The WaveCore - A Scalable Architecture for Real-time Audio Processing

Math Verstraelen

Members of the Graduation committee:

Prof. dr. P. M. G. Apers	University of Twente (chairman and secretary)
Prof. dr. ir. G. J. M. Smit	University of Twente (promotor)
dr. ir. J. Kuper	University of Twente (assistant-promotor)
Prof. dr. R. Bader	Universität Hamburg
Prof. dr. ir. C. H. Slump	University of Twente
Prof. dr. ir. M. J. G. Bekooij	University of Twente
dr. ir. A. B. J. Kokkeler	University of Twente
Prof. dr. ir. B. de Vries	Eindhoven University of Technology

UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics and Computer Science, Computer Architecture for Embedded Systems (CAES) group

Copyright © 2017 Math Verstraelen, Horst, The Netherlands.

This thesis was typeset using $\text{\LaTeX}$, TikZ, and Notepad++. This thesis was printed by Gildeprint Drukkerijen, The Netherlands.

ISBN 978-90-365-4240-1
DOI 10.3990/1.9789036542401

THE WAVECORE - A SCALABLE ARCHITECTURE FOR REAL-TIME AUDIO PROCESSING

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. T.T.M. Palstra
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 20 januari 2017 om 12.45 uur

door

Martinus Johannes Wilhelmina Verstraelen

geboren op 27 november 1967
te Horst

Dit proefschrift is goedgekeurd door:

Prof. dr. ir. G. J. M. Smit (promotor)
dr. ir. J. Kuper (assistant-promotor)

*Voor mijn ouders:
Jeu Verstraelen
Mia Verstraelen-Weijs*

ABSTRACT

vii

The subject of this thesis is real-time music/acoustical signal synthesis and processing, using a scalable domain-specific processor architecture. Basically there are two application classes within this domain. The first class is digitization of analog audio effects (e.g. musical instrument effects). Examples are reverberation (i.e. simulation of an acoustical space like concert hall), amplifier emulation and modulation effects like phasing and flanging. The second class is synthesis of sound for electronic musical instruments (e.g. keyboard, virtual piano, or sound rendering systems in virtual reality).

These applications consist of two distinct parts: real-time physical models and user interaction models. An interaction model embodies the interface between the user and the physical model. A physical model is a mathematical abstraction of a physical object, which needs to be executed in real-time. This is the actual signal processing/synthesis part of the application. We focus on three classes of physical models. The first class is virtual analog modeling. These models are based on a mathematical abstraction of an analog signal processing device. The second class is digital waveguide modeling. This class of modeling is based on a coarse abstraction of an acoustical object. The delay-line plays a central role within digital waveguide modeling. The third class is detailed physical modeling of musical instruments and acoustical spaces. This class is the most computational intensive.

From a computational perspective, a physical model has different requirements compared to an interaction model. The most challenging requirements for a physical model are low latency (i.e. elapsed time between signal input and processed output), time predictability of real-time execution and scalability. For interaction models the most important requirements are flexibility and best-effort execution. These are conflicting requirements, which lead to problems in the state-of-the-art approaches within the music/acoustical signal processing application domain.

The following contributions of this thesis address these problems:

- A scalable many-core (i.e. MIMD) processor: the *WaveCore*. Specific aspects of the *WaveCore* are a highly optimized instruction-set with domain-specific elements, and explicit delay-line support. The *WaveCore* is optimized to locality of reference, leading to a high degree of time-predictability of execution and low processing latency.
- A heterogeneous processor architecture, consisting of a general purpose pro-

cessor for executing interaction models and the *WaveCore* processor, for executing physical models.

- A programming model for this heterogeneous processor architecture, which is based on a data-flow concept. This programming model is conceptually close to the mathematical nature of many physical modeling problems, and matches closely to the *WaveCore* processor architecture. This makes the *WaveCore* efficient and programmable at a domain-specific abstraction level while keeping a high degree of control over processing latency, parallelism and time-predictability of execution.
- Application development tools, consisting of a compiler, a simulator and an FPGA-based *WaveCore* processor platform.
- Mapping and analysis of a wide range of audio effects and synthesis applications on the *WaveCore* processor.

We evaluated the *WaveCore* processor architecture, and the associated programming methodology from three different viewpoints:

- 1) The first viewpoint is application cases. We have implemented several digital audio effects applications. The associated physical models are mathematical abstractions of analog circuitry (virtual analog modeling). In addition to this we implemented reverberation models, based on the digital waveguide principle. Finally we implemented detailed physical models of simple geometrical objects (e.g. string, membrane, plate) and combined these to show the feasibility of building virtual music instruments. For all these application cases we investigated the mapping of physical models onto the *WaveCore* processor, and we studied the programming abstraction and mapping efficiency.
- 2) The second viewpoint is programming. We implemented a graphic equalizer application in a domain-specific functional programming language for musical applications, called *Kronos*. This application has been automatically compiled to the heterogeneous architecture, while keeping full control over latency and real-time constraints.
- 3) The third viewpoint is a benchmark analysis. We compared the *WaveCore* to other processors which are, or can be used for physical modeling. Within these benchmarks we compared the different solutions for area efficiency, instruction-set efficiency, time predictability and application development effort.

Our research has shown that the proposed heterogeneous processor architecture covers the investigated applications quite well. Moreover, we showed that the *WaveCore* programming methodology fits well to existing domain specific functional languages. Despite the high abstraction level of programming, the mapping onto the processor has shown to be efficient, and yields a predictable and ultra-low latency. Moreover, the benchmark analysis has shown that the *WaveCore* is efficient with respect to area and mapping of applications.

SAMENVATTING

ix

Het onderwerp van dit proefschrift is real-time muziek/akoestische signaal-verwerking en -synthese, door gebruik te maken van een schaalbare en domein-specifieke processor architectuur. In principe zijn er twee toepassings-groepen binnen het applicatie domein. De eerste groep behelst het digitaliseren van analoge audio effecten (bijv. muziek instrument effecten). Voorbeelden zijn nagalm (simulatie van akoestische ruimte), emulatie van versterkers en modulatie effecten zoals *phasing* en *flanging*. De tweede groep behelst geluids synthese voor elektronische muziek instrumenten (bijv. keyboard, virtuele piano, of geluids synthese in virtuele realiteit toepassingen).

Deze toepassingen bestaan uit twee gedeelten: real-time fysische modellen en gebruikers interactie modellen. Een interactie model realiseert de koppeling tussen de gebruiker en het fysische model. Een fysisch model is een wiskundige beschrijving van een fysisch object, welke in real-time uitgevoerd moet kunnen worden door een processor. Dit model is het feitelijk signaal-verkings of -synthese gedeelte van de toepassing. We focuseren op drie klassen van fysische modellen. De eerste klasse is virtueel analoge modellen. Deze modellen zijn een wiskundige beschrijving van een apparaat welk analoge signalen verwerkt. De tweede klasse behelst digitale golfgeleider modellen. Deze klasse van modellering is gebaseerd op een ruwe abstractie van een akoestisch object. De vertraging-lijn speelt een centrale rol binnen deze klasse van modellering. De derde klasse omvat gedetailleerde fysische modellering van muziek instrumenten en akoestische ruimten. Deze klasse is het meeste reken-intensief.

Vanuit een reken complexiteits perspectief stellen fysische modellen andere eisen, vergeleken met de eisen die gesteld worden aan interactie modellen. De meest uitdagende eisen voor een fysisch model zijn een korte verwerkingstijd (dit is de tijd die nodig is voor het verwerken van een ingans-sigitaal tot het produceren van een uitgangssigitaal), de voorspelbaarheid van de real-time executie van het model, en schaalbaarheid. Voor interactie modellen zijn flexibiliteit en best-effort executie de meest belangrijke eisen. Dit zijn conflicterende eisen, welke leiden tot problemen in de huidige aanpak binnen het domein van muziek/akoestische signaal verwerkings toepassingen.

De volgende bijdragen van dit proefschrift adresseren deze problemen:

- » Een schaalbare multi-kern (MIMD) processor: de *WaveCore*. Specifieke aspecten van de *WaveCore* zijn een sterk geoptimaliseerde instructie-set, en expliciete ondersteuning van verbindings-lijnen. De *WaveCore* is geoptimaliseerd naar het lokaliteits principe, welk leidt tot een hoge mate van de voorspelbaarheid van tijd-executie en een lage signaal verwerkingstijd.
- » Een heterogene processor architectuur, welke bestaat uit een algemeen toepasbare processor (voor de executie van interactie modellen) en de *WaveCore* processor (voor de executie van fysische modellen).
- » Een programmeermodel voor deze heterogene processor architectuur, welke gebaseerd is op een data-flow model. Op het conceptuele vlak ligt dit programmeermodel dicht bij de wiskundige eigenschappen van vele fysische modellen, en sluit nauw aan bij de processor architectuur van de *WaveCore*. Dit maakt de *WaveCore* efficient, en programmeerbaar op een domein-specifiek abstractie niveau, terwijl de programmeur een hoge mate van grip heeft op de verwerkingstijd, de schaalbaarheid, en voorspelbaarheid van tijd-executie.
- » Gereedschappen voor het ontwikkelen van toepassingen, bestaande uit een compiler, een simulator en een FPGA-gebaseerd *WaveCore* processor platform.
- » De mapping en analyse van een brede verscheidenheid aan audio effecten en synthese toepassingen, op de *WaveCore* processor.

We hebben de *WaveCore* processor, en de bijbehorende programmeermethode, vanuit drie invalshoeken geëvalueerd:

- 1) De eerste invalshoek is toepassingen. We hebben verscheidene digitale audio effecten geïmplementeerd. De bijbehorende fysische modellen zijn wiskundige abstracties van analoge circuits (virtueel analoge modellering). Daarnaast hebben we nagalm modellen geïmplementeerd, gebaseerd op de digitale golfgeleider methode. Tot slot hebben we gedetailleerde fysische modellen geïmplementeerd van eenvoudige geometrische objecten (zoals snaar, membraan en plaat), en hebben deze object modellen gecombineerd om de haalbaarheid aan te tonen voor het bouwen van virtuele muziek instrumenten. Voor al deze toepassingen hebben we de afbeeldingseigenschappen op de *WaveCore* onderzocht, en hebben we de efficientie en programmeer-abstractie geanalyseerd.
- 2) De tweede invalshoek is programmeren. We hebben een grafische equalizer toepassing geïmplementeerd in een domein-specifieke programmeertaal, genaamd Kronos. Deze applicatie hebben we automatisch kunnen afbeelden op de heterogene architectuur, met hierbij een hoge mate van controle behoudend op verwerkingstijd en real-time restricties.
- 3) De derde invalshoek is een vergelijkings analyse. We hebben *WaveCore* vergeleken met andere processoren die gebruikt, of gebruikt kunnen worden, voor het afbeelden van fysische modellen. Binnen deze analyse hebben we gekeken naar oppervlakte efficientie, instructie-set efficientie, real-time voorspelbaarheid en het gemak waarmee een toepassing ontwikkeld kan worden binnen de gegeven proces-

sor technologie.

Ons onderzoek heeft aangetoond dat de voorgestelde heterogene processor architectuur een groot gedeelte van het beoogde toepassingsgebied afdekt. Voorts hebben we laten zien dat de *WaveCore* programmeermethode goed past bij bestaande domein-specifieke programmeermethoden. Ondanks het hoge abstractieniveau van het programmeermodel leidt deze methode tot efficiënte afbeeldingen op de processor, met een hoge mate van tijd-voorspelbaarheid van executie bij een ultralage verwerkingstijd. Bovendien heeft het vergelijkingsonderzoek laten zien dat de *WaveCore* efficiënt is met betrekking tot hardware oppervlakte.

ACKNOWLEDGEMENTS

xiii

There and back again...

The journey starts during spring time in 2009 when I was one of the blessed ones within NXP Semiconductors to be affected by a reorganization. As a result I got a "sabbatical" period of 4 months to rethink my career and opt for new opportunities, after a consecutive period of 17 years where I have been lucky to be involved in several R&D activities in the field of signal processing architectures. During this sabbatical break, I started an activity which can be considered as the seed of my PhD work. I combined my passion for music, as a guitar player, with interest for music electronics and experience in DSP. The goal of this project was to develop guitar effects in software, mapped on a PC soundcard. However, this project got out of control and one year later a prototype version of the WaveCore processed its first audio streams on FPGA.

My initial idea was to develop a product, based on the WaveCore. I thank Rob Woudsma and Kees Moerman for giving technical feedback on the initial concept. Moreover, I thank Patrick Heuts for sharing business and product thoughts and guiding me through the process of start-up innovation. My dear ex-colleagues at DAP Technology have been very supportive in the development of the WaveCore by providing an FPGA board, and giving valuable feedback and encouragements on this off-topic innovation. I am grateful to Cor Jansen, Jan de Vries, Jeroen de Zoeten for encouraging the big-effort development of the WaveCore and programming tools, which often distracted my attention from my regular work at DAP Technology. Thanks guys, I owe you a big favour! Furthermore, I thank Casper van Doorne for his advices on PCB design, and Loek Derks for the actual PCB design of a prototype. At the application side, I have been working together with CCRMA at Stanford University. This work included benchmarking and an investigation of functional language (Faust) mapping on the WaveCore. I thank Julius O. Smith for his advice. Furthermore, I thank Edgar Berdahl for our cooperation on benchmarking the WaveCore against other music signal processing platforms. Many thanks go to Andreas Degert from the Guitarix community, who wrote a prototype WaveCore backend for the Faust compiler.

