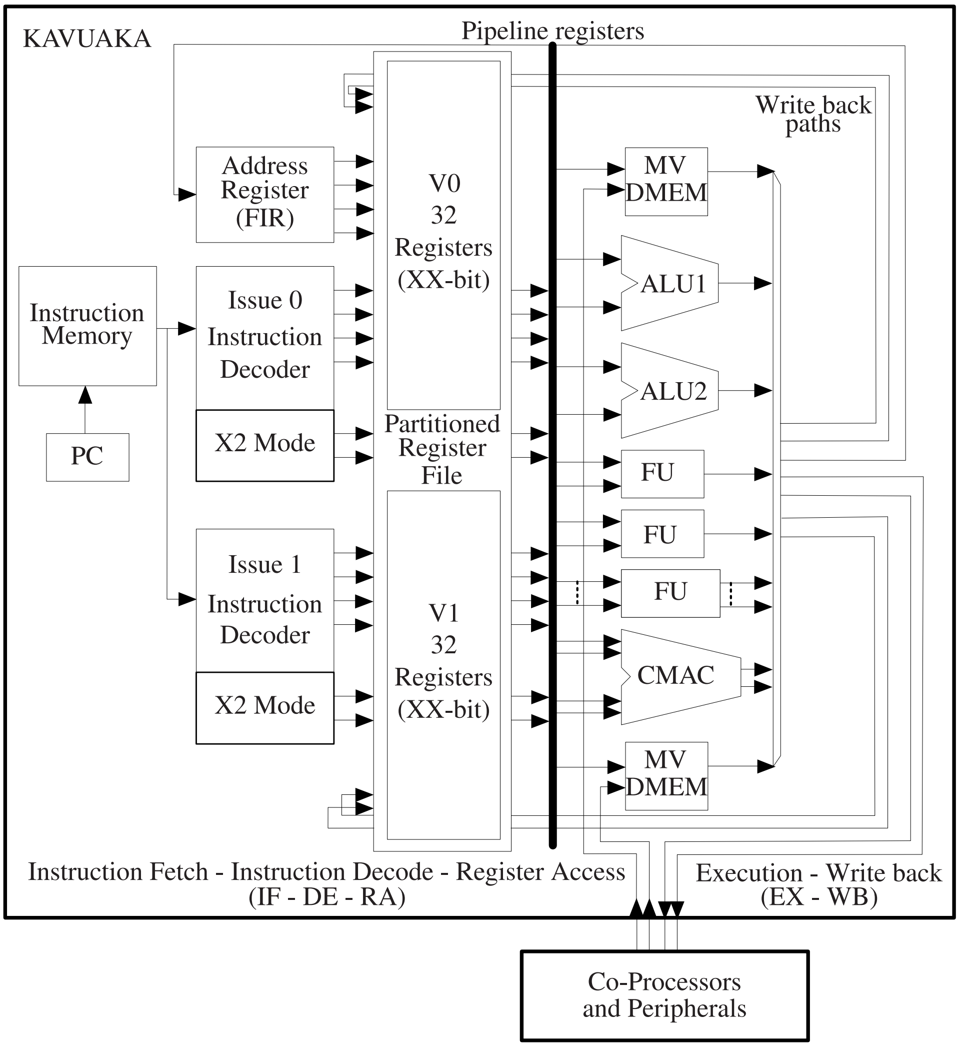


Figure 3.75: Baseline *KAVUAKA* processor architecture [56]. The processor is divided into two pipeline stages. Two instructions are decoded and executed by the specialized functional units, like a complex-valued MAC unit [128]. External CORDIC co-processors and peripherals are tightly attached.

divided into two processor clusters. Each cluster consists of one *big* processor core (64-bit or 48-bit) and one *little* processor core (32-bit or 24-bit). Furthermore, each cluster contains a dedicated data memory instance and a shared instruction memory. The sizes of the shared instruction and data memories are determined by the memory requirements of the hearing aid algorithms. A subset of these algorithms and their memory requirements are listed in Table 3.27. Based on these values, the total data memory size is fixed at 4096 words or 2048 words for each cluster. The total instruction memory size is 8192 words. With this configuration, most algorithm can be chained and the required SRAM instances fit on the die with an area of 3.6 mm^2. The Localization algorithm [6] is not considered for the ASIC this configuration due to the comparatively high data memory requirement.

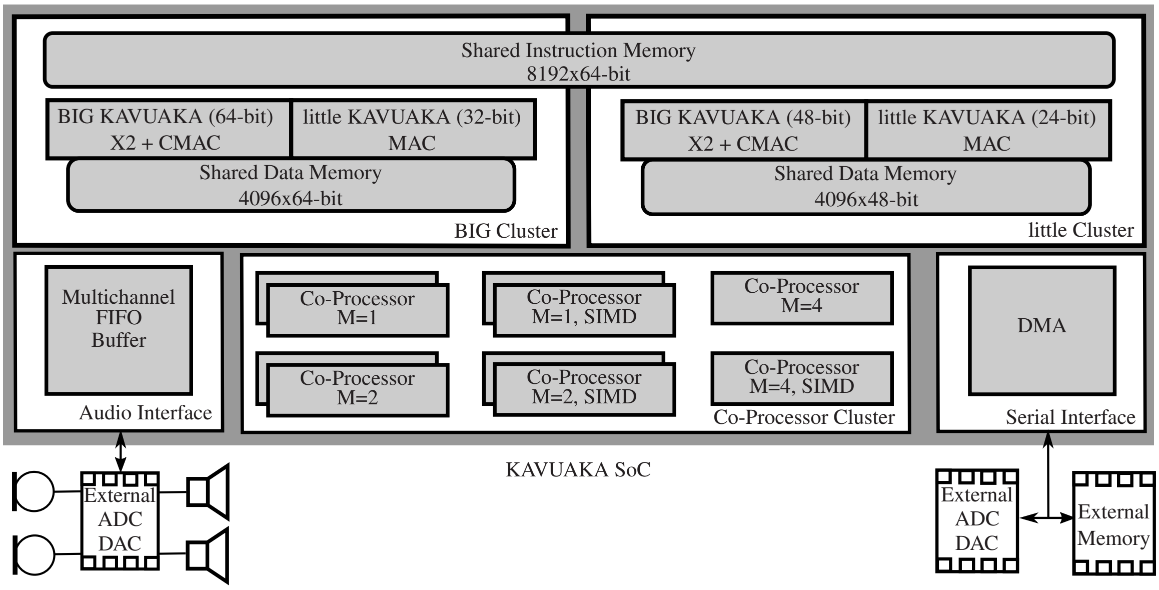


Figure 3.76: The hearing aid system-on-chip includes four processor cores, organized in two clusters, a co-processor array, one audio interface and a serial interface.

Each of the memory instances incorporates four dual port SRAM memories, in total $4 \times 1024 \times$ 64-bit and $4 \times 1024 \times 48$-bit. In case the little cores access the data memory, the memory interface switches to 32-bit or 24-bit word access, whereas the word access is switched to 64-bit or 48-bit in case of the big cores. The memory interface supports four words per access for the X2 mode (Section 3.2.4) with up to 256-bit per cycle for the 64-bit processor core configuration and 192-bit per cycle for the 48-bit processor core configuration.

Data can be copied between the data memories of each cluster using a global bus interface. Since the processor architecture does not support interrupts to guarantee an uninterrupted audio processing, a idle flag synchronization approach with low complexity is implemented. Each processor can activate a wait state for itself or other processors. The wait state is only

Table 3.27: Data and instruction memory requirements for a subset of hearing aid algorithms.

Algorithm	Data memory size (words (24-bit to 64-bit))	Instruction memory size (VLIW words (64-bit))
Localization	52592	1463
FFT and IFFT (1024-point)	3072	655
Binaural coherence filter	2184	941
Overlapadd	1250	347
Mel-frequency cepstrum	722	1593
Frame loop (512-samples)	512	106
Spectral-subtractive noise reduction	424	2622
Gammatone filter bank	334	254
Adaptive gain beamforming	113	371
IIR filter (17-tap)	103	118
FFT (scalar) (32-point)	80	581
Floating-point FFT (32-point)	80	4809
Microphone calibration	50	96
Adaptive filter beamforming	42	304
Fixed beamforming	16	140
Sample loop	0	57

triggered by the processor itself, usually when the pending processing depends on data provided by another processor or when all processing tasks are completed. During the wait state the processor's pipeline is filled with NOP instructions until the other processor completes its work and reactivates the waiting core. With this ping-pong based approach the processor are synchronized without the need for interrupt-based processing that requires more complex interrupt handlers and results in a less predictable processing time.

The instruction memory, consisting of four SRAM instances, each with 2048 × 64-bit, is shared between the processor cores. If one core is active, the whole instruction memory size is accessible by this core. If more than one core is active, the address range and thus the memory space is divided among the cores.

Different co-processor configurations (Section 3.2.3) are implemented for evaluation. In total, ten co-processors are part of the co-processor cluster (Figure 3.76). The co-processors implement the kernel equation of the coordinate rotation digital computer (CORDIC) algorithm to compute hyperbolic and trigonometric functions. Their architecture is presented in [164]. The parameter M, which indicates the number of CORDIC kernel units in the processing stage, is varied between one and four. When set to four, the co-processor computes four fractional digits per cycle. The SIMD parameter indicates whether the co-processor performs single instruction, multiple data (SIMD) operations. Each co-processor can be used by each of the processor. Each co-processor can be deactivated by clock gating.

The audio interface (Section 3.4.1), proposed in [248], enables a multi channel audio data transfer to external digital-to-analog converter (DAC) and analog-to-digital converter (ADC) converters. This inter-IC audio transfer is required, if the analog components and digital processing systems are integrated on separate chips using different ASIC process technologies, as proposed in [7]. The inter-IC sound I^2S bus interface standard is used here. Each processor accesses the first in first out (FIFO) memories ($16 \times 512 \times 16$-bit) of the audio interface based on the processor's data width and format. The interface can be configured to work with SIMD formatted data or single words.

The serial interface inter-integrated circuit (I^2C) (Section 3.4.2) is used for booting the system and, during normal operation, for accessing external devices like non-volatile memory or DACs and ADCs. The integrated DMAs manages the transfer of large amounts of data, e.g., initializes the look-up tables of the co-processors with the coefficients stored in the external memories.

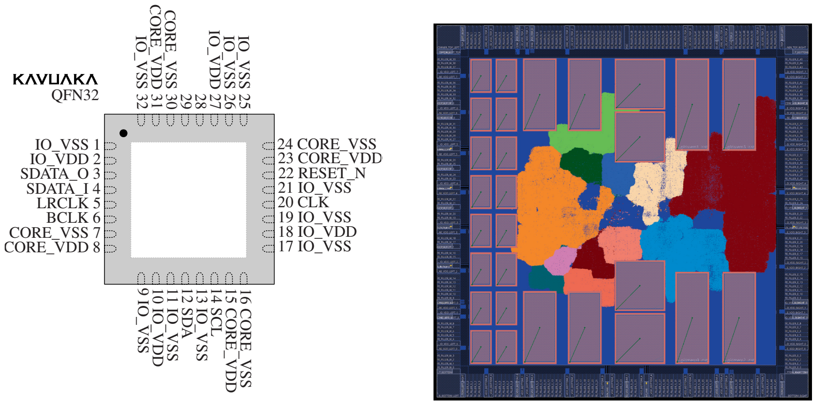
Each of the processor, co-processor, interface modules and memory instances can be activated separately or simultaneously. As presented in [19], clock gating is used for this purpose. The clock gates can be configured during runtime and the default settings may be overwritten during the boot process by the DMA unit. The clock gates reduce dynamic power consumption of the deactivated processors and co-processors for the power measurements.

The pinout diagram of the package for the *KAVUAKA* system-on-chip is shown in Figure 3.77a. Besides the power pins for the I/Os (IO_VSS and IO_VDD) and for the core (CORE_VSS and

CORE_VDD) there are pins for the audio interface (serial data (SDATA), left-right clock (LRCLK) and bit clock (BCLK)), pins for the data interface (serial data (SDA) and serial clock (SCL)), the reset pin (RESET) and clock (CLK) pin. An on scale bonding diagram of the package is depicted in Figure 3.77e. The 32 quad flat no leads package (QFN) package pins are connected to the *KAVUAKA* chip pins. The dimensions of the package are $5 \times 5\,\text{mm}$ (Figure 3.77f).

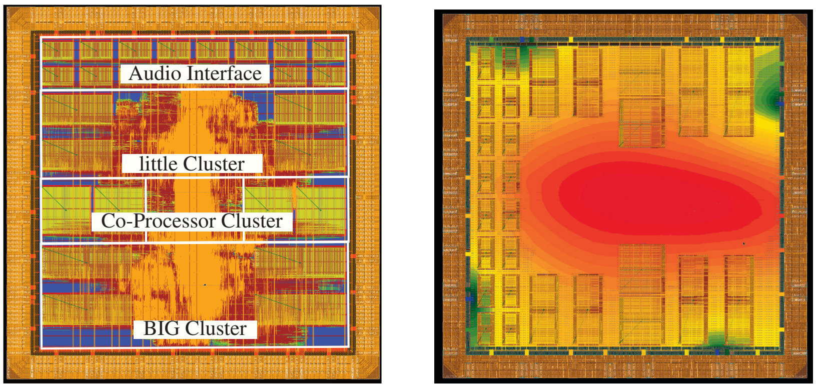
The complete hearing aid system (Figure 3.77) is synthesized using a 40 nm low-power standard cell ASIC library. The target clock frequency is set to 50 MHz. The layout view of the complete system is shown in Figure 3.77c. In total, 0.8 million standard cells, 0.4 million nets, 28 SRAM memories and 24 I/O cells are integrated. The die size is $1920\,\mu\text{m} \times 1920\,\mu\text{m}$, which is equal to $3.61\,\text{mm}^2$. The floorplan and the placement of the SoC is shown in Figure 3.77b. The biggest colored areas are the four *KAVUAKA* cores, whereas the smaller ones are the co-processors. The rectangular blocks are the SRAM instances. The core area is surrounded by the I/O ring. The total standard cell gate count of this system-on-chip is $825,284$. Eight metal layers are used for routing. The power consumption is estimated based on the static and dynamic power consumption of the gates, memories and pads. The voltage drop (IR drop analysis) across the wires is color coded. The red color indicates higher drop in voltage (Figure 3.77d). It is verified that the voltage drop is within the 10 % constraint.

The die micrograph of the fabricated *KAVUAKA* application-specific integrated circuit (ASIC) is shown in Figure 3.78. This application-specific integrated circuit (ASIC) is placed in socket on the printed circuit board (PCB). Further components of a hearing aid device, like power regulators, ADCs and DACs and a wireless communication module, are available on the PCB. Test pins and a connection to a FPGA board are available for test and verification purposes.



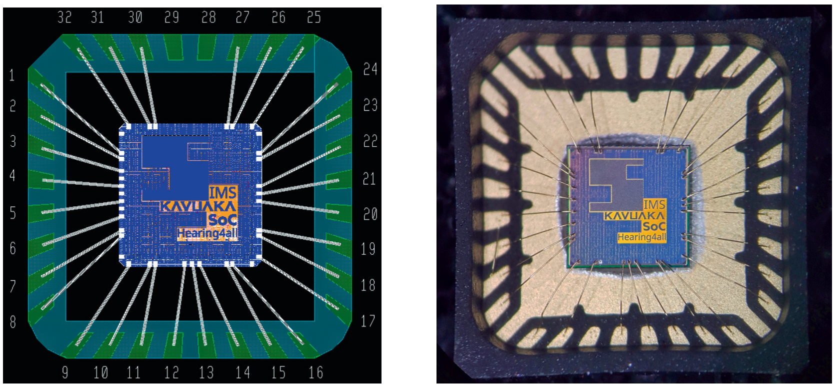
(a) Pinout diagram.

(b) Placement of *KAVUAKA* cores, co-processors and SRAM blocks and the I/O ring.



(c) Routed design.

(d) Result of the IR drop analysis.



(e) Bonding diagram.

(f) Photo of the bonded ASIC.

Figure 3.77: Pinout diagram, floorplan, routing, IR analysis, bonding diagram and photo of the fabricated *KAVUAKA* system-on-chip.

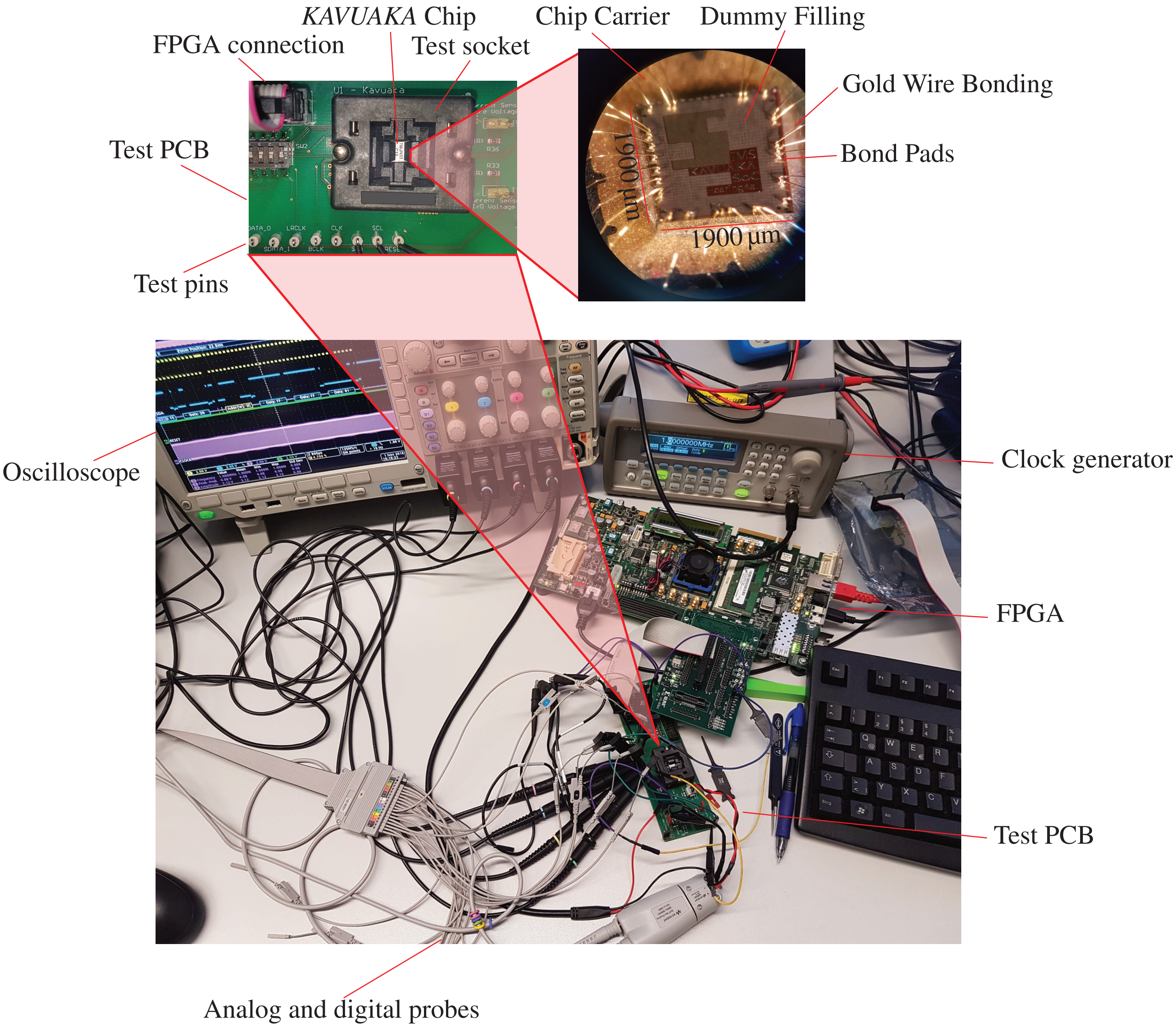


Figure 3.78: Photo of the chip socket on the printed circuit board and die micrograph of the *KAVUAKA* ASIC.

4 Operation Merging, Instruction Scheduling and Register Allocation

An efficient mapping of the algorithms to the custom and application-specific target hardware architecture is of high importance. The goal is to achieve a high utilization of the available hardware resources. The result is a high processing performance and low-power consumption [193]. This chapter describes how evolutionary algorithms are used for optimizations during operation merging, instruction scheduling, and register allocation. An overview of the code generator, which is consists of several optimization problems such as instruction merging, scheduling, and register allocation, is given in Figure 4.1.

The hearing aid algorithms, a subset is listed in Figure 3.66, for the target VLIW *KAVUAKA* processor are written in a custom assembly language. The human-readable instructions are mapped directly to micro-operations (MOs), which is defined in the instruction set architecture (ISA). The proposed code generation process [193, 249] translates these MOs into scheduled micro-instructions (MIs) taking into account the processor configuration, the associated control and data dependencies, and the optimization goals.

The presented code generator for the *KAVUAKA* processor supports different target processor architecture configurations. The processor configuration is specified in an extensible markup language (XML) file, which describes architectural parameters such as the pipeline and register file configuration, names of special registers, memory configuration, etc. It also describes the functional units, which implement the processor's instruction set architecture (ISA). For each operation, the syntax, the number of available units for parallel execution, the number and types of arguments, the available modes, the latency, and the binary encoding are specified. From the configuration, the code generator builds an internal model of the target processor, which is used for the syntactic and semantic analysis of the input programs and for binary writing.

When the code generator reads an input program for a given target processor architecture, it divides the sequence of MOs of the input code/algorithm into basic blocks (straight line microcodes (SLMs)), which are linear sequences of MOs. SLMs have only one entry point at the beginning and no branches, except possibly at the end. The code generator performs optimizations during operation merging (Section 4.1), instruction scheduling (Section 4.2), and register allocation (Section 4.3) locally for each of the SLMs, using static heuristics and evolutionary algorithms for multi-target optimizations (Section 4.2.2 and Section 4.3.1).

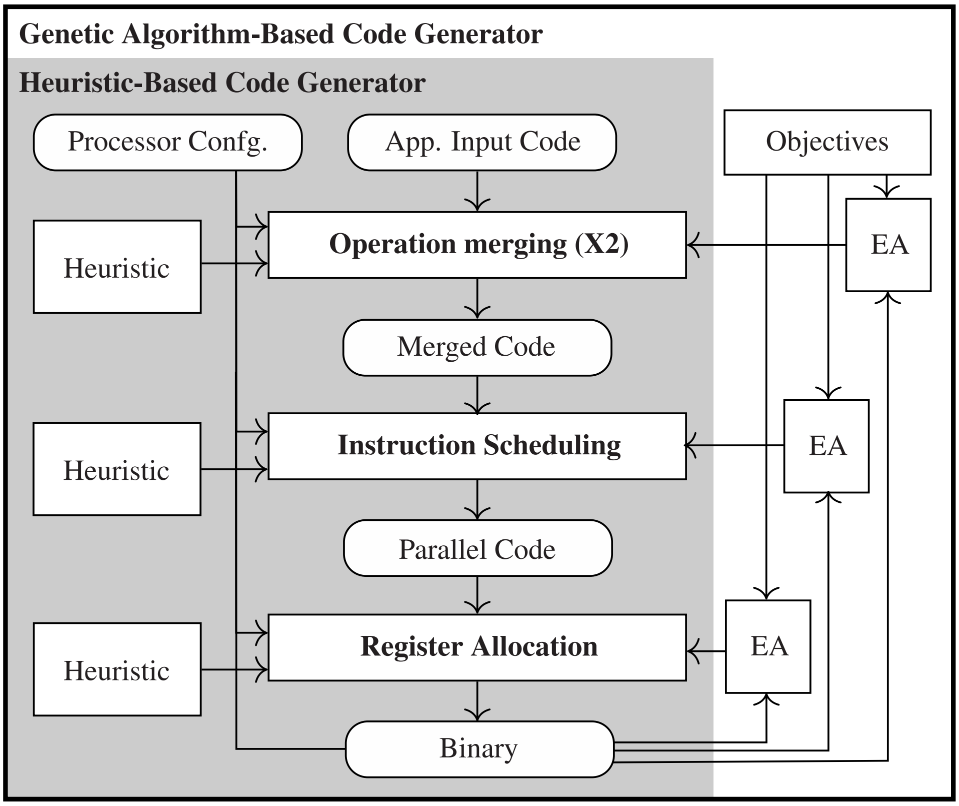


Figure 4.1: Structure of the code generator presented in [193,249]. The operations are merged, the instructions are scheduled and the registers are allocated for the input application code for a given processor configuration and objectives using a combination of static heuristics and evolutionary algorithm algorithms.

4.1 Operation Merging

The merging of operations, also called operation merging [84], is an optimization feature that merges several micro-operations (MOs) into one [193]. An assembler example for the operation merging is given in Figure 3.35. This feature is a kind of code selection, since it modifies the sequence of MOs from the input program before instruction scheduling. The MOs must be of the same type and use the same operation modes in order to be merged. For automatic merging, the code generator can search the basic block for mergeable MOs and replace them with functionally equivalent X2-MOs.

The first approach in [84] processes the MOs in a basic block sequentially. For each MO, the algorithm searches the operation sequence for a MO of the same type that can be merged. The first candidate found is used to create a new merged MO. This approach aims at an efficient merging of repetitive operation patterns, which are created, e.g., during loop unrolling. However, if the regular structure of repetitive patterns is interrupted by other operations or dependencies, more sophisticated merge decisions may be required. A merge of two MOs changes the data dependencies and also restricts the register allocation. The added dependencies could prevent subsequent MOs from merging and affect the overall code compaction.

The operation merging (X2) optimization uses evolutionary algorithms (EAs) to improve code compaction. The automatic operation merging based on EA first determines all possible merges, i.e., pairs of MOs, in a straight line microcode (SLM). In a second step the population is initialized with chromosomes for the merging, as shown in an example in Figure 4.2. To initialize the first generation, a chromosome without merging is added. All merging candidates are set to the stop entry, which is denoted by ‘X’. A second chromosome is added, which results in the same sequential merging sequence as the static heuristic described above. Therefore, the first generation contains two individuals with chromosomes which are predefined. The rest of the population is filled with random individuals, where the candidate lists in each gene are permuted randomly.

The next generations of individuals are generated from the individuals of the previous generations by copying the best three (elitism), using crossover, mutation (about 90 %) and generation of new random chromosomes. The population size remains constant for all generations. Crossover generates two child individuals from two parents selected by tournament selection. With this selection method, a subset (tournament set, by default three individuals) is selected from the individuals and the fittest of them is returned. Each gene in the first child is randomly selected from one of the parent’s gene (i.e., the order of the MOs in the candidate list is copied). For the second child, the gene from the other parent is selected. With a default mutation rate of m = 1 %, the individual genes in the children are mutated by mixing their merge candidate lists.

In order to evaluate the fitness of an individual, the merged SLM is passed on to the instruction scheduler and register allocation. Both can use the heuristic-based techniques described above

```
1 ADD        VxR0 , V0R2 , V0R0
2 SRI        VxR1 , VxR0 , #1
3 MAX        VxR2 , VxR1 , V0R2
4 PERMREG0   VxR2 , VxR0 , VxR2
5 MAX        VxR0 , V1R0 , V1R1
6 ADD        VxR2 , VxR0 , V0R0
7 ADD        VxR2 , VxR0 , V0R1
8 PERMREG0   VxR4 , VxR0 , VxR2
9 PERMREG1   VxR8 , VxR0 , V0R2
```

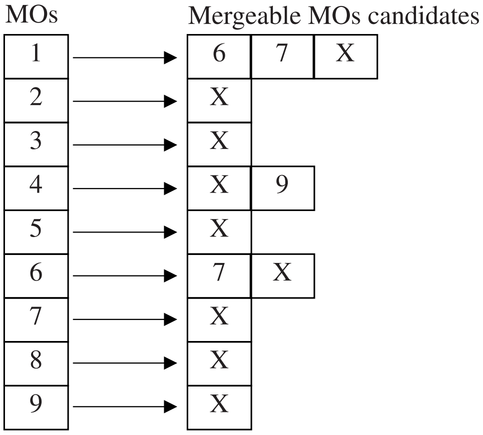


Figure 4.2: Exemplary assembler code and corresponding chromosome for the X2 instruction merging [193]. The chromosome for X2 merging lists all possible merging candidates for each MO. A stop entry, denoted by 'X', stops the candidate search and prevents merging of the corresponding MO.

or the evolutionary algorithms explained in the following sections. For example, the fitness value is the number of MIs in the scheduled program. The fitness value is computed to divide the population into different sections: The first section contains all individuals for whom a valid instruction scheduling and register allocation has been found. The second section contains all individuals for whom a scheduling was performed but no register allocation could be found. The third section contains individuals for whom a register allocation was found to be impossible. If instruction scheduling was not possible for the merged code of an individual, it is placed in the fourth section, where the individuals are sorted by the number of MOs, with merged MOs counting as one.

The population size in each generation is based on the number of possible operation merges in a SLM, which can be derived from the lists of merge candidates for operations. A command line parameter allows the user to specify the merge level of the compiler backend that affects the population size. If n is the number of possible merges and x is the merge level, then the population size S is computed as $S = n \cdot 2^{x-2}$. Merge level 0 disables automatic merging, while merge level one uses static heuristic. The assembler code can contain pragmas to override the merge level for individual SLMs, so that the programmer can influence the distribution of the computational effort over the SLMs. The EA-based merging uses a dynamic stop criterion. The algorithm stops, if a certain number of generations has not improved over the best solution.

To speed up the code generation, the instruction scheduling and register allocation steps for a chromosome can be skipped, if the SLM including the candidate list cannot be scheduled smaller than the best solution so far. From the data dependency graph of the merged SLM the critical path can be computed. Also, dividing the number of MOs by the number of issue slots gives a lower limit for the number of MIs after scheduling. If this minimum size is greater than an user-defined offset from the current best solution, the instruction scheduling is skipped.

4.2 Instruction Scheduling

Instruction scheduling performs parallelization of the code by converting the sequence of MOs written by the programmer into a sequence of micro-instructions (MIs), where each MI contains one MO for each issue-slot [193]. The heuristic-based instruction scheduler [84] uses list scheduling [250] to address this optimization problem.

The list scheduling constructs the MIs one after the other. The algorithm contains a set of all MOs whose data dependencies are satisfied and selects one from this list for placement in the current MI. Instead of exhaustively testing every possible MO and thereby creating a tree with different solutions, the selection is based on a heuristic function that assigns a weight to each MO. The static heuristic derives weights for the operations in the input program from data dependency graphs (DDGs) to influence the order in which they are picked for scheduling. An example for the weights and the instruction selection order for list scheduling is shown in Figure 4.3. If a SLM contains a branch or jump operation, that operation is handled after all other operations have been placed. According to the number of delay slots for branches, the last scheduled MIs are removed, the branch is placed, and the removed operations are rescheduled to fill the delay slots. The heuristic has a strong influence on code compaction. Static heuristics, e.g., the solution in [251], which derives the weight for a MO from the data dependency graph, can lead to suboptimal results, because scheduling decisions cannot be easily reversed if they turn out to prevent better scheduling later. A more flexible dynamic heuristic based on evolutionary algorithms (EAs) is proposed.

The EA-based instruction scheduler is an extension of the heuristic-based scheduler. The list scheduling uses weights to select MOs during scheduling. Instead of statically deriving the weights from the input program (the data dependency graph (DDG)), an EA is used to find a set of weights that results in an optimized scheduling. The chromosome for the EA-based scheduling algorithm contains one gene for each MO in the SLM. The gene assigns a weight to the corresponding MO.

The EA-based scheduling always includes the static heuristic-based solution. For the other chromosomes in the population, the weights are randomly chosen between 1 and the highest weight from the DDG.

The population size S for the EA-based instruction scheduling is derived from the number n of MOs in the SLMs and can be influenced by an optimization level o. It is computed as $S = n \cdot 2^{o-2}/5$ [249], but a minimum population size of 10 individuals is always used. The value $o = 0$ disables scheduling and $o = 1$ enables only static heuristic-based list scheduling.

```
1 ADD        VxR0 , V0R2 , V0R0
2 SRI        VxR1 , VxR0 , #1
3 MAX        VxR2 , VxR1 , V0R2
4 PERMREG0   VxR2 , VxR0 , VxR2
5 MAX        VxR0 , V1R0 , V1R1
6 ADD        VxR2 , VxR0 , V0R0
7 ADD        VxR2 , VxR0 , V0R1
8 PERMREG0   VxR4 , VxR0 , VxR2
9 PERMREG1   VxR8 , VxR0 , V0R2
```

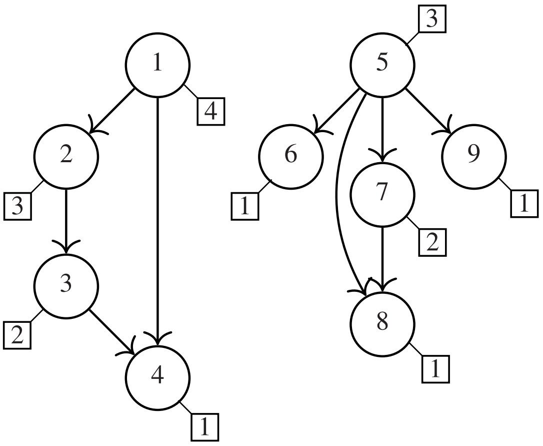


Figure 4.3: Exemplary assembler code and corresponding data dependency graph (DDG) [193]. Nodes represent MOs corresponding to code lines. Weights are shown as small numbers.