Jan Jacobs has been my supervisor during my internship at Océ back in the late 80s. In may 2012 I met Jan Jacobs again and he introduced me to the lecturers team at Zuyd Hogeschool in Heerlen. This historical meeting has been the starting point

of a collaboration on a WaveCore science lab and associated guest-lectures. After this meeting Jan asked me "why don't you start your own PhD program, based on the WaveCore"? A question that I had never thought of earlier... This is how I came in contact with CAES within Twente University, and how the PhD train started to roll. Jan, I am very thankful to you for your support, interesting discussions and friendship. Your out of the box thinking has been, and still is, an inspiration to many people. Ramon Jongen at Zuyd gave me the opportunity to develop a science lab, based on the WaveCore, where students can develop their own guitar effects. Ramon, many thanks for giving me this opportunity and your inspiring enthusiasm. Furthermore, I wish to thank graduate students Raoul van de Rijt, Rick Thijssen and Cliff Wings who contributed to the programming environment of the WaveCore.

And so I started my PhD program at the beginning of 2014. Despite the fact the I did all the work remotely (I visited CAES only once in a while), I could sense the productive and stimulating atmosphere in the research group. Gerard Smit and Jan Kuper are a stimulating and complementary team. Soon enough they taught me that I needed to descend from abstraction and focus my work to a concrete and practical scope. The most important part of the work has been the formulation of an answer to the very simple question "what is the WaveCore"? Finding this answer has been far from trivial. Gerard and Jan, thanks a lot! I also thank graduate student Ruud Harmsen, who showed that WaveCore and Clash can be friends. The DAFx2014 conference has been a very productive event. There I met Vesa Norilo, a PhD student working on functional languages. I also met Udo Zölzer, who brought me in contact with Rolf Bader and Florian Pfeifle, both at Hamburg University. Working together with Vesa Norilo (and later Vesa Välimäki) brought me the missing link between functional languages and the WaveCore, and the work became a significant part of the thesis. Vesa and Vesa, thanks a lot! Rolf Bader and Florian Pfeifle introduced me in the world of virtual instrument modeling. Working together with Florian and Rolf yielded another significant part of the thesis, and perhaps the most important justification of the WaveCore architecture. It has been a pleasure working with you, thanks!

As with many things, the last piece of the work (finalization of the thesis) has been the most demanding. This turned out to be sheer impossible to combine with a full time job, family and social life. I am grateful to my employer Intel for granting me two months sabbatical, in order to jump over the remaining few thesis hurdles. In particular I thank Aviad Hevrony for his unconditional support. Furthermore I thank numerous colleagues for their understanding.

Carrying out a PhD program, and combining this with a job, family and social life is demanding. Working in evening hours of many days and many weekends implies a big sacrifice, specifically to the family. Therefore, my biggest gratitude goes to the ones closest to me. To my two sons Lars and Glenn: I am sorry that I did not spend the amount of quality time with you, which I was supposed to. And

I thank you to put my feet back on the ground at those moments that the produced audio from the FPGA board revealed terrible screaming bugs: dad, please stop! I thank Lars, being a mathematics student, for his contribution to the thesis. The last person I dearly thank is my wife Yvonne. I could never have made it to this point without her love, understanding and doing all these extra domestic tasks while I was busy...

xv

So the work has come full circle, and I am back at a stage to continue where I started the journey. Perhaps I can now finally start using this programmable technology to create interesting guitar effects...

Math

Horst, jan 2017

CONTENTS

xvii

1	INTRODUCTION	1
1.1	Scope	1
1.2	Problem statement and research questions	5
1.2.1	<i>Problem statement</i>	5
1.2.2	<i>Research questions</i>	6
1.3	Contributions	7
1.4	Thesis structure	8
2	REAL-TIME MUSIC/ACOUSTICAL SIGNAL PROCESSING	11
2.1	Introduction	11
2.2	Applications and modeling	12
2.2.1	<i>Real-time physical models</i>	12
2.2.2	<i>Interaction models</i>	15
2.2.3	<i>Modeling techniques and computational requirements</i>	16
2.3	Processor technology	17
2.3.1	<i>General purpose processors</i>	20
2.3.2	<i>Programmable Digital Signal Processors</i>	21
2.3.3	<i>Field-programmable gate array</i>	24
2.3.4	<i>Multi-core and coarse-grained architectures</i>	27
2.4	Programming methodologies	32
2.4.1	<i>High-Level DSP Languages</i>	32
2.4.2	<i>Musical Programming Languages</i>	33
2.5	Conclusions	34
2.5.1	<i>Justification for a domain-specific processor concept</i>	35
3	THE REMAP PROCESSOR ARCHITECTURE	37
3.1	Introduction	37
3.1.1	<i>Dataflow model for ReMap applications</i>	39
3.2	Programming model	45
3.2.1	<i>Process graph definition</i>	45

3.2.2	<i>Process definition</i>	48
3.2.3	<i>WaveCore process partitioning</i>	51
3.2.4	<i>The Primitive Actor</i>	54
3.2.5	<i>WaveCore programming language</i>	57
3.3	WaveCore processor architecture	61
3.3.1	<i>Scalable cluster of Processing Units</i>	61
3.3.2	<i>Relation between programming model and the WaveCore</i>	63
3.3.3	<i>Processing Unit architecture</i>	65
3.3.4	<i>Graph Iteration Unit</i>	66
3.3.5	<i>Load/Store Unit</i>	75
3.4	WaveSlang compiler	80
3.4.1	<i>WPG consistency checking</i>	83
3.4.2	<i>Spatial mapping</i>	83
3.4.3	<i>Temporal mapping</i>	87
3.4.4	<i>Shared memory transaction synthesis</i>	93
3.4.5	<i>Instruction synthesis</i>	94
3.5	Development tools	94
3.5.1	<i>Process graph simulator</i>	95
3.5.2	<i>HW target interface</i>	96
3.5.3	<i>Development board</i>	96
3.6	Discussion and conclusions	98
4	APPLICATION CASES IN THE CONTEXT OF WAVECORE MAPPING	103
4.1	Introduction	103
4.2	Virtual analog modeling	104
4.2.1	<i>Basic functions</i>	105
4.2.2	<i>Analog guitar effects</i>	112
4.3	Digital Waveguide synthesis	125
4.3.1	<i>The roots: Karplus-Strong and d'Alembert</i>	125
4.3.2	<i>Digital reverberation</i>	127
4.4	Finite-Difference Time Domain modeling	131
4.4.1	<i>Wave equations and discrete approximations</i>	133
4.4.2	<i>Capturing finite-difference schemes into WaveCore processes</i>	134
4.5	Composition of physical geometry models	137
4.5.1	<i>Experiment 1: primitive six-string device in an acoustic room</i>	137
4.5.2	<i>Experiment 2: scalable plate model</i>	139
4.5.3	<i>Experiment 3: real-time membrane model on FPGA</i>	140
4.6	Discussion	142
4.6.1	<i>Programming model</i>	142
4.6.2	<i>Application spectrum and processor scalability</i>	143

5	HETEROGENEOUS CODE GENERATION FOR A GRAPHIC EQUALIZER	147
5.1	Introduction	147
5.2	Background	148
5.2.1	Graphic Equalization	148
5.2.2	Specific Graphic EQ algorithm	149
5.3	Kronos – a high level music DSP language	150
5.3.1	Heterogeneous code generation	152
5.4	Discussion	154
5.4.1	Equalizer Response	154
5.4.2	Performance	155
5.5	Conclusions	155
6	EVALUATION	159
6.1	Introduction	159
6.2	Physical analysis of the WaveCore processor	160
6.2.1	Area efficiency	160
6.2.2	Processor instance optimization	165
6.3	Performance analysis	167
6.3.1	ISA and compiler performance analysis	167
6.3.2	Geometry mapping performance	169
6.3.3	FPGA benchmark: the Banjo	171
6.4	WaveCore design methodology	176
6.5	Discussion and conclusions	178
7	CONCLUSIONS	183
7.1	Summary	183
7.2	Conclusions	184
7.2.1	Claims	184
7.3	Discussion and future work	185
A	WAVESLANG CODE EXAMPLE	189
B	WAVESLANG COMPILATION METRICS	195
	ACRONYMS	201

BIBLIOGRAPHY	205
LIST OF PUBLICATIONS	213
INDEX	215

INTRODUCTION

ABSTRACT – *The focus of this thesis is on real-time music/acoustic signal synthesis and processing. Application examples are electrical guitar effects, digital synthesizers and digital music instruments (e.g. digital piano). We observe that the applied processor technologies within this domain are quite diverse, while the mathematical formulation of the related algorithms which run on these processors are quite similar. Within the state of the art, which is elaborately described in chapter 2, we observe a number of problems which are related to the applied processor technologies. These problems are predictability of execution, long processing latency, and programmability problems due to a mismatch of abstraction between the mathematical formulation of algorithms and the applied programming methodologies. These observations are the basis of our main contribution: a domain specific processor called WaveCore, and its programming methodology called WaveSlang.*

1.1 SCOPE

Digitization of processes, in the broadest sense, has a dramatic effect on our society and has yielded a wealth of applications. Many of these digitized processes are integrated in our daily life in a pervasive way (e.g. communication, infotainment, services, medical diagnostics). This digitization started in the early 70s and has been an on-going process ever since. Its pace of progress is to a large extent linked to the progress in semiconductor technology (Moore's law). Like within many other domains, the digitization of musical and acoustical algorithms finds its roots in the need/desire to enhance the characteristics of the original analog reference (e.g. acoustical music instruments, analog synthesizers), or to enable applications which had not been possible before. Examples of such characteristics are reduction of production cost (e.g. emulation of a piano with a cheap keyboard) and function enhancement. A central aspect in the digitization process is a mathematical description of the physical problem. Such a description is called a physical model (e.g. the model of a piano). Subsequently the second part of the digitization process is the

FIGURE 1.1 – DAFx application (e.g. guitar effect)

mapping of the derived physical model on a suitable computer architecture. The constraints of a given processor platform (e.g. computational capacity, real-time properties) determine whether the mapping of the physical model is feasible.

We limit our scope to real-time synthesis and processing of music/acoustic signals. Examples of REal-time Music/Acoustical signal synthesis and Processing (*ReMap*) applications that fit to this scope are virtual music instruments (e.g. synthesizer, digital piano) and Digital Audio Effects (*DAFx*) (e.g. acoustical space emulation, modulation effects). DAFx are widely applied to a wide range of musical instruments (e.g. guitar, bass guitar, synthesizers).

ReMap applications can be classified in two groups: DAFx applications which process a signal that is produced by an audio source like a musical instrument (e.g. an electrical guitar), and applications which produce music/acoustical signals (e.g. synthesizer). Abstract models of these two application classes are depicted in fig. 1.1 and fig. 1.2. A DAFx application consists of a real-time part (the orange-coloured blocks in fig. 1.1), and a control part which usually does not execute in real-time. The control block responds to the human (user) interface. Its function is to compute parameters from the user interface (e.g. knob settings, sliders), and to pass the computed parameters to the processor block. Examples of such parameters are filter coefficients or parameters that control sound characteristics like volume. The processor block contains the actual processing function which modifies the input signal. Apart from derived parameters from the control block, the processor can also be controlled by properties of the input signal itself. This is visualized in fig. 1.1 by the analyzer block, which is executed in real-time. An example of an analysis function is to derive the slow-varying envelope from the input signal, which can be used to compute parameters for the processor (i.e. dynamic filter or dynamic range compression [109]).

A signal synthesis application, as depicted in fig. 1.2, produces an audio signal. The generation of this signal is performed by a model which executes in real-time: the generator. The generator function is most often based on a mathematical abstraction of a physical sound producing object (e.g. a virtual music instrument). This implies that such a model describes the mechanical (e.g. vibrations) charac-

FIGURE 1.2 – Signal synthesis application (e.g. synthesizer)

teristics of the physical object. Therefore, such a model is called a *physical model*. Application examples of signal synthesis models are synthesizers and virtual reality (e.g. real-time sound rendering within computer games). The stimulation of the physical model within the generator is implemented by the "stimulation & control" block in fig. 1.2. This function acts as an interface between the "player" (i.e. the musician or the gamer) and the generator. Typically this interface function does not run in real-time, at least not at the same rate as the physical model within the generator. The generated acoustical signal is in many applications not directly made audible but post-processed, in order to add additional effects. An example of such an effect, which is added by the processor block, is reverberation (i.e. simulation of an acoustical space, like a concert hall). Note that the *processor* function within a signal synthesis application is similar to the *processor* function within a signal processing application. Physical modeling techniques are applied to both *generator* functions, as well as *processor* functions.

So in general, a ReMap application consists of a physical model and a control-part which we call an "interaction model". The physical model runs in real-time and may be composed of several sub-modules like a generator and post-processing model. The interaction model acts as an interface between the user and the physical model.

The computational complexity of ReMap physical models varies significantly [85] [91]. For instance, a detailed physical model of a piano (i.e. virtual piano) requires a huge amount of computational resources [63]. The result is that these models can only be mapped on expensive massively parallel computing platforms. In order to make it economically feasible to execute physical models in real-time, these models need to be restricted in terms of computational complexity. Reduced complexity physical models are often called "physically informed" models [85]. Only the most dominant aspects are modeled within physically-informed models.

The most important characteristics of ReMap applications are the following:

1. *Time predictability of execution*

The human ear is very sensitive to irregularities (e.g. missing samples) in reproduced audio. Even a single missing sample in an audio stream is audi-

ble, and perceived as disturbing. Therefore, an important characteristic of real-time physical model execution is that this should be highly predictable, with an extremely low probability of discontinuities in the sample stream.

2. *Low latency*

The elapsed time between the excitation of an instrument (e.g. electrical guitar, or a virtual instrument), and perception of the produced sound, is called latency. This latency should be shorter than noticeable by the musician. A noticeable latency is typically larger than 15-20ms. A distance between the musician and the speaker of 1 meter implies a latency of approximately 3 ms. If the signal, which is generated by the (virtual) instrument, needs to be processed, then the processing latency should therefore be less than 12ms. In general, the latency that is introduced by the execution of a physical model should be as short as possible.

3. *Physical model complexity range*

The complexity of a ReMap physical model varies significantly, depending on the level of detail of the model. For virtual analog models (e.g. a digitized equalizer model) such a model often represents a straightforward 1-dimensional chain of relatively simple signal processing functions, like filter sections. For detailed modeling of instruments or acoustical spaces, these models may represent complex 3-dimensional geometrical topologies from which the audio signal is rendered.

4. *Mathematical formulation*

A large majority of ReMap physical models can be expressed mathematically (e.g. differential equations) [85]. Discrete-time approximations of such models most often can be expressed as synchronous static data flow graphs. Elementary functions are arithmetic operations and delays. Apart from the unity delay (z^{-1}) which is commonly used in digital signal processing, we identify a variable programmable delay-line (z^{-M}) as an elementary function [84].

5. *Functional programming paradigm*

A functional programming methodology fits well to the mathematical nature of ReMap applications. We see this reflected in the domain specific programming methodologies. Examples of ReMap domain specific languages are Faust [58] and Kronos [54].

ReMap applications (in particular, those based on physically-informed models) are predominantly mapped on programmable digital signal processors (DSP). This class of processors are optimized for real-time stream processing. However, the general purpose processor (GPP) is also widely applied within the ReMap domain. Examples of GPP based platforms within this context are PCs, tablets and smartphones. Both programmable digital signal processor (DSP)s and GPPs are not sufficiently powerful for executing detailed physical models. In this domain we often see the application of field-programmable gate array (FPGA)s as an alternative to (multi-core) programmable processors [18] [19] [64].

1.2 PROBLEM STATEMENT AND RESEARCH QUESTIONS

1.2.1 PROBLEM STATEMENT

We observe the following problems within state-of-the-art ReMap applications (see chapter 2 for elaboration on these problems):

1) *Processing latencies*

The latency that is induced with the execution of ReMap physical models on programmable GPP and DSP processor platforms consists of three parts: (1) latency in analog-to-digital converter (ADC), (2) buffering/processing, and (3) latency in digital-to-analog converter (DAC). A programmable GPP or DSP usually employs instruction/data caches. Caches rely on the locality of reference principle: the assumption is that the next instruction/data is located close to the current instruction/data in both address-space and time. The consequence is that programs need to be optimized to this locality principle as well, in order to minimize the cache-miss rate and therefore maximize the performance. Block-based processing (i.e. processing blocks of data, rather than processing individual samples) is a means to improve the locality of reference. Block-based processing implies the necessity to buffer the sample-stream, and therefore introduces latency.

2) *Time predictability of execution*

GPPs are designed for flexibility, driven by imperative and versatile programming languages like C++. This versatility requires a unified memory model and the ability to multi-task, supported by an operating system. For a processor this implies the necessity for caches (to support a high-performance unified memory model) and interrupt based task switching. Task switching and caches introduce a degradation of the time predictability of execution of real-time tasks. Time predictability can therefore often not be guaranteed. Particularly for ReMap applications a guaranteed real-time execution is extremely important. Large buffers can help to improve the time predictability and guarantee of real-time execution. However, the disadvantage of large buffers is longer processing latency.

3) *Scalability*

Detailed physical modeling is often based on finite element methods, like finite-difference time-domain (FDTD). Such models consist of a grid of computational elements, which represents the physical structure of the modeled object (e.g. musical instrument) [8] [5]. In practice, such a model can be complex (i.e. large number of grid points) and strongly connected (i.e. many dependencies between adjacent grid points). The required computational capacity usually exceeds the capacity of a single processor core. Hence, partitioning of the model and allocating partitions to a multi-core processor architecture is unavoidable. Mapping of a strongly connected and partitioned model on a multi-core processor is a complex problem [99] [31]. We observe that FPGAs are the only devices on which detailed physical ReMap models have been successfully mapped [64].

4) Programmability

ReMap physical models are mostly data-flow oriented. We already mentioned that a functional programming methodology fits well to this class of problems. However, an imperative programming style, like C++, is usually required for compiling an algorithm to a state-of-the-art GPP or programmable DSP. Usually, the programming support for a DSP consists of a library of optimized functions which can be called from a C-style program. Optimizing beyond these library functions is usually difficult, because it requires a detailed understanding of the processor architecture (e.g. pipeline, parallelism, memory hierarchy). The result is that the program source in which a ReMap algorithm is coded does not transparently reflect critical aspects of the model, like real-time constraints. This can seriously hamper the predictability of the developed code w.r.t. real-time behaviour. Programmability, and real-time predictability becomes even more complicated when a ReMap model needs to be partitioned over a multi-core architecture, using an imperative language like C++.

1.2.2 RESEARCH QUESTIONS

Following on the identified problems within the ReMap application domain, we formulate a number of research questions. These questions are the basis of our contributions, which are explained in section 1.3. The most important questions are:

Q1: Is it possible to design a scalable, low-latency processor architecture which covers the majority of the ReMap application domain?

The main motivation behind this question is that a programmable processor architecture for detailed physical modeling does not exist. Our goal is to define a scalable and efficient processor architecture, with low processing latency, which serves the majority of the ReMap domain.

Q2: Is it possible to design a programming language which both matches transparently to the aimed processor architecture, and to existing domain-specific ReMap programming languages?

Our aim is to be able to describe ReMap models in an existing functional language, while the mapping of the algorithm leads to time-predictable and efficient mapping on the aimed processor architecture.

Q3: Is it possible to define a ReMap application design methodology, based on the processor and programming language as proposed in Q1 and Q2?

This enables the automatic generation of optimized ReMap implementations from a functional language description. Such a design methodology is abstract in the sense that processor architecture details are not exposed to the developer. An advantage is a relative short development time, and a less steep learning curve for the developer. Another advantage of this is that an optimized low-latency ReMap implementation (i.e. a processor instance and associated program code) can be

automatically generated from an abstract functional application description.

1.3 CONTRIBUTIONS

7

Our work has resulted in the following contributions:

1. *The WaveCore processor (chapter 3.3)*
We have developed the WaveCore, which is a programmable and scalable many-core processor. Optimized instances of the processor can be derived for several application classes within the ReMap domain. A unique property of the WaveCore processor is that configurable delay-lines are supported in its instruction set. Next to that, the processor is optimized for predictable and ultra-low latency processing [MJW:6].
2. *A heterogeneous processor architecture (chapter 3.3)*
This architecture consists of a general purpose processor for executing interaction models and the WaveCore processor, for executing physical models.
3. *A domain specific data-flow programming language (chapter 3.2)*
We have developed a data-flow programming language called WaveSlang, specifically for the ReMap application domain. The semantics of WaveSlang match closely to the mathematical properties of a broad class of ReMap algorithms.
4. *Application development tools (chapter 3.4)*
We developed mapping tools for the WaveCore processor. These mapping tools automatically partition, allocate and schedule a WaveSlang program (i.e. a data-flow graph) to the WaveCore processor. Next to the compiler we also developed a simulator, and a FPGA-based development platform to which we mapped the WaveCore.
5. *Functional language compatibility of WaveSlang (chapter 5)*
WaveSlang is highly compatible with a ReMap specific functional programming language, called Kronos [56]. In chapter 5 we demonstrate how a ReMap application, described in the functional Kronos language, is automatically compiled to WaveSlang.
6. *Application case study (chapter 4)*
We analyzed the mapping of a set of applications on the WaveCore processor. Out of the broad ReMap application domain we chose both simple [MJW:6] and complex [MJW:5] application examples.
7. *Performance analysis and benchmarking (chapter 6)*
We have compared the efficiency of the WaveCore processor, in terms of area and language mapping efficiency, against other existing processors.
8. *WaveCore as design methodology (chapter 6)*
We have evaluated WaveCore as a design methodology. The strength of this methodology is that it enables automated generation of ReMap implementations (i.e. an optimized WaveCore processor instance and compiled WaveSlang code) from an abstract functional description of the application.

1.4 THESIS STRUCTURE

The structure of this thesis is as follows:

In chapter 2 we present an overview of the state-of-the-art in ReMap applications, identify physical modeling techniques, programming methodologies and associated processor architectures.

In chapter 3 we specify our main contribution: the WaveCore processor and its programming language WaveSlang. We will outline the processor architecture, programming methodology, compiler and development environment.

In chapter 4 we will analyze our target application domain and select a number of applications. From these applications we will derive physical models for which we subsequently analyze how and how efficient these can be mapped on the WaveCore processor.

In chapter 5 we will analyze how an existing ReMap specific programming language fits to the WaveCore processor. We do this through a relevant ReMap example: a graphical equalizer.

In chapter 6 we evaluate the technology through benchmark analysis on three different aspects: physical efficiency of the processor, the efficiency of the compiler and instruction-set architecture (*ISA*), and an analysis of WaveCore as a design methodology.

We wrap up in chapter 7 with the formulation of answers to our research questions and our contributions. Furthermore, we discuss aspects which are not or partially covered in the work and related future work.

REAL-TIME MUSIC/ACOUSTICAL SIGNAL PROCESSING

ABSTRACT – We analyze the state of the art of real-time music/acoustic signal processing (ReMap). We do this from three interdependent viewpoints: applications, processor technologies and programming models. For the majority of applications we observe that the associated physical models are data-driven by nature and can be described as static data-flow graphs. The three most common processors which are applied to map physical models are GPP, programmable DSP, and FPGA. Interaction models cannot always be modeled as data-flow graphs. These models, which are the implementation of the interface between the user and the physical model, are best suited to be mapped on a GPP. The preferred ReMap application programming style is declarative, because the modeling problems can often be represented in a mathematical form (e.g. differential equations). However, we observe that usually imperative programming languages like C++ are applied today. We use these findings, together with the problems that we introduced in chapter 1 to derive the requirements for the ReMap domain-specific processor architecture and an associated programming model.

2.1 INTRODUCTION

We analyze the state-of-the-art of the ReMap application domain from three viewpoints. These viewpoints are (1) applications, (2) processor technologies and (3) programming methodologies. From this state of the art analysis, we will derive the requirements for the WaveCore processor and its programming language WaveSlang.

2.2 APPLICATIONS AND MODELING

Within this section we analyze the characteristics of ReMap applications and related physical modeling. We distinguish two parts in a ReMap applications:

1. *Real-time physical models*
The real-time part of a ReMap application consists of one or more physical models. Like explained in chapter 1, this real-time part can either be a signal-analysis/processor or a synthesis model.
2. *Interaction models*
The real-time part of a ReMap application is controlled (i.e. stimulation of a virtual instrument or control of a DAFx) by an interaction model which usually does not run in real-time. This interaction model acts as the interface between the player and the real-time part of the ReMap application.

2.2.1 REAL-TIME PHYSICAL MODELS

Like explained in the introduction chapter 1, we assume that the real-time part of ReMap applications is composed of one or more physical models. A physical model is an abstract (mathematical) representation of the associated physical object. Examples of such objects which are relevant to ReMap applications are:

- » An (acoustical) music instrument, such as piano, trumpet, or guitar. The associated physical model serves as the audio production engine within the ReMap application (e.g. synthesizer or virtual reality).
- » Analog audio processing electronics, like a graphic equalizer (*GEQ*), non-linear tube amplifier, a spring or plate reverb. Associated ReMap applications are virtual amplifiers (to simulate the characteristic sound of a typical amplifier) or virtual acoustical space simulation.
- » Analog audio generation electronics, like a Moog oscillator [62] (e.g. as processing function within vintage analog synthesizer simulation).
- » Acoustical (analog) processing systems, like a rotating-speaker cabinet [85], a concert hall, bathroom.

The perceptual quality of the audio signal which is generated/processed by the physical model is related to the level of detail of the physical model. It is likely that a detailed physical model of an acoustical guitar produces a natural, perceptual high quality sound. Likewise, a detailed model of a Moog ladder filter [15] within an analog synthesizer model, combined with a detailed model of a rotating speaker cabinet is likely to produce a typical vintage analog synthesizer sound. Depending on the detail level of the model, the computational requirements for executing such a model in real-time (i.e. the number of arithmetic operations per second) can be significant. A reduction of the complexity of such a physical model is often necessary in order to execute such a model on a commercial state-of-the-art processing device in real time. Such a reduced-complexity model is often called

a "physically informed" model [84] [96] [106]. We highlight 3 classes of physical modeling. These classes cover the ReMap application domain reasonably well.

Detailed physical modeling

The aim of detailed physical modeling is to obtain a virtual object (e.g music instrument) that produces sound which matches closely to the real object. The most important aspect of detailed physical modeling is to understand how acoustical (i.e. pressure) waves propagate through the object, and how to describe this in a mathematical formulation. The basic technique for this class of modeling is to numerically solve wave equations by means of a FDTD derivative¹ of the mathematical formulation [8]. Such a FDTD scheme can be classified as a static, synchronous data-flow graph [MJW:5]. A virtual music instrument, based on FDTD modeling, can be constructed through the combination of geometry models (e.g 1D string, 2D plate, 2D membrane, 3-dimensional enclosed air space). For each geometrical model, a detailed study of the wave propagation characteristics may be obtained through detailed measurement (e.g. 2D microphone array) of the physical object [65]. An example of a virtual instrument (i.e. Banjo) which runs in real time on FPGA is described in [64], and is studied in more detail in chapter 6 of this thesis. FDTD modeling is elegant in the sense that the computation "schemes" [8] directly reflect the physical geometry that is modeled. The sound which is rendered from a detailed physical model can be strikingly realistic. An important advantage of this technique is that it allows to build detailed physical models in an intuitive way, which is conceptually close to building/shaping the real physical counterpart. Hence, detailed physical modeling can also be applied as a tool to develop musical instruments [2]. The most important disadvantage of detailed physical modeling is the computational complexity, which can be huge. Apart from virtual instruments, the FDTD technique is also applied to audio processing models such as the spring reverb [9]. For further reading on detailed physical modeling we recommend [69] [68].