The number of MOs in a SLM allows the estimation of the compaction complexity and is therefore used to concentrate the computational effort on the larger SLMs. The assembler code may contain pragmas to override the global optimization level for individual SLMs.

For the next generation, the three fittest individuals from the previous generation are copied (elitism). Approximately 1 % of the new population is randomly generated to introduce new material into the gene pool and thereby cover a wider range of the search space. The remaining part of the new generation is formed by recombination (i.e., crossover followed by mutation). These distributions have been determined empirically. For recombination, two parent individuals from the previous generation are selected by tournament selection with a tournament size of three individuals and their genes are combined to form a new individual. Each gene in the child individual is selected independently from the others with equal probability from one parent. The individuals from the crossover are subject to a mutation that randomly overwrites the weights in one gene with a small probability of 0.8 %.

The evaluation of the individuals in the population is done by sorting the individuals to allow tournament selection for the generation of the next generation. First, instruction scheduling is performed using the parameters from chromosome. After that the registers are allocated. Then the population is divided into different sections. The first section contains all individuals for which a register allocation of the compacted code could be found. The individuals in this section are sorted by fitness values, which represents the size of the compacted program (i.e., number MIs). The second section contains all individuals for which a register allocation was not attempted or was not successful. In this section, the individuals are sorted by the size of the compacted program or, if identical for two individuals, by the fitness of the register allocation, where the EA-based register allocation takes precedence over the heuristic register allocation.

From the DDG of the SLM and the number of MOs the theoretical minimum number of MIs in the compacted SLM is derived. If this size is reached during scheduling, the EA stops. Otherwise, the algorithm stops after a certain number of generations without improving the fitness of the best individual. If the size of the best individual changes, the stop counter is also reset. This stop criterion is started when the first valid solution is found. If no valid solution can be found, the algorithm stops after a specified number of rounds. The register allocation (REGA) step for a compacted SLM can be skipped completely, if the number of MIs after scheduling of the current chromosome is more than a given offset of the current skip size, which corresponds to the size of the worst individual in the elite set. In this case, a penalty is added to the fitness of the chromosomes. Skipping the register allocation helps to speed up the scheduling algorithm. If a EA-based register allocation is used, a heuristic-based register allocation is performed first. Only if this is not successful, the EA-based register allocation is performed. To speed up the evaluation of scheduled individuals a maximum number of EA-based register allocation can be specified. After the heuristic register allocation the individuals are sorted by fitness as described above. For all individuals with the same size as the smallest individual and the specified number of larger individuals the EA-based register allocation is performed.

4.2.1 Issue-Slot Based Predication Register Allocation

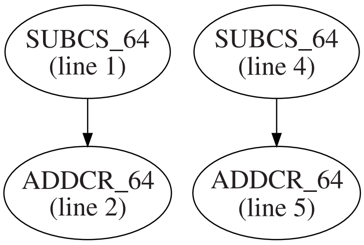
The issue-slot based predication technique presented in Section 3.2.5 requires the instruction scheduler to schedule conditional operations on a predefined issue-slot. If multiple predicate registers are used, the issue-slot of conditional micro-operations (MOs) has to be bound to the issue-slot of the corresponding predicate register. Therefore, the predicate register is allocated during instruction scheduling. The instruction scheduler determines the issue-slot for every instruction for a given application. The predicate register resource dependencies are checked during instruction scheduling using data dependency graphs (DDGs). The DDGs indicate, which conditional instructions are interdependent due to the data in the condition flags and must therefore be scheduled before other conditional instructions can be scheduled. A DDG representation for a code with two conditionally executed *addition (ADD_CR)* (condition read (CR)) instructions with conditional flags set by two *subtraction (SUB_CS)* (condition set (CS)) instructions is shown in Figure 4.4. There is no dependency between these two instructions, because there is no connection between the subtrees in the DDG. These can be scheduled in parallel on two issue-slots, using two different predicate registers set by the *subtraction (SUB_CS)* (condition set (CS)) instructions.

Due to the issue-slot constraint for conditional instructions, the scheduling process may fail for a SLM under certain circumstances. This may the case if no further conditional operations can be scheduled due to the scheduling order and resulting dependency conflicts. This happens if a set of conditional instructions of two subtrees of the DDG is scheduled on both issue-slots, but the remaining conditional operations of these subtrees cannot be scheduled due

Sequential assembler code with two conditions

```
1 SUBCS_64 V0R0 , V0R1 , V1R2
2 ADDCR_64 V0R3 , V0R4 , V1R5
3
4 SUBCS_64 V0R6 , V0R7 , V1R8
5 ADDCR_64 V0R9 , V0R10 , V1R1
```

Corresponding data dependency graph (DDG)



Scheduled code on two issue-slots

Issue-Slot 0 Issue-Slot 1

```
SUBCS_64 V0R6 V0R7 V1R8;     SUBCS_64 V0R0 V0R1 V1R2
ADDCR_64 V0R9 V0R10 V1R1;    ADDCR_64 V0R3 V0R4 V1R5
```

Figure 4.4: Exemplary assembler code with two independent conditions. The corresponding DDG representing the condition flag data dependency is depicted. The condition set (CS) and condition read (CR) instructions are scheduled on two issue-slots in parallel, using a separate predicate register.

to register dependencies on other not yet scheduled conditional operations of other subtrees. The conditional instruction of these other subtrees cannot be scheduled because the conditional flag registers are still in use. The scheduler detects this conflict and aborts scheduling and marks these SLMs as not schedulable. To resolve this issue, evolutionary algorithms for instruction scheduling are used. If a SLM cannot be scheduled due to imposed constraints of the conditional operations, its fitness value is set to a high constant value and the number of conditional operations, which could not be scheduled, is added to this constant. Therefore, individuals with more conditional instructions, which cannot be scheduled, have a lower fitness value. Their chance to be selected for the next generation is slightly reduced if no valid solution is found at that time. Compared to a heuristic-based scheduler, which may fail to find a valid scheduling, the EA-based scheduler is able to find scheduling solutions for the proposed issue-slot based predication technique if the number of trials is large enough. Furthermore, it can optimize the issue-slot and predicate register allocation for an overall efficient scheduling [193].

4.2.2 Towards Power-Aware Instruction Scheduling

The *KAVUAKA* system-on-chip (SoC) of Section 3.5 is designed and implemented as a research application-specific integrated circuit (ASIC). In this case study, different hearing aid hardware configurations and power optimization techniques are evaluated based on-chip power measurements. These optimizations affect different architecture and software implementation levels. Additionally, the potential of different power-aware compiler and scheduler techniques for application-specific hearing aid processors is evaluated. The goal is to reduce the power consumption of ASIP architectures by exploiting the flexibility offered by these programmable hardware architectures. In this case, the instruction set architecture (ISA) is analyzed in order to create an accurate power model of the underlying processor architecture.

The first step for generating an instruction-set power consumption model requires the measurement of the base power of the instruction set of the *KAVUAKA* processor. The technique used here is described in [252]. The base power for every instruction is measured while the processor executes a loop containing only the target instruction at a clock frequency of 50 MHz. The measurement results of the base power consumption for a subset of arithmetic and load/store instructions is depicted in Figure 4.5.

For these measurements, all registers of the processor were initialized with zeros. The opcodes of the instructions are varied by using different condition flags and SIMD subword modes. Results of the measurements show, that the maximum deviation of power consumption for different instructions is about 35 %.

With the data from the base power measurements, the power consumption of any program can be estimated for each cycle. A measurement of the total chip power consumption over time for a simple test program is shown in Figure 4.6. The program consists of a sequence

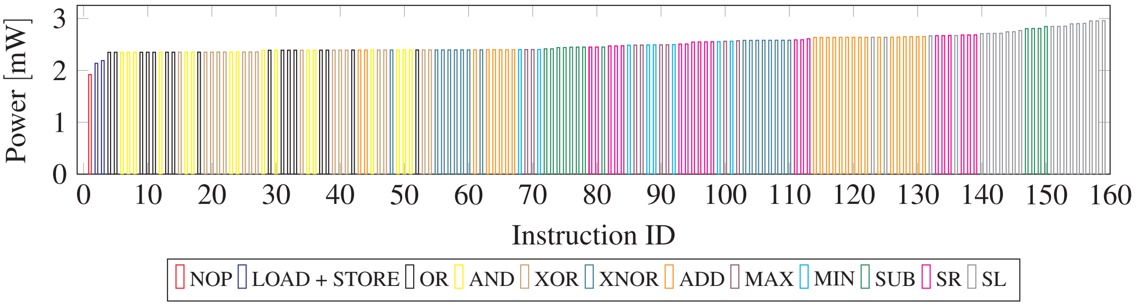


Figure 4.5: Base power consumption for a set of arithmetic instructions

of *ADD* statements followed by a sequence of *NOP* instructions. The clock frequency is 10 MHz and the 32-bit *KAVUAKA32* core is used. The simulated, measured and estimated (model-based) power consumption values are compared. The measured power consumption is the total system-on-chip (SoC) power, which includes all components shown in Figure 3.76. The simulated power is given for the SoC and the *KAVUAKA32* core for worst case operating conditions (−40°), whereas the measurement is performed under typical conditions (25°). The developed model estimates the power consumption with a maximum error of 2 %.

4.3 Register Allocation

The presented compiler backend allows using variables in the form of virtual registers [193]. Their scope is limited to a basic block and they are allocated to physical registers after instruction scheduling. When the parser detects a MO writing to a virtual register, the register is renamed to an unused register for this and all subsequent MOs, in order to remove artificial data dependencies (e.g., write-after-read).

Register allocation has to consider constraints, e.g., register pairs for merged MOs and number of read and write ports per register file partition, and is therefore split in two steps. First, the virtual register is assigned to a register file partition. A heuristic function balances the distribution of registers between the available partitions, as proposed in [251]. Then, in the selected partition, a free register is allocated. When allocating single registers, the allocation subroutine tries to keep pairs of consecutive registers free, in order to support allocation of register pairs from merged operations. No spill code is generated in case the number of live variables is higher than the number of free registers.

Similar to the heuristic-based register allocation, a evolutionary algorithm (EA)-based allocation is proposed, which first assigns the virtual registers to the available register files and selects the one of the registers in a second step. Instead of deriving this distribution from static properties of the compiled program, the EA encodes and evolves the distribution in the chro-

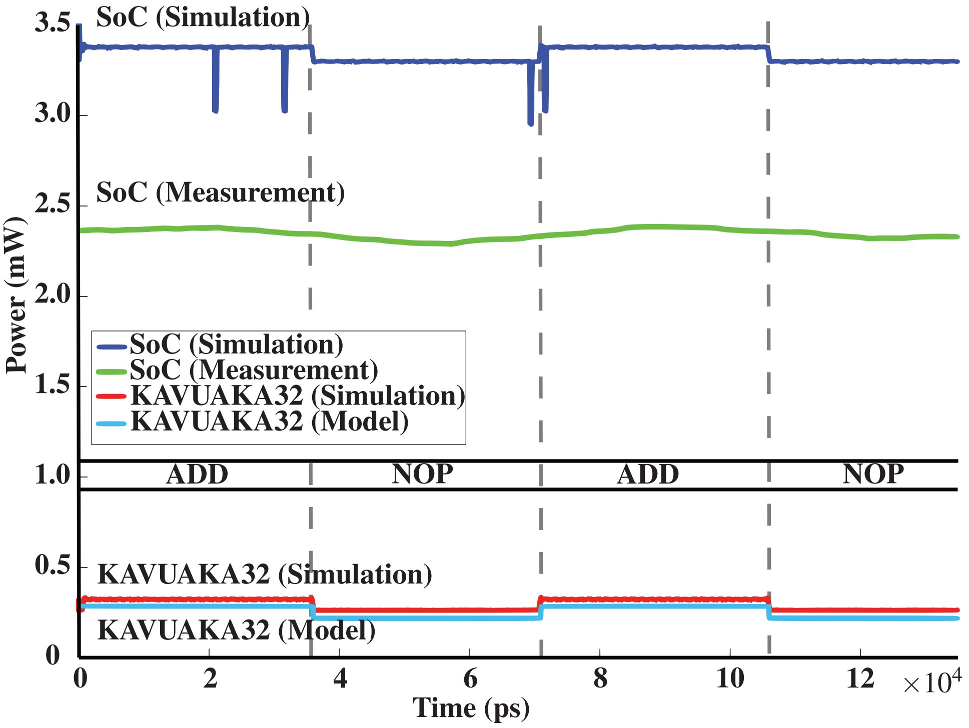


Figure 4.6: Measured, simulated and estimated dynamic power consumption over time for a sequence of ADD and NOP instructions.

mosomes. Due to operation merging and register file configuration, dependencies between registers can arise, which are analyzed in a register dependency graph (RDG) for the scheduled SLM. This helps reducing the search space and allocation complexity, as for each cluster of interdependent registers only one representative has to be encoded in the chromosome. An example RDG for a scheduled assembler code is shown in Figure 4.7. Due to the X2 operation merging constraints, registers in a pair (like VxR3+VxR7) have to be allocated in the same register file, denoted in the graph by an edge with an equal sign. These two target registers require two write ports in one register file. Therefore, the target register VxR5 of the parallel operation has to be scheduled in the other register file, denoted by the edge with the unequal sign.

```
5 MAX VxR4 , V1R0 , V1R1 ;                           1 ADD VxR0 , V0R2 , V0R0
8 PERMREG0 VxR8 , VxR0 , VxR2 ;                      2 SRI VxR1 , VxR0 , #1
3 MAX VxR2 , VxR1 , V0R2 ;                           7 ADD VxR6 , VxR4 , V0R1
8 PERMREG0_X2 VxR3+VxR7 , VxR0+VxR4 , VxR2+VxR6 ;    6 ADD VxR5 , VxR4 , V0R0
```

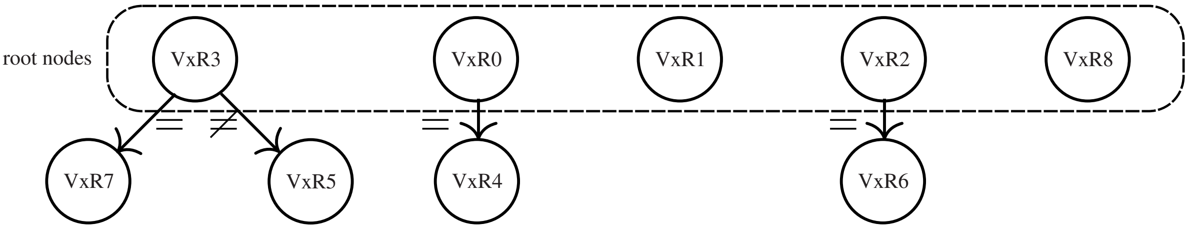


Figure 4.7: Exemplary scheduled assembler code and corresponding register dependency graph (RDG) [193]

The chromosome contains a gene for each independent virtual register in a SLM, which stores the associated register file for that register. During initialization, the registers are distributed randomly on the available register files. In contrast to EA-based scheduling and merging, no individual with information from the static heuristic is created for register allocation, as the overhead for computing such an individual is not justified. However, the EA-based register allocation (REGA) is only performed, when the static heuristic-based register allocation failed and can therefore only improve the results. The recombination of two parent chromosomes picks each gene randomly from one of the parents and mutates a gene in the child chromosome with a default probability of 1 %, by choosing a random register file for the register. Parents are selected by tournament selection from the previous generation.

The population size S is derived from the number of independent registers n and an optimization level P as $S = n \cdot 2^{P/8-5}$ with a minimum population size of five individuals. To generate the next generation of individuals, the best individual is copied (elitism), about 10 % of the new population is generated randomly and the remainder of the population is filled by recombination.

If a valid register allocation is found, the EA stops. For invalid register allocations, a fitness value is computed by considering the different possible reasons for invalid register allocations with decreasing priority: (1) The number of extra registers needed, when the number of live variables exceeds the number of registers. (2) Unbalanced use of register file partitions. (3) The number of read and write port conflicts after register allocation. This fitness value is used in tournament selection, the stopping criterion, and also evaluated in the EA-based instruction scheduling.

The EA stops after a certain number of generations without any improvement in the fitness of the fittest individual, or as soon as a valid register allocation is found. If the scheduling objective is performance, this stopping criterion is sufficient, but if other objectives (e.g., power consumption) are considered, the algorithm should not stop on the first valid register allocation as other allocations might improve the objective function.

4.3.1 Power-Aware Register Allocation

A fitness function is computed by the evolutionary algorithm for the register allocation. It is used to optimize the power consumption of the application-specific instruction-set processors (ASIPs) during compile time. Besides the instruction opcodes [253–255], the addresses of the source and target registers cause switching activity in the address decoder of the multi port register file [207]. A power model of the underlying processor architecture is required in order to make necessary and substantiated decisions during the power optimization process.

In order to verify, that the register allocation has a significand influence on the total power consumption of the processor, synthetic benchmarks, comparing the best and worst case register addressing in terms of Hamming distance of two consecutive instructions, are used. In the best case, the source and target registers of two consecutive instructions are the same. Therefore, the resulting Hamming distance between each register address of the consecutive instructions is zero. A code example for evaluating the best case register allocation is given in Figure 4.8. For the worst case, every bit of the source and target register address is toggled. The worst case register allocation is given in Figure 4.9, which results in the maximum Hamming distance. This effect was verified by gate-level power simulations, as shown in Table 4.1, Figure 4.10 and Figure 4.11. Table 4.1 lists a detailed summary of the causes of power consumption and their occurrence in the processor architecture for the different tests. Leakage power consumption is small compared to the dynamic power consumption. The power consumption of the register file accounts for up to 19.2 % of the total system-on-chip (SoC) power. The total power of the *KAVUAKA* processor is depicted for each of the register allocation test programs in Figure 4.10. The comparison between the tests with registers, which were initialized with constant or random data, shows a similar pattern, where the influence of the register allocation on the total processor power consumption is visible.

To further study the influence of the register allocation on the power consumption, six additional synthetic benchmarks with random registers and instructions are evaluated. The sum of

```
#repeat 100
:0 ADD V0R0, V0R0, V0R0          :1 OR  V1R0, V1R0, V1R0
:0 ADD V0R0, V0R0, V0R0          :1 OR  V1R0, V1R0, V1R0
#endrepeat
```

Figure 4.8: Best case register allocation. Test program for power consumption evaluation.

```
#repeat 100
:0 ADD V0R0,  V0R0,  V0R0        :1 OR  V1R0,  V1R0,  V1R0
:0 ADD V0R31, V0R31, V0R31       :1 OR  V1R31, V1R31, V1R31
#endrepeat
```

Figure 4.9: Worst case register allocation. Test program for power consumption evaluation.

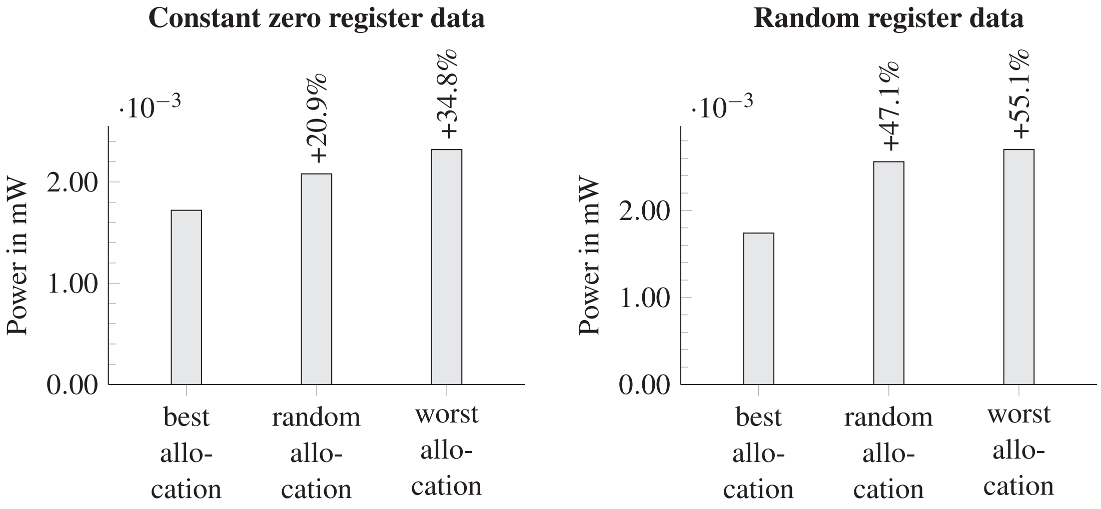


Figure 4.10: Average power consumption of the *KAVUAKA* 64-bit processor for best (min. Hamming distance), random and worst (max. Hamming distance) case register allocation. The average power consumption values are given for the case where registers are initialized with constant zeros (left figure) and the registers are initialized with random data (right figure).

Table 4.1: Internal, switching and leakage power consumption for a register addressing test. Results from an ASIC gate-level simulation with the TSMC 40 nm HVT low-power technology at 50 MHz.

Hierarchy	Internal Power	Switching Power	Leakage Power	Total Power	Percentage of SoC Power
Register allocation: best case; Register data:random					
KAVUAKA 64-bit	1.35e-03	3.81e-04	2.02e-06	1.74e-03	16.8 %
Register file	9.84e-04	2.37e-04	6.62e-07	1.22e-03	11.8 %
Register allocation: best case; Register data:zeros					
KAVUAKA 64-bit	1.35e-03	3.69e-04	2.05e-06	1.72e-03	16.7 %
Register file	9.82e-04	2.32e-04	6.80e-07	1.22e-03	11.8 %
Register allocation: random case; Register data:random					
KAVUAKA 64-bit	1.85e-03	1.53e-03	2.00e-06	3.39e-03	28.1 %
Register file	1.29e-03	1.03e-03	6.49e-07	2.32e-03	19.2 %
Register allocation: random case; Register data:zeros					
KAVUAKA 64-bit	1.49e-03	8.05e-04	2.05e-06	2.29e-03	20.9 %
Register file	1.04e-03	4.73e-04	6.80e-07	1.52e-03	13.9 %
Register allocation: worst case; Register data:random					
KAVUAKA 64-bit	1.62e-03	1.08e-03	2.02e-06	2.70e-03	23.6 %
Register file	1.07e-03	5.34e-04	6.62e-07	1.61e-03	14.1 %
Register allocation: worst case; Register data:zeros					
KAVUAKA 64-bit	11.50e-03	8.14e-04	2.05e-06	2.32e-03	21.0 %
Register file	1.02e-03	3.70e-04	6.79e-07	1.39e-03	12.6 %

all Hamming distances of all consecutively addressed registers are compared to the simulated power consumption in the register file read and write ports. A linear relation between the Hamming distance and the power consumption in the read and write ports of the register file can be observed in Figure 4.11.

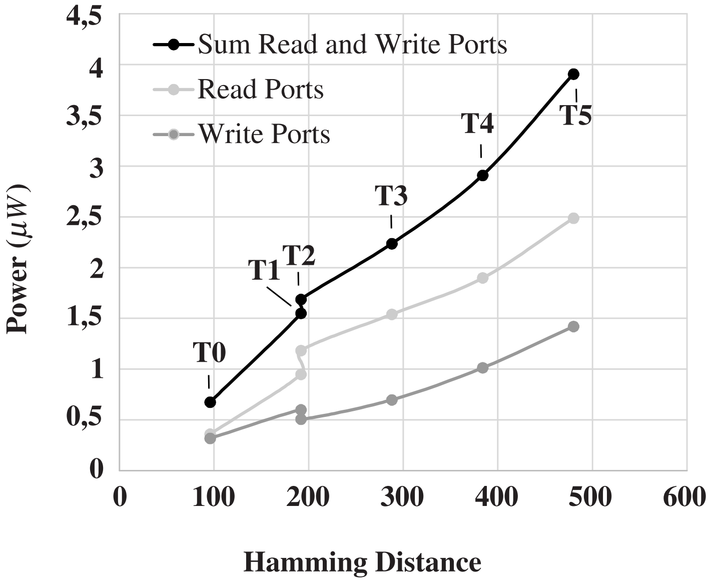


Figure 4.11: Simulated power consumption for read and write ports of the register file based on the cumulative Hamming distance of the register addresses for six different benchmarks. [56]

In order to predict the power consumption caused by the register allocation, the model presented in [256] is applied. A linear regression approach is used to estimate the power consumption based on the Hamming distance of consecutive register addresses. The comparison between the model and the simulated power consumption is shown in Figure 4.12 for switching between different target and source registers.

In order to optimize the power consumption of a particular program, a heuristic for power aware register allocation is developed. During register allocation, the heuristic selects the next free register based on the minimal Hamming distance of its address compared to the last register address at the same port. If the heuristic power aware register allocation is compared to the standard register allocation, which selects registers one after another during allocation, the register file power consumption can be reduced for each synthetic benchmark with random instructions and virtual registers (Figure 4.13).

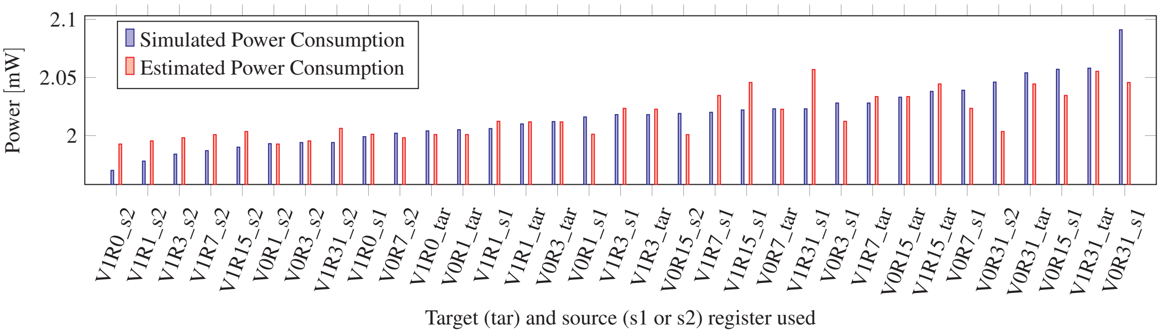


Figure 4.12: Simulated and estimated power consumption for a switch from register V0R0 to the listed target (tar) and source (s1 or s2) registers. [56]

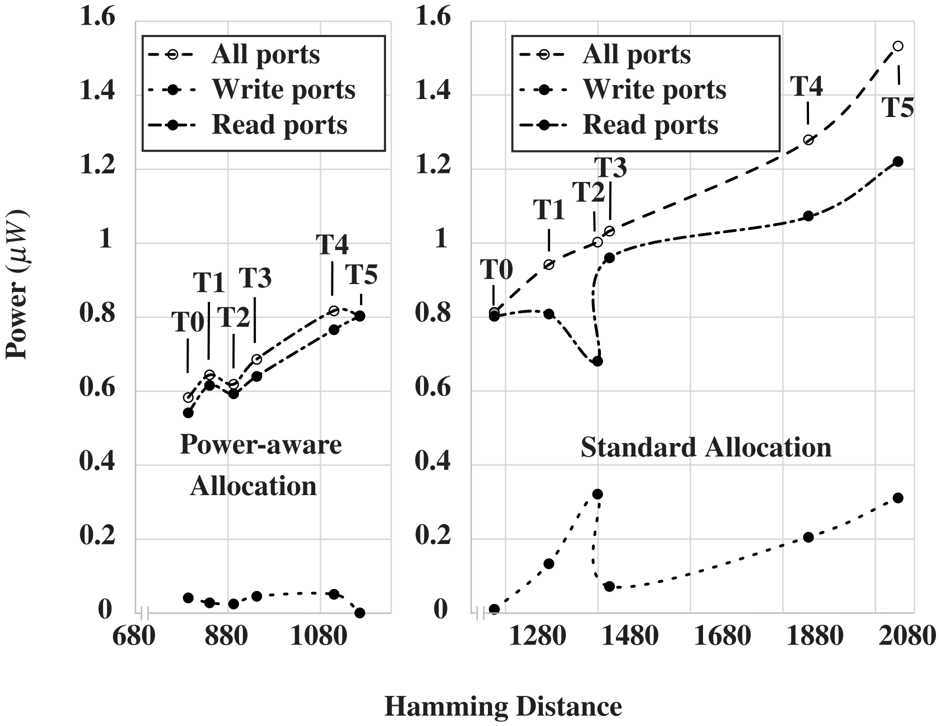


Figure 4.13: Simulated register file port power consumption for the power-aware and standard register allocation. The Hamming distance between register addresses was summed for each of the six synthetic benchmarks (T0-T5).

5 Evaluation and Design Space Exploration

In this chapter, the implementations of representative hearing aid algorithms are analyzed to evaluate the hardware extensions presented in Chapter 3. The increase in performance and the associated reduction in power consumption are evaluated. Additionally, it is analyzed how optimizations of the hardware architectures influence the algorithm performance and the audio quality.

5.1 Case Study: Beamforming Algorithms

In this section, different beamforming algorithms are studied to evaluate the application-specific hardware extensions described in Section 3.2 and Section 3.3.

Hearing impaired people suffer from reduced speech intelligibility. As a result, these people have difficulties with communication and social interaction. This is especially the case in complex acoustic scenarios in which several people speak simultaneously or other undesired noise sources exit. This so-called cocktail-party effect [257] leads to a considerable reduction of speech intelligibility, so that communication with other people is a serious challenge. Hence, in addition to standard hearing aid processing algorithms, such as frequency-dependent amplification and dynamic range compression, noise reduction algorithms in hearing aids are crucial to improve speech intelligibility [23, 63].

Modern digital hearing aids are equipped with two or more microphones [5, 23], enabling multi-channel digital signal processing [258, 259]. In order to increase speech intelligibility for the hearing impaired, beamforming algorithms [63, 223, 259, 260] are frequently used to suppress undesired components and to increase the signal-to-noise ratio (SNR) of the target speech signal. Fixed beamformers, also denoted as data-independent beamformers, are typically designed such that the signals arriving from a certain direction are passed through without any distortion. In contrast to fixed (data-independent) beamformers, adaptive (data-dependent) beamformers also exploit the signal statistics of the noise component in order to adapt to changing noise fields. The achieved algorithm performance in terms of improvement in signal-to-noise ratio (SNR) and in speech intelligibility (STOI) for hearing aid users depends on the implementation of the algorithm and the acoustic scenario.

Besides the performance of the algorithms, the battery life of digital hearing aids is an important aspect for user acceptance. Battery life depends on power consumption, which is partially consumed by the hearing aid's digital processor. The power consumption of the processor depends on its architecture, its utilization, its operating clock frequency, and other factors. Specializing a processor architecture for a target application by customizing its instruction-set architecture is a common technique used to reduce power consumption while keeping the required processing performance. The *KAVUAKA* ASIP and its application-specific extensions are evaluated based on selected beamforming algorithms and their performance.

The focus of this section is the power consumption evaluation of hearing aid ASIP optimizations based on the performance of different beamforming algorithms. Compared to the related work and [4, 10, 68, 104], which only mention the average or maximum power consumption of the processing hardware, this work provides a detailed and application-specific power consumption evaluation. The power consumption evaluation includes the description of several ASIP optimizations and their impact on the power consumption during the processing of different exchangeable beamforming algorithms. Furthermore, the performance of different beamforming algorithms using objective measures is evaluated in this section. The performance of these algorithms is then compared to the power consumption, which enables a trade-off analysis between the algorithm performance and power consumption. The proposed hardware optimizations are not only evaluated in terms of the average power consumption, silicon area overhead, and processing performance, but also in terms of the application-specific power consumption.

This section is structured as follows. The evaluated beamforming algorithms are listed in Section 5.1.1. The objective performance evaluation and the power consumption evaluation are presented in Section 5.1.2 and Section 5.1.3. At the end of the case study, the overall evaluation including the algorithm performance is given in Section 5.1.4 and the processing performance is compared with related architectures in Section 5.1.5.

5.1.1 Evaluated Beamforming Algorithms

The following three dual-microphone monaural beamforming algorithms have been used for the evaluation:

- Fixed Beamformer [63, 260]
- Adaptive Gain Beamformer [55, 63, 260]
- Adaptive Filter Beamformer [63, 260, 261]

The fixed differential beamformer, which is described in [63], is shown in Figure 5.1. This beamformer uses two omnidirectional microphones, which are closely spaced at a distance d. Due to this distance, sound waves arriving from different angles reach the microphones at different times. The rear microphone signal of the hearing aid is delayed by a constant time value τ and subtracted from the front microphone signal, resulting in a directional pattern. A common assumption for the use of this directional pattern is that the desired speaker is located in front of the hearing aid user, whereas the interfering sounds are located behind the hearing aid user. The resulting directional patterns, which indicate the sensitivity for different sound source angles, are shown in Figure 5.2 for three different constant delays. The resulting directional patterns are called cardioid, supercardioid, and hypercardioid.

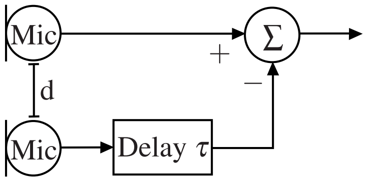


Figure 5.1: Fixed differential beamformer according to [63]. One of the microphone signals is delayed by a constant time value τ and subtracted from the other microphone signal.

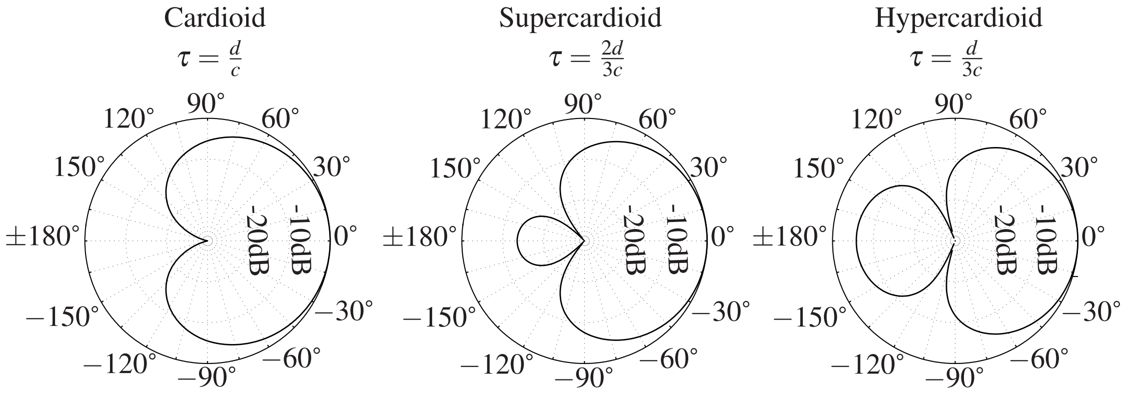


Figure 5.2: Fixed beamformer patterns for different constant delays τ: Cardioid ($\tau = \frac{d}{c}$), supercardioid ($\tau = \frac{2d}{3c}$) and hypercardioid ($\tau = \frac{d}{3c}$), where d is the distance between the microphones and c is the speed of sound.

Contrary to fixed beamformers, adaptive beamformers are able to adapt to the changing spatial characteristics of the interfering sounds. Two adaptive beamforming algorithms are shown in Figure 5.3 and Figure 5.4. The microphone inputs are first used to generate a front- and a back-facing cardioid response. For the adaptive gain beamformer (Figure 5.3) the back-facing response $c_2(n)$ is weighted with a time-varying scalar $W(n)$ and subtracted from the front-facing response $c_1(n)$, such that the output of the adaptive gain beamformer is defined by

Equation 5.1:

$$y(n) = c_1(n) - W(n) \cdot c_2(n) \tag{5.1}$$

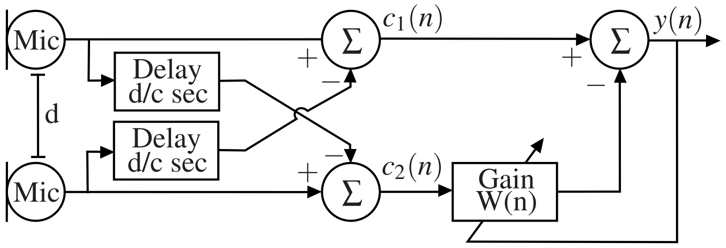


Figure 5.3: Adaptive gain beamformer according to [55, 260]. The output is the subtraction of the front-facing cardioid and the adaptively weighted back-facing cardioid response.

Assuming a frame-by-frame processing with frame index m and the frame length $M = 64$, the following adaptation for the gain factor $W(m)$ is used [55] (Equation 5.2):

$$\begin{aligned} W(m) &= \frac{\hat{R}_{c_1c_2}(m)}{\hat{R}_{c_2c_2}(m)} \\ \hat{R}_{c_1c_2}(m) &= \frac{\alpha}{M}\sum_{n=1}^{M} c_1(n)c_2(n) + (1-\alpha)\hat{R}_{c_1c_2}(m-1) \\ \hat{R}_{c_2c_2}(m) &= \frac{\alpha}{M}\sum_{n=1}^{M} c_2(n)^2 + (1-\alpha)\hat{R}_{c_2c_2}(m-1) \end{aligned} \tag{5.2}$$

where $\alpha = 0.5$ is an adjustable recursive smoothing parameter.

The adaptive filter beamformer, presented in [63] and shown in Figure 5.4, uses an adaptive finite impulse response (FIR) filter with a compensation delay in order to allow for acausal filter taps. The filter coefficients are updated using the following least mean squares (LMS) adaptation based on Equation 5.3:

$$w_k(n+1) = \beta \cdot w_k(n) + \mu \cdot c_2(n) \cdot y(n) \tag{5.3}$$

with $\beta = 1.0$ and $\mu = 0.3$ as adjustable parameters to control the adaptation speed.

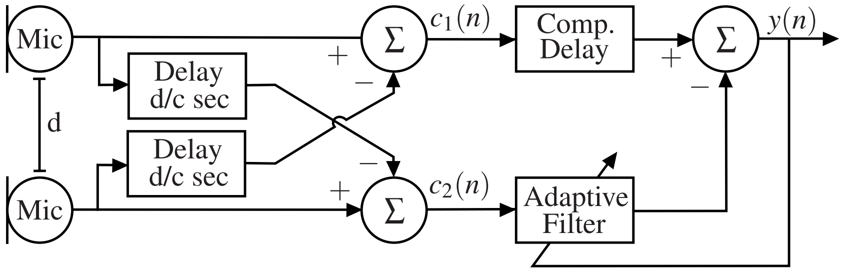


Figure 5.4: Adaptive filter beamformer according to [63]. The output is the subtraction of the front-facing cardioid and the adaptively filtered back-facing cardioid response.

5.1.2 Objective Performance Evaluation of the Beamforming Algorithms

Objective instrumental measures are used to compare the performance of different beamforming algorithms. In Section 5.1.3, the results of this evaluation are used to compare the algorithm performance obtained with hardware related requirements, such as static and dynamic power consumption for running these algorithms on the hearing aid processor.

The algorithm evaluation is based on the setup described in [262] for binaural pre-processing strategies. An acoustic test scenario is created using a database of behind-the-ear impulse responses [263]. Each of the behind-the-ear hearing aid is equipped with two microphones at a distance of about 7.6 mm and is mounted on an artificial head. From this database, the anechoic impulse responses[1] have been used to generate the hearing aid microphone signals for an acoustic scenario comprising one target and one interfering sound source. The target signal is a male speaker (*male.wav*) and the interfering signal is babble noise (*babble_olsa.wav*), both recorded from the Oldenburger Satztest (OLSA) [264] at a sampling frequency of 16 kHz. The target source is always at 0° in front of the head, while different interfering source angles ranging from 0° to 180° around the head are considered. To evaluate the performance of the fixed and the adaptive beamforming algorithms, the following instrumental measures are used:

- perceptual evaluation of speech quality (PESQ) [265]
- short-time objective intelligibility (STOI) [266]
- intelligibility-weighted signal-to-noise ratio (ISNR) [267]

The results of this evaluation, i.e., the PESQ, STOI and ISNR scores for all considered angles of the interfering source, are shown in Figure 5.5. The fixed beamformer is configured to generate either a cardioid, a supercardioid, or a hypercardioid response with a constant spatial

---

[1] Although in practice obviously also reverberation is present and more robust versions of the discussed (adaptive) beamforming algorithms should be considered [258, 259], it is assumed that the considered anechoic scenario suffices for the trade-off analysis in this work.

null in its directional pattern (Figure 5.2). Based on the PESQ, STOI, and ISNR scores, it can be observed that the adaptive beamformers yield a better performance on average for interfering sound source angles larger than 90° compared to the fixed beamformers. The reason for this performance difference is the adaptation to the angle of the interfering sound source.

The data type used for the algorithm implementation and for the results shown in Figure 5.5 is double-precision floating-point. For power consumption reasons, fixed-point processors are used in hearing aids. Therefore, the influence of the quantization and rounding errors of the fixed-point beamforming algorithm implementations on the algorithm performance are studied. Four different implementations with floating-point and three fixed-point data types with different word lengths are compared. The average PESQ, STOI, and ISNR scores for interfering source angels larger than 90° are given in Table 5.1. The 32-bit fixed-point implementation offers about the same algorithm performance compared to the double-precision implementation (maximum of 1 % deviation). The 24-bit and the 16-bit implementations have the same fixed-point formats as the 32-bit implementation with a reduced fraction length. The algorithm performance of the 24-bit adaptive filter beamformer is slightly decreased whereas both 16-bit adaptive beamformers become unstable and do not converge due to reduced fixed-point word length [268], resulting in a not working adaptation.

In addition to the simulated performance of the beamforming algorithms, measurements were performed in an anechoic chamber as shown in Figure 5.6. The loudspeakers generate the input signal for the hearing aid, which is located on a rotating base. The microphone signals are processed by the 64-bit *KAVUAKA* processor with a sampling frequency of 16 kHz and a sample width of 16-bit. The generated output of the *KAVUAKA* processor is feed into an audio analyzer, which processes the signal and stores the results [269]. The results of sensitivity measurements of the fixed and adaptive gain beamformer for three different frequencies are depicted in Figure 5.7. The measured sound pressure level (SPL) shows the supercardioid pattern for the fixed beamformer and the pattern of the adaptive gain beamformer. These patterns match the simulated results.

5.1.3 Power Consumption Evaluation

The static and dynamic power consumption of *KAVUAKA* running the beamforming algorithms is estimated using ASIC gate-level switching activity and sign-off power analysis. The minimum required clock frequency is determined and applied separately for each of the applications and processor configurations. The switching activity is recorded after all internal registers are filled with realistic data for the time taken to process 700 audio samples.

The total average power consumption of four *KAVUAKA* configurations with four different datapath width (24-bit to 64-bit) running the beamforming algorithms are shown in Figure 5.8. The minimum required clock frequencies for processing the beamforming algorithms, which

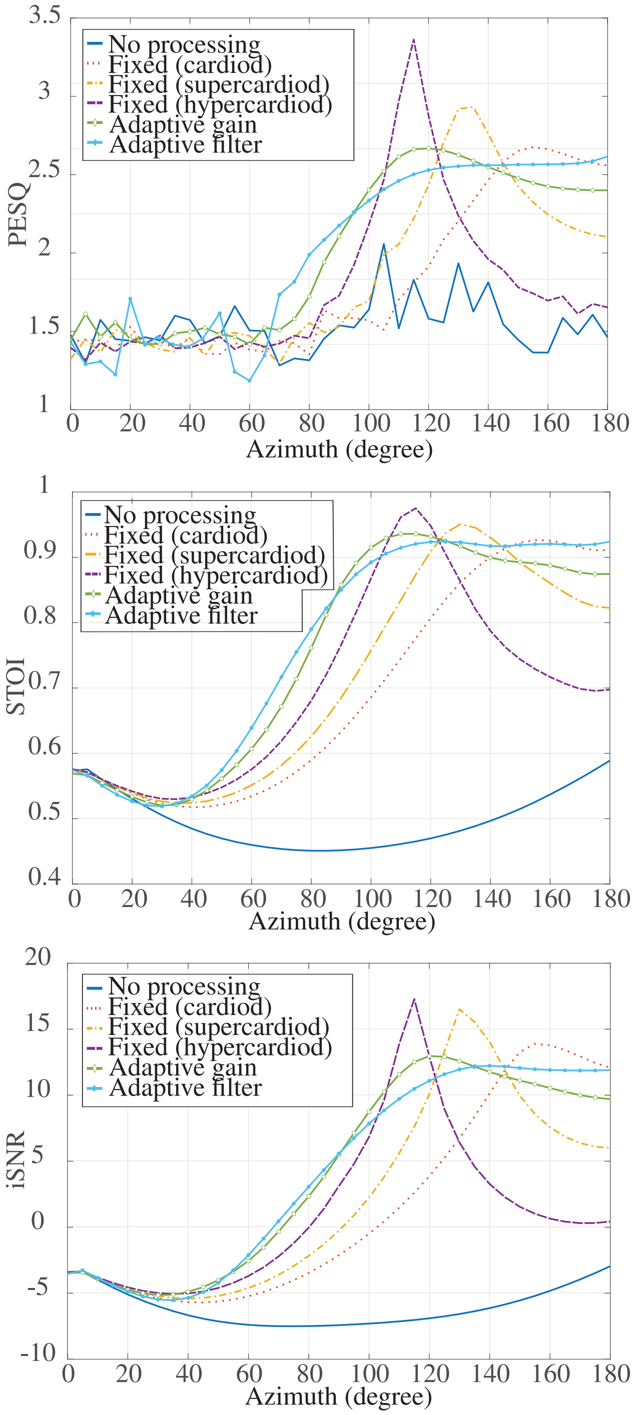


Figure 5.5: PESQ, STOI, and ISNR scores for the different fixed and adaptive beamforming algorithms as a function of the azimuth angle of the interfering sound source. Double precision floating-point is used in this case.

Table 5.1: Performance of beamforming algorithms: Average PESQ, STOI and ISNR scores for interfering source angles larger than 90°. Deviations of fixed-point implementations compared to double-precision floating-point (FP) implementations are given in percent.

Double Precision Floating-Point

	PESQ	STOI	ISNR
No processing	1.60	0.50	-5.92
Fixed (cardioid)	2.19	0.83	7.29
Fixed (supercardioid)	2.26	0.85	8.03
Fixed (hypercardioid)	2.10	0.80	5.19
Adaptive gain	2.48	0.90	10.61
Adaptive filter	2.49	0.91	10.73

32-bit **Fixed-Point**

	PESQ (% FP)	STOI (% FP)	ISNR (% FP)
No processing	1.60 (0%)	0.50 (0%)	-5.92 (0%)
Fixed (cardioid)	2.19 (0%)	0.83 (0%)	7.29 (0%)
Fixed (supercardioid)	2.26 (0%)	0.85 (0%)	8.03 (0%)
Fixed (hypercardioid)	2.10 (0%)	0.80 (0%)	5.19 (0%)
Adaptive gain	2.48 (0%)	0.90 (0%)	10.61 (-1%)
Adaptive filter	2.49 (0%)	0.90 (-1%)	10.72 (-1%)

24-bit **Fixed-Point**

	PESQ (% FP)	STOI (% FP)	ISNR (% FP)
No processing	1.60 (0%)	0.50 (0%)	-5.92 (0%)
Fixed (cardioid)	2.19 (-1%)	0.83 (0%)	7.29 (0%)
Fixed (supercardioid)	2.26 (0%)	0.85 (0%)	8.03 (0%)
Fixed (hypercardioid)	2.10 (0%)	0.80 (0%)	5.19 (0%)
Adaptive gain	2.47 (-1%)	0.90 (-1%)	10.61 (-1%)
Adaptive filter	2.38 (-5%)	0.89 (-2%)	10.07 (-7%)

16-bit **Fixed-Point**

	PESQ (% FP)	STOI (% FP)	ISNR (% FP)
No processing	1.60 (0%)	0.50 (-1%)	-5.92 (-1%)
Fixed (cardioid)	2.18 (-1%)	0.83 (0%)	7.27 (-1%)
Fixed (supercardioid)	2.26 (-1%)	0.85 (0%)	8.01 (-1%)
Fixed (hypercardioid)	2.10 (0%)	0.80 (0%)	5.17 (-1%)
Adaptive gain	2.18 (-12%)	0.90 (-8%)	7.21 (-23%)
Adaptive filter	0.58 (-77%)	0.34 (-63%)	-12.4 (-216%)

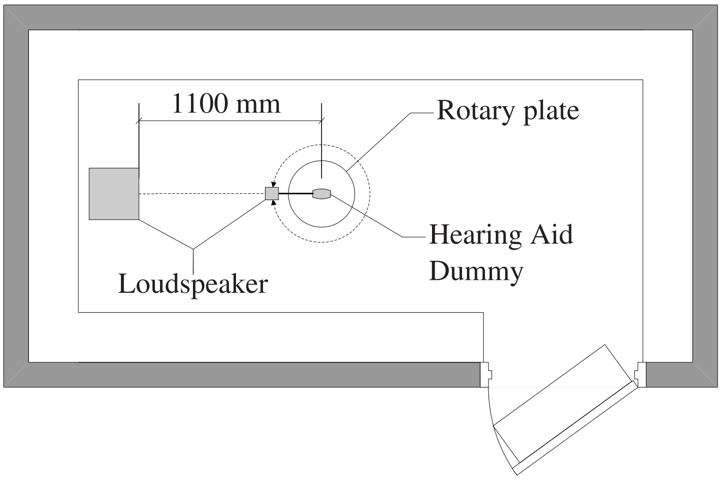


Figure 5.6: Measurement setup in an anechoic chamber.

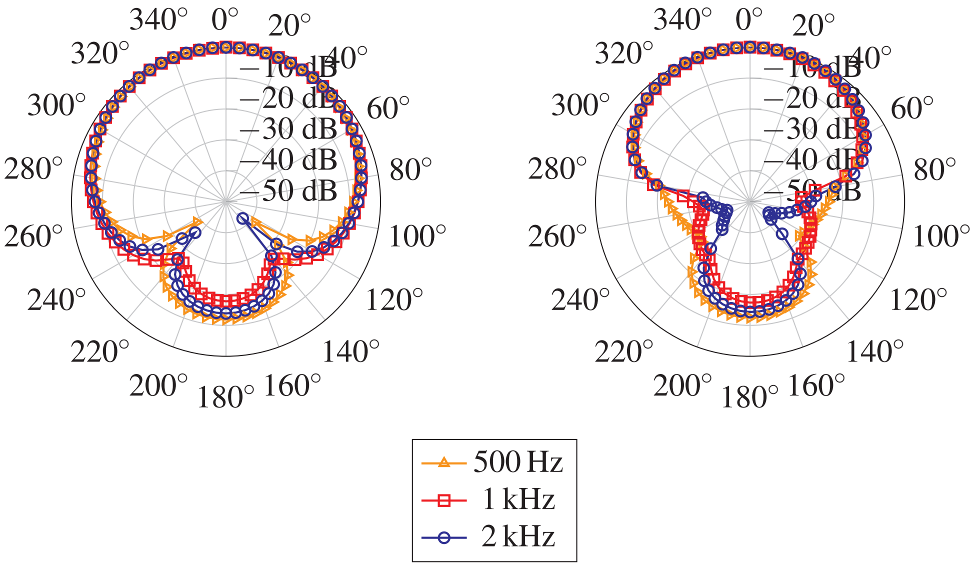


Figure 5.7: Polar pattern of the fixed and adaptive gain beamformer for 500 Hz, 1 kHz and 2 kHz single sine source. The adaptation is set to continuous adaptation. The adaption rate is set to 0.01.

are listed in Table 5.2, are used. Due to its comparatively low computing complexity, the fixed beamformer offers the lowest power consumption with a minimum value of 0.014 mW for the 24-bit configuration. The adaptive gain beamformer with a division computed by software library requires up to 0.624 mW, running on the 64-bit SIMD processor configuration. The power consumption of all beamforming algorithms scales linearly with the silicon area.

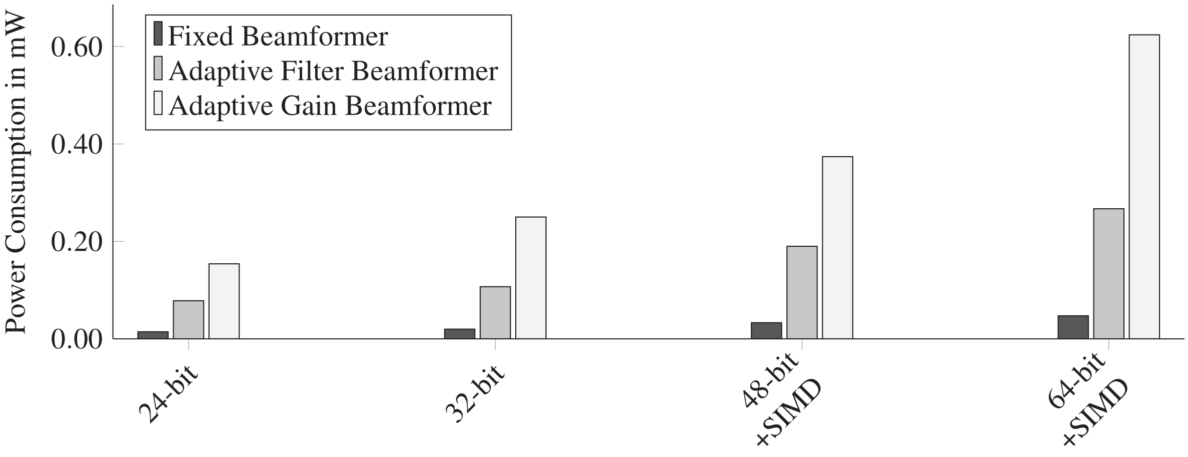


Figure 5.8: Total average power consumption for different datapath widths and SIMD modes.

The parallelizability of the beamforming algorithms is evaluated using the dynamic count of the instructions per cycle (IPC) and the minimum required clock frequencies, which are listed in Table 5.2. The instruction level parallelism is high, since the IPC values reach almost the maximum achievable value of two on the two issue-slot processor. The data level parallelism provided by the SIMD mechanism is not beneficial. The filter structures of the beamformers can not be packed efficiently into the SIMD subwords due to data dependencies and data flows. This leads to higher minimum clock frequencies, due to the extra required operations to repack the subwords. The repacking task requires more cycles than the reduction of cycles by SIMD processing. New permutation or rotation operations are required to reduce the cycles required for repackaging.

5.1.4 Overall Evaluation Including Algorithm Performance

The performance of the beamforming algorithms is compared to the hardware-related requirements, i.e., the silicon area and the static and dynamic power consumption. All register file (RF) implementations, the reference, dummy and isolation and different datapath configurations, which are described in Section 3.3, are evaluated, resulting in 24 different configurations of the *KAVUAKA* processor. The total average power consumption of these configurations,

Table 5.2: Dynamic instructions per cycle (IPC) and minimum required clock frequencies

Algorithm	Fixed		Adaptive Filter		Adaptive Gain	
	non-SIMD	SIMD	non-SIMD	SIMD	non-SIMD	SIMD
IPC	1.86	1.80	1.93	1.94	1.87	1.95
min. MHz	0.41	0.42	2.55	2.91	7.89	10.09

running the beamforming algorithms from Section 5.1.1, is compared with the silicon area requirements. The results are shown in Figure 5.9. The fixed beamformer (marker style: $\star$) consumes the lowest power, even when implemented on the 64-bit configuration. The hardware accelerated adaptive gain beamformer (marker style: $+$) consumes less power than the adaptive filter beamformer (marker style: x) and the software-based adaptive gain beamformer. The silicon area scales with the datapath width and increases with the configurations equipped with the hardware division co-processor (CP). The most efficient hardware and algorithm combination with the minimum area-power product is the 24-bit processor without a co-processor and without SIMD support running the fixed beamformer.

In order to evaluate the algorithm performance, the PESQ, STOI, and ISNR scores are plotted over the total power consumption in Figure 5.10. The fixed beamformer offers the lowest power consumption, but the algorithm performance is lower compared to the adaptive beamformers. Besides the 24-bit adaptive filter beamformer, both adaptive beamformers offer almost identical performance. When prioritizing the performance against the silicon area, the best combination is the 24-bit adaptive gain beamformer with the hardware co-processor and the dummy register file.

Due to the hardware acceleration using the CORDIC co-processor, the minimum clock frequency required for the adaptive gain beamformer can be decreased by up to 81 %, as shown in Table 5.3. The resulting static and dynamic power consumption with and without the co-processor is given in Figure 5.11. Despite the increased silicon area of the co-processor, the total power consumption drops by 62 % to 79 %, when using the hardware acceleration instead of the software computation of the division operation for the adaptive gain beamformer.

5.1.5 Comparison of the Processing Performance with related ASIPs

In this design space exploration, the processing performance and the estimated power consumption of the *Cadence Tensilica LX7* [270], the *HiFi4* [121], the *Fusion F1* [122], the *Fusion G3* [271], the *Fusion G6* [271] and the *Texas Instruments C6748* [123, 272] digital signal processors (DSPs) are evaluated. The *Cadence Tensilica LX7* is a high-level software programmable application-specific base processor architecture for digital signal processing.

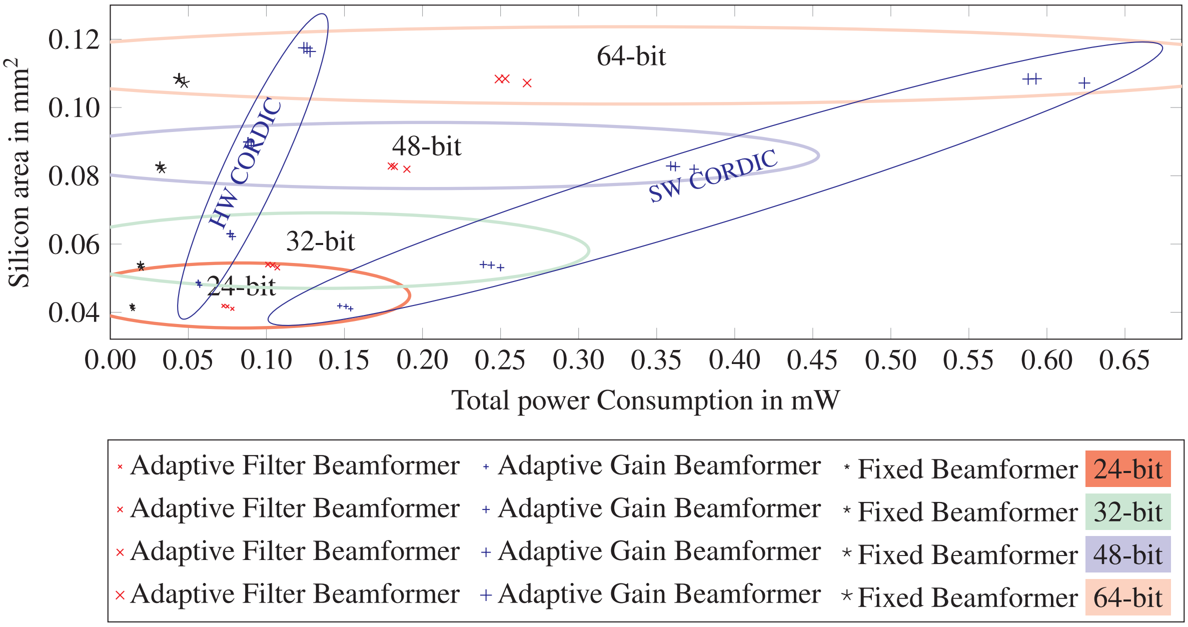


Figure 5.9: Total average power consumption compared to silicon area requirement. The colored ellipses hold all processor configurations with 24-bit, 32-bit, 48-bit and 64-bit with the reference, dummy and isolation register file implementation. Results from an ASIC synthesis with the TSMC 40 nm HVT low-power ASIC technology. The minimum required clock frequencies are used.

Table 5.3: Minimum operating clock frequencies in MHz for the adaptive gain beamformer for hardware configurations with and without a co-processor (CP)

24-bit	24-bit +CP	32-bit	32-bit +CP	48-bit	48-bit +CP	64-bit	64-bit +CP
6.2 MHz	1.9 MHz -69%	7.9 MHz	2.0 MHz -74%	7.9 MHz	1.7 MHz -78%	10.1 MHz	1.9 MHz -81%

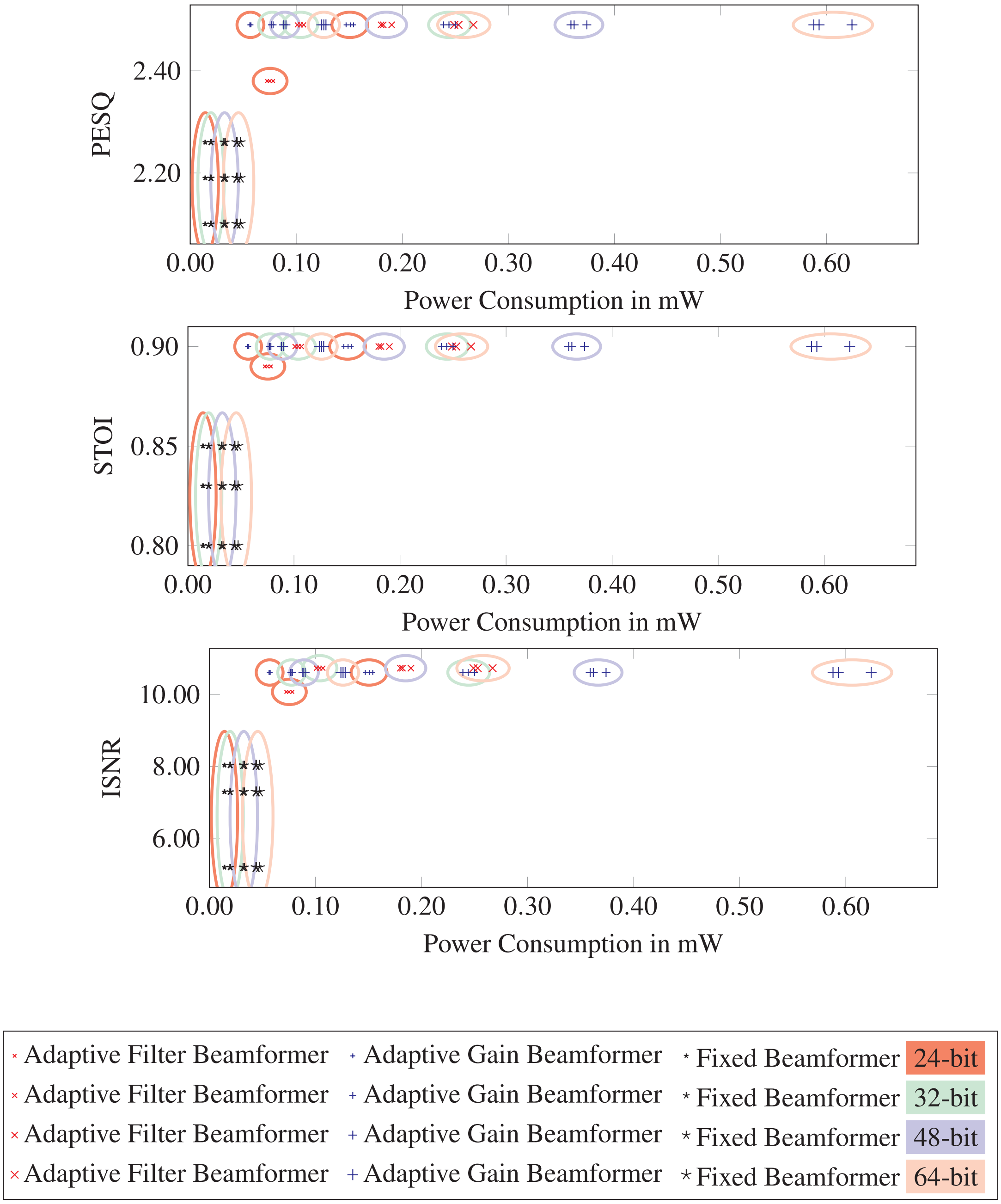


Figure 5.10: Total average power consumption compared to the average values of perceptual evaluation of speech quality (PESQ), short-time objective intelligibility (STOI) and intelligibility-weighted signal-to-noise ratio (ISNR) for interfering source angles larger than 90°.