Digital waveguide modeling

The reduction of the computational complexity of detailed physical models, without sacrificing too much on perceptual quality, is an important objective to make it economically feasible to apply such models to ReMap applications within the consumer domain. A reduced-complexity physical model is often called a "physically informed model". Physically-informed means that only the most dominant physical aspects of the object are taken into account within the model. Digital WaveGuide (DWG) modeling [84] [85] can be classified as a physically-informed modeling technique. The basic principle within digital wave-guide (DWG) is that the modeled object is represented as an abstract structure through which acoustical waves propagate. A DWG model can be seen as a lumped component model, where

¹A Finite Difference Time Domain model is a discrete model which describes the behaviour of (acoustical) waves through a medium

the delay-line, and wave-scattering junctions are basic components. The function of a delay-line is to simulate a travelling wave through a 1-dimensional lossless medium (e.g. a string). The function of a wave-scattering junction is to model the interaction of waves (e.g. interference) and to model losses within the medium. A DWG model therefore primarily consists of a network of delay-lines and junctions [53]. Modeling of a travelling wave through a delay-line is computationally cheap. A delay-line can be implemented by means of a memory, and a circular read and write pointer. Similar to detailed physical models, DWG models can also be described as static synchronous data-flow graphs. For further reading we refer to [66] [38] [78] [81] [83] [96].

Virtual analog modeling

The objective of virtual analog modeling is to derive an abstract mathematical representation of an analog electronics circuit. These models are applied as real-time DAFx kernels, and hence emulate the associated analog circuit behaviour. Examples of virtual analog models which originate from analog electronics are virtual amplifiers, a broad class of guitar effects (e.g. phaser, flanger, wahwah, distortion), and virtual analog synthesizers. One way of obtaining a virtual analog model is through a mathematical analysis of the main overall function of the analog reference circuit. For instance, a mathematical analysis of an analog equalizer circuit yields a differential equation (linear time-invariant (*LT*I) system [57]). This differential equation can be approximated by an associated difference equation, which finally can be mapped on a data-flow graph. This data-flow graph is the implementation of the virtual analog model of the equalizer circuit, and can be executed in real-time on for instance a DSP processor.

Another method to derive a virtual analog model from an analog circuit is through modeling of the electronics schematic topology. This implies that the modeled circuit is represented as electronic component models/functions and the interconnections between these component models. This same method is also applied in analog circuit simulators for electronic design automation (*EDA*) purposes. Fettweis developed a methodology in the late 1960's, called wave digital filter (*WDF*) [21]. The objective of *WDF* is to digitize lumped electrical circuits, composed of basic functions like inductors, capacitors, resistors, etc. The main advantage of this methodology is that it only requires to model the behaviour of the electronic components [37] and the interconnection structure of the circuit. Hence, this methodology does not require an understanding of the overall analog circuit behaviour, in order to model it. Apart from analog circuit modeling, *WDF* also found its application in the (automated) modeling of acoustical systems. An example of a virtual tube amplifier, based on *WDF*, is described in [60].

Modeling of non-linearity is an important aspect in virtual analog modeling. Usually, non-linear behaviour is an unwanted phenomenon. However, in ReMap applications, non-linearity characterizes the typical sound of for instance music instru-

ments or overdriven guitar amplifiers. Modeling of non-linearity is often complex. It usually requires oversampling to avoid aliasing [57], or in some cases it requires iterative solver algorithms (e.g. Newton-Raphson) for non-linear differential equations ([104]). An example of a non-linear virtual analog model of a distortion guitar pedal is described in [106], [101].

Virtual analog models can often be classified as static and synchronous data-flow graphs. Exceptions are models which require iterative solvers. Such a solver approximates a non-linear function through iteration, where the accuracy of the approximation depends on the approximated function and the number of iterations. The drawback of this method is that the number of iterations, per time-step, is not fixed. Hence, iterative methods cannot be modeled with static data-flow graphs. This implies that the predictability of real-time execution cannot be guaranteed. In order to overcome this problem, iterative solvers are sometimes remodeled as a non-linear function look-up table and a separate filter function [104] [106]. The result of this remodeling is that the function is then simplified to a static synchronous data-flow graph. Further reading and background on virtual analog modeling can be found in [95] [94] [107] [105] [75] [25] [51] [106].

2.2.2.2 INTERACTION MODELS

In chapter 1 we introduced the "interaction model". The interaction model embodies the interface between the physical model(s), which are executed in real-time, and control events (e.g. produced by a musician). We distinguish two classes of interaction models: (1) interaction models for DAFx applications (fig. 1.1) and (2) interaction models for signal synthesis applications (fig. 1.2).

Interaction models for DAFx

The most important function of an interaction model for DAFx is to translate user settings (e.g. knobs, sliders) to parameters which feed into the real-time physical model. An example is a set of sliders for a graphical equalizer [MJW:3]. Each slider represents the attenuation of a particular frequency band. The purpose of the interaction model for this particular example is to translate the slider settings to a set of filter coefficients. The filter itself is a real-time physical model. The mentioned translation might be straightforward, but can also be a complex function. Typically, the interaction model for DAFx applications is event-driven (e.g. the user moves one of the sliders) and hence is executed occasionally.

Interaction models for signal synthesis applications

The real-time part of a signal synthesis application consists of a physical model which is executed in real-time, and which generates (i.e. renders) audio signals. An example of such a physical model is a virtual piano. The purpose of the associated interaction model for the virtual piano example is to translate keystrokes, which are initiated by the musician, to stimuli which are injected to the physical model.

Depending on the complexity of the physical model, this mentioned translation can be complex as well. The keystroke needs to be translated to the movement of a virtual hammer which hits a string model at a certain position. Furthermore, the string may be damped by a pedal, which is controlled by the foot of the musician. The translation of the pedal-setting to physical model parameters is in that case also part of the interaction model function. Musical instrument interaction modeling is an active research topic [20]. Similar to interaction modeling for DAFx, this is event-driven. The complexity, however, can be far more complex.

2.2.3 MODELING TECHNIQUES AND COMPUTATIONAL REQUIREMENTS

In the previous subsections we reviewed 3 physical model categories, and the characteristics of interaction models. These physical models can almost always be captured in static data-flow graphs [85], [96], [95], supported with linear systems theory [57].

The computational complexity of these models vary significantly. The execution constraints for these models are as follows:

1. The processor to which the physical model is mapped must be scalable, in order to support both (relatively simple) physically informed models as well as (computationally demanding) detailed physical models.
2. The model must be executed in strict real-time, with a predictable execution time.
3. The latency (i.e. elapsed time between stimulation of the model and produced audio) must be as short as possible.

The requirements for interaction models are significantly different compared to those for the physical models. These interaction models translate events (e.g. generated from a computer game, a musician which plays a keyboard, or a user who changes control parameters for a guitar pedal) to parameters which feed into the physical model(s). The execution characteristics for interaction models are as follows:

1. Best-effort execution. Despite the fact that events need to be translated to physical model parameters as quick as possible (i.e. short latency), there is a less strict constraint on the elapsed time between the event and the availability of mentioned parameters. The elapsed time between the event, and the computation of derived parameters for the physical model, needs to be shorter than noticeable. This time, however, is still much longer than a typical tread period for a physical model. Example: if the maximum allowed latency is $7ms$, at a firing rate of $48kHz$ for the physical model, then this implies that the interaction model should respond within a time-frame of 336 physical model thread periods.
2. The processor to which the interaction model is mapped needs to be highly flexible (i.e. general purpose), and does not necessarily have to be scalable.

FIGURE 2.1 – Processor technology landscape: performance versus flexibility

3. The structure of the interaction model is not necessarily a static data-flow graph.
4. The complexity of interaction models may vary significantly, depending on the ReMap application.

This fundamental difference between physical models, and associated interaction models leads to the preference for a heterogeneous ReMap execution platform. Physical models fit well to processors which are optimized for real-time streaming (e.g. DSP), while interaction models fit well to general purpose processors. In the following chapter we will analyze processor architectures which are relevant to both model classes.

2.3 PROCESSOR TECHNOLOGY

Throughout history, a variety of processor technologies have evolved. The evolution of different processor architectures has always been driven by the state of the art in silicon process technology (i.e. feasible clock frequency, number of transistors, power budget) and the requirements from the application domain for which an implementation is to be developed. Roughly, the evolved processor landscape can be visualized by categorizing performance versus flexibility. Flexibility implies the possibility to solve a wide range of possible application problems. A good example of a flexible processor architecture is the GPP, which can be found in any personal computer. At the other side of the landscape we find processor architectures which are highly specialized for a specific task which cannot be performed by a GPP because of performance demands. An example of this is a software defined digital modem for wireless communication (e.g. a WiFi modem pipe), implemented in application-specific integrated circuit (ASIC) technology. An overview of the processor landscape is depicted in fig. 2.1. A general way to classify a processor

FIGURE 2.2 – Computation in space (left) or time (right)

architecture is by focussing on its implicit parallelism: the amount of computations which can be executed concurrently. When we zoom in into the extremes then we find the definition of computation in space on the right-hand side of the landscape (maximum performance) and computation in time (maximum flexibility) on the left-hand side. Figure 2.2 illustrates both extremes through the example of the computation of the quadratic relation in (2.1).

$$y = A.x^2 + B.x + C \quad (2.1)$$

At the left-hand side of fig. 2.2 we find an implementation in *space* of (2.1) which is entirely implemented in a structure that is capable to compute y instantaneously: each arithmetic operator is implemented as a hardware structure in a fixed graph which is dedicated to compute (2.1), but cannot do anything else and hence is ultimately inflexible. At the right-hand side of fig. 2.2 we find an implementation of the same equation. However, this implementation uses a programmable arithmetic and logic unit (ALU), a bit of storage to administrate variables, and a control structure which loops through the instructions that are necessary to compute the result y step by step. This implies a implementation of (2.1) in *time* by means of a programmable processor. This implies ultimate flexibility (e.g. the structure can do a lot more than computing the quadratic relation) at the cost of performance: it takes 5 steps to compute the result. In general, the most efficient solution for a given set of application-derived problems is a balance between computation in space and time. This means that the optimal balance between performance (i.e. parallelism) and flexibility (i.e. programmability or configurability) yields an efficient processor solution for the given application class.

The progress of Moore's law has a big influence on the processor landscape as well. For many years there has been impressive evolution in GPP architectures, which primarily focussed on squeezing as much parallelism as possible out of single instructions: enhancement of instruction-level parallelism (ILP). This has led to generically applicable superscalar central processing unit (CPU) architectures.

There are however a combination of factors which has virtually ended the evolution of ILP in single-core GPPs: the three "walls".

1. *The ILP wall*

There is an upper limit on exploiting parallelism at instruction level for a single core. This upper limit is restricted by the complexity to schedule the amount of parallelism at instruction level. Several techniques have evolved to improve the utilization of execution units within a single core processor, like out-of-order execution, register renaming, branch prediction, speculative execution and hyper threading [61]. All these techniques have significantly complicated single-core CPU architectures, while the benefits of each subsequent improvement diminished, at the cost of an increase in silicon area and power consumption. A background of these ILP techniques can be found in chapter 2 of [28].

2. *The memory wall*

There is an ever growing gap between the feasible speed of logical cells compared to the speed of memories access, for subsequent silicon process generations. This speed gap has led to the evolution of complex memory hierarchy architectures (i.e. multi-level cache architectures), as small embedded memories tend to be much faster than bigger on-chip memories and large off-chip double data rate SDRAM (*DDR*) memories. The necessity for this memory hierarchy has led to a significant area increase of the on-chip (cache) memory, compared to the area of the functional units within a superscalar processor core (i.e. the logic wherein the actual computations are performed). Only a relatively small portion of the silicon area is devoted to actual data processing.

3. *The energy wall*

Power density (i.e. Watt/mm^2) increased with subsequent (smaller) process technologies. The main reasons for this is an ever increasing clock frequency, while transistor leakage increased because of smaller semiconductor feature sizes. This effect is one of the reasons that clock frequency of processors within subsequent process nodes has stopped increasing, in order to keep dissipation within feasible bounds.

These three "walls" have become the main driving factors towards the "multi-core era", which started around 2004 [13]. The logical consequence of the ILP wall has been that GPPs have become multi-core chips. The same trend can be observed in programmable DSPs, which have a lot in common with GPP. Fabric devices, like FPGA and coarse-grained reconfigurable architecture (*CGRA*), however, scale naturally with Moore's law. Hence, from a conceptual viewpoint the evolution of these devices is less disruptive, apart from the fact that modern FPGAs have become heterogeneous, as we will show in section 2.3.3.

In the next subsections we will probe into the state of the art of the different proces-

processor architectures which are most relevant to the domain of musical signal processing and related physical modeling. These are GPP (e.g. low-complexity physical, and interaction models), DSP (e.g. low to medium complexity physical models) and FPGA/CGRA (e.g. high complexity detailed physical modeling). Our aim is to extract an understanding of the strengths/weaknesses of these architectures for the ReMap domain.