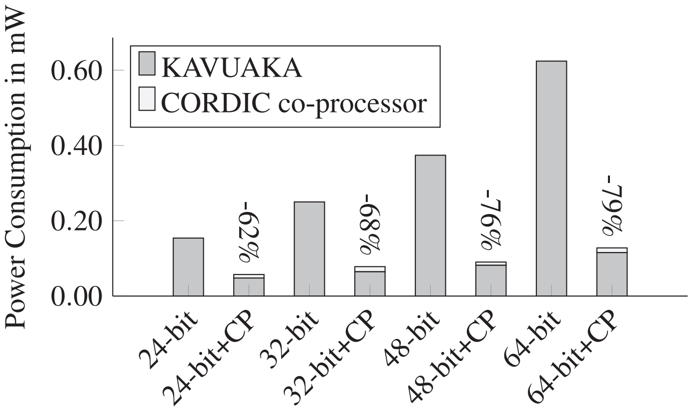


Figure 5.11: Total power consumption with and without a co-processor (CP).

It features a 32-bit reduced instruction-set computer (RISC) architecture. This base instruction set architecture (ISA) is extended by the instruction set extensions (co-processors) called *HiFi4*, *Fusion F1*, *Fusion G3* and *Fusion G6*. The *C6748* is a 32-bit fixed- and floating-point DSP for a wide range of low-power applications. The architectural features of the related processors are listed in Table 5.4. The number of load and store units is important for the processing performance, because the RFs of the processors can not hold all data for processing the beamforming algorithms. The required digital filter processing requires MAC operations. To a certain extent, the processing of the beamforming algorithms can be parallelized with SIMD instructions and a VLIW architecture. The *Tensilica* processors have two separated RFs, one for the *LX7* base processor and one for the co-processors.

Table 5.4: Architectural features of the related and evaluated ASIPs.

	LX7	Fusion F1	HiFi 4	Fusion G3	Fusion G6	TI C6748
Load (bits)	1×32	1×32	2×32	2×128	2×256	4×32
Store (bits)	1×32	1×32	1×32	1×128	1×256	4×32
Instructions per Cycle*	1	2	4	4	4	8
MACs per Cycle*	0	1	4	4	8	2
Issue Slots	1	2	4	4	4	8
SIMD subwords*	0	0	2	4	8	4
Pipeline Stages	5	5	5	7	7	11
General-Purpose Register	64	64	64	64	64	64
CP Register	0	24×32-bit	32×32-bit	32×128-bit	32×256-bit	0

* (32-bit operands)

The beamforming algorithms were implemented on all processors according to the proposed framework presented in [273]. Application- and processor-specific fixed-point C-code is used. The number of processing cycles required and the minimum required clock frequencies for a sampling frequency of 16 kHz for each beamforming algorithm is listed for each associated processor in Table 5.5. The *LX7* processor, without a hardware multiplier and only one operation per cycle, requires significantly more instructions than the *Fusion F1* with two operations per cycle and one hardware multiplier. The *Fusion*, *HiFi* and *C6748* processors do not need significantly fewer cycles for the computation. The additional parallelism of the architectures cannot be used to speed up the computation of the beamforming algorithms. Due to a missing hardware divider of the *C6748* processor, the implementation of the beamformer with adaptive gain requires an iterative conditional subtraction algorithm for the division operation [274], resulting in a comparatively high number of cycles required for this particular beamformer. Although the *KAVUAKA* processor can perform only two instructions and two 32-bit MACs operations per cycle, the required number of processing cycles is lower compared to all other related processors. The reason for this is the indirect addressing mode [83], the CORDIC co-processor (Section 3.2.3), the short pipeline architecture (Section 3.3.3) and the low-level assembler implementation and optimization.

Table 5.5: Number of required processing cycles and minimum required clock frequencies 16 kHz for beamforming algorithms on different processors.

Processor	Fixed	Adaptive Gain	Adaptive Filter
LX7	536 (8.576 MHz)	1582 (25.312 MHz)	3822 (61.152 MHz)
Fusion F1	61 (0.976 MHz)	325 (5.200 MHz)	385 (6.160 MHz)
HiFi4	40 (0.640 MHz)	292 (4.672 MHz)	254 (4.064 MHz)
HiFi4+mul	40 (0.640 MHz)	245 (3.920 MHz)	254 (4.064 MHz)
Fusion G3	49 (0.784 MHz)	226 (3.636 MHz)	287 (4.592 MHz)
Fusion G6	54 (0.864 MHz)	231 (3.696 MHz)	258 (4.128 MHz)
C6748	54 (0.864 MHz)	1889 (30.224 MHz)	281 (4.496 MHz)
KAVUAKA	24 (0.390 MHz)	103 (1.650 MHz)	167 (2.670 MHz)

The comparison in terms of silicon area and estimated power consumption is shown in Figure 5.12. The power consumption is estimated based on the minimum required clock frequencies listed in Table 5.5. The processors are synthesized with the TSMC 40 nm HVT low-power technology. The *KAVUAKA* processor requires the least amount of silicon area. The power consumption is lower than all other processors. This is a result of the smaller register file (RF) and the presented application-specific optimizations, which lead to an overall less complex processor architecture. Since the capabilities of the *HiFi4*, *Fusion G3* and *Fusion G6* processors cannot be fully utilized with the beamforming algorithms, more silicon area is required and higher average power consumption is caused. The same applies to the *C6748* processor, whose power consumption is estimated with the model presented in [275]. Its average power consumption is in the range between 0.864 mW for the fixed beamformer and 29.51 mW for the beamformer with the adaptive filter.

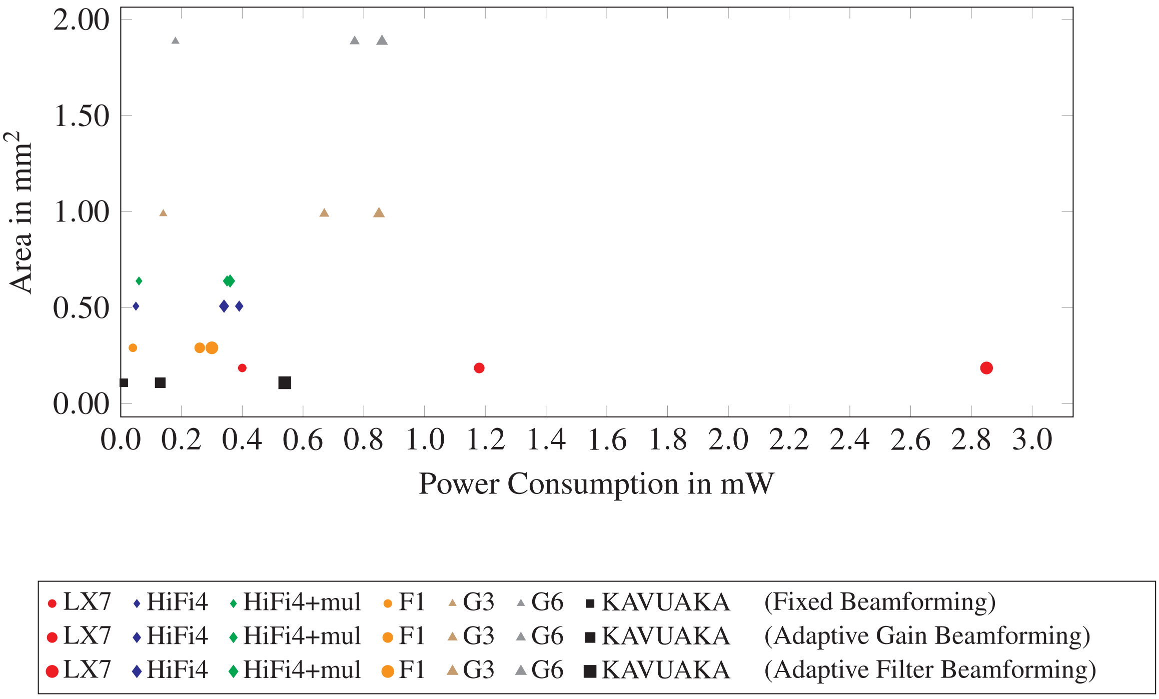


Figure 5.12: Average power consumption compared to the processor area for different beamforming algorithms.

5.2 Case Study: Speech Enhancement

The aim of speech enhancement algorithms is to improve speech intelligibility of degraded speech [60, 276]. These algorithms attempt to reduce or suppress the additive broadband noise component of the speech signal. Noise reduction algorithms are a major component of a hearing aid processing chain as shown in Figure 2.4. One of first proposed and most commonly used noise reduction algorithm is the spectral-subtractive noise reduction. It is based on the assumption that an existing noisy speech signal $Y(\omega)$ is composed of two parts, the pure speech component $X(\omega)$ and an additive noise component $D(\omega)$ (Equation 5.4):

$$Y(\omega) = X(\omega) + D(\omega) \tag{5.4}$$

The noise signal is measured, when no speech is present. The magnitude of the noise spectrum $|D(\omega)|$ noise signal is subtracted from the input signal to obtain an estimated clean signal spectrum (Equation 5.5):

$$\hat{X}(\omega) = [|Y(\omega)| - |\hat{D}(\omega)|]e^{j\phi_y(\omega)} \tag{5.5}$$

The phase of the noisy speech signal $Y(\omega)$ is copied. The complete algorithm is shown in Figure 5.13. The spectrum of the input speech signal is calculated with a windowed fast Fourier transform. This spectrum is used to measure and estimate the noise power, when the signal-to-noise ratio is below a defined threshold. The amplified speech signal is recovered with a IFFT. It is composed of the phase information of the speech input and the calculated magnitude from the difference between the input speech and the estimated noise signal.

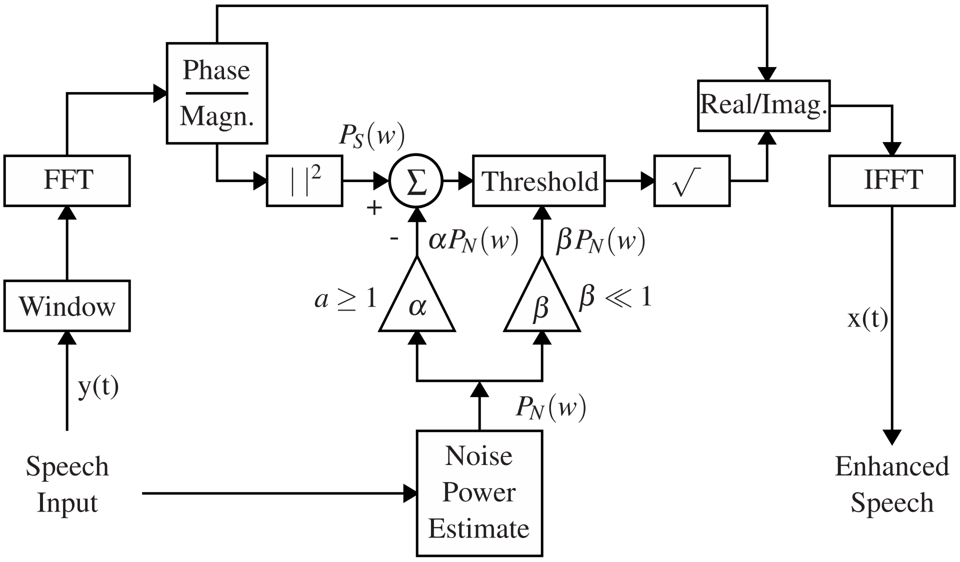


Figure 5.13: Block diagram of the nonlinear spectral subtraction speech enhancement algorithm. [276]

The application-specific hardware and software implementation and optimization of this algorithm is described below. Focus and goal is the evaluation of the hardware architectures presented in Chapter 3. In particular, the performance increase due to specialization and parallelization techniques is examined. The metric for measuring performance is the number of processing cycles required per frame, as described in Section 3.3.3.

The implementation of this noise reduction algorithm is comprised of several subroutines, such as computation of the absolute value (ABS_LOOP), vector summation (VEC_SUM), phase angle (PHASE_ANGLE), sine, cosine and square root computations (SIN_COS_PHASE and

SQRT_PHASE). In order to generate a dense and compacted code, the compiler backend presented in Chapter 4 is used for code generation. The extent to which evolutionary optimization algorithms improve the overall code quality in terms of code compaction and instructions per cycle is evaluated. The code composition in the form of straight line microcode sizes for the mentioned subroutines is analyzed for two different implementations of the noise reduction algorithm. These differ whether the trigonometric and square root function are computed with software CORDIC libraries or with co-processors (Section 3.2.3). As a result, the two implementations use different kernels for the trigonometric and square root function computation. The number of micro-operations (MOs) and straight line microcodes (SLMs) are listed in Table 5.6.

Table 5.6: Number of micro-operations (MOs) and straight line microcodes (SLMs) for two noise reduction algorithm implementations.

Noise reduction algorithm	Number of SLMs	Number of MOs
with software CORDIC libraries	136	1887
with CORDIC co-processors	118	1745

The number of SLMs and MOs differ by 15.2 % and 7.6 %. The distribution in terms of MO count per SLM is also different, which is depicted in Figure 5.14 and Figure 5.15 for the two implementations. The SLM distribution of the implementation with CORDIC co-processors contains more MOs on average. Therefore, less small sized SLMs are part of the implementation. Fewer branches increase the optimization capability of the compiler backend.

The minimum clock frequencies and the compile time for the two implementations are shown in Figure 5.16 and Figure 5.17. The minimum clock frequency varies between 67.0 MHz and 70.6 MHz for the implementation with CORDIC software libraries and between 12.4 MHz and 14.3 MHz for the implementation with CORDIC co-processors. In addition to the instruction scheduling optimization level, the automatic operating merging optimization (X2), which is described in Section 4.1, is used. Furthermore, the influence on the performance is also depicted for both implementations. The optimization level for the instruction scheduling ($o2 - o16$, Section 4.2) and the operation merging ($x0 - x2$, Section 4.1) is varied. Optimization levels below $o2$ fail to schedule the instructions, due to the conditional set and read instruction scheduling (Section 4.2.1). As described in Section 3.2.3, the co-processors decrease the required cycles per computation. Therefore, the minimum required frequency is decreased for this implementation, as the noise reduction algorithm contains multiple of these operations (Figure 5.13). In order to use more than one co-processor in parallel, as proposed in Section 3.5, software (SW) pipelining is applied for the computation of the angle of a complex number ($z = x + iy$) with the arctangent computation of x/y. The performance improvement with software pipelining is shown in Figure 5.17. The minimum required frequency is even further decreased due to greater compiler optimizations. The larger SLM sizes of the imple-

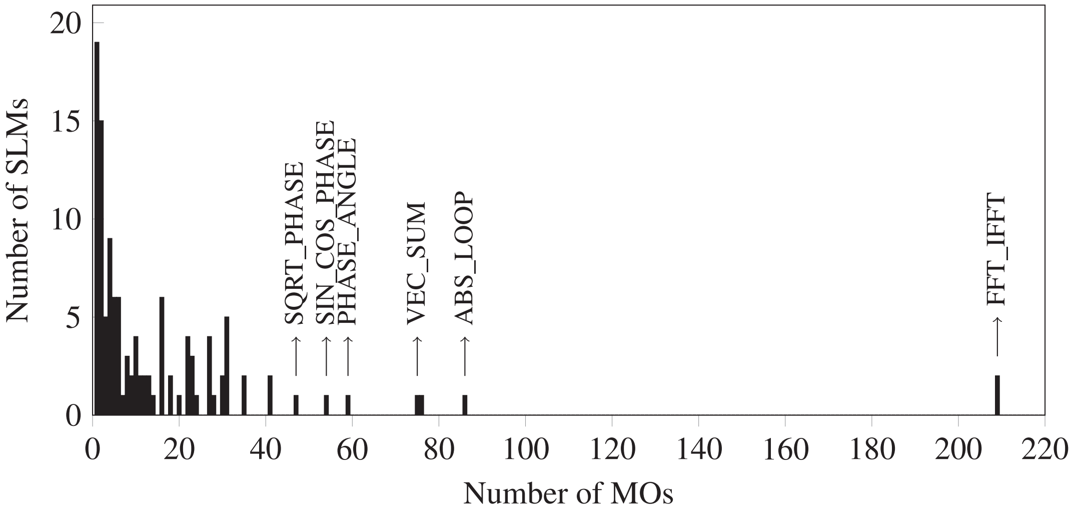


Figure 5.14: Number of straight line microcodes (SLMs) and micro-operations (MOs) for the noise reduction algorithm. Hyperbolic and trigonometric functions are computed with CORDIC software libraries in this implementation.

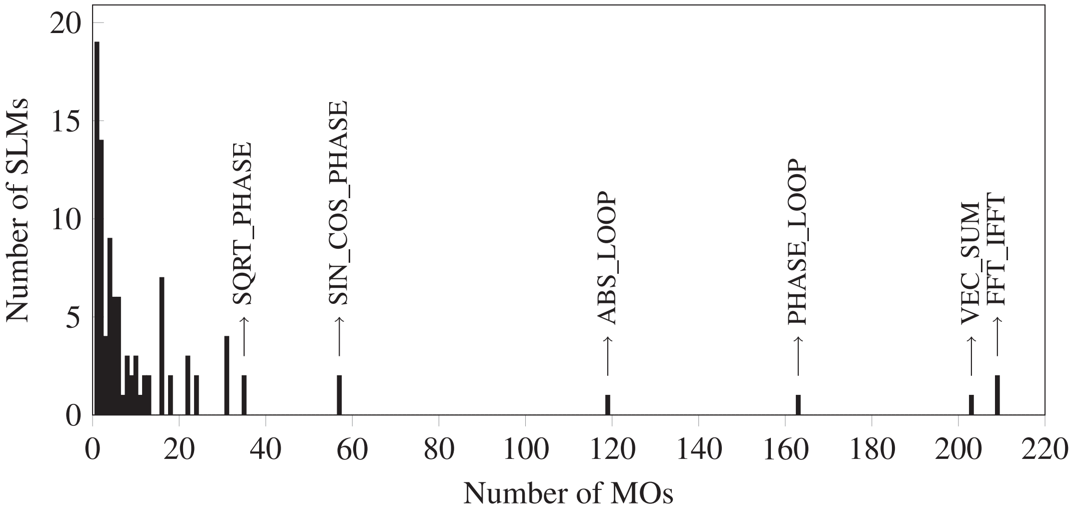


Figure 5.15: Number of straight line microcodes (SLMs) and micro-operations (MOs) for the noise reduction algorithm. Hyperbolic and trigonometric functions are computed with CORDIC co-processors in this implementation.

mentation with hardware co-processors lead to more merged X2 operations. The processing performance is increased by about 13.6 % compared to 2.9 % in case of software CORDIC libraries implementation. Optimization levels below *o*9 fail to schedule the instructions, due to the conditional set and read instruction scheduling (Section 4.2.1).

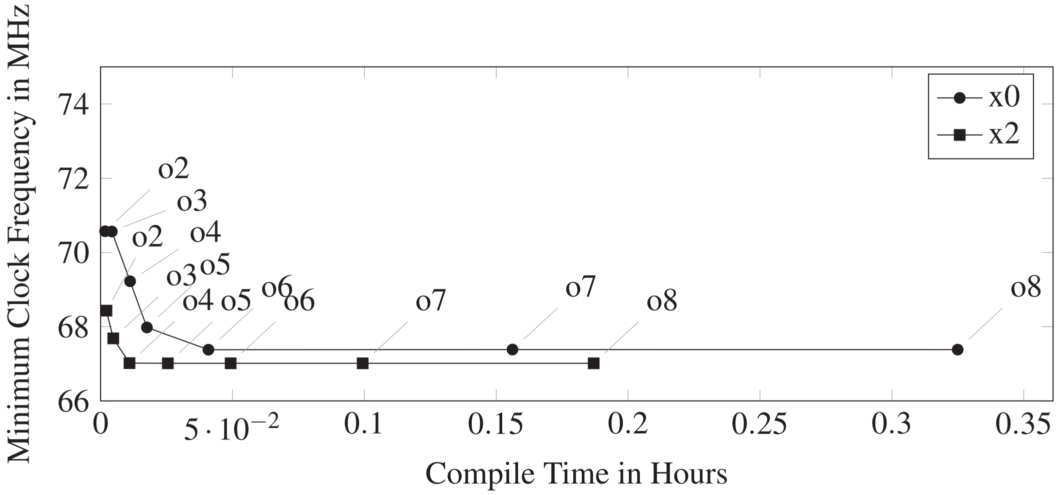


Figure 5.16: Compile time versus minimum required clock frequencies for the noise reduction algorithm for different optimization levels ($o \leq 8$). Hyperbolic and trigonometric functions are computed with CORDIC software libraries.

The minimum required operating frequency of 12.54 MHz is required with the *KAVUAKA* processor for real-time processing with a frame size of 128 samples at a sampling frequency of 16 kHz. The same algorithm is implemented on a *TMS320VC5509A* fixed-point DSP in [277]. The DSP includes two MAC units, two ALUs and three read and two write buses. The minimum required operating frequency for this general purpose DSP is 95.4 MHz with a frame size of 128 samples at a sampling frequency of 16 kHz. This processing performance is comparable to the *KAVUAKA* processor without a HW co-processor (Figure 5.16). The estimated average power consumption is 26.1 mW based on [278].

5.3 Case Study: Speaker Localization

In this section, the evaluation of the sound source localization algorithm originally proposed in [279, 280] is presented. This localization algorithm uses two binaural cues to estimate the azimuth of sound sources. These cues are interaural level difference (ILD) and interaural time difference (ITD) features, which are motivated by human auditory processing. The azimuth-dependent patterns of these cues are used by a probabilistic model (gaussian mixture

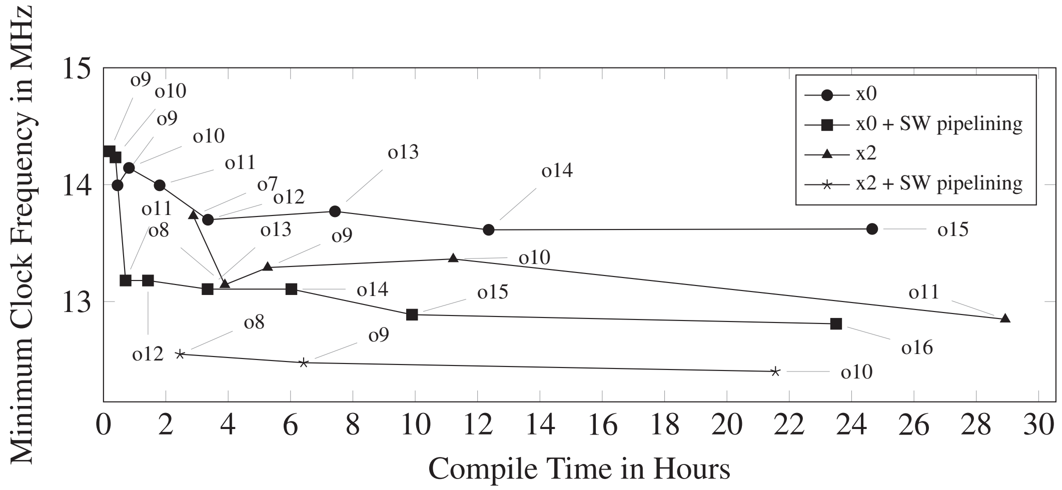


Figure 5.17: Compile time versus minimum required clock frequencies for the noise reduction algorithm for different optimization ($o \leq 16$) and merging levels ($x \leq 2$). Hyperbolic and trigonometric functions are computed with CORDIC co-processors.

model (GMM)) for sound source localization. A block diagram of the signal flow is shown in Figure 5.18.

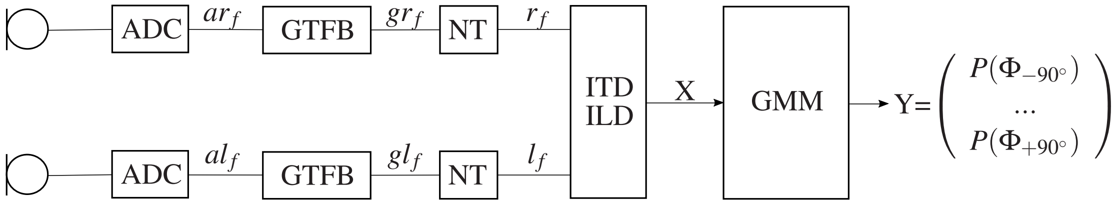


Figure 5.18: Signal flow diagram of the GMM-based binaural localization algorithm.

The binaural cues are extracted from a 16 ms window of $B = 256$ audio samples. The sampling frequency is 16 kHz. The auditory processing of the human system is modelled with a front end consisting of a gammatone filter bank (GTFB) and a neural transduction computation [279]. The gammatone filter bank with 32 auditory channels f and fourth-order phase-compensated filters models the frequency selectivity of the human cochlea. Channel dependent gains are applied to simulate the middle ear transfer function. Half-wave rectification and square root compression are used for neural transduction (NT) of inner hair cells and are specified in Equation 5.6:

$$r_f(t) = \begin{cases} \sqrt{gr_f(t)}, & gr_f(t) \geq 0 \\ 0, & else \end{cases} \tag{5.6}$$

The interaural time difference (ITD) between the binaural auditory signals is calculated for each channel of the GTFB using a normalized cross-correlation for each frame index t. This correlation is also called interaural cross correlation [281] and is defined by Equation 5.7:

$$C_f(t,\tau) = \frac{\sum_{b=0}^{B-1}\left(l_f(t\cdot B/2-b)-\bar{l}_f\right)\left(r_f(t\cdot B/2-b-\tau)-\bar{r}_f\right)}{\sqrt{\sum_{b=0}^{B-1}\left(l_f(t\cdot B/2-b)-\bar{l}_f\right)^2}\sqrt{\sum_{b=0}^{B-1}\left(r_f(t\cdot B/2-b-\tau)-\bar{r}_f\right)^2}} \tag{5.7}$$

where l_f and r_f are the output signals of the neural transduction (NT) processing and $\bar{l}_f$ and $\bar{r}_f$ are the mean values of these signals, respectively. τ is the time lag, whose range is determined by the maximum expected time difference of -1 ms to 1 ms. The maximum of this cross-correlation for every τ is given in samples (Equation 5.8):

$$\hat{\tau}_f(t) = \arg\max_{\tau} C_f(t,\tau) \tag{5.8}$$

The resolution of the ITD value is limited by the sampling frequency f_s, e.g., for a sampling frequency of 16 kHz the resolution is limited to 7.321° due to the delay of two successive audio samples [280]. To improve the resolution, an exponential interpolation can be applied (Equation 5.9) to interpolate the the peak position relative to the estimated integer peak position $\hat{\tau}_f(t)$:

$$\delta_i(t) = \frac{\log C_f(t,\hat{\tau}_f(t)+1) - \log C_f(t,\hat{\tau}_f(t)-1)}{4\log C_f(t,\hat{\tau}_f(t)) - 2\log C_f(t,\hat{\tau}_f(t)-1) - 2\log C_f(t,\hat{\tau}_f(t)+1)} \tag{5.9}$$

The estimate of the ITD in seconds is than computed by combining the integer and fractional value (Equation 5.10):

$$\widehat{ITD}_i(t) = \frac{\hat{\tau}_f(t)+\delta_i(t)}{f_s} \tag{5.10}$$

The second feature that is used is the ILD. This feature is computed for every channel i of the gammatone filter bank by comparing the integrated energy over the frame of length B of the two signals (Equation 5.11):

$$\widehat{ILD}_i(t) = 20\cdot\log_{10}\left(\frac{\sum_{b=0}^{B-1} r_i(t\cdot B/2-b)^2}{\sum_{b=0}^{B-1} l_i(t\cdot B/2-b)^2}\right) \tag{5.11}$$

Therefore, the binaural feature space is two dimensional (Equation 5.12):

$$X_f = \{(\widehat{ILD}_f(1),\widehat{ITD}_f(1)),\ldots,(\widehat{ILD}_f(T),\widehat{ITD}_f(T))\} \tag{5.12}$$

where T represents the number of observations for every gammatone channel.

Gaussian mixture models (GMMs) are used to recognize the azimuth-dependent pattern of the binaural cues by computing the directional probability distribution of the two-dimensional feature space consisting of ITD and ILD features. The GMM is computed for each sound source direction λ with the weighted feature vector $\vec{x}$ (Equation 5.13):

$$p(\vec{x}|\lambda) = \sum_{j=1}^{K} w_j p_j(\vec{x}) \tag{5.13}$$

where w_j are the trained weights for the K Gaussian components. The Gaussian function is given by (Equation 5.14):

$$p_j(\vec{x}) = \frac{1}{(2\pi)^{D/2}|\Sigma_j|^{1/2}} \exp\left[-\frac{1}{2}(\vec{x}-\vec{\mu}_j)^t \Sigma_j^{-1}(\vec{x}-\vec{\mu}_j)\right] \tag{5.14}$$

with the mean value $\vec{\mu}_j$ and the covariance matrix Σ_j for each mixture component j. The estimated sound source direction ϕ is determined by the maximum of the log-likelihood of the current observation $\vec{x}_{f,t}$ for the gammatone channels F (Equation 5.15):

$$\hat{\phi}(t) = \underset{1\leq k\leq K}{\arg\max} \underbrace{\sum_{f=1}^{F} \underbrace{\log p(\vec{x_{f,t}}|\lambda_{f,\varphi_k})}_{\text{log likelihood}}}_{\text{across frequency integration}} \tag{5.15}$$

The directional estimation of the sound source is calculated for each frame of the input signals. The resolution of the model is 5° in the range of −90° to 90°. This results in 37 spatial positions, each classified by 15 Gaussian components. For all 32 filter bands this results in a model with $32 * 37 * 15 = 17,760$ components. For the two-dimensional case, each of the Gaussian components λ_φ is described with the five parameters $w_j, \mu_{j,1}, \Sigma_{j,1}, \mu_{j,2}, \Sigma_{j,2}$ according to the Equation 5.16. This results in $17760 \times 5 = 88,800$ trained parameters.

$$\lambda_\varphi = (w_j, \mu_{j,1}, \Sigma_{j,1}, \mu_{j,2}, \Sigma_{j,2}) \qquad \forall j = 1, \cdots \tag{5.16}$$

5.3.1 Implementation and Application-Specific Optimizations

The implementation and evaluation of the localization algorithm on the *KAVUAKA* processor is presented in this section. To achieve real-time processing with a clock frequency below 50 MHz, several of the proposed application-specific hardware and software features (Chapter 3) are applied and evaluated. The total number of cycles per audio frame of 256 samples for each optimization step is shown in Figure 5.19. Each optimization step is described in this section. Starting from the initial baseline implementation with a required minimum clock frequency of 524.15 MHz, a real-time processing with a clock frequency of 30.08 MHz is possible with all optimizations.

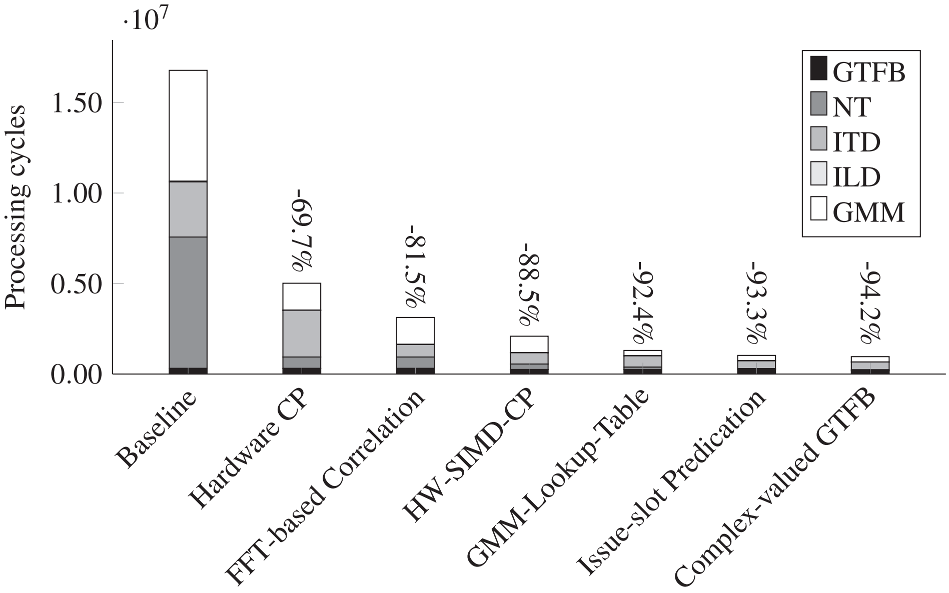


Figure 5.19: Required processing cycles after each optimization phase.

Gammatone filter bank

The feature extraction phase of the localization algorithm [279, 280] includes a gammatone filter bank (GTFB) with 32 channels for each microphone channel. There are different GTFB implementations presented in the literature [221, 279, 282–285]. These are approximations of the analog gammatone filters originally presented in [286] and differ in their implementations. There are impulse- or frequency-invariant implementations with multi-zero, one-zero or all-pole with complex or real filter output [221]. The frequency responses of a nonlinear one-zero and all-pole gammatone filter [283], the linear with complex-valued coefficients and complex-conjugated pole pairs [279] and the linear all-pole gammatone filter with complex output [221] are shown in Figure 5.20. In this case study, the performance of these GTFB implementations is compared using application-specific hardware extensions.

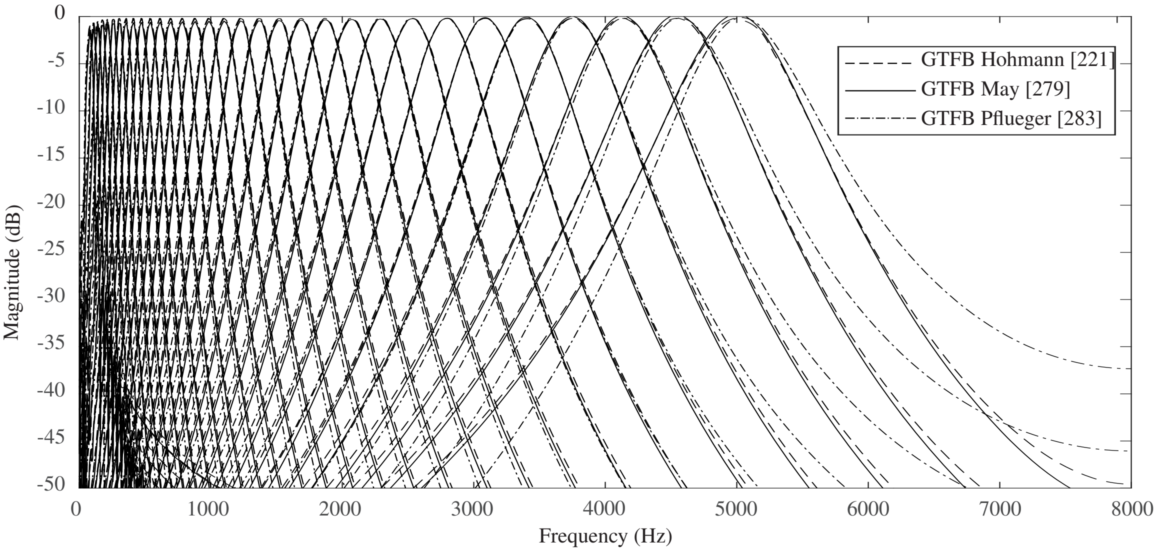


Figure 5.20: Frequency response of the gammatone filter banks (GTFBs) presented in [221, 279, 283].

Nonlinear One-Zero and All-Pole Gammatone Filter

The nonlinear one-zero and all-pole gammatone filter is proposed in [283]. The implementation of a single channel is shown in Figure 5.21. Each gammatone filter consists of four second-order IIR filters. The coefficients b_1 and $a_{1,1},\ldots,a_{2,4}$, the input $x(b)$ and the output $y(b)$ are real-valued. The gammatone filter is implemented on the *KAVUAKA* processor with multiply-accumulate (MAC) instructions and hardware indirect circular addressing modes [83, 280].

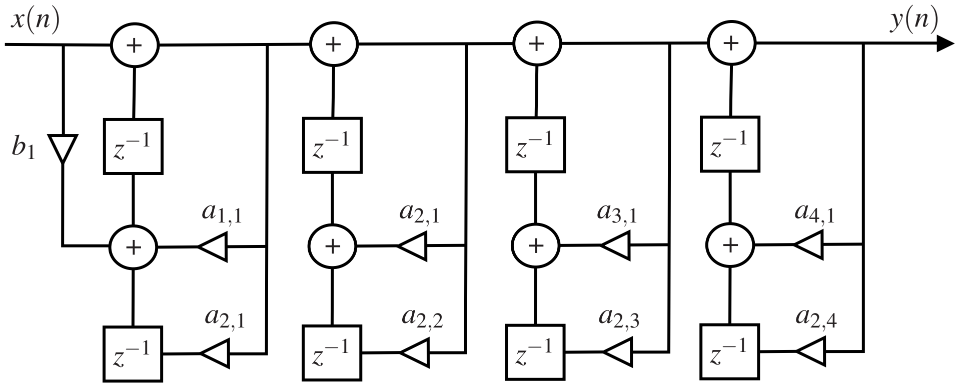


Figure 5.21: Nonlinear all-pole and one-zero gammatone filter [283].

Linear All-Pole Gammatone Filter with Complex-Valued Output

The linear all-pole design of the 4th-order linear gammatone filter is presented in [221]. The implementation of the filter bank is shown in Figure 5.22. The analyzer and the synthesizer stage for K channels are shown. The filter coefficients $a_0, \ldots, d_3$, and the phase and gain factors $n_0, \ldots, m_K$ are complex-valued. For the gammatone calculation, complex-valued multiplications, additions and subtractions are required. The complex-valued multiply-accumulate (CMAC) functional unit described in Section 3.2.1 is used for these computations. Additionally, the circular indirect address calculation is used for the filter implementation [83].

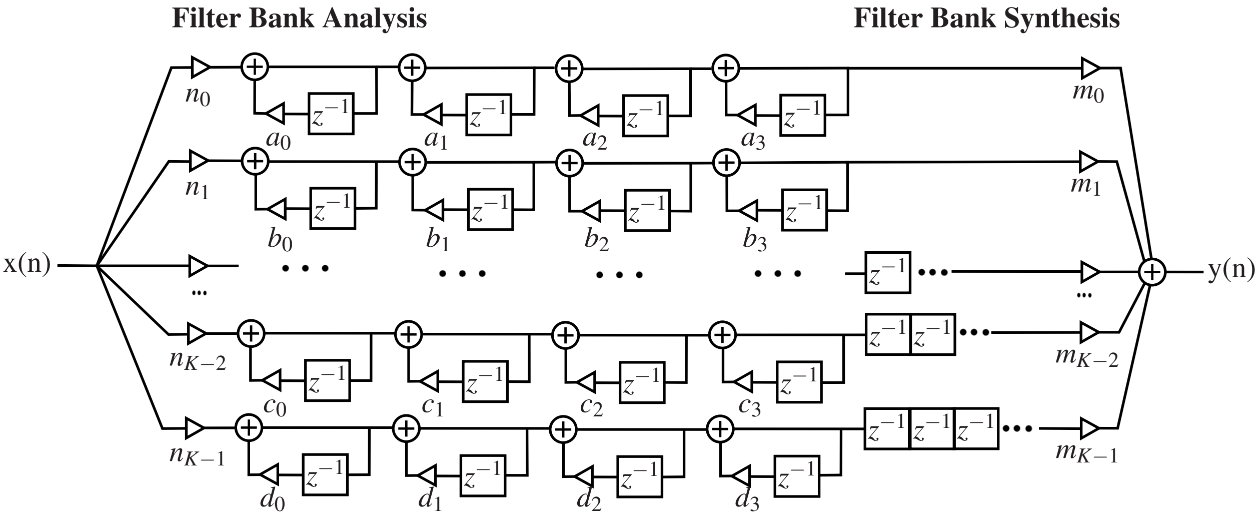


Figure 5.22: An impulse-invariant, all-pole gammatone filter with complex output [221]. The normalized inputs $x(b) \cdot n_k$ are filtered by cascading first-order complex band pass filters and are synthesized using a delay line with phase and gain correction factors.

Fast Fourier Transform

A short-time frequency analysis method such as a windowed fast Fourier transform (FFT) is also suitable for use as a filter bank in hearing aids [221]. However, the bands in the frequency domain are linearly distributed, whereas the bands in the human auditory system are distributed almost logarithmically. In addition, the audio delay for FFT-based processing is higher compared to the GTFB-based processing when the same resolution is applied in the lower bands. An advantage of the FFT implementation is the lower processing performance requirement. The implementation of the FFT on the *KAVUAKA* processor with the complex-valued multiply-accumulate (CMAC) functional unit (FU) is described in Section 3.2.1.

Comparison of the GTFB Implementations

Table 5.7 shows a comparison between the GTFB filter bank implementations. The following parameter set is used for the comparison:

1. Center frequencies of the gammatone filters: 1000 Hz
2. Number of filters used for the filter bank: 32
3. Lower cutoff frequency: 80 Hz
4. Upper cutoff frequency: 5000 Hz

The FFT-based implementation is added for comparison. 256 bins are used. The results show that the processing performance and the audio delay is significantly different based on the implementations. The lowest number of processing cycles is required for the FFT-based implementation. However, the resulting delay is the greatest. The delay of GTFB implementations depends on the filter order and the number of cascaded filters. The linear all-pole gammatone filter with complex output [221] needs less cycles due to the complex-valued multiply-accumulate (CMAC) functional unit of the *KAVUAKA* processor, which calculates the required complex-valued filter operations within a single cycle. Therefore, fewer cycles are required compared to the higher order nonlinear one-zero and all-pole gammatone filter implementation [283].

Table 5.7: Comparison of gammatone filter bank (GTFB) and FFT implementations on the *KAVUAKA* processor.

Gammatone filter bank	Nonlinear one-zero and all-pole [283]	Linear all-pole with complex output [221]	FFT-based [221]
Cycles per sample	981	362	7.48
IPC	1.92	2.33	2.17
Audio delay	10.70 ms	3.75 ms	16.00 ms

Neural Transduction

The most processing demanding operation of the neural transduction is the square root compression. The square root operation is computed with the software CORDIC libraries for the baseline implementation (Figure 5.19). In order to speed up the computation, two CORDIC co-processors (CPs) with SIMD and four CORDIC modules are used (Section 3.2.3) [280]. The achieved speedup is around 55.8.

Interaural Level and Time Difference

The ITD and ILD computation requires around 18.3 % of the total cycles of the baseline implementation (Figure 5.19). The number of cycles is decreased by computing the cross-correlation in the frequency-domain based on the convolution theorem. The convolution of the signals is replaced by the pointwise product of their Fourier transforms. The proposed complex-valued MAC (Section 3.2.1) is used to compute a complex-valued FFT. The fast Fourier transform of both input signals are computed with one FFT computation with 512 bins [280]. The speed up compared to the baseline implementation with a time-domain convolution is 4.87. The computation of the ILD (Table 5.11) is by far less demanding and is excluded from further optimizations.

Gaussian Mixture Model

Although the required processing time for the computation of the gaussian mixture model (GMM) is reduced by a factor of 6.72 with SIMD CORDIC co-processors. However, the GMM computation still requires around 43 % of the total processing time. To further improve the speed, a LUT-based log-add algorithm is used [280, 287]. With this algorithm, the computation of exponential function (Equation 5.14) is replaced with summation of logarithmic terms. In this implementation, the table size is set to 2^{14} 16-bit elements, resulting in a maximum accuracy of 2^{-14}. The required number of cycles for the computation of the GMM are reduced from 910k to 296k cycles. However, 2^{12} additional 64-bit memory words are required for the LUT.

5.3.2 Processing Performance Comparison to a Related Architecture

The same localization algorithm is evaluated on the transport triggered architecture (*TTA*) in [288]. The *TTA* is a customized C programmable processor design, which supports register file bypassing and offers a scalable instruction level parallelism. By extending the application-specific instruction set architecture (ISA), a speed-up of 151 × compared to the baseline general purpose implementation of the processor is achieved. SIMD instructions are implemented with 32 subwords for each gammatone channel, resulting in 1024-bit vector operations. Among these vector operations are *ABS, ADD, AND, EQ, GT, LT, MAX, MIN, OR, SHL, SHR, SUB, and XOR*, which are added to the architecture as custom operations. Specialized operations are added to speed up the computation of the CORDIC algorithm, i.e., an element-wise conditional addition and subtraction instruction, a count leading zeros instruction and a combined multiply shift operation. Two equally sized register files with 16 1024-bit registers and one write port and up to two read ports are part of the proposed *TTA* architecture. The data memory is connected with a dedicated load and store unit.

The comparison between the *TTA* and *KAVUAKA* 64-bit implementation is shown in Table 5.8. The required core area and minimum clock frequencies are listed. The *TTA* processor requires around 3.94 × more silicon area than the *KAVUAKA* 64-bit processor. The vector register file requires 43.2 % of the core cell area, compared to 40 % in case of the *KAVUAKA* processor (Section 3.3.2). The higher area requirement is based on the datapath architecture of the *TTA*, which is by far more complex compared to the VLIW architecture of the *KAVUAKA* processor. Due to the lower operating clock frequency of the *KAVUAKA* processor and the smaller chip size, the estimated average power consumption is almost nine times lower.

Table 5.8: Comparison of processor core area, minimum required clock frequencies, and the estimated average power consumption. Results from an ASIC synthesis with the TSMC 40 nm HVT low-power ASIC technology at 50 MHz.

	TTA	*KAVUAKA* 64-bit
Core cell area in	743,016 μm^2	188,511 μm^2
Minimum clock frequency	47 MHz	30.08 MHz
Estimated Average Power Consumption	11.9 mW	1.35 mW

5.4 Case Study: Speech Recognition

In this section, the application of DNN-based performance measures for predicting error rates in automatic speech recognition on hearing aid processors is discussed [24, 289, 290]. The goal is to predict the word error rate in automatic speech recognition and to use it for the optimization of hearing aid parameters, such as the beamforming angle in a spatial scene. In order to evaluate the feasibility of this algorithm for hearing aids, the hardware processing performance and the processing delay are evaluated. The processing performance is determined in cycle counts per frame. A suitable clock frequency of 50 MHz was selected for the hearing aid processor to meet the real-time processing requirements, taking into account the resulting constraints on power consumption, silicon area and processing delay [8, 104].

The features for the DNN-based performance measures are 40-dimensional log-Mel-spectral coefficients. Ten frames of these features are fed to the input layer of the DNN for temporal information. The frame size in this case is 10 ms and the sampling frequency is 16 kHz. The feature extraction includes fast Fourier transforms (FFTs), logarithmic functions and discrete cosine transformations. The computation of feature extraction requires $32,958$ cycles on the *KAVUAKA* 64-bit processor for one frame. At a processor clock frequency (f_C) of 50 MHz this part requires a processing time (T_P) of 0.659 ms (Equation 5.17):

$$T_P = \frac{cycles}{f_C} \tag{5.17}$$

The *KAVUAKA* 64-bit processor is equipped with specialized functional hardware units, like a complex-valued MAC unit described in Section 3.2.1 for speeding up the computation of a fast Fourier transform. The requirements for the computing performance of the feature extraction part are low compared to the processing of the forward path of the DNN. The technical feasibility depends on the computational complexity of the used DNNs. The complexity is determined by the number of parameters of the acoustic model. Therefore 20 different feed-forward networks are trained and evaluated, i.e., each combination of two to six hidden layers (HL) with 256, 512, 1024 and 2048 hidden units (HU). An extrapolated estimate of the number of processing cycles required for a forward path for each evaluated DNN is given in Figure 5.23. These results are based on a hand optimized implementation on the processor architecture shown in Figure 3.75.

The processing of the forward path of the network are mainly matrix multiplications. The activation functions are sigmoid for the hidden units and softmax for the output layer. Arithmetic operations such as exponential and division operations are required to compute the activation functions. The required cycle counts for these operations are considered by summing the processing cycle counts when using the coordinate rotation digital computer (CORDIC) hardware accelerator (Section 3.2.3) as suggested in [291]. Despite the high number of 32 cycles per operation for exponential and division computation, the required number of cycles for the forward path with matrix multiplications and additions is far greater. Therefore, the computation of the forward path dominates the total cycle count. Even if the exponential and division operations could be computed within one cycle, the total number of cycles would only be reduced by 1.8 % on average. The forward path matrix multiplications are calculated as follows:

$$\begin{pmatrix} v_1 \\ \vdots \\ v_m \end{pmatrix} = \begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{pmatrix} \times \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} + \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \tag{5.18}$$

where v and u represent the values of the neurons, w is the weight matrix and b is the bias vector. The processing complexity in terms of number of required operations depends on the parameters m and n.

To meet the performance requirements for real-time processing, the required processing cycles must be processed at a given clock frequency (f_C) of 50 MHz. To achieve this, the available specialized hardware units, instruction-level parallelism and data-level parallelism of the

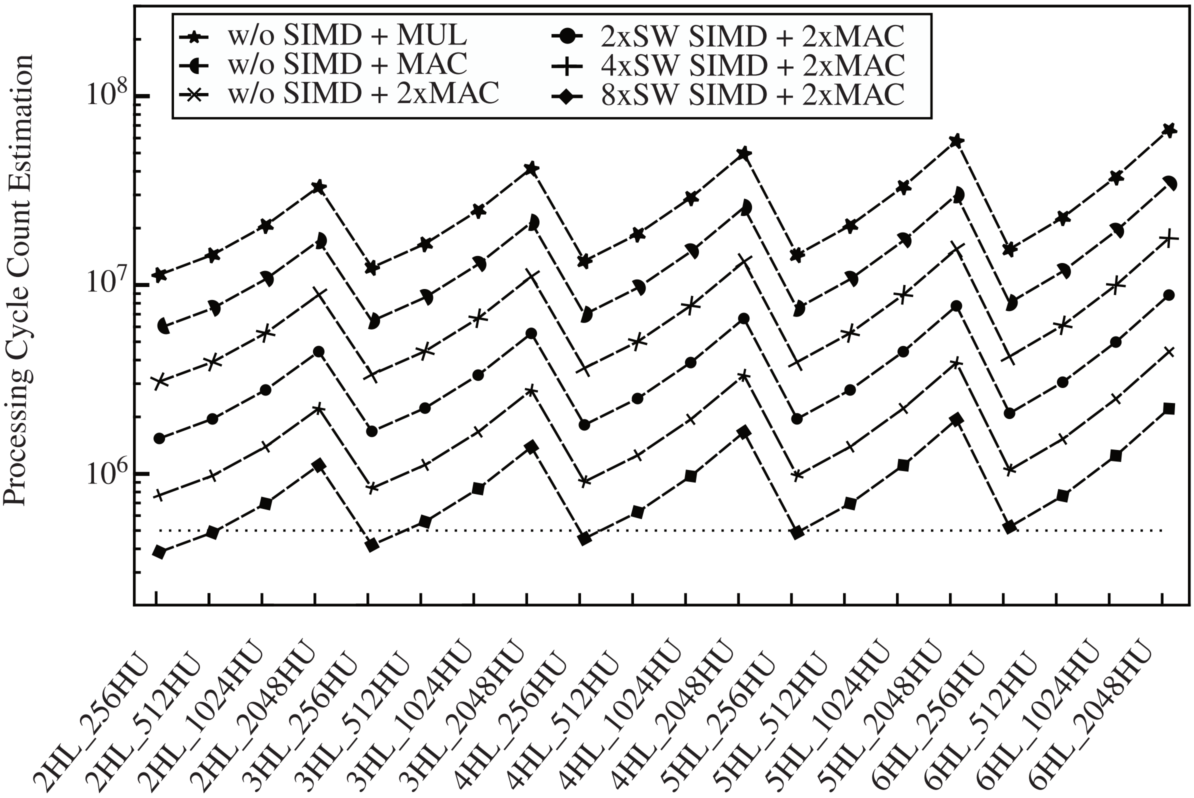


Figure 5.23: Extrapolated estimate of the required cycle count for the forward path for all DNNs with 256 up to 1024 hidden units (HU) and two to six hidden layers (HL). The number of cycles are estimated for processor architecture with and without single instruction, multiple data on two to eight data points or subwords, and either a MUL or multiple MAC units. The black dashed line represents the maximum cycle count for online processing with 50 MHz.

processor architecture are exploited. A commonly used hardware unit for neural network acceleration is the multiply-accumulate (MAC) unit [7, 25, 100], which performs a product of two numbers and adds the result to the accumulator. This hardware unit can be used directly for these matrix multiplications (Section 3.2.1). Instruction level parallelism is typically exploited by using a very long instruction word instruction set architecture (ISA) [8, 83, 133]. This enables the execution of two or more instructions in parallel per cycle. As an example, two MAC units can be used by processor architectures [7,25,100], which execute two or more instructions in parallel. A well-known and applied data level parallelism mechanism is single instruction, multiple data (SIMD), which is applicable in hearing aids [75, 207]. Typical SIMD mechanisms perform the same operation in parallel on multiple input data organized in various subwords (SW) of equal size (2xSW, 4xSW, . . .).

Due to low data or control dependencies in the calculation of the DNN forward path, the computation for the *KAVUAKA* architecture can be almost completely parallelized on the instruction and data level. The scheduled code of the inner loop of the handwritten assembler code for the calculation of the matrix multiplications defined in Equation 5.18 is shown in Figure 5.24. The *KAVUAKA* 64-bit processor architecture supports instruction level parallelism, by processing two instructions per cycle on the issue-slot 0 and issue-slot 1. The specialized MAC operations (*MAC_32* instructions) are SIMD instructions, processing two subwords (SW) with 32-bit each in parallel. These *MAC_32* instructions are scheduled in parallel to memory accesses (*move (MV)* instructions). The *MAC_32* instruction writes the result in two target registers, which together have twice the data width, to avoid overflows during fixed-point operations. Loop unrolling was used to achieve 1.88 instructions per cycle (IPC) on average. As a result, ten vector elements can be multiplied and added in nine cycles.

Using the model *2HL_512HU* with the best performance regarding the prediction error [24] and the model with the least number of parameters *2HL_256HU*, the number of required cycles can be reduced by a factor of about 30 (Figure 5.23) with the mentioned hardware architecture extensions compared to the sequential implementation without SIMD instructions and a single multiplication instruction (*w/o SIMD + MUL*). Based on these results, the lowest achievable processing time (Equation 5.17) for the complete DNN is 7.69 ms with 384634 cycles and a clock frequency of 50 MHz.

According to the M-measure [289], the temporal distance between two propagated frames separated by 100 ms can be calculated as soon as the latter frame is processed. The M-measure calculation includes several independent division and logarithmic operations. If these operations were also calculated with the CORDIC hardware accelerator within 32 cycles, 88000 estimated cycles are required in addition to the forward path of the second frame. If two hardware accelerators (Section 3.2.3) are used in parallel, the processing time is about 0.88 ms at a processor/co-processor clock rate of 50 MHz.

The estimated total processing time of this algorithm depends on the temporal distance of the frames to be averaged. In order to find an optimal observation window, a detailed memory

```
//Issue-slot #0        ; Issue-slot #1
:LOOP
MV      V1R0,   FIR0+  ; MV     V1R1,      FIR1+
MV      V1R2,   FIR0+  ; MV     V1R3,      FIR1+
MV      V1R0,   FIR0+  ; MAC_32 V0R2+V0R3, V1R0,   V1R1
MV      V1R1,   FIR0+  ; MAC_32 V0R2+V0R3, V1R2,   V1R3
MV      V1R2,   FIR1+  ; MV     V1R3,      FIR0+
MV      V1R0,   FIR1+  ; MAC_32 V0R2+V0R3, V1R0,   V1R2
MV      V1R2,   FIR1+  ; MAC_32 V0R2+V0R3, V1R1,   V1R0
LOOPR   V0R1,   LOOP   ; NOP
NOP                    ; MAC_32 V0R2+V0R3, V1R3,   V1R2
```

Figure 5.24: Scheduled assembler code of the inner loop for the matrix multiplications. Two statements are processed in parallel. Up to three registers (*V0R0-V1R31*) are addressed per instruction, the first register being the target register. Move (MV) instructions copy data from main memory using pointers stored in file indirect registers (FIREGs). Multiply-accumulate (MAC) instructions are executed in parallel with the MV instructions. The MAC results are stored in two destination registers to get full precision. The suffix _32 specifies the subword mode for SIMD operations.

analysis of the particular structure, on which the implementation is to be performed, is necessary. However, managing up to 20 frames in memory does not result in a considerable number of processing cycles. Considering a reduced fixed-point data width, the accuracy of the DNN, and the M-measure algorithm performance, the implementation of the M-measure on hearing aid hardware with sufficient parallelization capability is feasible under real-time constraints and the overall audio latency is less than the frame shift of 10 ms.

5.5 Comparison to Other Related Hearing Aids

This section compares hardware characteristics of ASIP-based hearing systems from the literature with the ASIP hearing system presented in this thesis. The characteristics include the ASIC technology (Section 5.5.1), power consumption (Section 5.5.2), circuit area (Section 5.5.3), operating clock frequency (Section 5.5.4), audio datapath width (Section 5.5.5), and memory sizes (Section 5.5.6).

5.5.1 ASIC Technology and Supply Voltage

The advantages of the steadily decreasing feature sizes of CMOS semiconductor technology are exploited in commercial and research hearing aids. The feature sizes of modern hearing aids from 1996 to 2020 are shown in Figure 5.25. Hearing aids with an analog front end

(AFE), including analog-to-digital converters (ADCs), programmable gain amplifiers (PGAs), or digital-to-analog converters (DACs), are marked. These hearing aids are either mixed-signal or analog hearing aid designs, which have on average larger feature sizes due to more restrictive design rules and greater sensitivity to noise [4]. To overcome these limitations, the authors of [7, 56, 72, 73] propose a chip-level integration with two separate chips. Each chip is integrated with a different ASIC technology, to independently utilize the more appropriate feature size for both, the digital and the analog components of the hearing aid. The rate, at which the feature size shrinks, decreased significantly for hearing aid implementations in recent years. This is due to the higher costs for the design and manufacturing with smaller feature sizes [4]. The *KAVUAKA* system-on-chip (SoC) is manufactured with a 40 nm ASIC technology. This value is among the lowest reported in the literature. Only the chip for smart hearables is manufactured with a smaller feature size of 28 nm [7, 12]. The supply voltages of hearing aid implementations are shown in Figure 5.26. Since the feature size remained almost constant over the last years (Figure 5.25), the supply voltage also remains almost constant. This is especially noticeable for hearing aids with analog components (Figure 5.26). The lowest supply voltages of 0.55 V to 0.8 V are used in digital hearing aid designs. Those hearing aid implementations, that employ undervoltage techniques through dynamic voltage scaling and use voltages close to the threshold voltage, are listed in Table 5.9. Dynamic voltage scaling is not used for the *KAVUAKA* processor. Therefore, the supply voltage of 1.10 V is relatively high compared to the recently presented commercial and research hearing aids.

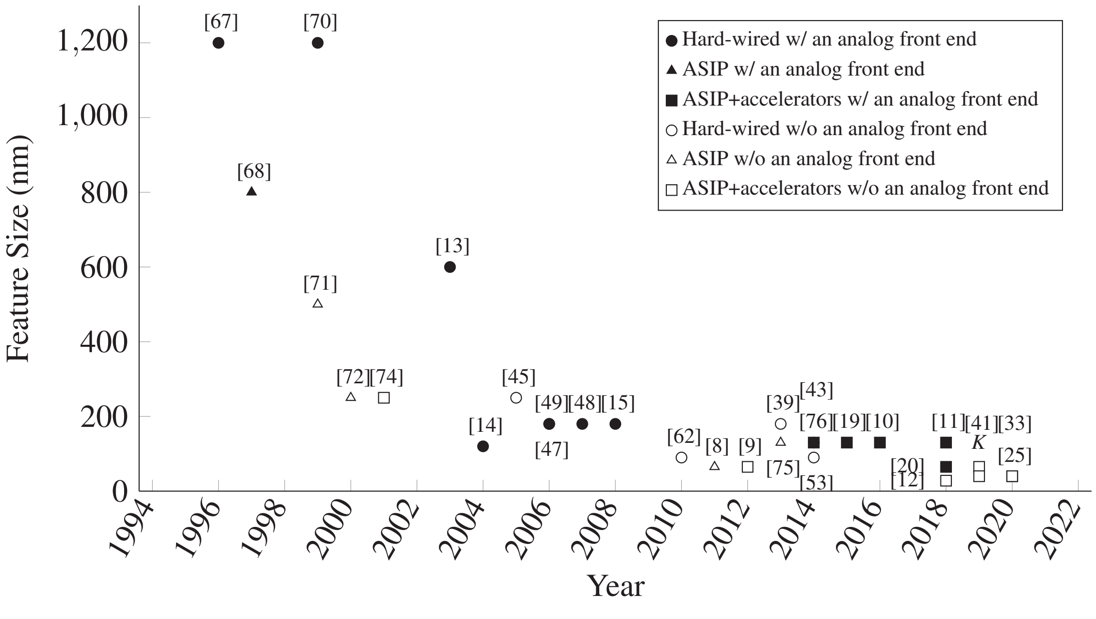


Figure 5.25: Feature sizes of commercial and research hearing aids. *KAVUAKA* is marked with the letter K.

Table 5.9: Operating voltages of commercial and research hearing aids.

Work	Operating Voltage	Default Voltage	ASIC Technology
[8]	0.80 V	1.00 V	65 nm CMOS
[33]	0.70 V	0.90 V	40 nm CMOS
[62]	0.60 V	1.00 V	90 nm CMOS
[53]	0.60 V	1.00 V	90 nm CMOS
[25]	0.60 V	0.90 V	40 nm CMOS
[12]	0.55 V	1.05 V	28 nm CMOS

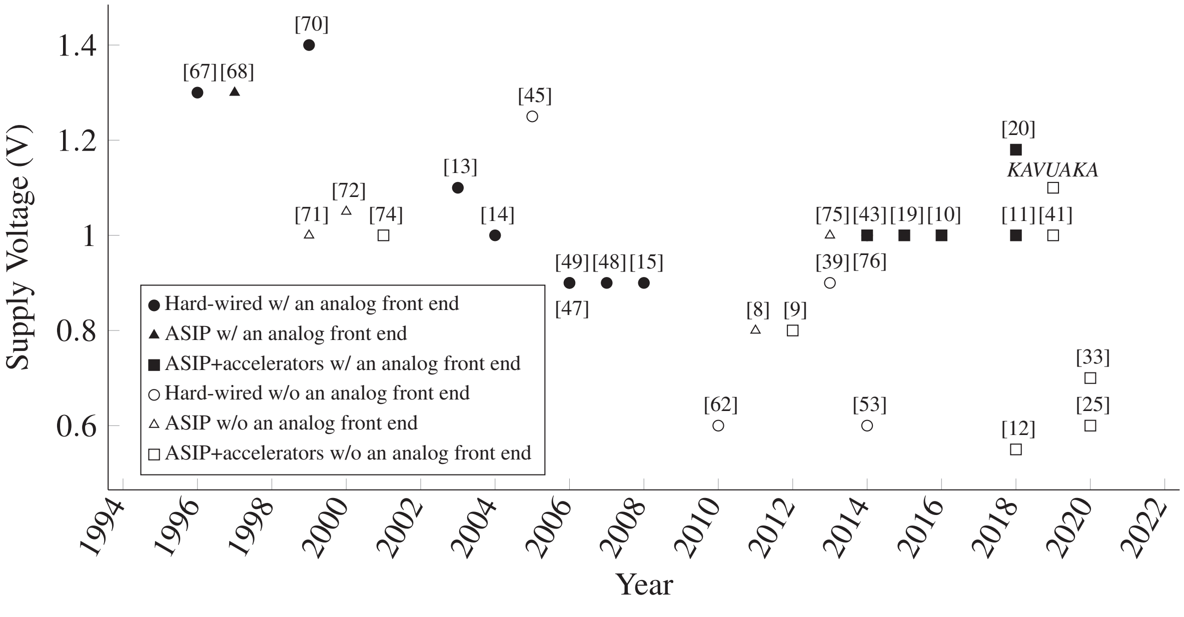


Figure 5.26: Supply voltages of commercial and research hearing aids.

5.5.2 Power Consumption

The average power consumption determines the battery life for the hearing aids. During normal operation, all components of the hearing aid processing system are usually constantly active. The average power consumption for the hearing aid implementations is shown in Figure 5.27. The computational complexity of the algorithms determines, among other things, the power consumption. The lowest achieved average power consumption for the given implementations is 10 µW. The hearing aids [49] and [74] consume this power for an adaptive signal-to-noise ratio (SNR) monitor based on an envelope detection and adaptive FIR and IIR filter calculations. On the other hand, when targeting hearables or smart headphones instead of hearing aid devices, deep-learning based noise reduction techniques require an average power consumption up to 4 mW [12]. Hard-wired architectures offer a comparatively low-power consumption compared to the ASIP architectures. The power distribution for the components of the mixed-signal hearing aid [11] is 36 % for the analog front end, 39 % for the digital signal processor (DSP), 11 % for the power on reset circuit and 13 % for the remaining components. The digital signal processor of the hearing aid presented in [19], on the other hand, consumes up to 71 %, while the analog parts consume the remaining 29 %. The comparatively low average power consumption of 0.6 mW of the *KAVUAKA* 64-bit processor is measured during the computation of adaptive beamforming algorithms [207].