2.3.1 GENERAL PURPOSE PROCESSORS

The GPP is primarily designed for flexibility, and secondary for performance (i.e. operations/second). The GPP is in principle based on a "von Neumann" architecture. This architecture consists of a unified memory, a control unit and one or more ALUs. This architecture has a tight link to classical (imperative) programming languages, like C or C++, which is illustrated by the following key characteristics:

- » **Memory model**
Data structures are allocated by the compiler to one unified memory space. From a processor architecture perspective this implies that data storage is physically implemented as one (usually cached) shared memory. Likewise, instructions (i.e. compiled program code) are fetched by the processor from the same shared memory. However, usually from a different memory region than data, and usually through an instruction cache.
- » **Stack/Heap**
In order to support procedural software design (e.g. C-functions) and recursion, a stack is used. A stack is a piece of memory which is used to temporarily store variables which are passed through function calls. From processor architecture point of view, this implies dedicated hardware support like a stack pointer.
- » **Interrupts**
In order to support operating systems or multi-threading, an interrupt mechanism is available in the processor architecture.
- » **Sequential instruction flow**
The support for loops, for, and if-then-else structures, and function calls is translated by the compiler to an instruction flow which is not necessarily linear. This means that the program counter (which is basically a pointer to the instruction memory) does almost never increment linearly while the program is executed. Branch and jump support, and program counter save/restore on stack, are architectural features which are required to support mentioned software structures.
- » **Parallelism**
The ALU usually contains multiple functional units (like multipliers, adders, logical units, etc.). Parallelism is not explicitly exposed in imperative programming languages, and for a compiler it is often difficult to extract parallelism from a program and map this on the available parallel units in an

ALU. Utilization of the parallelism within the ALU is often performed by the processor hardware, and may be complex.

For an elaborate background on GPP processor architectures, in relation to imperative programming models, we refer to [61].

Ultimate flexibility does not come for free. The main general disadvantages of the GPP for our ReMap application domain are:

- » Non-precise execution time
Instruction and data caching relies on the principle of locality: both temporal and spatial. If a program executes on a large data-set (too large to be cached) or has many non-linearities in the instruction flow (e.g. due to nested procedures, interrupts and thread switching), then the mentioned locality is disrupted. This implies cache misses. Each cache-miss implies that the processor needs to be stalled (i.e. halted) until the requested data/instructions are fetched from the next level in the memory hierarchy. Stalling a processor at an unpredictable time inherently means a non-precise execution time. For real-time applications (e.g. ReMap) this can become problematic, or it is at least difficult to predict the flawless execution time of an algorithm.
- » Execution latency
In order to enhance the locality, it is beneficial to process blocks of data (i.e. use buffers for an array of samples) within one execution thread, rather than single data elements (e.g. audio samples). Execution on blocks of data, however, implies that the latency (i.e. the elapsed time between input samples, processing, and generation of output samples) inherently increases. The latency increase is proportional to the sizes of input and output buffers.
- » Silicon area inefficiency
A relatively large portion of the area of a GPP is dedicated to the memory hierarchy (i.e. primarily caches). This implies that the GPP is usually not the most efficient solution for number-crunching, like ReMap applications.

Within ReMap applications, the GPP is a well-suited architecture for interaction models because of the flexibility requirement for such models. However, the execution requirements for real-time physical models are conflicting (to some extent) to the mentioned disadvantages of GPPs. We see GPPs being applied as processors that execute entire ReMap applications [89]. For these GPP based ReMap applications we, however, clearly observe limitations on the complexity of the physical models (this thesis, chapter 6). Likewise, we observe that minimizing the processing latency is often a challenge when applying the GPP for physical modeling.

2.3.2 PROGRAMMABLE DIGITAL SIGNAL PROCESSORS

A DSP essentially is a programmable architecture, which shares many similarities with a GPP. However, a DSP is (as its name suggests) optimized to digital signal

processing in a generic sense. These optimizations are derived from common properties of signal processing algorithms, and related periodicity:

1. *Zero overhead looping*

DSP algorithms often contain "for-loops". Such loops iterate a certain code block for a definite number of times, that can be determined before the for-loop starts. Zero-overhead looping is a control mechanism in hardware (HW), which enables the execution of such loops without the overhead of modifying/checking a loop iteration variable and explicit jump instruction.

2. *Modulo-protected data memory address generation*

Cyclic buffers are often used within DSP algorithms. Such buffers can be constructed by means of address pointers which are incremented by each data memory reference. Often, these pointers can be programmed in such a way that these move within a defined range, and automatically wrap-around when they reach a range-bound value.

3. *Specialized datapath arithmetic*

A classical example of a data-path function which is inspired by finite impulse response (FIR) filter (4.3) algorithm is the Multiply-Accumulate operation. Within the FIR filter application, this operation multiplies a delayed copy of the input signal $x[n-k]$ with a filter coefficient a_i , and subsequently adds the result to an accumulation variable (i.e. $acc = acc + a_i \cdot x[n-k]$). Another example of data-path arithmetic which is typically useful in DSP is complex multiplication (i.e. multiplication of complex numbers). The datapath of virtually all modern programmable DSPs contain parallelism, in the sense that multiple functional units are independently available within a single instruction.

4. *Parallel load, process and writeback*

The multiply-accumulate (MAC) operation, as mentioned in the context of the FIR filter iteration, is often combined with a parallel fetch of the operand and coefficient, and packed in one dedicated instruction: fetch operand and coefficient, multiply and accumulate. In general, data fetching and data operations are often combined at instruction level within DSP architectures.

5. *Specialized addressing modes*

Common DSP algorithms, like fast Fourier transform (FFT) can be implemented efficiently when applying bit-reversed addressing. Such specific addressing modes are supported natively within many DSPs.

Programmable DSP architectures are often highly optimized, which tend to lead to an irregular architecture and non-orthogonal instruction-set. This implies that these DSP processors are often hard to program (e.g. assembly level, or C-based with compiler intrinsics to explicitly exploit the architecture specifics). Typically, optimized library modules, which are offered by the DSP vendor, are used. A background on DSP architectures can be found in [32]. Dedicated DSP chips are typically used in cost-effective embedded systems. Likewise, DSP cores are often applied as

accelerator functions within multi-coe systems. Because of its efficiency, DSP tend to have good low power characteristics.

Like the GPP, DSPs have seen the same three "walls" of the "von Neumann" architecture. Similar architectural developments, like the introduction of caches within DSPs, and enhanced ILP (e.g. very long instruction word (*VLIW*) and/or single instruction, multiple data (*SIMD*)) typically have shaped the evolution of DSP architectures. Logically, multi-core DSP architectures are common nowadays.

DSPs are widely applied within the application domain of musical signal processing and related modeling. The main applications can be found in embedded architectures, which are particularly applied in the domain of virtual analog modeling and digital-waveguide. Example products are guitar-effects processors (e.g. stomp-boxes or multi-effects processors, covering virtual analog like non-linear amplifier modeling and digital waveguide modeling such as reverberation) and synthesizers. Similar to the application of the GPP, complete ReMap applications, including interaction models and physical models, are implemented on DSPs. However, the DSP is better equipped to execute physical models, rather than interaction models (with GPP this is the other way around). Typical issues, related to the application of DSP, are the following:

» Scalability

Similar to GPP, multi-core DSP programming is difficult. In particular, partitioning of an algorithm and mapping of the partitions onto the different cores, with a unified memory model is a difficult problem.

» Portability

Because of the irregularities in many DSP processor architectures, the programming model is often low-level: the details of the processor architecture are often exposed to the assembly-level or compiler intrinsics programming structures. Apart from the required expertise to program such machines efficiently, this implies a portability problem. When a specific DSP algorithm implementation needs to be ported to a different processor (e.g. processor variant, or to another processor architecture), then this portability problem materializes.

» Programmability

We already mentioned that the irregular, non-orthogonal instruction set of many DSP architectures imply steep learning curves, and specialism. Often, software (*SW*) libraries of common DSP functions are made available by the DSP vendor. This makes the technology somewhat more accessible.

Some examples of ReMap applications on programmable DSP can be found in [93] [3] [24] [86].

FIGURE 2.3 – Modern FPGA device

2.3.3 FIELD-PROGRAMMABLE GATE ARRAY

The essence of an FPGA is that it contains a fabric of configurable logic blocks, combined with a configurable interconnect network. The principle of a configurable logic block (*CLB*) is depicted in fig.2.3. A CLB allows to define an arbitrary n -input boolean function, implemented as a random-access memory (*RAM*)-based lookup-table. Next to this function, the CLB contains a configurable flip-flop through which the output of the lookup table (*LUT*) can be routed. In between the LUT and the output stage (i.e the flip-flop part) is a function called "Carry & control Logic". This part enables, amongst others, the efficient construction of an adder for which a carry-chain is required. Each CLB can be connected to others through a configurable interconnect network. This implies that a CLB fabric and its associated interconnect network can implement virtually any digital circuit. An FPGA needs to be configured before it can be used. This means that the FPGA is in principle a chip with an undefined function when it is not configured. Hence, the FPGA is an extremely versatile digital processing device. The design flow for an FPGA (i.e. at least the "front-end" design phase) has many similarities with the design flow for an ASIC. The objective is to develop a digital circuit, usually in a hardware description language (*HDL*) language, which can be automatically mapped on the FPGA. Applications for FPGAs are as versatile as its configurability. In the early days, FPGAs have primarily been used to "glue" chips together in a system on printed circuit board (*PCB*). While Moore's law progressed, FPGAs have become widely used as application specific chips, similar to ASICs. FPGAs are also widely used in the design process for ASICs for verification and validation purposes (i.e. ASIC prototyping). The main differences between FPGA and ASIC are that FPGAs usually consume more silicon area to achieve a similar function and ASICs are usually developed for high volume and cost optimized functions. Nowadays, FPGAs have the capacity to embed mid-range complex system-on-chip (*SoC*)s. Next to the fabric of CLBs, a modern FPGA contains a wealth of other specific building blocks

(e.g. embedded memory, GPP), like shown in fig. 2.3. A state-of-the-art FPGA device is heterogeneous and may contain the following basic building blocks:

1. *CLB*
Fabric of configurable logic blocks and associated configurable routing mesh.
2. *Configurable IO blocks*
Input and output signal pins of the FPGA can usually be configured (e.g. electrical levels, slew-rate, differential/single-ended, registered/combinatorial).
3. *Block RAM*
Embedded RAM can be combined with logic by means of block-RAM cells. These blocks can usually be configured as single or dual-port memory devices.
4. *Phase-locked loop (PLL)/delay-locked loop (DLL)*
Several clock domains can be configured by means of these blocks. Likewise, synchronization with external clock signals can be obtained.
5. *DSP functions*
These are basic arithmetic units, like dedicated multipliers. These units can be configured in order to build more complex DSP data-path functions such as a programmable floating-point unit (e.g. add/subtract, multiply/add, divide, logarithmic operations).
6. *Embedded processor core*
Embedded GPP (e.g. ARM) enable the configuration of heterogeneous embedded systems, where one or more GPPs can be combined with soft-core intellectual property (IP)s, which are mapped on the CLB fabric.
7. *High-speed serial IO (transceivers)*
These blocks can be applied to configure high speed interfaces like PCIe or universal serial bus (USB).
8. *Configurable interconnect*
All enumerated functional blocks can be connected in a versatile way by means of a configurable interconnect fabric.

An FPGA is a so-called "bit-parallel", or "fine-grained" configurable device. Bit-parallelism follows from the fact that each LUT within a CLB has a 1-bit output, and works in parallel to all other LUTs. Each LUT output can be routed to virtually any other LUT input through the configurable interconnect network. Soft-cores (i.e. IP blocks with arbitrary functions like processors, state-machines, DSP functional blocks) which are described in a register transfer level (RTL) language, can be mapped on an FPGA. The main cost of this versatility is that the interconnect between the logical primitives (i.e. the CLBs) is expensive in both silicon area and power consumption. However, the increasing amount of "DSP functions" within modern FPGAs, along with other non-CLB functional blocks like embedded GPP

cores, is a trend towards more efficient silicon utilization. Hence, the inclusion of these special blocks decreases the efficiency gap between FPGAs and ASICs. Sampling rates are usually modest Within ReMap applications: the typical frequency range of interest is dominated by the audible frequency range of the human ear (i.e. 20Hz-20kHz). In some cases, a physical model needs to be executed at a multiple of this rate (e.g. non-linear behaviour). Despite the fact that the required sampling rates may be a multiple of the audio sampling rate, these frequencies are typically way below the feasible clock frequency within FPGA designs (e.g. 200MHz clock frequency). On the other hand, a detailed physical model requires a vast amount of computational resources, which need to be executed for every sampling period.

High level design tools for FPGA are often focussed on a direct mapping of an algorithm on the available FPGA resources, leading to space-optimized solutions which greedily occupy the available logic resources. Such a solution typically is able to run at a high sampling rate (e.g. 200MHz), often much higher than the required sampling rate. For physical models with relatively low sampling rate this implies that an FPGA would in general be used inefficiently when a space-optimized design methodology is applied.

Hence, it is not trivial to apply FPGAs efficiently for physical modeling. In principle, an efficient space/time optimized model of computation needs to be derived which subsequently can be mapped on an FPGA. Such a model leads naturally to a programmable many-core architecture. Mapping of a physical model on such a many-core architecture means that the model needs to be partitioned in space and time. Each partition needs to be mapped on a single core, and the interfaces between the partitions need to be mapped on the many-core interconnect network. Each core executes a physical model partition sequentially (e.g. temporal domain). The allocation of different physical model partitions to different cores implies parallel execution of these partitions (e.g. spatial domain). The development of a many-core architecture for physical modeling on FPGA is not trivial:

1. A domain specific many-core processor needs to be designed and subsequently implemented in a HDL.
2. The algorithm which needs to be mapped on the many-core processor needs to be partitioned in such a way that each partition can be mapped on a single core. Furthermore, the connections between these partitions need to be identified and mapped on the interconnect network of the many-core topology. This process can either be done manually, or in an automated way. Automation often implies the development of specific mapping software.
3. Given the fact that we are dealing with a domain specific processor solution, most probably automated mapping tools needs to be developed.