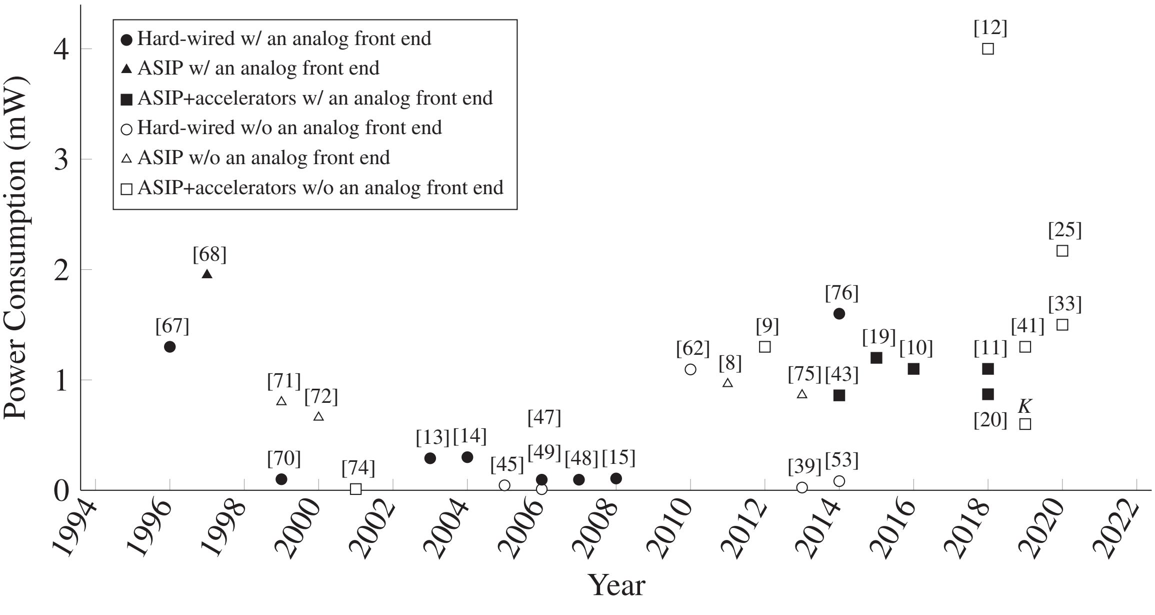


Figure 5.27: Power consumption of commercial and research hearing aids. *KAVUAKA* is marked with the letter K.

5.5.3 Circuit Area

The silicon area for each hearing aid is shown in Figure 5.28. The analog front end or wireless connection modules, which are not part of every hearing aid, require additional silicon area, which must be considered when comparing implementations. The area distribution for the mixed-signal hearing aid, which is presented in [19, 43], is 30 % for the analog and 70 % for the digital part. The digital part consists of a 24-bit application-specific instruction-set processor and five dedicated accelerators. The analog part consists of an audio front end with a programmable gain amplifier (PGA), an analog-to-digital converter (ADC) and a class-D amplifier for the pulse density modulation (PDM) output. The total size is 9.50 mm^2 and this is the maximum chip size since 2004. The analog hearing aid presented in [76], which is manufactured using a 0.13 µm and a 0.35 µm technology, requires 66 % of the area for the AGC, 15 % for the driver and 20 % for the filter circuit. The wireless control part of the analog hearing aid, which presented in [67], is based on a dual tone multi frequency (DTMF) receiver, occupies 1.16 × 4.6 mm, which is 16 % of the total chip size of 5.7 × 4.9 mm. The silicon area of a hearing aid may be pad limited. As a result, the total area is larger than effectively required for the digital or analog core parts. This is the case for the second largest ASIP-based hearing aid system in this comparison, which does not include an analog front end [72]. Its size is 20 mm^2. Although the *KAVUAKA* hearing aid system-on-chip contains four ASIPs on a single chip, the silicon area of 3.6 mm^2 is relatively small.

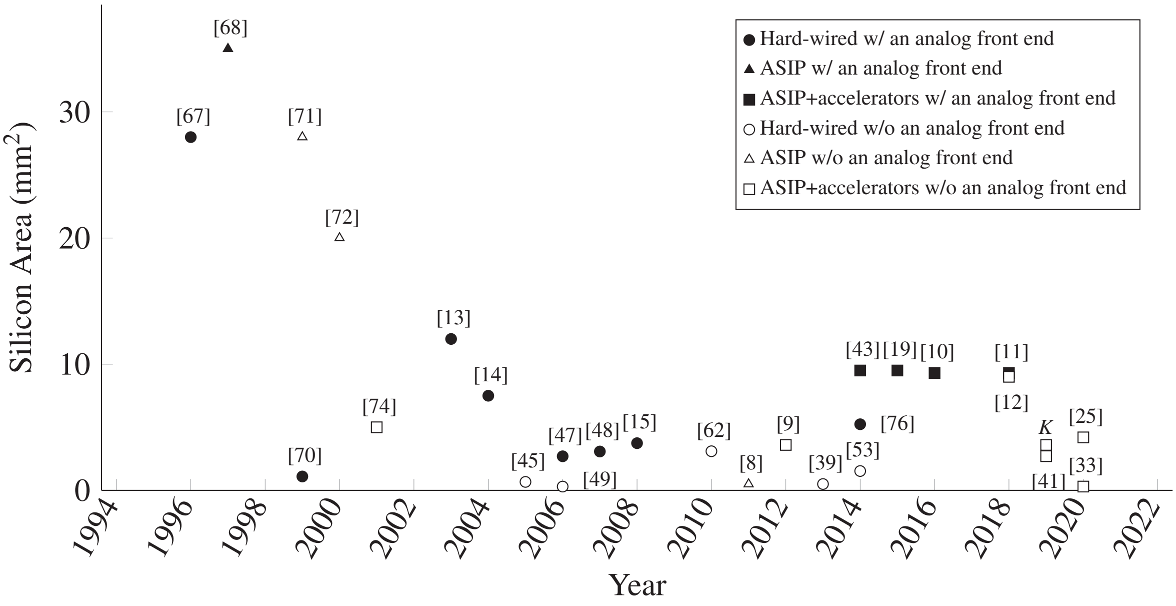


Figure 5.28: Silicon area of commercial and research hearing aids. *KAVUAKA* is marked with the letter K.

5.5.4 Operating Clock Frequency

The required operating clock frequency depends on the computing complexity of the hearing aid algorithms and the architecture-dependent processing power of the digital signal processing system (Figure 5.29). Most hard-wired hearing aids operate at comparatively low operating clock frequencies around 0.032 MHz to 8.000 MHz. The processing is sample-based, i.e., each processing unit or component like a digital filter or amplifier processes one sample per clock cycle. In [53, 62], a more computationally intensive sample-based processing is applied, using a noise reduction algorithm based on multiband spectral subtraction and an enhanced entropy voice activity detection. The audio samples are stored in local ping-pong buffer and processed sequentially for each sub-band at a clock frequency of 3 MHz to 8 MHz for the various processing blocks. Digital hearing aids with an application-specific instruction-set processor as the central processing unit require somewhere in the region of a thousand instructions to process the algorithms. An implementation of a related noise reduction algorithms (*mband*) on an ASIP with hardware accelerators [41] needs 2176 cycles for computation. Parallelism at data or instruction level, or customized application-specific instructions [8, 9, 11, 19, 41, 43, 56, 71, 74, 75] can reduce the clock frequency requirement. Accelerators are used for intensive tasks, where the pure software implementation on an ASIP is not feasible.

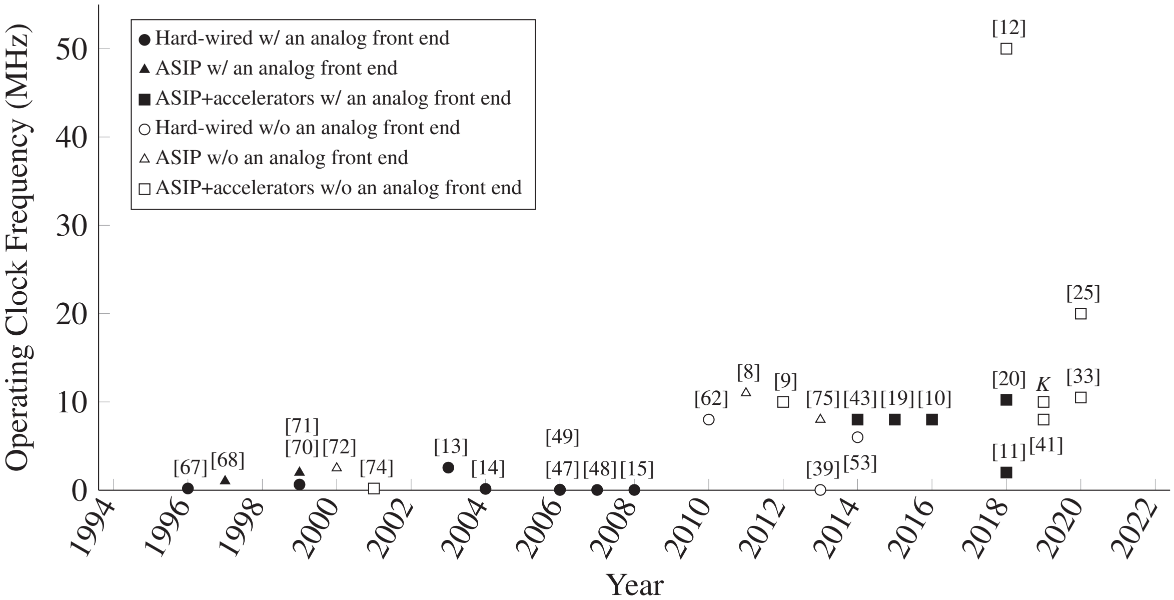


Figure 5.29: Operating clock frequency of commercial and research hearing aids. *KAVUAKA* is marked with the letter K.

5.5.5 Audio Datapath Width

All digital hearing aids presented in this comparison use fixed-point hardware architectures for signal processing, due to lower hardware cost in terms of area and power requirements compared to floating-point hardware [75]. The audio datapath width of the fixed-point data, i.e., the number of bits per audio sample, is a crucial parameter for the design and implementation of hearing aids, as it determines the maximum achievable signal-to-noise ratio (SNR). A high SNR value is a strict requirement for hearing aids [11, 48]. Each additional datapath bit increases the SNR by about 6 dB. However, this parameter also affects the area, power consumption, and processing performance of all components in the processing chain, digital processing blocks, memories, ADCs, and DACs [8, 9, 43, 56, 80]. The authors of [33] present a word length optimization to reduce the area and power of their MAC unit accelerator. They propose to optimize the number of bits based on the results of short-time objective intelligibility (STOI) measurements. Alternatively, signal-to-noise ratio (SNR) measurements are used in [41]. In [8], a 16-bit processor is extended with specific functional units that use 32-bit and 40-bit intermediate results to improve the fixed-point accuracy. Two separate processors are used in [80]. The 32-bit Arm Cortex M3 processor is used for debugging and wireless connectivity and the 24-bit ASIP processes the audio samples. In Table 5.10, a comparison of architectures implementing an audio datapath with fixed width is given. Most designs have a datapath width of 16-bit, for the digital and analog parts. The datapath width can be switched in some ASIP based architectures, which are listed in Table 5.11. This is possible by using different execution units with different datapath width, *micro*SIMD subword modes [292] or specialized accelerators. To take advantage of the increased dynamic range of floating-point data types, the architectures listed in Table 5.12 add hardware support for floating-point processing. The approaches used are block floating-point, static floating-point, or emulated floating-point. The maximum operation frequency of *KAVUAKA* processor is 50 MHz. For the beamforming algorithms, 10 MHz are required (Section 5.1).

5.5.6 On-Chip Memory

Due to strict power and area restrictions, on-chip memory is the only implementation option for the hearing aids listed in Table 5.13. On-chip area is limited and memory size is critical to the overall size of the chip. The area for the SRAM macros for the mixed-signal hearing aid presented in [19] is 1.35 mm^2. Compared to the logic size of 5.39 mm^2 and the analog size of 2.77 mm^2 the area of the SRAM is 14 % of the total chip size for a 130 nm ASIC technology. The memory size depends on the complexity and type of the audio processing algorithms. Algorithms with a comparably high memory requirements are those based on trained models or data. Among those are localization algorithms [6, 27], deep learning based speech enhancement and speech recognition algorithms [24–26, 66]. As an example, the gaussian mixture model (GMM) of the localization algorithm requires about 90 % of the total memory require-

Table 5.10: Fixed audio datapath width architectures of commercial and research hearing aids.

Work	Analog data path	Digital data path	Sampling Frequency	Processor Architecture
[62]	—	16-bit	24 kHz	hard-wired
[9, 25]	—	16-bit	—	ASIP+accelerator
[25]	—	16-bit	16 kHz	ASIP+accelerator
[47]	16-bit	16-bit	16 kHz	hard-wired
[8, 71]	—	16-bit	—	ASIP
[53, 62]	—	16-bit	24 kHz	hard-wired
[75]	—	16-bit	20 kHz	ASIP
[72]	—	22-bit	—	ASIP
[10, 11, 19, 43, 80]	16-bit	24-bit	16 kHz	ASIP+accelerator

Table 5.11: Variable audio datapath width architectures of commercial and research hearing aids.

Work	Analog data path	Digital data path	Sampling Frequency	Processor Architecture
[68, 69]	13-bit	13 to 24-bit	16 kHz	ASIP
[74]	—	12 to 25-bit	16 kHz	ASIP+accelerator
[33]	16-bit	6 to 32-bit	16 kHz	ASIP+accelerator
[41]	16-bit	24 to 32-bit	16 kHz	ASIP+accelerator
[8]	—	24 to 40-bit	16 kHz	ASIP
KAVUAKA	—	8 to 64-bit	16 kHz	ASIP+accelerator

Table 5.12: Optional floating-point audio datapaths of commercial and research hearing aids.

Work	Digital datapath	Processor Architecture
[20, 71, 80]	block floating-point	ASIP+accelerator
[9]	static floating-point	ASIP+accelerator
KAVUAKA	emulated floating-point	ASIP+accelerator

ment of this algorithm [6, 27]. In this case 44,400 of 48,816 words are required only for the trained model. Another example is the hearing aid with the highest amount of on-chip memory, which is designed for computing intensive task as neural networks for speech enhancement [25]. The hearing aid with the least amount of on-chip memory is designed for IIR filters [68, 69].

Table 5.13: On-chip memory sizes of commercial and research hearing aids.

Work	Total	Details
[68, 69]	0.85 kB	0.368 kB instruction RAM, 0.096 kB data RAM and 0.384 kB coefficient RAM
[74]	1.23 kB	0.62 kB data memory for mini-cores, 0.438 kB instruction memory and 0.172 kB coefficient memory
[9]	5.00 kB	Four processing elements (PEs) with 512 B instruction memory, 512 B shared memory for inter-PE communication and 2.5 kB local memory
[43]	6.00 kB	6 kB data memory
[71]	22.50 kB	6.125 kB RAM and 16.375 kB ROM
[72]	68.00 kB	4 kB instruction ROM and 64 kB DSP parameter RAM
[20, 78, 80]	110.00 kB	Six separate logical memory banks, 24-bit data memory, 32-bit DSP instruction memory
KAVUAKA	140.00 kB	28 SRAMs, 65 kB instruction memory, 57 kB data memory and 16 kB audio interface memory
[25]	327.00 kB	Four processing cluster, each with 64 kB for the CNNs and 2 kB for the FFT accelerators

6 Conclusion

Digital signal processing in hearing aids is continuously being developed and is becoming more and more computationally intensive. This results in challenges for the development of suitable, powerful, and efficient hardware architectures for hearing aids, especially with regard to the small physical size of hearing aids and the battery operation. These challenges motivate this thesis. A new low-power application-specific instruction-set processor for digital hearing aids is proposed, which is called *KAVUAKA*. The architecture of *KAVUAKA* is customized for the low-power hearing aid application and is optimized for the processing performance requirements of hearing aid applications. The basis for these optimizations are benchmarks based on state-of-the-art reference hearing aid algorithms. The goal is to design, evaluate, and optimize the hearing aid processor architecture in terms of power consumption, silicon area, and processing performance requirements.

One hardware specialization proposal for hearing aid application-specific instruction-set processors is a **real- and complex-valued SIMD multiply-accumulate (MAC) unit**. A commonly used partial product multiplier architecture for SIMD multiplications is extended to also support complex-valued multiply-accumulate and butterfly operations. Since the same multiplier structure can be used for both operation modes, the silicon area overhead is about 30 % and the estimated average power consumption is about 19 % smaller compared to duplicating MAC-units, as proposed in the related work. A fast Fourier transform (FFT) benchmark demonstrates that the proposed complex-valued SIMD-MAC unit outperforms MAC architectures of related digital signal processors by integrating a SIMD butterfly operation into the complex-valued MAC architecture. The *KAVUAKA* processor is at least 16 % faster in the computation of 128-point FFT compared to the related programmable digital signal processors.

Another proposed hardware specialization is the **efficient emulation of floating-point arithmetic for fixed-point SIMD processors**. The emulated floating-point operations are optimized for commonly available microSIMD architecture mechanisms. Based on a custom floating-point format and arithmetic, which include the independent processing of the significand and the exponent in different subwords, a comparatively fast emulation speed is achieved reducing the required cycles by a factor of 3 on average. The proposed floating-point emulation is faster than any related software floating-point emulation framework implemented on comparable digital signal processors. The proposed library can decrease the development time for signal processing algorithms on fixed-point processors.

With regard to parallelization, a new **issue-slot based predication technique for VLIW processors** is presented. This technique does not require additional instruction encoding bits to address a predicate register. The predicate registers are addressed based on the issue-slot, on which the conditional instructions are scheduled. Multiple predicate registers can be accessed simultaneously. The core area overhead for multiple predicate registers is about 1 %. Two case studies show a performance increase by about 4 % when using two instead of one predicate register in parallel, reaching almost the maximum possible instructions per cycle.

An **evolutionary algorithm-based code generator** is presented, which maps the application input code, i.e., the hearing aid applications, on the *KAVUAKA* processor. Evolutionary algorithm are used for operation merging, instruction scheduling, and register allocation in order to adapt the code generation to different target architecture configurations and constraints and to find an optimal mapping. This code generator is extended to schedule the conditional instructions for issue-slot based predication technique automatically. Additional extensions include techniques for a power-aware instruction scheduling and register allocation. Results show a possible power consumption reduction of up to 55 %. These initial results and concepts for energy aware compiler techniques are studied in an on-going project called *'Comparison of Evolutionary and Machine Learning-Based Algorithms for Energy-Aware Instruction Scheduling' (Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) — PA 2762/2—1).*

A **low latency multichannel audio interface for low-power SIMD digital signal processors** is presented. The architecture is FIFO-based and supports commonly used SIMD vector data formats. Compared to related architectures, which make use of a direct memory access (DMA), no double buffering and interrupts are used. Since no processing overhead is required to perform data permutations, no interrupt routine is required, and no additional buffers are needed, the load, required for handling the audio samples, of the coupled *KAVUAKA* processor is small. For the highest investigated interrupt rate, the average processor load is around 50 % lower for the *KAVUAKA* processor compared to the related processor. The hardware-induced audio latency is 34 % lower compared to related audio interfaces for frame size of 64 samples. A low-power mechanism based on the FIFO-based architecture is presented, which can set the processor to a low-power mode, if no audio samples are ready to be processed.

Finally, **four case studies** with representative hearing aid algorithms are used to evaluate the proposed ASIP optimizations. In case of the **beamforming algorithms**, 24 different optimized processor configurations and three beamforming algorithm combinations are studied. The results show that one of the best combinations in terms of the algorithm performance is the adaptive gain beamformer running on a 24-bit processor without SIMD, with a division coprocessor and optimized register file, including address isolation and dummy registers. This combination offers the highest possible algorithm performance, while the power consumption is by a factor of 11 smaller than using the unoptimized 64-bit processor with SIMD and the same algorithm. The smallest possible combination, with reduced performance (i.e., fixed beamformer with $-2\,\text{dB}$ ISNR), requires 2.2 times less silicon area than the largest combination. The M-measure implementation for **automatic speech recognition** in hearing aids is deemed feasible with more hardware parallelization and model weights quantization. The lowest achievable processing time of the full forward pass is 7.69 ms with a clock frequency of 50 MHz. The optimized M-measure computation would add 0.88 ms when using 10 frames as temporal separation. The **speaker localization** algorithm requires a minimum clock frequency of 30 MHz and the estimated average power is 1.35 mW, which is almost 10 × less than the related transport-triggered architecture. The minimum operating frequency for the **speech enhancement** is reduced by a factor of 6 with co-processors and compiler optimizations.

In order to verify the power consumption simulations and estimations, four different configurations of the *KAVUAKA* processor were **integrated as a system-on-chip** using a 40 nm ASIC technology. The SoC also includes ten co-processors for trigonometric and hyperbolic computations, the multi-channel audio interface and a serial interface. The die size is 3.6 mm^2 and the core area for the 64-bit processor configuration is 0.134 mm^2. The average power consumption is 2.4 mW for the complete SoC and 0.6 mW for the 64-bit *KAVUAKA* core at a clock frequency of 10 MHz. An comparison with state-of-the-art hearing aid ASICs is presented. Analog, mixed-signal and digital processors are part of the comparison, with emphasis on ASIPs, which are compared to dedicated hardware architectures. Trends in ASIC technologies, average power consumption, silicon area, and operating clock frequencies are presented. The average power consumption of 0.6 mW as well as the silicon area of 3.6 mm^2 of the *KAVUAKA* ASIP are among the lowest compared to related ASIPs. Due to new sophisticated algorithms like deep neural network based noise reduction and binaural sound source localization, there is a definite trend towards more flexibility and computing performance requirements. With maximum operating frequency of 50 MHz and the proposed specialized hardware enhancements for the *KAVUAKA* processor, a comparatively high flexibility and processing performance is achieved.

References

[1] World Health Organization and others, "Addressing the rising prevalence of hearing loss," 2018. https://apps.who.int/iris/handle/10665/260336. License: CC BY-NC-SA 3.0 IGO.

[2] National Institute on Deafness and Other Communication Disorders, "Quick statistics about hearing," *National Institutes of Health*, 2016. https://www.nidcd.nih.gov/health/statistics/quick-statistics-hearing.

[3] World Health Organization and others, "Deafness and Hearing Loss," 2020. https://www.who.int/news-room/fact-sheets/detail/deafness-and-hearing-loss.

[4] Semiconductor Components Industries, LLC, "Solving the Hearing Aid Platform Puzzle," *Tech. Rep.*, 2014. https://www.onsemi.com/pub/Collateral/TND6092-D.PDF.

[5] H. Puder, "Hearing aids: an overview of the state-of-the-art, challenges, and future trends of an interesting audio signal processing application," in *Image and Signal Processing and Analysis, 2009. ISPA 2009. Proceedings of 6th International Symposium on*, pp. 1–6, IEEE, 2009.

[6] C. Seifert, J. Thiemann, L. Gerlach, T. Volkmar, G. Payá-Vayá, H. Blume, and S. van de Par, "Real-time implementation of a GMM-based binaural localization algorithm on a VLIW-SIMD processor," in *Multimedia and Expo (ICME), 2017 IEEE International Conference on*, pp. 145–150, IEEE, 2017.

[7] Y. Pu, D. Butterfield, J. Garcia, J. Xie, M. Lin, R. Sauhta, R. Farley, S. Shellhammer, M. Derkalousdian, A. Newham, *et al.*, "An Ultra-low-power 28nm CMOS Dual-die ASIC Platform for Smart Hearables," in *2018 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pp. 1–4, IEEE, 2018.

[8] P. Qiao, H. Corporaal, and M. Lindwer, "A 0.964 mW digital hearing aid system," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pp. 1–4, IEEE, 2011.

[9] K.-C. Chang, Y.-W. Chen, Y.-T. Kuo, and C.-W. Liu, "A Low Power Hearing Aid Computing Platform Using Lightweight Processing Elements," in *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, pp. 2785–2788, IEEE, 2012.

[10] C. Chen, L. Chen, J. Fan, Z. Yu, J. Yang, X. Hu, Y. Hei, and F. Zhang, "A 1V, 1.1 mW mixed-signal hearing aid SoC in 0.13 um CMOS process," in *Circuits and Systems (ISCAS), 2016 IEEE International Symposium on*, pp. 225–228, IEEE, 2016.

[11] C. Chen and L. Chen, "A 79-dB SNR 1.1-mW Fully Integrated Hearing Aid SoC," *Circuits, Systems, and Signal Processing*, pp. 1–17, 2018.

[12] Y. Pu, C. Shi, G. Samson, D. Park, K. Easton, R. Beraha, A. Newham, M. Lin, V. Rangan, K. Chatha, *et al.*, "A 9-mm^2 ultra-low-power highly integrated 28-nm CMOS SoC for Internet of Things," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 3, pp. 936–948, 2018.

[13] D. Gata, W. Sjursen, J. Hochschild, J. Fattaruso, L. Fang, G. Iannelli, Z. Jiang, C. Branch, J. Holmes, M. Skorez, *et al.*, "A 1.1-V 270-μA Mixed-Signal Hearing Aid Chip," *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, vol. 38, no. 4, pp. 683–683, 2003.

[14] Serra-Graells, Francisco and Gomez, Lluís and Huertas, José L, "A True-1-V 300-μW CMOS-Subthreshold Log-Domain Hearing-Aid-On-Chip," *IEEE Journal of Solid-State Circuits*, vol. 39, no. 8, pp. 1271–1281, 2004.

[15] S. Kim, S. J. Lee, N. Cho, S.-J. Song, and H.-J. Yoo, "A fully integrated digital hearing aid chip with human factors considerations," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 266–274, 2008.

[16] G. Payá-Vayá, *Design and Analysis of a Generic VLIW Processor for Multimedia Applications*. PhD thesis, Leibniz Universität Hannover, 2011.

[17] H. Strass, "Digital processors aid hearing devices," 2009. Embedded Computing Design, http://embedded-computing.com/articles/digital-aid-hearing-devices/.

[18] S. Doclo, "Distributed Microphone Array Signal Distributed Microphone Array Signal Processing for Hearing Aids," tech. rep., Signal Processing Group, University of Oldenburg, 2010.

[19] L.-M. Chen, Z.-H. Yu, C.-Y. Chen, X.-Y. Hu, J. Fan, J. Yang, and Y. Hei, "A 1-V, 1.2-mA fully integrated SoC for digital hearing aids," *Microelectronics Journal*, vol. 46, no. 1, pp. 12–19, 2015.

[20] Semiconductor Components Industries, LLC, *Wireless-Enabled Audio Processor for Hearing Aids*, 2018. https://www.onsemi.com/pub/Collateral/E7150-D.PDF.

[21] M. Kock, S. Hesselbarth, M. Pfitzner, and H. Blume, "Hardware-accelerated design space exploration framework for communication systems," *Analog Integrated Circuits and Signal Processing*, vol. 78, no. 3, pp. 557–571, 2014.

[22] K. Kąkol and B. Kostek, "A study on signal processing methods applied to hearing aids," in *2016 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, pp. 219–224, IEEE, 2016.

[23] V. Hamacher, J. Chalupper, J. Eggers, E. Fischer, U. Kornagel, H. Puder, and U. Rass, "Signal Processing in High-End Hearing Aids: State of the Art, Challenges, and Future

Trends," *EURASIP Journal on Applied Signal Processing*, vol. 2005, pp. 2915–2929, 2005.

[24] A. M. C. Martinez, L. Gerlach, G. Payá-Vayá, H. Hermansky, J. Ooster, and B. T. Meyer, "DNN-based performance measures for predicting error rates in automatic speech recognition and optimizing hearing aid parameters," *Speech Communication*, vol. 106, pp. 44–56, 2019.

[25] Y.-C. Lee, T.-S. Chi, and C.-H. Yang, "A 2.17-mW Acoustic DSP Processor With CNN-FFT Accelerators for Intelligent Hearing Assistive Devices," *IEEE Journal of Solid-State Circuits*, 2020.

[26] S. R. Park and J. Lee, "A fully convolutional neural network for speech enhancement," *arXiv preprint arXiv:1609.07132*, 2016.

[27] T. May, S. Van De Par, and A. Kohlrausch, "A probabilistic model for robust localization based on a binaural auditory front-end," *IEEE Transactions on audio, speech, and language processing*, vol. 19, no. 1, pp. 1–13, 2011.

[28] R. Rehr and T. Gerkmann, "SNR-Based Features and Diverse Training Data for Robust DNN-Based Speech Enhancement," *arXiv preprint arXiv:2004.03512*, 2020.

[29] M. Tammen, D. Fischer, B. T. Meyer, and S. Doclo, "DNN-Based Speech Presence Probability Estimation for Multi-Frame Single-Microphone Speech Enhancement," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 191–195, IEEE, 2020.

[30] R. Varzandeh, K. Adiloğlu, S. Doclo, and V. Hohmann, "Exploiting Periodicity Features for Joint Detection and DOA Estimation of Speech Sources Using Convolutional Neural Networks," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 566–570, IEEE, 2020.

[31] V. Hamacher, E. Fischer, U. Kornagel, and H. Puder, "Applications of adaptive signal processing methods in high-end hearing aids," *Topics in Acoustic Echo and Noise Control, edited by E. Hansler and G. Schmidt (Springer, Berlin)*, pp. 599–636, 2006.

[32] L. Griffiths and C. Jim, "An alternative approach to linearly constrained adaptive beamforming," *IEEE Transactions on antennas and propagation*, vol. 30, no. 1, pp. 27–34, 1982.

[33] Y.-J. Lin, Y.-C. Lee, H.-M. Liu, H. Chiueh, T.-S. Chi, and C.-H. Yang, "A 1.5 mW Programmable Acoustic Signal Processor for Hearing Assistive Devices With Speech Intelligibility Enhancement," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2020.

[34] Y. Ephraim and D. Malah, "Speech enhancement using a minimum-mean square error short-time spectral amplitude estimator," *IEEE Transactions on acoustics, speech, and signal processing*, vol. 32, no. 6, pp. 1109–1121, 1984.

[35] Y. Ephraim and D. Malah, “Speech enhancement using a minimum mean-square error log-spectral amplitude estimator,” *IEEE transactions on acoustics, speech, and signal processing*, vol. 33, no. 2, pp. 443–445, 1985.

[36] H.-G. Hirsch and C. Ehrlicher, “Noise estimation techniques for robust speech recognition,” in *1995 International conference on acoustics, speech, and signal processing*, vol. 1, pp. 153–156, IEEE, 1995.

[37] P. Scalart *et al.*, “Speech enhancement based on a priori signal to noise estimation,” in *1996 IEEE International Conference on Acoustics, Speech, and Signal Processing Conference Proceedings*, vol. 2, pp. 629–632, IEEE, 1996.

[38] K. El-Maleh and P. Kabal, “Comparison of voice activity detection algorithms for wireless personal communications systems,” in *CCECE’97. Canadian Conference on Electrical and Computer Engineering. Engineering Innovation: Voyage of Discovery. Conference Proceedings*, vol. 2, pp. 470–473, IEEE, 1997.

[39] L. Shaer, I. Nahlus, J. Merhi, A. Kayssi, and A. Chehab, “Low-power digital signal processor design for a hearing aid,” in *2013 4th Annual International Conference on Energy Aware Computing Systems and Applications (ICEAC)*, pp. 40–44, IEEE, 2013.

[40] B. Farhang-Boroujeny and Z. Wang, “Adaptive filtering in subbands: Design issues and experimental results for acoustic echo cancellation,” *Signal Processing*, vol. 61, no. 3, pp. 213–223, 1997.

[41] S.-W. Kim, M.-J. Kim, and J.-S. Kim, “High-performance DSP platform for digital hearing aid SoC with flexible noise estimation,” *IET Circuits, Devices & Systems*, vol. 13, no. 5, pp. 717–722, 2019.

[42] H.-F. Chi, S. X. Gao, and S. D. Soli, “A novel approach of adaptive feedback cancellation for hearing aids,” in *1999 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 3, pp. 195–198, IEEE, 1999.

[43] Y. Jia, C. Liming, Y. Zenghui, and H. Yong, “A sub-milliwatt audio-processing platform for digital hearing aids,” *Journal of Semiconductors*, vol. 35, no. 7, p. 075008, 2014.

[44] H. Teutsch and G. W. Elko, “First-and second-order adaptive differential microphone arrays,” in *Proc. IWAENC*, vol. 1, Citeseer, 2001.

[45] F. Carbognani, F. Burgin, L. Henzen, H. Koch, H. Magdassian, C. Pedretti, H. Kaeslin, N. Felber, and W. Fichtner, “A 0.67-mm^2 45-μW DSP VLSI Implementation of an Adaptive Directional Microphone for Hearing Aids,” in *Proceedings of the 2005 European Conference on Circuit Theory and Design, 2005.*, vol. 3, pp. III/141–III/144 vol. 3, 2005.

[46] W. P. Sjursen, “Hearing aid digital filter,” Sept. 18 2001. US Patent 6,292,571.

[47] S. Kim, N. Cho, S. . Song, D. Kim, K. Kim, and H. . Yoo, “A 0.9-V 96-μW Digital Hearing Aid Chip with Heterogeneous Σ-Δ DAC,” in *2006 Symposium on VLSI Circuits, 2006. Digest of Technical Papers.*, pp. 55–56, 2006.

[48] S. Kim, N. Cho, S.-J. Song, and H.-J. Yoo, “A 0.9 V 96 μW Fully Operational Digital Hearing Aid Chip,” *IEEE journal of solid-state circuits*, vol. 42, no. 11, pp. 2432–2440, 2007.

[49] J. Yoo, S. Kim, N. Cho, S.-J. Song, and H.-J. Yoo, “A 10-μW digital signal processor with adaptive-SNR monitoring for a sub-1 V digital hearing aid,” in *IEEE International Symposium on Circuits and Systems*, 2006.

[50] C. Ris and S. Dupont, “Assessing local noise level estimation methods: Application to noise robust ASR,” *Speech communication*, vol. 34, no. 1-2, pp. 141–158, 2001.

[51] J. Lee, J. Kim, and G. Yoon, “Digital envelope detector for blood pressure measurement using an oscillometric method,” *Journal of medical engineering & technology*, vol. 26, no. 3, pp. 117–122, 2002.

[52] C. Jia and B. Xu, “An improved entropy-based endpoint detection algorithm,” in *International Symposium on Chinese Spoken Language Processing*, 2002.

[53] C.-W. Wei, C.-C. Tsai, Y. FanJiang, T.-S. Chang, and S.-J. Jou, “Analysis and implementation of low-power perceptual multiband noise reduction for the hearing aids application,” *IET Circuits, Devices & Systems*, vol. 8, no. 6, pp. 516–525, 2014.

[54] S. Kamath and P. Loizou, “A multi-band spectral subtraction method for enhancing speech corrupted by colored noise,” in *ICASSP*, vol. 4, pp. 44164–44164, Citeseer, 2002.

[55] F.-L. Luo, J. Yang, C. Pavlovic, and A. Nehorai, “Adaptive Null-Forming Scheme in Digital Hearing Aids,” *Signal Processing, IEEE Transactions on*, vol. 50, no. 7, pp. 1583–1590, 2002.

[56] L. Gerlach, G. Payá-Vayá, and H. Blume, “KAVUAKA: A Low Power Application Specific Hearing Aid Processor,” in *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 99–104, Oct 2019.

[57] J. M. Kates, “Principles of digital dynamic-range compression,” *Trends in amplification*, vol. 9, no. 2, pp. 45–76, 2005.

[58] N. Westerlund, M. Dahl, and I. Claesson, “Speech enhancement for personal communication using an adaptive gain equalizer,” *Signal Processing*, vol. 85, no. 6, pp. 1089–1101, 2005.

[59] J. Chen, J. Benesty, Y. Huang, and S. Doclo, “New insights into the noise reduction Wiener filter,” *IEEE Transactions on audio, speech, and language processing*, vol. 14, no. 4, pp. 1218–1234, 2006.

[60] P. Loizou, *Speech Enhancement: Theory and Practice, Second Edition*. Taylor & Francis, 2013.

[61] Y.-T. Kuo, T.-J. Lin, W.-H. Chang, Y.-T. Li, C.-W. Liu, and S.-T. Young, “Complexity-effective auditory compensation for digital hearing aids,” in *2008 IEEE International Symposium on Circuits and Systems*, pp. 1472–1475, IEEE, 2008.

[62] C.-W. Wei, Y.-T. Kuo, K.-C. Chang, C.-C. Tsai, J.-Y. Lin, Y. FanJiang, M.-H. Tu, C.-W. Liu, T.-S. Chang, and S.-J. Jou, “A low-power Mandarin-specific hearing aid chip,” in *2010 IEEE Asian Solid-State Circuits Conference*, pp. 1–4, IEEE, 2010.

[63] J. Kates, *Digital Hearing Aids*. Plural Publishing, Incorporated, 2008.

[64] J. Chen and B. C. Moore, “Effect of individually tailored spectral change enhancement on speech intelligibility and quality for hearing-impaired listeners,” in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 8643–8647, IEEE, 2013.

[65] J. Benesty and Y. Huang, *Adaptive Signal Processing: Applications to Real-World Problems*. Signals and Communication Technology, Springer Berlin Heidelberg, 2013.

[66] C.-Y. Lai, Y.-W. Lo, Y.-L. Shen, and T.-S. Chi, “Plastic multi-resolution auditory model based neural network for speech enhancement,” in *2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, pp. 605–609, IEEE, 2017.

[67] J. F. Duque-Carrillo, P. Malcovati, F. Maloberti, R. Pérez-Aloe, A. H. Reyes, E. Sánchez-Sinencio, G. Torelli, and J. M. Valverde, “VERDI: An acoustically programmable and adjustable CMOS mixed-mode signal processor for hearing aid applications,” *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, pp. 634–645, 1996.

[68] H. Neuteboom, B. M. Kup, and M. Janssens, “A DSP-Based Hearing Instrument IC,” *IEEE Journal of Solid-State Circuits*, vol. 32, no. 11, pp. 1790–1806, 1997.

[69] H. Neuteboom, M. Janssens, J. Leenen, B. Kup, E. Dijkmans, B. De Koning, V. Frowijn, R. De Bleecker, E. Van der Zwan, S. Note, *et al.*, “A single battery, 0.9 V operated digital sound processing IC including AD/DA and IR receiver with 2 mW power consumption,” in *1997 IEEE International Solids-State Circuits Conference. Digest of Technical Papers*, pp. 98–99, IEEE, 1997.

[70] J. S. Martinez, S. S. Bustos, J. S. Suñer, R. R. Hernández, and M. Schellenberg, “A CMOS Hearing Aid Device,” *Analog Integrated Circuits and Signal Processing*, no. 21, pp. 163–172, 1999.

[71] F. Moller, N. Bisgaard, and J. Melanson, “Algorithm and architecture of a 1 V low power hearing instrument DSP,” in *Proceedings. 1999 International Symposium on Low Power Electronics and Design (Cat. No. 99TH8477)*, pp. 7–11, IEEE, 1999.

[72] P. Mosch, G. van Oerle, S. Menzl, N. Rougnon-Glasson, K. Van Nieuwenhove, and M. Wezelenburg, “A 660-μW 50-Mops 1-V DSP for a Hearing Aid Chip Set,” *IEEE Journal of Solid-State Circuits*, vol. 35, no. 11, pp. 1705–1712, 2000.

[73] R. Klootsema, O. Nys, E. Vandel, D. Aebischer, P. Vaucher, O. Hautier, P. Bratschi, F. Bauduin, G. Van Oerle, A. Jakob, *et al.*, “Battery supplied low power analog-digital front-end for audio applications,” in *Proceedings of the 26th European solid-state circuits conference*, pp. 118–121, IEEE, 2000.

[74] O. Paker, J. Sparso, N. Haandbæk, M. Isager, and L. S. Nielsen, “A heterogeneous multiprocessor architecture for low-power audio signal processing applications,” in *VLSI, 2001. Proceedings. IEEE Computer Society Workshop on*, pp. 47–53, IEEE, 2001.

[75] Y. Ku, J. Sohn, J. Han, Y. Baek, and D. Kim, “A High Performance Hearing Aid System with Fully Programmable Ultra Low Power DSP,” in *Consumer Electronics (ICCE), 2013 IEEE International Conference on*, pp. 352–353, 2013.

[76] X. Wang, H. Yang, F. Li, T. Yin, G. Huang, and F. Liu, “A programmable analog hearing aid system-on-chip with frequency compensation,” *Analog Integrated Circuits and Signal Processing*, vol. 79, no. 2, pp. 227–236, 2014.

[77] N. Werner, G. Payá-Vayá, and H. Blume, “Case study: Using the xtensa LX4 configurable processor for hearing aid applications,” *Proceedings of the ICT. OPEN*, 2013.

[78] Berkeley Design Technology, Inc., “ON Semiconductor’s Hearing Aid SoCs: Distributed Performance That’s Easy on Batteries,” 2014.

[79] Semiconductor Components Industries, LLC, “NXP puts CoolFlux DSP in to hearing aids,” 2008. https://www.electronicsweekly.com/market-sectors/embedded-systems/nxp-puts-coolflux-dsp-in-to-hearing-aids-2008-11/.

[80] Semiconductor Components Industries, LLC, *EZAIRO 7111 HYBRID: Audio Processor for Digital Hearing Aids*, 2018. https://www.onsemi.com/pub/Collateral/E7111-D.PDF.

[81] K. Samtani, J. Thomas, S. Deepu, and S. S. David, “Area and power optimised ASIC implementation of adaptive beamformer for hearing aids,” in *Biomedical Circuits and Systems Conference (BioCAS), 2017 IEEE*, pp. 1–4, IEEE, 2017.

[82] Semiconductor Components Industries, LLC, “WOLA Filterbank Coprocessor: Introductory Concepts and Techniques.”

[83] J. Hartig, L. Gerlach, G. Payá-Vayá, and H. Blume, “Customizing a VLIW-SIMD Application-Specific Instruction-Set Processor for Hearing Aid Devices,” in *Signal Process. Syst. (SiPS), 2014 IEEE Workshop on*, pp. 1–6, IEEE, 2014.

[84] G. Payá-Vayá, J. Martín-Langerwerf, F. Giesemann, H. Blume, and P. Pirsch, “Instruction Merging to Increase Parallelism in VLIW Architectures,” in *System-on-Chip, 2009. SOC 2009. International Symposium on*, pp. 143–146, IEEE, 2009.

[85] G. Payá-Vayá, J. Martín-Langerwerf, and P. Pirsch, “A multi-shared register file structure for VLIW processors,” *Journal of Signal Processing Systems*, vol. 58, no. 2, pp. 215–231, 2010.

[86] D. Patterson and J. Hennessy, *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. ISSN, Elsevier Science, 2020.

[87] J. Squire, “CMSC 411 Computer Architecture,” tech. rep., University of Maryland Baltimore County, 2019.

[88] M. Lu *et al.*, *Arithmetic and logic in computer systems*, vol. 169. Wiley Online Library, 2004.

[89] B. Venkataramani and M. Bhaskar, *Digital Signal Processors: Architecture, Programming and Applications*. McGraw-Hill Education (India) Pvt Limited, 2002.

[90] L. L. Corporation, *LSI403LP Digital Signal Processor*. LSI Logic Corporation, 2002. www.lsi.com.

[91] B. Ackland, A. Anesko, D. Brinthaupt, S. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C. Nicol, J. O’Neill, *et al.*, “A single-chip, 1.6-billion, 16-b MAC/s multiprocessor DSP,” *Solid-State Circuits, IEEE Journal of*, vol. 35, no. 3, pp. 412–424, 2000.

[92] W. Hinrichs, J. P. Wittenburg, H. Lieske, H. Kloos, M. Ohmacht, and P. Pirsch, “A 1.3-GOPS parallel DSP for high-performance image-processing applications,” *Solid-State Circuits, IEEE Journal of*, vol. 35, no. 7, pp. 946–952, 2000.

[93] Y.-H. Huang, H.-P. Ma, M.-L. Liou, and T.-D. Chiueh, “A 1.1 G MAC/s sub-word-parallel digital signal processor for wireless communication applications,” *Solid-State Circuits, IEEE Journal of*, vol. 39, no. 1, pp. 169–183, 2004.

[94] S.-R. Kuang and J.-P. Wang, “Design of power-efficient pipelined truncated multipliers with various output precision,” *Computers & Digital Techniques, IET*, vol. 1, no. 2, pp. 129–136, 2007.

[95] Texas Instruments Inc., *TMS320C6747 Fixed- and Floating-Point DSP*. Texas Instruments Inc., www.ti.com, 2014.

[96] Y. Luo, Z. Zhang, X. Huang, J. Wu, and X. Chen, “Architecture and implementation of a vector MAC unit for complex number,” in *Communications and Networking in China (CHINACOM), 2014 9th International Conference on*, pp. 589–594, IEEE, 2014.

[97] Texas Instruments Inc., *TMS320C5517 Fixed-Point Digital Signal Processor*. Texas Instruments Inc., www.ti.com, 2014.

[98] J. Hartig, “Implementation and Evaluation of a VLIW-SIMD Application-Specific Instruction-Set Processor for Hearing Aid Systems,” Master’s thesis, Leibniz Universität Hannover, 2013.

[99] S. Krithivasan and M. J. Schulte, "Multiplier architectures for media processing," in *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Seventh Asilomar Conference on*, vol. 2, pp. 2193–2197, IEEE, 2003.

[100] C. Arm, S. Gyger, J.-M. Masgonty, M. Morgan, J.-L. Nagel, C. Piguet, F. Rampogna, and P. Volet, "Low-Power 32-bit Dual-MAC 120 μW/MHz 1.0 V icyflex1 DSP/MCU Core," *Solid-State Circuits, IEEE Journal of*, vol. 44, no. 7, pp. 2055–2064, 2009.

[101] C. Arm, J.-M. Masgonty, M. Morgan, C. Piguet, F. Rampogna, P. Volet, *et al.*, "Low-Power Quad-MAC 170 uW/MHz 1.0 V MACGIC DSP Core," in *Solid-State Circuits Conference, 2006. ESSCIRC 2006. Proc. of the 32nd European*, pp. 223–226, IEEE, 2006.

[102] C.-K. Chen, P.-C. Tseng, Y.-C. Chang, and L.-G. Chen, "A digital signal processor with programmable correlator array architecture for third generation wireless communication system," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, vol. 48, no. 12, pp. 1110–1120, 2001.

[103] Y.-H. Huang and T.-D. Chiueh, "A sub-word parallel digital signal processor for wireless communication systems," in *ASIC, 2002. Proceedings. 2002 IEEE Asia-Pacific Conference on*, pp. 287–290, IEEE, 2002.

[104] H. Roeven, J. Coninx, and M. Ade, "CoolFlux DSP-The embedded ultra low power C-programmable DSP core," in *Proc. Intl. Signal Proc. Conf. (GSPx)*, Citeseer, 2004.

[105] S. Ong, M. H. Sunwoo, and M. Hong, "A fixed-point multimedia DSP chip for portable multimedia services," in *Signal Processing Systems, 1998. SIPS 98. 1998 IEEE Workshop on*, pp. 94–102, IEEE, 1998.

[106] B.-W. Kim, J.-H. Yang, C.-S. Hwang, Y.-S. Kwon, K.-M. Lee, I.-H. Kim, Y.-H. Lee, and C.-M. Kyung, "MDSP-II: A 16-bit DSP with mobile communication accelerator," *Solid-State Circuits, IEEE Journal of*, vol. 34, no. 3, pp. 397–404, 1999.

[107] K. Tatas, G. Koutroumpezis, D. Soudris, and A. Thanailakis, "Architecture design of a coarse-grain reconfigurable multiply-accumulate unit for data-intensive applications," *INTEGRATION, the VLSI journal*, vol. 40, no. 2, pp. 74–93, 2007.

[108] Freescale Semiconductor, Inc., *Software Optimization of DFTs and IDFTs Using the StarCore SC3850 DSP Core*. Freescale Semiconductor Inc., 2010, Available: www.freescale.com/.

[109] S. Agarwala, T. Anderson, A. Hill, M. D. Ales, R. Damodaran, P. Wiley, S. Mullinnix, J. Leach, A. Lell, M. Gill, *et al.*, "A 600-MHZ VLIW DSP," *Solid-State Circuits, IEEE Journal of*, vol. 37, no. 11, pp. 1532–1544, 2002.

[110] Y.-L. Tsao, W.-H. Chen, M. H. Tan, M.-C. Lin, and S.-J. Jou, "Low-power embedded DSP core for communication systems," *EURASIP Journal on Applied Signal Processing*, vol. 2003, pp. 1355–1370, 2003.

[111] C. Rowen, P. Nuth, and S. Fiske, "A DSP architecture optimized for wireless baseband," in *System-on-Chip, 2009. SOC 2009. International Symposium on*, pp. 151–156, IEEE, 2009.

[112] I. Freescale Semiconductor, *Symphony DSP56724/DSP56725 Multi-Core Audio Processors Reference Manual*, 2009.

[113] Analog Devices Inc., "Blackfin Embedded Processor: ADSP-BF531," tech. rep., ADSP-BF532/ADSP-BF533 Datasheet. Januray, 2011.

[114] A. Danysh and D. Tan, "Architecture and implementation of a vector/SIMD multiply-accumulate unit," *Computers, IEEE Transactions on*, vol. 54, no. 3, pp. 284–293, 2005.

[115] M. Basiri and N. M. Sk, "An efficient hardware based MAC design in digital filters with complex numbers," in *Signal Processing and Integrated Networks (SPIN), 2014 International Conference on*, pp. 475–480, IEEE, 2014.

[116] M. S. Schmookler, J. Tyler, H. Nguyen, M. Putrino, A. Mather, J. Lent, C. Roth, M. N. Pham, and M. Sharma, "A low-power, high-speed implementation of a PowerPC (tm) microprocessor vector extension," in *14th IEEE Symposium on Computer Arithmetic (ARITH-14'99)*, p. 12, IEEE Computer Society, 1999.

[117] L. Huang, N. Xiao, Z. Wang, Y. Wang, and M. Lai, "Efficient multimedia coprocessor with enhanced SIMD engines for exploiting ILP and DLP," *Parallel Computing*, vol. 39, no. 10, pp. 586–602, 2013.

[118] C.-L. Wey and J.-F. Li, "Design of reconfigurable array multipliers and multiplier-accumulators," in *Circuits and Systems, 2004. Proceedings. The 2004 IEEE Asia-Pacific Conference on*, vol. 1, pp. 37–40, IEEE, 2004.

[119] K. Nadehara, T. Miyazaki, and I. Kuroda, "Radix-4 FFT implementation using SIMD multimedia instructions," in *Acoustics, Speech, and Signal Processing, 1999. Proceedings., 1999 IEEE International Conference on*, vol. 4, pp. 2131–2134, IEEE, 1999.

[120] L.-D. Van and J.-H. Tu, "Power-efficient pipelined reconfigurable fixed-width Baugh-Wooley multipliers," *Computers, IEEE Transactions on*, vol. 58, no. 10, pp. 1346–1355, 2009.

[121] Cadence Design Systems, Inc., *Tensilica HiFi DSP Family - Configurable processors for audio, voice, and speech processing*, 2017.

[122] Cadence Design Systems, Inc., *Tensilica Fusion F1 DSP - Configurable processor for Internet of Things applications*, 2016.

[123] T. Instruments, "TMS320C6748 Fixed-and Floating-Point DSP," *Data Sheet, SPRS591, Texas Instruments Inc*, 2011.

[124] U. Zölzer, *Digital Audio Signal Processing*. Wiley, 2008.

[125] E. Hänsler and G. Schmidt, *Speech and Audio Processing in Adverse Environments*. Springer Publishing Company, Incorporated, 1st ed., 2010.

[126] J. S. Lee and M. H. Sunwoo, "Design of new DSP instructions and their hardware architecture for high-speed FFT," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 33, no. 3, pp. 247–254, 2003.

[127] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule, "Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications," *Micro, IEEE*, vol. 34, no. 2, pp. 34–43, 2014.

[128] L. Gerlach, G. Payá-Vayá, and H. Blume, "An Area Efficient Real- and Complex-Valued Multiply-Accumulate SIMD Unit for Digital Signal Processors," in *Signal Process. Systems (SiPS), 2015 IEEE Workshop on*, pp. 1–6, IEEE, 2015.

[129] P. D. S. Labs, *CoolFlux DSP*. NXP, www.coolfluxdsp.com, 2004.

[130] T. Stetzler, N. Magotra, P. Gelabert, P. Kasthuri, and S. Bangalore, "Low Power Real-Time Programmable DSP Development Platform for Digital Hearing Aids," in *Acoustics, Speech, and Signal Processing, 1999. Proceedings., 1999 IEEE International Conference on*, vol. 4, pp. 2339–2342, IEEE, 1999.

[131] Freescale Semiconductor, Inc., *Six-Core Digital Signal Processor*. Freescale Semiconductor Inc., www.freescale.com, 2013.

[132] A. A. Al Sallab, H. Fahmy, and M. Rashwan, "Optimized hardware implementation of fft processor," in *Design and Test Workshop (IDT), 2009 4th International*, pp. 1–5, IEEE, 2009.

[133] L. Liu, Z. Yang, S. Li, and M. Yan, "Implementation of high-throughput FFT processing on an application-specific reconfigurable processor," in *Computer Science and Network Technology (ICCSNT), 2012 2nd International Conference on*, pp. 1284–1288, IEEE, 2012.

[134] S. S. Kidambi, F. El-Guibaly, and A. Antoniou, "Area-efficient multipliers for digital signal processing applications," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, vol. 43, no. 2, pp. 90–95, 1996.

[135] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[136] Analog Devices Inc., *SHARC Processor: ADSP-21161N*. Analog Devices Inc., www.analog.com, 2013.

[137] TSMC, "40nm Technology," tech. rep., Taiwan Semiconductor Manufacturing Company Limited, 2018.

[138] C. Inacio and D. Ombres, "The DSP decision: Fixed point or floating?," *IEEE Spectrum*, vol. 33, no. 9, pp. 72–74, 1996.

[139] D. Menard, D. Chillet, and O. Sentieys, "Floating-to-fixed-point conversion for digital signal processors," *EURASIP journal on applied signal processing*, vol. 2006, pp. 77–77, 2006.

[140] M. Christensen and F. J. Taylor, "Fixed-point-IIR-filter challenges," *EDN Netw*, vol. 51, no. 23, pp. 111–122, 2006.

[141] T. Hilaire, A. Volkova, and M. Ravoson, "Reliable fixed-point implementation of linear data-flows," in *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*, pp. 92–97, IEEE, 2016.

[142] R. Oshana, *DSP for Embedded and Real-Time Systems*. Expert guide, Elsevier Science, 2012.

[143] Analog Devices Inc., "Fast Floating-Point Arithmetic Emulation on Blackfin® Processors," *Anlog Devices*, 2007.

[144] M. Sarkar, "A Floating-Point to Fixed-Point Conversion Methodology for Audio Algorithms," tech. rep., Citeseer, 2004.

[145] J. Hauser, "SoftFloat," *available from http://www. jhauser. us/arithmetic/SoftFloat. html*, 2002.

[146] S. K. Raina, *FLIP: a floating-point library for integer processors*. PhD thesis, École Normale Supérieure de Lyon, 2006.

[147] D. Connors, Y. Yamada, and W. Hwuy, "A Software-Oriented Floating-Point Format for Enhancing Automotive Control Systems," in *Workshop on Compiler and Architecture Support for Embedded Computing Systems (CASES98)*, 1998.

[148] Z. Nikolić, H. T. Nguyen, and G. Frantz, "Design and implementation of numerical linear algebra algorithms on fixed point DSPs," *EURASIP Journal on Advances in Signal Processing*, vol. 2007, no. 1, p. 087046, 2007.

[149] T. Instruments, "TMS320C6000 Technical Brief," *Literatür No: SPRU197D, Texas, ABD*, 1999.

[150] C.-P. Jeannerod, J. Jourdan-Lu, and C. Monat, "Non-generic floating-point software support for embedded media processing," in *Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium on*, pp. 283–286, IEEE, 2012.

[151] J. J. Pimentel and B. M. Baas, "Hybrid floating-point modules with low area overhead on a fine-grained processing core," in *Signals, Systems and Computers, 2014 48th Asilomar Conference on*, pp. 1829–1833, IEEE, 2014.

[152] S. Z. Gilani, N. S. Kim, and M. Schulte, "Virtual floating-point units for low-power embedded processors," in *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pp. 61–68, IEEE, 2012.

[153] S. Z. Gilani, N. S. Kim, and M. Schulte, "Energy-efficient floating-point arithmetic for digital signal processors," in *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, pp. 1823–1827, IEEE, 2011.

[154] X. Wang and M. Leeser, "VFloat: a variable precision fixed-and floating-point library for reconfigurable hardware," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 3, no. 3, p. 16, 2010.

[155] M. K. Jaiswal, R. C. Cheung, M. Balakrishnan, and K. Paul, "Unified architecture for double/two-parallel single precision floating point adder," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 61, no. 7, pp. 521–525, 2014.

[156] V. Reddy, S. Z. Gilani, E. Gunadi, N. S. Kim, M. J. Schulte, and M. H. Lipasti, "REEL: Reducing effective execution latency of floating point operations," in *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, pp. 187–192, IEEE Press, 2013.

[157] R. B. Lee, "Subword parallelism with MAX-2," *IEEE micro*, vol. 16, no. 4, pp. 51–59, 1996.

[158] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo, *et al.*, "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[159] C.-P. Jeannerod, C. Mouilleron, J.-M. Muller, G. Revy, C. Bertin, J. Jourdan-Lu, H. Knochel, and C. Monat, "Techniques and tools for implementing IEEE 754 floating-point arithmetic on VLIW integer processors," in *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, pp. 1–9, ACM, 2010.

[160] T. Hon and M. Marsono, "Hardware Design Space Exploration of CORDIC Algorithm for Run-Time Reconfigurable Platform," in *The First International Conference on Green Computing*, pp. 1–7, 2013.

[161] J. S. Walther, "A unified algorithm for elementary functions," in *Proceedings of the May 18-20, 1971, spring joint computer conference*, pp. 379–385, 1971.

[162] M. Andrews, S. McCormick, and G. Taylor, "Evaluation of functions on microcomputers: square root," *Computers & Mathematics with Applications*, vol. 4, no. 4, pp. 359–367, 1978.

[163] J. Crenshaw, *Math toolkit for real-time programming*. CRC Press, 2000.

[164] S. Nolting, G. Payá-Vayá, I. Schmädecke, and H. Blume, "Evaluation of a Generic Radix-4 CORDIC Coprocessor Tightly Coupled with a Generic VLIW-SIMD ASIP Architecture," *Proceedings of the ICT. OPEN*, 2012.

[165] L. Gerlach, S. Nolting, H. Blume, G. Payá-Vayá, H. Stolberg, and C. Reuter, "A Highly Optimized Arithmetic Software Library and Hardware Co-processor IP for Fixed-Point

VLIW-SIMD Processor Architectures," in *Technology Transfer in Computing Systems (TETRACOM Technology Transfer Project (TTP), 2016), Prague, Czech Republic*, 2016.

[166] M. Yang, J. Wang, Y. Wang, and S. Zheng, "Optimized parallel implementation of polynomial approximation math functions on a DSP processor," in *Proceedings of the 44th IEEE 2001 Midwest Symposium on Circuits and Systems. MWSCAS 2001 (Cat. No. 01CH37257)*, vol. 1, pp. 344–347, IEEE, 2001.

[167] T. Instruments, "Texas Instruments Test Results MATHLIB 3.1.2.1 c674x," tech. rep., Texas Instruments, 2019.

[168] K. G. Lenzi and O. Saotome, "Optimized math functions for a fixed-point DSP architecture," in *19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'07)*, pp. 125–132, IEEE, 2007.

[169] G. Payá-Vayá, J. Martín-Langerwerf, C. Banz, F. Giesemann, P. Pirsch, and H. Blume, "VLIW Architecture Optimization for an Efficient Computation of Stereoscopic Video Applications," in *Green Circuits and Systems (ICGCS), 2010 International Conference on*, pp. 457–462, IEEE, 2010.

[170] P. Faraboschi, J. A. Fisher, and C. Young, "Instruction scheduling for instruction level parallel processors," *Proceedings of the IEEE*, vol. 89, no. 11, pp. 1638–1659, 2001.

[171] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-M. W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2, pp. 138–150, 1995.

[172] B. R. Rau and J. A. Fisher, *Instruction-Level Parallelism*, p. 883–887. GBR: John Wiley and Sons Ltd., 2003.

[173] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley, "Dataflow predication," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pp. 89–102, IEEE, 2006.

[174] R. A. Starke, A. Carminati, and R. S. de Oliveira, "Evaluation of a low overhead predication system for a deterministic VLIW architecture targeting real-time applications," *Microprocessors and Microsystems*, vol. 49, pp. 1–8, 2017.

[175] D. N. Pnevmatikatos and G. S. Sohi, *Guarded execution and branch prediction in dynamic ILP processors*, vol. 22. IEEE Computer Society Press, 1994.

[176] C. W. Kesseler, "Compiling for VLIW DSPs," in *Handbook of Signal Processing Systems*, ch. 3, Springer, 2018.

[177] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier, 2005.

[178] J. Crawford and F. J. Huck, “Next Generation Instruction Set Architecture,” in *Microprocessor Forum*, 1997.

[179] M. Verma and P. Marwedel, “Memory wall problem,” in *Advanced memory optimization techniques for low-power embedded processors*, vol. 1, ch. 1.1.1, Springer, 2007.

[180] A. Artes, J. L. Ayala, J. Huisken, and F. Catthoor, “Survey of low-energy techniques for instruction memory organisations in embedded systems,” *Journal of Signal Processing Systems*, vol. 70, no. 1, pp. 1–19, 2013.

[181] D. Jaggar and D. Seal, *ARM architecture reference manual*. Prentice Hall, 2018.

[182] T. Instruments, “TMS320C67x/C67x+ DSP CPU and Instruction Set Reference Guide, November 2006,” *Literature Number: SPRU733A*, 2006.

[183] M. Jayapala, F. Barat, P. O. De Beeck, F. Catthoor, G. Deconinck, and H. Corporaal, “A low energy clustered instruction memory hierarchy for long instruction word processors,” in *International Workshop on Power and Timing Modeling, Optimization and Simulation*, pp. 258–267, Springer, 2002.

[184] T. Instruments, “TMS320C62x/C67x Power Consumption Summary,” 2004.

[185] R. S. Bajwa, M. Hiraki, H. Kojima, D. J. Gorny, K. Nitta, A. Shridhar, K. Seki, and K. Sasaki, “Instruction buffering to reduce power in processors for signal processing,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 5, pp. 417–424, Dec 1997.

[186] R. Chen, “The Itanium Processor,” tech. rep., Microsoft, 2015.

[187] Intel, *Intel Itanium Architecture: Software Developer’s Manual Volume 3: Intel Itanium Instruction Set Reference*. Intel.

[188] R. B. Lee and A. M. Fiskiran, “PLX: A fully subword-parallel instruction set architecture for fast scalable multimedia processing,” in *Proceedings. IEEE International Conference on Multimedia and Expo*, vol. 2, pp. 117–120, IEEE, 2002.

[189] B. Valentine and O. Sohm, “Optimizing the JPEG2000 binary arithmetic encoder for VLIW architectures,” in *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, pp. V–117, IEEE, 2004.

[190] C. J. Hughes, “Single-instruction multiple-data execution,” *Synthesis Lectures on Computer Architecture*, vol. 10, no. 1, pp. 1–121, 2015.

[191] L. Gerlach, G. Payá-Vayá, and H. Blume, “Efficient Emulation of Floating-Point Arithmetic on Fixed-Point SIMD Processors,” in *Signal Processing Systems (SiPS), 2016 IEEE International Workshop on*, pp. 254–259, IEEE, 2016.

[192] C. Iordache and P. T. P. Tang, “An Overview of Floating-point Support and Math Library on the Intel/spl reg/XScale/spl trade/architecture,” in *Proceedings 2003 16th IEEE Symposium on Computer Arithmetic*, pp. 122–128, IEEE, 2003.

[193] F. Giesemann, L. Gerlach, and G. Payá-Vayá, “Evolutionary Algorithms for Instruction Scheduling, Operation Merging, and Register Allocation in VLIW Compilers,” *Journal of Signal Processing Systems*, 2020.

[194] K. Han, J. Ahn, and K. Choi, “Power-Efficient Predication Techniques for Acceleration of Control Flow Execution on CGRA,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 2, p. 8, 2013.

[195] MATLAB, *CORDIC-based approximation of cosine*. The MathWorks Inc.

[196] K. Diefendorff and P. K. Dubey, “How multimedia workloads will change processor design,” *Computer*, vol. 30, no. 9, pp. 43–45, 1997.

[197] G. Popelka, B. Moore, R. Fay, and A. Popper, *Hearing Aids*. Springer Handbook of Auditory Research, Springer International Publishing, 2016.

[198] M. Pedram and J. M. Rabaey, *Power aware design methodologies*. Springer Science & Business Media, 2002.

[199] W. T. Padgett and D. V. Anderson, “Fixed-point signal processing,” *Synthesis Lectures on Signal Processing*, vol. 4, no. 1, pp. 1–133, 2009.