This principle of space/time partitioning and resource utilization by means of dedicated softcores on FPGA has been applied for detailed physical modeling in several studies [52], and shows that it is far from trivial to derive a domain specific space/-

time optimized processor concept.

Another related problem with FPGAs is that these processors are hard to "program". The traditional HDL design methodology requires specific engineering skills which are usually not the same as those required and common by acoustical/sound engineers. Some attempts have been made to bridge this gap through coupling of domain specific functional programming to FPGA technology [90]. This study, however, is yet another example of a space-optimized solution for which we have shown that these do not yield efficient exploitation of FPGA resources within our domain of interest.

Conclusion

FPGAs offer a great potential for massively parallel computational intensive applications, like detailed physical models. Interaction models are usually more efficiently mapped on GPPs, rather than on the CLB fabric. Given the fact that today often a GPP is integrated on the same silicon die as the CLB fabric, the FPGA can be a very efficient heterogeneous processor architecture for ReMap applications. It is however far from trivial to exploit the versatile capability of these devices *most efficiently*, due to the typical modest sampling rates within ReMap applications. The development of a domain specific application solution on FPGA has many similarities with the development of a similar solution on ASIC. This requires multi-disciplinary engineering skills. There exists a significant silicon efficiency gap between FPGAs and ASICs with respect to area and power mainly due to the large interconnect overhead. This gap, however, is decreasing with the diversification of primitive FPGA building blocks.

2.3.4 MULTI-CORE AND COARSE-GRAINED ARCHITECTURES

In the introduction of this section we explained how the effects of Moore's law have led to the multi-core era. Multi-core on itself implies a broad domain of different processor architectures. We focus on three types of many-core architectures which are interesting to consider:

1. Course-grained reconfigurable architectures
2. GPP-inspired many-core architectures
3. Domain-specific many-core architectures

Course-grained reconfigurable architectures

The development of CGRAs was triggered by the area and performance gap between FPGAs and ASICs. The aim of a CGRA is to reduce the configurable interconnection area overhead, which is dominant in FPGA architectures. A CGRA consists of a fabric of "coarse-grained" building blocks which can be interconnected in a versatile way. The fundamental difference between a FPGA and a CGRA is that the latter is based on a determined data-width (e.g. 16-bit), rather than the 1-bit data

FIGURE 2.4 – Montium Tile Processor

granularity within the FPGA. Typically, a coarse-grained building block is a configurable ALU with a fixed data width. The basic idea is that a network of configured ALUs (e.g. multipliers, adders, etc.) can implement a wide variety of DSP functions (e.g. FIR filter). Unlike the FPGA, the CGRA usually requires less configuration data (less routing, less instances of basic primitive functions in an implementation) and less interconnect wiring. So this means that the CGRA is more efficient for data-path centric structures but less versatile, compared to a FPGA. A general overview on CGRA can be found in [26]. Similar to the FPGA, a straightforward application of CGRAs often leads to a space-centric model of computation. Such models can be very efficient for applications with sampling rates that are within a similar range, compared to the clock frequency of the fabric. For ReMap applications, such a straightforward usage of CGRAs is not as efficient because sampling rates are typically way below the feasible fabric clock frequency.

For applications with relatively low sampling rates a fabric of ALUs can be utilized more efficiently when these ALUs are scheduled both in the spatial *and temporal* domain (similar to a FPGA). Scheduling the ALU in time implies that the ALU needs to be wrapped in a (small) programmable core. Next to the ALU this core should contain an instruction sequencer, local storage to administrate variables and state, and interfaces to neighbour cores. Based on this concept, quite a few architectures evolved [50] [6] [108]. We highlight one example, called the Montium by Recore Systems [29].

The Montium consists of one or multiple Montium tile processor (*TP*)s, linked within a network-on-chip (*NoC*) topology. The block diagram of the Montium TP is depicted in fig. 2.4. Each TP consists of an array of 5 ALU slices, an instruction sequencer and associated decoder, and a NoC interface (i.e. the "communication and

configuration" block). Each ALU contains fixed-point arithmetic units, enabling operations like multiply-accumulate. The Montium TP can be classified as a VLIW core. However, its control structure and related programming model differs significantly from classical programmable cores. The Montium TP has a configurable instruction set. These instructions basically consists of a set of separate control words for the different units in the processor (e.g. 5xALU control, 10x memory control, and communication control), which functions can be manipulated in full detail in parallel at this level. This low-level control, which is actually the lower-layer of the configuration concept, is stored in small RAMs within the instruction decoder unit. In other words: the instruction decoder is configurable, and thereby the instruction set of the machine is configurable. The actual (compact) instructions are stored in a program memory within the instruction sequencer. An instruction at this level consists of a number of fields, where each field resembles a lookup address for the associated control part of the machine (e.g. an address which selects one out of N instructions within ALU control).² This specific control structure results in a simple instruction decoder of the processor, which typically results in small configuration file sizes and low power. Likewise, each instruction enables full parallel control over all the resources within the TP. Typical applications for the Montium are a broad class of DSP functions in several domains. Despite the fact that the Montium has never been applied as musical/acoustical modeling processor, its architecture seems to be suitable for this purpose. Programming the Montium, however, requires quite a lot of expertise and in-depth knowledge of the architecture. The programming methodology is based on a derivative of the C-language, linked to a dedicated programming tool-chain.

GPP-inspired many-core architectures

The Epiphany, developed by Adapteva, [98] is a many-core architecture which is inspired by a classical superscalar CPU model and associated programming (C-based). The architecture is based on a matrix of NoC-interconnected "mesh-nodes", a control processor (e.g. based on a Xilinx Zynq SoC with a dual-core ARM GPP), and external shared DDR memory. The matrix of mesh-nodes (see fig. 2.5) acts as a co-processor to the GPP. Each mesh-node comprises a floating-point reduced instruction set computing (RISC) processor core, a direct memory access (DMA) engine, local memory (32kByte shared instruction/data) and a NoC interface. Each local memory within every mesh node is visible in the programming model as a distinct part of one shared memory map. This implies that each core has read/write access to the memory of each other core, and to a shared external memory. The Epiphany architecture is scalable in the sense that the number of cores within the matrix can be configured up to 64k cores. The current state of the art is that a 64-core processor is available in silicon in a 28nm process. Claimed applications are diverse: smartphone applications (real-time face recognition, speech recognition, translation, and augmented reality), low-power supercomputing, and

²A similar configurable instruction concept was developed by Philips Semiconductors within a DSP for wireless communication [42]

FIGURE 2.5 – Epiphany processor

floating-point acceleration for embedded applications based on FPGA architectures (based on IP model). Within the domain of physical modeling we found a relevant publication on a heat distribution application model, based on FDTD modeling (i.e. a model based on stencil computation) [99]³. This publication reveals that the application scales well on the given architecture. Programming, however, is not trivial due to shared memory consistency management and limited memory per core. Particularly, partitioning of a large FDTD model has shown to imply in-depth knowledge of the many-core architecture (in particular the shared memory distribution and related consistency).

Domain-specific many-core architectures

The ASynchronous Array of simple Processors (ASaP) has been developed by VLSI Computation Laboratory at the University of California [108]. The architecture is a scalable many-core, of which the cores are interconnected in a NoC topology, like depicted in fig. 2.6. The interconnection principle is based on streams of data which flow between adjacent cores, and buffered through first-in-first-out buffer (FIFO)s. Each core is enabled to exchange data between its four neighbour cores. Communication between cores which are not within this neighbourhood needs to be achieved explicitly by means of pass-through actions, carried out by the core which has to pass the information. Each core has its own clock generation unit (osc), which is entirely independent. This implies that the ASaP core fabric is based

³Heat distribution models are similar to physical models for acoustical applications in the sense that they are based on difference propagation through a physical medium

FIGURE 2.6 – ASaP processor

on the globally asynchronous locally synchronous (*GALS*) principle. This *GALS* principle means that each core operates within its own island of synchronicity, and hence asynchronously w.r.t. other cores in the fabric. The FIFO buffering between adjacent cores implicitly forms a solid clock domain boundary between adjacent cores. Cores which are not used by a particular application are switched-off. This architecture concept yields good scalability properties and is energy efficient, because of the utilization of the locality principle and the *GALS* property. An ASaP core is a single-issue fixed-point core (i.e. parallelism within the data-path is limited to one operation per instruction). Apart from the ALU, the core contains small embedded memories: 64-word instruction memory, 128-word data-memory, configuration units (i.e. static and dynamic configuration). Note that the core neither has instruction/caches, nor a DMA capability. This implies that the entire task needs to fit on the core w.r.t. both instruction and data footprint, and that data exchange entirely relies on neighbour cores through the FIFOs.

The ASaP programming model is task-level oriented. The basic idea is that many DSP algorithms can be decomposed as a sequence of tasks which are connected in a "processing pipe". One example is JPEG image compression, which can be decomposed in different relatively independent subsequent tasks like discrete cosine transform (*DCT*), quantization, zig-zag reordering and entropy coding. Such a class of algorithms which can be decomposed to a chain of subsequent, loosely coupled tasks (a compute pipe) is sometimes categorized as "embarrassingly parallel". Each task is supposed to be coded in the C-language and linked into a block-diagram oriented way of programming. The ASaP compiler then automatically takes care of core allocation and mapping of each task onto an allocated core.

Typical applications for ASaP are modem pipes (e.g. IEEE 802.11a/g wireless LAN transmitter and receiver), JPEG image compression, and other algorithms which can be relatively easily decomposed into a chain of loosely coupled tasks. For many ReMap algorithms, the ASaP processor could be used quite well. Many of these algorithms can also be expressed as a processing pipe, with relatively independent tasks. However, we see difficulties on the following aspects:

1. *Mapping of relatively large strongly cohesive tasks*

Detailed physical modeling implies periodic iteration of FDTD schemes [8]. The nature of such schemes is that these can be represented as very large strongly connected graphs (this thesis, section 4.4). Ideally, such a scheme is captured in one task. However, the size of such graphs usually implies partitioning, where each partition needs to be mapped on a dedicated core. The interconnections between partitions implies intensive data exchange between neighbourhood partitions. In particular, this aspect might be a problem for mapping this class of problems on the ASaP processor. Moreover, when data from within the task needs to be transported to the boundary of the core-fabric (e.g. stimuli or response data to/from the FDTD scheme), then the additional load to the forwarding cores in the fabric might become problematic.

2. *Access to shared memory*

Mapping of delay-lines in practice means the necessity of using a relatively large shared memory. Particularly within the sub domain of digital-waveguide modeling, often a large amount of distributed delay-lines might be required in the algorithm. The decomposition of such an algorithm onto different cores probably implies that many cores need to have access to one shared external memory. This seems to be difficult to map on the ASaP processor, since access to external memory is not straightforward.

2.4 PROGRAMMING METHODOLOGIES

Perhaps the most utilized method of programming signal processors is via classical imperative languages such as C or the native assembly language of the target chip. Libraries of typical signal processing primitives (such as digital filters, delay lines and signal transforms) are offered by chip vendors and third parties alike. This approach is relatively straightforward and tends to result in efficient utilization of the hardware, as long as the algorithms being implemented can be expressed in terms of such signal processing primitives. When the algorithm in question is more complicated, the traditional method of low level programming is no longer suitable. Implementation of novel processing primitives typically requires a high level of expertise on both the algorithm and the hardware in question, and the resulting work is tightly coupled with a specific processing architecture. These factors make signal processing code averse to being portable.

2.4.1 HIGH-LEVEL DSP LANGUAGES

Several approaches exist for programming signal processors without a high level of C or Assembly expertise. National Instruments LabVIEW [92] is a graphical interface to the *G language*, which is based on data flow. However, the LabVIEW system is proprietary and opaque, so utilization of custom hardware like DSPs is not easily achievable. Another commercial DSP design methodology is based on

MathWorks' MatLab and SimuLink (e.g. [39]). SimuLink supports generation of C code or direct synthesis of signal processing cores via a hardware description language. These approaches are focused on multi-domain simulation and are likely too elaborate for design and application of musical signal processors.

2.4.2 MUSICAL PROGRAMMING LANGUAGES

Musical Programming Languages are a topic of active research. This section presents a brief overview of influential design paradigms and language implementations specific to musical signal processing. The most widely adopted paradigm for musical signal processing is the unit generator (*ugen*) concept [74, pp. 787–810]. The *ugen* concept was solidified by the MUSICn family, of which CSound [14] is the contemporary example. Pure Data [67] and SuperCollider [48] are important developments of the concept. Programming *ugen* is declarative by nature. The programming task focuses on signal flow and topology, rather than the sequential ordering of program statements. *Ugen* languages can be criticized for their lack of higher level program constructs. Especially graphical front ends such as Pure Data [67] – an example of a very successful implementation – make it quite difficult to express data types, control flow or program composition, all of which enhance programmer productivity. Several attempts have been made to address this problem. CLM [77] attempts to merge the *ugen* concept with Common Lisp [87], an acclaimed high-level language. Since making high-level signal processing programs efficient is difficult, CLM delimits signal processors to a certain subset of Common Lisp and transcompiles that to C. The resulting programming paradigm is a mixture of styles, featuring a C-like programming style with Common Lisp syntax. SuperCollider [48] is a more actively developed idea in the same vein. SuperCollider is influenced by SmallTalk (e.g. [34]), and integrates *ugens* built in the C language. As a result, SuperCollider programs are tied with the implementation of the run time library, and the actual signal processors are opaque to the user. Faust [59] attempts to address signal processing in a functional high-level idiom. Faust features an expressive block diagram composition system capable of succinctly describing many typical signal flows. Faust transcompiles to C, with recent work aimed towards direct compilation using the LLVM [43] framework. Kronos [54] is the language used and adapted for an experimental heterogeneous ReMap application, which is presented in chapter 5 of this thesis, and published in [MJW:3]. Within this experiment, distinct code is generated for both the interaction model, as well as the real-time physical model of a graphic equalizer application. Influenced by Faust, Kronos offers a functional signal processing paradigm, enhancing it with advanced metaprogramming capabilities, automatic factorization and an advanced compiler pipeline [55]. Kronos makes use of LLVM [43] for native code generation. For the purposes of the mentioned experimental, heterogeneous ReMap compilation, an experimental code generator has been developed, along with facilities for heterogeneous compilation of a uniform source program for several distinct hardware targets [MJW:3].

2.5 CONCLUSIONS

We gave an overview of the domain specific area of musical signal processing from application, processor and programming point of view. An abstract of this overview is as follows:

1. *Applications*

ReMap applications consist of real-time physical models, and interaction models. We distinguish three categories of physical models: virtual analog models, digital-waveguide models, and detailed physical models. A real-time physical model can often be described as synchronous dataflow (SDF). A more thorough support on this statement is given in chapter 4 of this thesis. Interaction models implement the interface between the user (e.g. musician) and the physical model. Hence, interaction models usually respond to events which are translated to physical model parameters. Because of the event-driven nature of interaction models, such models do not necessarily fit to SDF computational models.

2. *Processor architecture*

The processor architectures which are used for ReMap applications are related to the complexity of the physical models. Most common processor architectures are the GPP (predominantly interaction models, and low complexity physical models), programmable DSPs (low to mid complexity physical models, and to some extent interaction models), and FPGAs (detailed physical models, but not very well suited for interaction models). Particularly, for real-time detailed physical modeling a many-core architecture is required. We, however, did not find a suitable many-core processor architecture which is actually being used for this purpose. In general, interaction models fit best to a GPP, because such models perform control processing (i.e. best-effort execution, event-driven). Physical models fit best to DSP/FPGA architectures (i.e. hard real-time requirements).

3. *Programming model*

The data-flow nature of the majority of physical models naturally results in a preference to model algorithms in a declarative way which reflects the mathematical properties. This results in a block diagram based modeling methodology (e.g. ugen) where blocks reflect basic signal processing functions (like filters). Domain-specific languages, like Faust or Kronos, satisfy the need to express modeling problems in a functional manner. Most of the commercially available processors, however, have a C-based programming entry. Hence, a compiler for a domain specific language often generates a C-based compiled representation, which then can finally be compiled to a target processor. The main disadvantages of this C-based approach are that the generated code potentially needs to be partitioned over the available processor cores. This is often a manual process, and is known to be difficult in a many-core context.

We can extract the following problems from the observations in the state-of-the-art of ReMap processing:

Problem 1: scalability in relation to detailed physical modeling.

For detailed physical modeling, a programmable many-core processor architecture seems not to exist. Quite some research has been done on dedicated, FPGA-based approaches. Nevertheless, we did not find an existing and suitable programmable architecture.

Problem 2: real-time constraints.

Low processing latency and time-predictability are critical requirements in DAFx applications. These requirements are often conflicting with commonly applied processor architectures (i.e. GPP or programmable DSP).

Problem 3: programmability.

Functional programming methodologies, often domain-specific, are commonly used within the ReMap domain. However, the commonly applied processor architectures have a C-based compiler. This results in programming methodologies which are not transparent: it is for instance difficult to have full control over the real-time constraints while programming.

These problems give input to the question whether it is feasible to define a ReMap specific processor technology. The difference in nature between interaction models and physical models call for a heterogeneous approach: interaction models fit best on GPP processors, while physical models fit best on a multi-core architecture. In the next section we will propose such a domain-specific multi-core processor, which we have called the WaveCore. This WaveCore processor is optimized to physical modeling, and interacts with associated interaction model(s) which are mapped on a GPP processor architecture. Hence, this WaveCore approach requires a heterogeneous processor platform.

THE REMAP PROCESSOR ARCHITECTURE

ABSTRACT – We identified the following four main problems within processing architectures for ReMap applications: long latencies, difficulties in time-predictability, scalability and programmability. We propose a ReMap processor architecture which addresses these problems. This architecture consists of a GPP (i.e. for mapping interaction processes), the WaveCore processor (i.e. for mapping physical models), shared memory and peripherals. We developed a data-flow programming language, called WaveSlang, for this architecture. The WaveCore processor is a multiple-instruction multiple-data (MIMD) architecture, and consists of a scalable network of processing units (PU). Each PU embodies a small RISC processor with an instruction-set that is optimized to data-flow concepts. Furthermore, domain-specific optimizations, like native support for delay-lines, have been made to the WaveCore architecture. The semantics of the WaveSlang language match closely to mathematical properties of ReMap applications, and also match closely to the WaveCore processor hardware. This makes the WaveCore processor efficient, and programmable at a domain-specific abstraction level while keeping a high degree of control over processing latency, parallelism and time-predictability of execution. We developed the WaveCore processor, a compiler, simulator and a FPGA based development platform¹.

3.1 INTRODUCTION

We introduced a general execution model for ReMap applications in chapter 1 for DAFx applications (fig. 1.1) and synthesis applications (fig. 1.2). This general execution model consists of a real-time physical model and an associated interaction model. Next to that we derived the modeling and execution requirements for real-time physical models and found that these fit best to a scalable multi-core DSP

¹Parts of this chapter have been published in [MJW:6], [MJW:5] and [MJW:3]

FIGURE 3.1 – Heterogeneous ReMap processor architecture

processor architecture. Furthermore we found that the execution requirements for interaction models fit best to a GPP architecture. Therefore, we propose a heterogeneous processor architecture for ReMap applications. An abstract block diagram of this "ReMap processor architecture" is depicted in fig. 3.1. Our aim is to map interaction processes onto the GPP, and real-time physical models on the WaveCore processor, which is our main contribution. The "peripherals" in fig. 3.1 contain all devices that interact with either the real-time physical model (e.g. ADC/DAC) or interaction model (e.g. keyboard). The purpose of the shared memory is to exchange data between interaction model, physical model and peripherals. The "Programmable Timer" generates periodic events, and is meant for synchronization purposes. The ReMap processor architecture contains a control bus and a data bus. The GPP is the master on the control bus (solid connection dot), while the WaveCore processor and I/O peripherals are slave devices to the GPP (open connection dot). All devices have access to the shared memory (solid dots on the data bus).

The most important requirements for the ReMap processor architecture are the following:

1. *Scalable physical modeling*
The WaveCore processor should be scalable. This is required in order to enable the mapping of relatively low complexity physically informed models, virtual analog models and large complexity detailed physical models.
2. *Ultra low latency*
It should be possible to stream an input signal (from peripherals) to the WaveCore processor, process this signal, and redirect the output signal to an output device with a predictable and low latency. Typically, this latency should be shorter than 7 ms.
3. *Real-time guaranteed execution*
Execution of a physical model on the WaveCore processor should be as

predictable as possible with respect to execution time.

4. Domain specific programming

It must be possible to program the ReMap processor architecture, using a domain specific programming methodology. Moreover, from a programmers perspective it must be possible to control the real-time properties of the algorithms. This means that the mapping of a ReMap algorithm onto the heterogeneous ReMap processor architecture should be transparent.

The organization of this chapter is as follows:

We start with a description of a synchronous data-flow model on which the WaveCore processor is based. The aim is that this data-flow model should enable the application developer to describe entire ReMap applications, where real-time physical models and interaction models can be explicitly included. Next to that we define a ReMap programming model in section 3.2, which we call WaveSlang. WaveSlang is a data-flow language which supports the heterogeneous ReMap processor architecture as a compile target. In section 3.3 we define the WaveCore processor architecture. We elaborate on the developed WaveSlang mapping tools (i.e. compiler, simulator and application development board) in section 3.5. We conclude this chapter in section 3.6.

3.1.1 DATAFLOW MODEL FOR REMAP APPLICATIONS

Dataflow is a model of computation which, as its name suggests, is driven by the availability of data [46]. A dataflow model of computation can be described as a graph. An example of a dataflow graph is depicted in fig. 3.2. A dataflow graph consists of *vertices* and *edges*. Each *vertex* represents a function which processes (i.e. consumes) data elements, called *tokens*, and generates (i.e. produces) tokens. The vertices are connected in a graph by *edges*. Each edge carries data, and hence transports data between vertices. A vertex is executed (i.e. *fired*) when the *firing-rule* is satisfied. Usually, the firing-rule describes what has to be done when there are one or more tokens available at the incoming edges of a particular vertex. Upon firing, a vertex consumes input tokens, processes these and produces one or more tokens which are made available at the connected outbound (i.e. outgoing) edges of the vertex. In the example graph in fig. 3.2, vertex V1 fires when a tokens are available on incoming edges E1 and E5.

SDF is a specific class of a dataflow, with the following characteristics:

1. Each vertex consumes and produces a predefined fixed number of tokens, each time that the vertex is fired.
2. The execution time of each firing is fixed.
3. The production of tokens is unconditional, which means that there is no data dependency in the execution model.

The consequence of these characteristics is that the execution schedule (i.e. the fire-ordering of the vertices, according to the data dependencies in the graph) can

FIGURE 3.2 – Dataflow graph example

FIGURE 3.3 – SDF example: FIR filter

be derived at compile-time [45] for SDF. Consequently, this implies that firing of each individual vertex in the SDF graph can be scheduled at a fixed timestamp, without the need to explicitly check for the availability of token(s) at the incoming edges. We illustrate these aspects by means of a FIR filter example in fig. 3.3. The FIR-filter in fig. 3.3 can be modeled as a SDF-graph, which is depicted in the same figure. The vertices v_6 , v_7 , v_8 , and v_9 represent the multipliers b_3 , b_2 , b_1 and b_0 respectively. Vertex v_2 represents a unity delay. Vertices v_3 and v_4 represent an addition with a unity delay at the output (i.e. the grey-coloured boxes in the FIR filter schematic). Vertex v_1 replicates input token $x[n]$ to its four outgoing edges. A valid static schedule for the example SDF graph is defined in table 3.1. Two subsequent graph execution cycles (initiated at $t=0$ and $t=T_{sample}$) are captured in the table. All vertices are fired within three subsequent phases ($n=1$, $n=2$, $n=3$). Within phase $n=1$, vertex v_1 is fired. During phase $n=2$, the vertices v_6 , v_7 , v_8 and v_9 are

TABLE 3.1 – Periodic SDF schedule of the example FIR filter

T=0			T=Tsample		
n=1	n=2	n=3	n=1	n=2	n=3
v1	v6	v2	v1	v6	v2
-	v7	v3	-	v7	v3
-	v8	v4	-	v8	v4
-	v9	v5	-	v9	v5

FIGURE 3.4 – WaveCore process graph

fired in parallel. Finally, during phase $n=3$ the remaining vertices $v2$, $v3$, $v4$ and $v5$ are fired in parallel. This schedule is static, which means that for every execution cycle the identical firing order and parallelism is applied. This implies that there is no need to check for the availability of input tokens in order to fire the vertices. It is, however, required that the input data is available, and that output tokens can be written at any condition.

In the introduction chapters 1 and 2 we identified that real-time physical models within ReMap applications predominantly can be described as synchronous data-flow graphs [44], while interaction models are event-driven (i.e. not tied to a periodic time cadence). A suitable dataflow model in which both real-time physical models and interaction models can be described, therefore needs to be hybrid. Hybrid in this context means that we need to define a dataflow model in which both SDF and event-driven processing can be combined. We define a dataflow model, which is based on a process graph description. This process graph description in its general form is depicted in fig. 3.4. This model, called WaveCore process graph (WPG) has the following characteristics:

1. *Process graph description*

A WPG contains a number of processes which are connected to each other by means of edges. The WPG describes a ReMap application, including real-time physical model and interaction model. Furthermore, the WPG allows to describe interfaces to external devices, like ADC or DAC (i.e. the I/O processes in fig 3.4).

FIGURE 3.5 – Synchronous physical model timing

FIGURE 3.6 – Asynchronous physical model timing

2. *Explicit process scheduling*

We use the SDF property that a statically derived schedule does not require the explicit checking for token availability as a firing rule for each individual process. We introduce explicit SDF process scheduling by means of a central scheduler. This scheduler emits fire-events through the "FireLinks". Every process which is connected to a specific firelink responds to the fire-event by firing the process once.

Hence, a firelink can be seen as a clock-wire. Each process, which is connected to a firelink, is executed once per "clock-cycle" (i.e. fire-event).

Through this centralized scheduler, we can minimize token buffering. This implies that the latency can be kept to a bare minimum, and predictability of execution is increased because SDF process execution is strictly synchronized to the central scheduler.

3. *Real-time physical modeling*

Real-time physical modeling is supported by means of SDF-like processes. Each SDF process in the WPG may be synchronous to a different time-base (i.e. multi-rate processing).

4. *Interaction modeling*

Interaction models are supported by means of control-type processes. These processes are event driven (e.g. plucking a virtual guitar string) and therefore not associated to a firelink. Interaction processes communicate with physical model processes through edges. The direction of these edges may be from interaction process to physical model process (e.g. filter coefficients), or vice versa (e.g. status information from the physical model).