[200] B. Wu, J. Zhu, and F. N. Najm, “Dynamic-range estimation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 9, pp. 1618–1636, 2006.

[201] Y. Cao and H. Yasuura, “A system-level energy minimization approach using datapath width optimization,” in *ISLPED’01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design (IEEE Cat. No. 01TH8581)*, pp. 231–236, IEEE, 2001.

[202] T. T. Hoang and P. Larsson-Edefors, “Data-width-driven power gating of integer arithmetic circuits,” in *2012 IEEE Computer Society Annual Symposium on VLSI*, pp. 237–242, IEEE, 2012.

[203] O. Ergin, D. Balkan, K. Ghose, and D. Ponomarev, “Register packing: Exploiting narrow-width operands for reducing register file pressure,” in *37th International Symposium on Microarchitecture (MICRO-37’04)*, pp. 304–315, IEEE, 2004.

[204] G. H. Loh, “Exploiting data-width locality to increase superscalar execution bandwidth,” in *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings.*, pp. 395–405, IEEE, 2002.

[205] M. Weißbrich, “Implementierung und Evaluierung eines generischen Datenpfades für einen VLIW-SIMD Hörgeräteprozessor,” Master’s thesis, Leibniz Universität Hannover, 2016.

[206] J. Tomarakos and C. Duggan, “32-Bit SIMD SHARC architecture digital audio signal processing applications,” *Journal of the Audio Engineering Society*, vol. 48, no. 3, pp. 220–229, 2000.

[207] L. Gerlach, G. Payá-Vayá, S. Liu, M. Weißbrich, H. Blume, D. Marquardt, and S. Doclo, "Analyzing the Trade-Off between Power Consumption and Beamforming Algorithm Performance using a Hearing Aid ASIP," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2017 International Conference on*, pp. 88–96, IEEE, 2017.

[208] C. F. Fang, R. A. Rutenbar, and T. Chen, "Fast, accurate static analysis for fixed-point finite-precision effects in DSP designs," in *Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, p. 275, IEEE Computer Society, 2003.

[209] A. V. Oppenheim and C. J. Weinstein, "Effects of finite register length in digital filtering and the fast Fourier transform," *Proceedings of the IEEE*, vol. 60, no. 8, pp. 957–976, 1972.

[210] N. S. Hockley, F. Bahlmann, and B. Fulton, "Analog-to-digital conversion to accommodate the dynamics of live music in hearing instruments," *Trends in Amplification*, vol. 16, no. 3, pp. 146–158, 2012.

[211] A. Belov, "Layout Implementation and Evaluation of an ASIP-Based Hearing Aid System," Master's thesis, Leibniz Universität Hannover, 2015.

[212] G. Payá-Vayá, J. Martín-Langerwerf, H. Blume, and P. Pirsch, "A Forwarding-sensitive Instruction Scheduling Approach to Reduce Register File Constraints in VLIW Architectures," in *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pp. 151–158, IEEE, 2010.

[213] N. Goel, A. Kumar, and P. R. Panda, "Power reduction in VLIW processor with compiler driven bypass network," in *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*, pp. 233–238, IEEE, 2007.

[214] M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, and R. Zafalon, "Low-power data forwarding for VLIW embedded architectures," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 10, no. 5, pp. 614–622, 2002.

[215] V. Guzma, T. Pitkänen, and J. Takala, "Use of compiler optimization of software bypassing as a method to improve energy efficiency of exposed data path architectures," *EURASIP Journal on Embedded Systems*, vol. 2013, no. 1, p. 9, 2013.

[216] D. Ponomarev, G. Kucuk, O. Ergin, and K. Ghose, "Reducing Datapath Energy Through the Isolation of Short–Lived Operands," in *2003 12th International Conference on Parallel Architectures and Compilation Techniques*, pp. 258–268, IEEE, 2003.

[217] Z. Hu and M. Martonosi, "Reduing Register File Power Consumption by Exploiting Value Lifetime Charateristis," in *in Workshop on Complexity-Effective Design (WCED*, Citeseer, 2000.

[218] M. Munch, B. Wurth, R. Mehra, J. Sproch, and N. Wehn, “Automating RT-level operand isolation to minimize power consumption in datapaths,” in *Proceedings Design, Automation and Test in Europe Conference and Exhibition 2000 (Cat. No. PR00537)*, pp. 624–631, IEEE, 2000.

[219] A. Correale Jr, “Overview of the power minimization techniques employed in the IBM PowerPC 4xx embedded controllers,” in *Proceedings of the 1995 international symposium on Low power design*, pp. 75–80, ACM, 1995.

[220] H. Dillon, *Hearing aids.* Hodder Arnold, 2008.

[221] V. Hohmann, “Frequency analysis and synthesis using a Gammatone filterbank,” *Acta Acustica united with Acustica*, vol. 88, no. 3, pp. 433–442, 2002.

[222] T. Herzke, H. Kayser, F. Loshaj, G. Grimm, and V. Hohmann, “Open signal processing software platform for hearing aid research (openMHA),” in *Proceedings of the Linux Audio Conference*, pp. 35–42, 2017.

[223] J. Kießling, “Achievements and limitations of modern hearing instruments,” tech. rep., Justus-Liebig-Universität Gießen, 2014.

[224] Oticon, *Oticon Product Guide 2016/2017*.

[225] J. DiCristina, “Introduction to hearing aids and important design considerations,” *Maxim Integrated, AppNote-4691*, 2010.

[226] Freescale Semiconductor, Inc., *Enhanced Serial Audio Interface (ESAI)* , 2004.

[227] J. Tomarakos, “Interfacing the ADSP-21161 SIMD SHARC DSP to the AD1836 (24-bit/96kHz) Multichannel Codec,” *Analog Devices*, 2001.

[228] E. Van Tuijl, J. van den Homberg, D. Reefman, C. Bastiaansen, and L. van der Dussen, “A 128fs Multi-bit Sigma Delta CMOS audio DAC with real-time DEM and 115dB SFDR,” in *2004 IEEE International Solid-State Circuits Conference (IEEE Cat. No. 04CH37519)*, pp. 368–369, IEEE, 2004.

[229] J. Lewis, “Common inter-IC digital interfaces for audio data transfer,” *EDN-Electronic Design News*, vol. 57, no. 16, p. 46, 2012.

[230] Philips Semiconductors, “I2S Bus Specification [S/OL]. 1996.”

[231] G. Jia, “An I2S (Inter-IC Sound Bus) Application on Kinetis,” *Automotive and Industrial Solutions Group*, 2012.

[232] D. TMS320C6000, *Multichannel Audio Serial Port (McASP) Reference Guide*. www.ti.com, 2005.

[233] A. McPherson and V. Zappi, “An environment for submillisecond-latency audio and sensor processing on BeagleBone Black,” in *Audio Engineering Society Convention 138*, Audio Engineering Society, 2015.

[234] Freescale Semiconductor, Inc., *K61 Sub-Family*. Freescale Semiconductor, Inc., https://www.freescale.com, 2015.

[235] M. Galda and C. Ro
vznov, "Audio Output Options for Kinetis," *Freescale Semiconductor Document Number: AN4369 Application Note*, 2012.

[236] Texas Instruments Inc., *Tuning Audio Latency on C6747*. http://processors.wiki.ti.com, 2014.

[237] X. Aragones, J. L. Gonzalez, and A. Rubio, *Analysis and solutions for switching noise coupling in mixed-signal ICs*. Springer Science & Business Media, 2013.

[238] T. Instruments, *TMS320C642x DSP Multichannel Buffered Serial Port (McBSP) Interface*. www.ti.com, 2007.

[239] Analog Devices Inc., "Blackfin Embedded Processor: ADSP-BF516," tech. rep., ADSP-BF516 Datasheet. Januray, 2014.

[240] I. K. Chayleva, M. A. Botev, V. P. Dobreva, and B. B. Petrov, "Methods and Techniques for real-time audio data streaming to and from high capacity local DSP SDRAM memory," *E-University TU Sofia*, 2012.

[241] L. Jingjiao, R. Chaoqun, and D. Lifang, "Audio Interface Based on eDMA and I2S of Kinetis," *Microcontrollers and Embedded Systems*, vol. 1, p. 014, 2013.

[242] T. Instruments, "TLV320AIC3106 Low-Power Stereo Audio CODEC for Portable Audio/Telephony," *Rev. F*, vol. 24, 2014.

[243] T. Instruments, "TLV320AIC3104 Low-Power Stereo Audio CODEC for Portable Audio/Telephony," *Rev. F*, vol. 24, 2016.

[244] Analog Devices Inc., "SigmaDSP Stereo, Low Power, 96 kHz, 24-Bit Audio Codec with Integrated PLL," *ADAU1761, Data Sheet*.

[245] D. S. Reay, *Digital signal processing and applications with the OMAP-L138 eXperimenter*. John Wiley & Sons, 2012.

[246] NXP Semiconductors N.V., "UM10204 I2C-bus specification and user manual," *User Manual*, vol. 4, 2014.

[247] J. J. Digiovanni, *Hearing Aid Handbook*. Cengage Learning, 2010.

[248] L. Gerlach, G. Payá-Vayá, and H. Blume, "A Low Latency Multichannel Audio Interface for Low Power SIMD Digital Signal Processors," in *ICT.OPEN 2016, ISBN: 978-90-73461-932*, 2016.

[249] F. Giesemann, G. Payá-Vayá, L. Gerlach, H. Blume, F. Pflug, and G. von Voigt, "Using a Genetic Algorithm Approach to Reduce Register File Pressure during Instruction Scheduling," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2017 International Conference on*, pp. 179–187, IEEE, 2017.

[250] J. A. Fisher, *The Optimization of Horizontal Microcode within and beyond Basic Blocks: An Application of Processor Scheduling with Resources*. PhD thesis, Courant Mathematics and Computing Laboratory U.S. Department of Energy, USA, 1979. AAI8010348.

[251] G. Paya-Vaya, J. Martin-Langerwerf, P. Taptimthong, and P. Pirsch, "Design Space Exploration of Media Processors: A Parameterized Scheduler," in *2007 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pp. 41–49, July 2007.

[252] L. Zhang, X. Wu, and Y. Zhao, "Instruction-Level Instantaneous Power Modeling for VLIW Processor," in *Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), 2015 IEEE 12th Intl Conf on*, pp. 1451–1456, IEEE, 2015.

[253] H. Blume and S. Hesselbarth, "Methoden zur applikationsspezifischen Verlustleitungsoptimierung für eingebettete Prozessoren," *15. ITG-Fachtagung für Elektronische Medien, 26.-27. Februar, Dortmund*, 2013.

[254] M. Wendt, M. Grumer, C. Steger, R. Weiss, U. Neffe, and A. Muehlberger, "Tool for automated instruction set characterization for software power estimation," *IEEE transactions on instrumentation and measurement*, vol. 59, no. 1, pp. 84–91, 2010.

[255] M. Wendt, M. Grumer, C. Steger, R. Weiss, U. Neffe, and A. Muhlberger, "Energy consumption measurement technique for automatic instruction set characterization of embedded processors," in *Instrumentation and Measurement Technology Conference Proceedings, 2007. IMTC 2007. IEEE*, pp. 1–4, IEEE, 2007.

[256] W. Wang, *An improved instruction-level power and energy model for RISC microprocessors*. PhD thesis, University of Southampton, 2017.

[257] J. Blauert, *The technology of binaural listening*. Springer, 2013.

[258] S. Doclo, W. Kellermann, S. Makino, and S. E. Nordholm, "Multichannel signal enhancement algorithms for assisted listening devices: Exploiting spatial diversity using multiple microphones," *IEEE Signal Processing Magazine*, vol. 32, no. 2, pp. 18–30, 2015.

[259] S. Doclo, S. Gannot, M. Moonen, and A. Spriet, "Acoustic beamforming for hearing aid applications," *Handbook on Array Processing and Sensor Networks, Wiley IEEE Press*, 2010.

[260] G. W. Elko, "Differential microphone arrays," in *Audio signal processing for next-generation multimedia communication systems*, pp. 11–65, Springer, 2004.

[261] S. Haykin, *Adaptive Filter Theory*. Always learning, Pearson, 2014.

[262] R. Baumgartel, M. Krawczyk-Becker, D. Marquardt, C. Volker, H. Hu, T. Herzke, and M. Dietz, “Comparing binaural pre-processing strategies I: Instrumental evaluation,” *Trends in Hearing*, vol. 19, pp. 1–16, 2015.

[263] H. Kayser, S. D. Ewert, J. Anemüller, T. Rohdenburg, V. Hohmann, and B. Kollmeier, “Database of Multichannel In-Ear and Behind-the-Ear Head-Related and Binaural Room Impulse Responses,” *EURASIP J. on Advances in Signal Processing*, vol. 2009, p. 6, 2009.

[264] K. Wagener, T. Brand, and B. Kollmeier, “Entwicklung und Evaluation eines Satztests für die deutsche Sprache III: Evaluation des Oldenburger Satztests [Development and Evaluation of a Sentence Test for the German Language III: Evaluation of the Oldenburg Sentence Test],” *Zeitschrift für Audiologie 1999c; 38: 86*, vol. 95, 1999.

[265] R. P. ITU-T, “862-perceptual evaluation of speech quality (PESQ): an objective method for end-to-end speech quality assessment of narrow-band telephone networks and speech codecs,” *International Telecommunication Union-Telecommunication Standardisation Sector*, 2001.

[266] C. H. Taal, R. C. Hendriks, R. Heusdens, and J. Jensen, “An evaluation of objective measures for intelligibility prediction of time-frequency weighted noisy speech,” *The J. of the Acoustical Society of America*, vol. 130, no. 5, pp. 3013–3027, 2011.

[267] J. Greenberg, P. Peterson, and P. Zurek, “Intelligibility-weighted measures of speech-to-interference ratio and speech system performance,” *The J. of the Acoustical Society of America*, vol. 94, no. 5, pp. 3009–3010, 1993.

[268] A. Frias Velazquez, R. d. J. Romero-Troncoso, A. Pizurica, and W. Philips, “Exact LMS learning curve analysis under finite word length effects,” in *20th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2009)*, pp. 218–222, STW Technology Foundation, 2009.

[269] R. . SCHWARZ, *AUDIO ANALYZER R&S UPL/UPL16/UPL66*. RHODE & SCHWARZ, 2011.

[270] Cadence Design Systems, Inc., *Xtensa LX7 Processor High-performance, configurable, and extensible controllers and DSPs*, 2016.

[271] Cadence Design Systems, Inc., *Tensilica Fusion G DSP Family Multi-purpose, fixed- and floating-point DSP with exceptional out-of-the-box performance*, 2017.

[272] Texas Instruments Inc., *Introduction to TMS320C6000 DSP Optimization*. Texas Instruments Inc., www.ti.com, 2011.

[273] J. Karrenbauer, L. Gerlach, G. Payá-Vayá, and H. Blume, “Design Space Exploration Framework for Tensilica-Based Digital Audio Processors in Hearing Aids,” in *2020 9th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pp. 1–6, IEEE, 2020.

[274] Texas Instruments Inc., *TMS320C6000 Integer Division*. Texas Instruments Inc., www.ti.com, 2000.

[275] Texas Instruments Inc., *C6748/46/42 power consumption summary*. Texas Instruments Inc., www.ti.com, 2019.

[276] M. Berouti, R. Schwartz, and J. Makhoul, "Enhancement of speech corrupted by acoustic noise," in *ICASSP'79. IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 4, pp. 208–211, IEEE, 1979.

[277] Y. Cai, J. Yuan, X. Ma, and C. Hou, "Low power embedded speech enhancement system based on a fixed-point DSP," in *2009 Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing*, pp. 132–136, IEEE, 2009.

[278] Texas Instruments Inc., *TMS320VC5503/C5506/C5507/C5509A Power Consumption Summary*. Texas Instruments Inc., www.ti.com, 2008.

[279] T. May, *Binaural Scene Analysis: Localization, Detection and Recognition of Speakers in Complex Acoustic Scenes*. PhD thesis, Technical University of Denmark, 2012. PhD thesis.

[280] T. Volkmar, "Analyse und Evaluation eines binauralen Lokalisationsalgorithmus auf einem VLIW-SIMD Prozessor," Master's thesis, Leibniz Universität Hannover, 2016.

[281] J. Blauert and S. Hearing, "The psychophysics of human sound localization," in *Spatial Hearing*, MIT Press, 1997.

[282] R. F. Lyon, "The All-Pole Gammatone Filter and Auditory Models," in *Acustica*, Citeseer, 1996.

[283] M. Pflueger, R. Hoeldrich, and W. Riedler, "Nonlinear all-pole and one-zero gammatone filters," *Acta Acustica united with Acustica*, vol. 84, no. 3, pp. 513–519, 1998.

[284] M. Slaney *et al.*, "An efficient implementation of the Patterson-Holdsworth auditory filter bank," *Apple Computer, Perception Group, Tech. Rep*, vol. 35, no. 8, 1993.

[285] L. Solbach, R. Wöhrmann, and J. Kliewer, "The complex-valued continuous wavelet transform as a preprocessor for auditory scene analysis," *Computational auditory scene analysis*, pp. 273–292, 1998.

[286] P. Johannesma, "The pre-response stimulus ensemble of neurons in the cochlear nucleus," in *Symposium on Hearing Theory, 1972*, IPO, 1972.

[287] P. EhKan, T. Allen, and S. F. Quigley, "FPGA Implementation for GMM-BasedSpeaker Identification," *International Journal of Reconfigurable Computing*, vol. 2011, 2011.

[288] N. Behmann, C. Seifert, G. Paya-Vaya, H. Blume, P. Jääskeläinen, J. Multanen, H. Kultala, J. Takala, J. Thiemann, and S. van de Par, "Customized high performance low power processor for binaural speaker localization," in *2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 392–395, IEEE, 2016.

[289] H. Hermansky, E. Variani, and V. Peddinti, “Mean temporal distance: Predicting ASR error from temporal properties of speech signal,” in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 7423–7426, IEEE, 2013.

[290] B. T. Meyer, S. H. Mallidi, A. M. C. Martinez, G. Payá-Vayá, H. Kayser, and H. Hermansky, “Performance monitoring for automatic speech recognition in noisy multichannel environments,” in *2016 IEEE Spoken Language Technology Workshop (SLT)*, pp. 50–56, IEEE, 2016.

[291] V. Tiwari and N. Khare, “Hardware implementation of neural network with Sigmoidal activation functions using CORDIC,” *Microprocessors and Microsystems*, vol. 39, no. 6, pp. 373–381, 2015.

[292] R. B. Lee, “Efficiency of microSIMD architectures and index-mapped data for media processors,” in *Electronic Imaging’99*, pp. 34–46, International Society for Optics and Photonics, 1998.

[293] L. Gerlach, G. Payá-Vayá, and H. Blume, “A Survey on Application Specific Processor Architectures for Digital Hearing Aids,” *Journal of Signal Processing Systems*, 2020.

[294] M. Weißbrich, L. Gerlach, H. Blume, A. Najafi, A. García-Ortiz, and G. Payá-Vayá, “FLINT+: A runtime-configurable emulation-based stochastic timing analysis framework,” *Integration*, 2019.

[295] L. Gerlach, F. Stuckmann, H. Blume, and G. Payá-Vayá, “Issue-Slot Based Predication Encoding Technique for VLIW Processors,” in *2020 9th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pp. 1–6, IEEE, 2020.

[296] M. Weißbrich, G. Payá-Vayá, L. Gerlach, H. Blume, A. Najafi, and A. García-Ortiz, “FLINT+: A runtime-configurable emulation-based stochastic timing analysis framework,” in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pp. 1–8, IEEE, 2017.

[297] R. Nowosielski, L. Gerlach, S. Bieband, G. Payá-Vayá, and H. Blume, “FLINT: Layout-oriented FPGA-based methodology for fault tolerant ASIC design,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, pp. 297–300, IEEE, 2015.

[298] R. Nowosielski, L. Gerlach, G. Payá-Vayá, S. Hesselbarth, and H. Blume, “Methodology for Observation and Evaluation of Fault Tolerance Implementations inside High Temperature ASICs,” in *ICT.OPEN 2013*, 11 2013.

[299] H. Blume, G. Payá Vayá, and L. Gerlach, “KAVUAKA : Chip Design für digitale Hörhilfen,” *Unimagazin 1/2 (2020)*, 2020.

[300] L. Gerlach, G. Payá-Vayá, and H. Blume, “Europractice Activity Report 2018-2019: The KAVUAKA Hearing Aid Processor,” tech. rep., Europractice, 7 Mrz 2019.

[301] J. Karrenbauer, L. Gerlach, G. Payá-Vayá, and H. Blume, “Automated Design Space Exploration of Digital Audio Processors for Hearing Aids,” 2019.

[302] L. Gerlach, G. Payá-Vayá, and H. Blume, “High-Performance, Low Power digital hearing aid ASIP/ASIC,” 2019. Tensilica Day—Trends in Modern Design of Configurable Processors 2019, Hannover, Germany.

[303] L. Gerlach, C. Seifert, G. Payá-Vayá, and H. Blume, “Real-Time Implementation of a GMM-Based Binaural Localization Algorithm on a Low Power Hearing Aid System,” 2018. Wirtschaftsempfang der UVN und der Leibniz Universität Hannover.

[304] L. Gerlach, G. Payá-Vayá, and H. Blume, “Analyzing the Trade-Off between Power Consumption and Beamforming Algorithm Performance using a Hearing Aid ASIP,” 2018. Tensilica Day—Trends in Modern Design of Configurable Processors 2018.

[305] L. Gerlach, C. Seifert, G. Payá-Vayá, and H. Blume, “Real-Time Implementation of a GMM-Based Binaural Localization Algorithm on a Low Power Hearing Aid System,” 2018. Wirtschaftsempfang der UVN und der Leibniz Universität Hannover.

[306] L. Gerlach, C. Seifert, G. Payá-Vayá, and H. Blume, “Real-Time Implementation of a GMM-Based Binaural Localization Algorithm on a Low Power Hearing Aid System,” 2018. Tag der Fakultät - Die akademische Jahresfeier.

[307] L. Gerlach, C. Seifert, G. Payá-Vayá, and H. Blume, “Real-Time Implementation of a GMM-Based Binaural Localization Algorithm on a Low Power Hearing Aid System,” 2018. Leibniz-Symposium “Maschinelles Lernen – Intelligente Digitalisierung”.

[308] G. Payá-Vayá, L. Gerlach, and H. Blume, “The KAVUAKA Hearing Aid Processor,” 2018. Tensilica Day—Trends in Modern Design of Configurable Processors 2018, Hannover, Germany.

[309] L. Gerlach, G. Payá-Vayá, and H. Blume, “Low-Power Optimization of a VLIW-SIMD ASIP for Hearing Aid Devices,” 2017. Tensilica Day—Trends in Modern Design of Configurable Processors 2017, Hannover, Germany.

[310] L. Gerlach, C. Seifert, G. Payá-Vayá, and H. Blume, “Instruction-Set Extension based on a 2D Sound Source Localization Algorithm on a Low Power Hearing Aid System,” 2016. Tensilica Day—Trends in Modern Design of Configurable Processors 2016, Hannover, Germany.

[311] L. Gerlach, G. Payá-Vayá, and H. Blume, “FPGA Based Rapid Prototyping for Exploring and Optimizing Hearing Aid Processors,” in *10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2015)*, 2015.

[312] G. Payá-Vayá, L. Gerlach, R. Nowosielski, and H. Blume, “FLINT: Layout-Oriented FPGA-Based Methodology for Fault Tolerant ASIC Design,” in *10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2015), Bremen, Germany*, 2015.

List of the Author's Publications

Journal Publications

- L. Gerlach, G. Payá-Vayá, and H. Blume, "A Survey on Application Specific Processor Architectures for Digital Hearing Aids," *Journal of Signal Processing Systems*, 2020
- A. M. C. Martinez, L. Gerlach, G. Payá-Vayá, H. Hermansky, J. Ooster, and B. T. Meyer, "DNN-based performance measures for predicting error rates in automatic speech recognition and optimizing hearing aid parameters," *Speech Communication*, vol. 106, pp. 44–56, 2019
- M. Weißbrich, L. Gerlach, H. Blume, A. Najafi, A. García-Ortiz, and G. Payá-Vayá, "FLINT+: A runtime-configurable emulation-based stochastic timing analysis framework," *Integration*, 2019
- F. Giesemann, L. Gerlach, and G. Payá-Vayá, "Evolutionary Algorithms for Instruction Scheduling, Operation Merging, and Register Allocation in VLIW Compilers," *Journal of Signal Processing Systems*, 2020

Conference Publications

- L. Gerlach, F. Stuckmann, H. Blume, and G. Payá-Vayá, "Issue-Slot Based Predication Encoding Technique for VLIW Processors," in *2020 9th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pp. 1–6, IEEE, 2020
- J. Karrenbauer, L. Gerlach, G. Payá-Vayá, and H. Blume, "Design Space Exploration Framework for Tensilica-Based Digital Audio Processors in Hearing Aids," in *2020 9th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pp. 1–6, IEEE, 2020
- L. Gerlach, G. Payá-Vayá, and H. Blume, "KAVUAKA: A Low Power Application Specific Hearing Aid Processor," in *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 99–104, Oct 2019
- L. Gerlach, G. Payá-Vayá, S. Liu, M. Weißbrich, H. Blume, D. Marquardt, and S. Doclo, "Analyzing the Trade-Off between Power Consumption and Beamforming Algorithm Performance using a Hearing Aid ASIP," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2017 International Conference on*, pp. 88–96, IEEE, 2017
- C. Seifert, J. Thiemann, L. Gerlach, T. Volkmar, G. Payá-Vayá, H. Blume, and S. van de Par, "Real-time implementation of a GMM-based binaural localization algorithm on a VLIW-SIMD processor," in *Multimedia and Expo (ICME), 2017 IEEE International Conference on*, pp. 145–150, IEEE, 2017

- F. Giesemann, G. Payá-Vayá, L. Gerlach, H. Blume, F. Pflug, and G. von Voigt, “Using a Genetic Algorithm Approach to Reduce Register File Pressure during Instruction Scheduling,” in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2017 International Conference on*, pp. 179–187, IEEE, 2017
- M. Weißbrich, G. Payá-Vayá, L. Gerlach, H. Blume, A. Najafi, and A. García-Ortiz, “FLINT+: A runtime-configurable emulation-based stochastic timing analysis framework,” in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pp. 1–8, IEEE, 2017
- L. Gerlach, G. Payá-Vayá, and H. Blume, “Efficient Emulation of Floating-Point Arithmetic on Fixed-Point SIMD Processors,” in *Signal Processing Systems (SiPS), 2016 IEEE International Workshop on*, pp. 254–259, IEEE, 2016
- L. Gerlach, G. Payá-Vayá, and H. Blume, “A Low Latency Multichannel Audio Interface for Low Power SIMD Digital Signal Processors,” in *ICT.OPEN 2016, ISBN: 978-90-73461-932*, 2016
- L. Gerlach, G. Payá-Vayá, and H. Blume, “An Area Efficient Real- and Complex-Valued Multiply-Accumulate SIMD Unit for Digital Signal Processors,” in *Signal Process. Systems (SiPS), 2015 IEEE Workshop on*, pp. 1–6, IEEE, 2015
- R. Nowosielski, L. Gerlach, S. Bieband, G. Payá-Vayá, and H. Blume, “FLINT: Layout-oriented FPGA-based methodology for fault tolerant ASIC design,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, pp. 297–300, IEEE, 2015
- J. Hartig, L. Gerlach, G. Payá-Vayá, and H. Blume, “Customizing a VLIW-SIMD Application-Specific Instruction-Set Processor for Hearing Aid Devices,” in *Signal Process. Syst. (SiPS), 2014 IEEE Workshop on*, pp. 1–6, IEEE, 2014
- R. Nowosielski, L. Gerlach, G. Payá-Vayá, S. Hesselbarth, and H. Blume, “Methodology for Observation and Evaluation of Fault Tolerance Implementations inside High Temperature ASICs,” in *ICT.OPEN 2013*, 11 2013

Further Publications

- H. Blume, G. Payá Vayá, and L. Gerlach, “KAVUAKA : Chip Design für digitale Hörhilfen,” *Unimagazin 1/2 (2020)*, 2020
- L. Gerlach, G. Payá-Vayá, and H. Blume, “Europractice Activity Report 2018-2019: The KAVUAKA Hearing Aid Processor,” tech. rep., Europractice, 7 Mrz 2019
- J. Karrenbauer, L. Gerlach, G. Payá-Vayá, and H. Blume, “Automated Design Space Exploration of Digital Audio Processors for Hearing Aids,” 2019

- L. Gerlach, G. Payá-Vayá, and H. Blume, “High-Performance, Low Power digital hearing aid ASIP/ASIC,” 2019. Tensilica Day—Trends in Modern Design of Configurable Processors 2019, Hannover, Germany
- L. Gerlach, C. Seifert, G. Payá-Vayá, and H. Blume, “Real-Time Implementation of a GMM-Based Binaural Localization Algorithm on a Low Power Hearing Aid System,” 2018. Wirtschaftsempfang der UVN und der Leibniz Universität Hannover
- L. Gerlach, G. Payá-Vayá, and H. Blume, “Analyzing the Trade-Off between Power Consumption and Beamforming Algorithm Performance using a Hearing Aid ASIP,” 2018. Tensilica Day—Trends in Modern Design of Configurable Processors 2018
- L. Gerlach, C. Seifert, G. Payá-Vayá, and H. Blume, “Real-Time Implementation of a GMM-Based Binaural Localization Algorithm on a Low Power Hearing Aid System,” 2018. Wirtschaftsempfang der UVN und der Leibniz Universität Hannover
- L. Gerlach, C. Seifert, G. Payá-Vayá, and H. Blume, “Real-Time Implementation of a GMM-Based Binaural Localization Algorithm on a Low Power Hearing Aid System,” 2018. Tag der Fakultät - Die akademische Jahresfeier
- L. Gerlach, C. Seifert, G. Payá-Vayá, and H. Blume, “Real-Time Implementation of a GMM-Based Binaural Localization Algorithm on a Low Power Hearing Aid System,” 2018. Leibniz-Symposium “Maschinelles Lernen – Intelligente Digitalisierung”
- G. Payá-Vayá, L. Gerlach, and H. Blume, “The KAVUAKA Hearing Aid Processor,” 2018. Tensilica Day—Trends in Modern Design of Configurable Processors 2018, Hannover, Germany
- L. Gerlach, G. Payá-Vayá, and H. Blume, “Low-Power Optimization of a VLIW-SIMD ASIP for Hearing Aid Devices,” 2017. Tensilica Day—Trends in Modern Design of Configurable Processors 2017, Hannover, Germany
- L. Gerlach, S. Nolting, H. Blume, G. Payá-Vayá, H. Stolberg, and C. Reuter, “A Highly Optimized Arithmetic Software Library and Hardware Co-processor IP for Fixed-Point VLIW-SIMD Processor Architectures,” in *Technology Transfer in Computing Systems (TETRACOM Technology Transfer Project (TTP), 2016), Prague, Czech Republic*, 2016
- L. Gerlach, C. Seifert, G. Payá-Vayá, and H. Blume, “Instruction-Set Extension based on a 2D Sound Source Localization Algorithm on a Low Power Hearing Aid System,” 2016. Tensilica Day—Trends in Modern Design of Configurable Processors 2016, Hannover, Germany
- L. Gerlach, G. Payá-Vayá, and H. Blume, “FPGA Based Rapid Prototyping for Exploring and Optimizing Hearing Aid Processors,” in *10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2015)*, 2015
- G. Payá-Vayá, L. Gerlach, R. Nowosielski, and H. Blume, “FLINT: Layout-Oriented FPGA-Based Methodology for Fault Tolerant ASIC Design,” in *10th International Sym-*

posium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2015), Bremen, Germany, 2015

Lebenslauf

Persönliche Angaben:

Name: Lukas Gerlach
Geburtsdatum, -ort: 12.01.1988, Duisburg

Wissenschaftlicher Werdegang:

2008 – 2013	Studium der Elektrotechnik (Diplom) an der Leibniz Universität Hannover
2010 – 2012	Hilfswissenschaftliche Tätigkeit, Institut für Grundlagen der Elektro- und Informationstechnik (GEM), Leibniz Universität Hannover
2012	Studienarbeit: Erweiterung einer Datenstruktur zur Verwaltung von 3D-Floorplans und eines Optimierungsalgorithmus um die Berücksichtigung von Softmodules
2013	Diplomarbeit: Konzipierung einer Designmethodologie für fehlertolerante Hochtemperatur-ASICs
2013 – 2020	Wissenschaftlicher Mitarbeiter und Promotionsstudium, Institut für Mikroelektronische Systeme (IMS), Leibniz Universität Hannover