5. *Edge behaviour*

An outgoing edge of an arbitrary process may be connected to more than one other processes. Produced tokens are *read* by one or more consumer process(es), but *are not removed* from the edge as a side-effect. The rationale for this choice is that this behaviour enables broadcasting of tokens by a single producer process to multiple consumer processes. The number of tokens which can be buffered within an edge is configurable when producer and connected consumer process(es) are *synchronous* (i.e. they share the same fire-link), and zero (i.e. no buffering) when producer and consumer process(es) are *asynchronous* (i.e. they share different fire-links). A token-buffer within a process edge behaves like a delay-line (i.e. $y[n] = x[n - M]$, with $y[n]$ the output of the edge, $x[n]$ the input of the edge and M the buffer-capacity in number of tokens).

We analyze the timing behaviour for asynchronous (i.e. different fire-links for the physical model and I/O processes) and synchronous (i.e. one shared fire-link for the physical model and I/O processes) WPG topologies in the following sections.

Interaction processes are event driven. This implies that the interaction between the real-time process(es) and the interaction process(es) is *per definition* asynchronous. There is no token-buffering in edges between asynchronous processes. The associated timing behaviour is depicted in fig. 3.6. An interaction process responds (i.e. fires) to an *event* at an arbitrary timestamp, and it requires a time period T_{cp} to fire this process. During T_{cp} , the interaction process samples the *Status* token which is *assumed to be produced* by the *real-time physical model*, and produces the outbound token *Param*. During T_{cp} we cannot specify the exact moment at which the process consumes/produces the mentioned tokens. As a result, the moment at which the interaction process *reads* the *Status* token may overlap with the moment that the *real-time physical model* updates mentioned *Status* token. This implies that the interaction process may read a partially updated *Status* token (i.e. partially *Status*[1] and partially *Status*[2], as indicated with the orange-coloured area in fig. 3.6), if the *Status* token consists of more than one data-element. We denote such a token as being *time-inconsistent*. Note, however, that the read token in itself is *data-consistent*: it contains all data elements. The same phenomenon may occur in the other direction, where a time-inconsistent *Param* token is transferred from the *real-time physical model* to the *interaction process*, as is also indicated in fig. 3.6. Note that time-inconsistent tokens also occur when real-time processes share different fire-links. This is indicated in fig. 3.6, where both the *Stim* and *Resp* tokens may be time-inconsistent. For instance, the second *Resp* token in fig. 3.6, which is read by the *I/O2* process, consists of *Resp1*, *Resp2* and *Resp3* fractions.

The programmer has to be aware of the side-effects of time-inconsistent token consumption, due to asynchronous scheduling of processes. In many cases, time-inconsistent tokens are not a problem. In particular, when the Nyquist criterion [57] is fulfilled for a signal (i.e. carried by a token-stream) which is transmitted between two asynchronous processes, then no side-effects will appear from partial token processing. If time-inconsistent tokens are not acceptable in an application, then a mechanism in software has to be implemented which exchanges tokens in an atomic fashion. One example to establish this is by labelling a token with a validation flag, where this flag is a data element within the token and updated after that the token content (i.e. payload) is updated by the producer process. Upon being fired, a consumer process can evaluate this flag, and conditionally process the token.

Interaction between synchronous processes

The timing behaviour of a WPG where the real-time processes share only one fire-link (i.e. $FL1 = FL2 = FL3$) is depicted in the example in fig. 3.5. Within this example, every edge between real-time processes buffers one token, in order to circumvent time-inconsistent token consumption. The implication of this buffer is a delay of one fire-period for each edge between these real-time processes. This is made

FIGURE 3.7 – WaveCore Process Graph example

visible in fig. 3.5, where the *Out* signal within process *I/O2* is delayed with two firing periods with respect to the *Stim* token stream.

3.2 PROGRAMMING MODEL

We introduced the WPG as a method to describe a ReMap application. A WPG is a process graph in which interaction processes, physical models and periphery can be described. This implies that it must be possible to map a WPG onto the heterogeneous ReMap processor architecture in fig. 3.1. In this section we present a full description of the WPG structure. Next to that we introduce a programming language, called *WaveSlang*. WaveSlang enables the description of a WPG in a textual format. The WaveSlang compiler, which will be introduced in section 3.4, automatically extracts the physical model from a WPG and maps it onto the WaveCore processor.

3.2.1 PROCESS GRAPH DEFINITION

A WPG consists of a set of processes which can be arbitrarily interconnected by means of process-edges. An example of a WPG is depicted in 3.7. A WPG supports four process-types:

1. *WaveCore process*

The physical model is composed of one or more WaveCore processes. A WaveCore process behaves like a SDF process. A WaveCore process executes (i.e. is fired) when it receives a fire event from the connected "firelink". Upon execution, a WaveCore process consumes one token per incoming edge, and produces one token per outgoing edge. Checking of token availability upon consumption is not done. The process assumes that the token

is available. As already mentioned, the physical model is mapped on the WaveCore processor.

2. *Control process*

Interaction models are composed of one or more control processes. A control process may generate tokens (i.e. control tokens) in an event-driven way. Like mentioned, interaction models are mapped on a GPP. Within the example, there is one edge E1 which is connected (i.e. branched) to WaveCore processes 1 and 3.

3. *Foreign process*

The purpose of a foreign process is to include SDF-like processes, other than WaveCore processes. Examples of foreign processes are ADC and DAC. Within the example, foreign process 1 might be a black-box model of an ADC, which synchronously generates tokens to WaveCore process 1 through edge E2. Hence, foreign process 1 responds to the same firelink FL1 as WaveCore process 1. Similarly, foreign process 2 might be a black-box model of a DAC, which synchronously consumes tokens from WaveCore process 2 through edge E6, synchronous to firelink FL3.

4. *Central scheduler process*

The central scheduler process has one or more outgoing edges. Through each edge, called "firelink", periodic fire-event (i.e. "clock") signals are transmitted. Each firelink is connected to one or more WaveCore or foreign processes. Each connected process is executed (fired) when it receives a fire-event. Basically, the scheduler can be seen as a set of modulo-counters, where each modulo-counter generates a periodic signal when the modulo state is reached. Hence, each fire-link is generated by a dedicated modulo-counter. Therefore, the scheduler is not aware of the required execution times of the processes. The execution times are static, and derived at compile-time. The central-scheduler process is executed by the *Programmable Timer* in fig. 3.1.

Through the definition of these process-types it is inherently supported to map a WPG on the heterogeneous system in fig. 3.1.

As mentioned, processes are interconnected with process-edges. A process-edge is always connected to one producer process, and may be connected (i.e branched) to more than one consumer-processes (e.g. process-edge E1 in 3.7). A process-edge is connected to an "outbound port" of a data-producing process, and connected to an "inbound port" of one or more consuming processes. A process produces one token through each of its outbound ports each time that it is executed ("fired"). Likewise, each process consumes one token per inbound port each time that it is fired. Checking for token availability upon consumption is not done. All process-types assume that their inbound tokens are available upon execution. Like mentioned, all processes except for control processes within the WPG are scheduled by means of the central scheduler. This scheduler periodically emits fire tokens to the connected

FIGURE 3.8 – Edge model and token buffering

processes through dedicated firing edges, called "firelinks" (i.e. FL1, FL2, and FL3 in fig. 3.7). The main advantages of the central scheduler are the following:

1. It is not necessary to check for the presence of inbound tokens.
2. Deadlock is not possible because each process is fired periodically.
3. Overall process graph execution timing is strict and predictable. Real-time deadlines can be considered per process. If all process deadlines are met, then the process graph deadline is met as well.

Edge and token definition

A process-edge connects an outbound port of a process to one or more inbound ports of another, or the same process. The process-edge serves as a medium to exchange tokens from one producer to one or more consumers. The processes themselves write or read tokens into/from the process-edge, synchronized to the firing link to which they are connected. This implies that a process-edge needs to contain a physical buffer, which can hold at least one token. A token consists of one or more elementary data entities, called primitive token (*PT*). The actual token size depends on the application (e.g. two *PT*s for a stereo audio channel). The process-edge model is depicted in 3.8. A process-edge consists of a circular token-buffer which can hold up to N tokens. At the token production side of the edge (connected to the outbound port of a process) a token write pointer is maintained. This pointer increments at each fire event at the producer process side, and wraps around when the maximum token address ($N-1$) is reached. The produced token is written in the addressed token buffer entry when the process is fired. In fig. 3.8, the token present on the *obEdge* is written in *TokenBuffer 1* when fire-link 'fire' is activated. A process which is connected to the inbound side of the process-edge also uses a circular pointer mechanism to fetch tokens from the same circular buffer at each firing event. The fact that a process-edge may be branched implies that each consuming process has access to the same physical token buffer, which is also indicated in section 3.8.

The required buffer capacity N is derived at compile time, and depends on the synchronicity of the consuming and producing processes to which the edge is connected. The rationale for this buffering capacity has been explained in section 3.1.1.

(1) Producer and consumer process are *synchronous*.

Processes are synchronous, when they share the same fire-link (i.e. a single fire-link which is branched to several processes). In this situation the edge behaves like a delay-line with static length $N-1$. If for example $N=2$, then the producer and consumer processes access alternating token-buffer entries (sometimes referred to as a "ping-pong" buffer). Effectively, a delay of one fire period is induced by the edge in this example. The minimum value of N is two, in order to circumvent time-inconsistent tokens (see section 3.1.1). Note that the write-pointer should be initialized to 0, and the read-pointer to 1, before processing.

(2) Producer and consumer process are *asynchronous*.

In this situation, the token transmission ordering will get corrupted when a buffer-length, different than 1, is configured. This is because the read and write pointer will increment at different rates due to the different fire-event frequencies. Therefore, a token buffer capacity needs to be configured to 1 in this situation.

3.2.2 PROCESS DEFINITION

The structure of a WPG process is depicted in 3.9. This structure is common for the three defined process-types: WaveCore, foreign and control processes. A process is composed as follows:

1. *Fire port*

The physical model part of the WPG consists of a SDF-like graph. This part of the WPG must be synchronized to a time-base, being a firelink. The purpose of the "fire-port" is to connect a firelink to the process.

2. *Inbound ports*

The purpose of an inbound port is to connect a WPG edge to the process (i.e. token consumption). A process has N inbound ports (N is a non-negative integer, including 0). Upon a fire-event that is received through the fire-port, the process-core (*PCORE*) fetches a token $T_{ibi}[n]$ through each inbound port $IbPort_i$.² from the token-buffer.

3. *Outbound ports*

The purpose of an outbound port is to connect a WPG edge to the process (i.e. token production). Each process has M outbound ports (M is a non-negative integer, including 0). Upon fire-link assertion, the *PCORE* writes a token $T_{obi}[n]$ to the token-buffer, through each outbound port.

²We use the notation $g[n]$ to specify the value g at discrete time-stamp $t = n.\Delta T$ with ΔT the firing period.

FIGURE 3.9 – Process structure

FIGURE 3.10 – Block diagram of the example WPG in fig. 3.7

4. PCORE

The PCORE is the implementation of the physical model. The PCORE is synchronous to the fire signal. The PCORE is executed (i.e. fired) once upon assertion of the fire signal. During execution, the PCORE consumes the inbound tokens (one token per inbound port), and produces the outbound tokens (one token per outbound port).

The block-diagram of the example WPG in fig. 3.10, in terms of the structures for processes, edge buffers and scheduler, is depicted in fig. 3.7. We further describe the PCORES for the three WPG process types (WaveCore, foreign and control processes) in the following sections.

PCORE for a foreign process

Like mentioned in the introduction, a foreign process behaves like an SDF process. Within the context of a WPG, the foreign process models a peripheral which is directly connected to the physical model within a ReMap application. As such, the foreign process is not mapped on the WaveCore processor. Examples of such peripherals are ADC, DAC, but might also be processors on which SDF graphs are mapped (e.g. programmable DSP or HW accelerator). From WPG perspective, the PCORE of a foreign process is a black-box model which is not further specified. The relevance of including black-box models in the WPG description is to describe how real physical processes are embedded in the ReMap application.

PCORE for a control process

A control process implements (part of) the interaction model of a ReMap application. We mentioned that the control process is not modeled as a SDF graph, unlike the physical model of the ReMap application. However, a control process is compliant to the definition of WPG edges. The firing rule of a control process is not defined in the context of a WPG. Typically, a control process is fired when a user event takes place. An example of such an event within a DAFx application is that the user changes an effect setting, like a slider in a graphic equalizer. Upon an event, the PCORE of the control process consumes/produces tokens in the same way as SDF processes: one token per edge. Similar to the PCORE of a foreign process, the PCORE of a control process is a black-box model in the context of a WPG.

PCORE for a WaveCore process

The WaveCore process is the only process within a WPG for which the PCORE is explicitly defined. The PCORE for a WaveCore process is composed of one or more WaveCore PCORE partition (WPP) instances. Furthermore, a WPP is composed of one or more primitive actor (PA) instances. The rationale to partition a WaveCore PCORE into WPPs is to enable the mapping on the scalable WaveCore architecture, like we will show later. The PCORE of a WaveCore process can be modelled as one or more Mealy-machines [49], as is depicted in fig. 3.11. The rationale to choose a Mealy-machine abstraction of the PCORE of a WaveCore process is because of the explicit distinction between state (i.e. registers) and combinatorial (i.e. computational) parts. This distinction forms the basis of the ability to partition a PCORE into multiple WPPs. Next to that, it enables automated partitioning and mapping of a PCORE on the scalable WaveCore architecture as we will show in section 3.4. A Mealy-machine abstraction of the WaveCore PCORE consists of two parts:

1. *A combinatorial part*

The combinatorial part of the Mealy-machine computes the outputs $Y[n]$, according to (3.2). Likewise, the combinatorial part computes the next-state $S[n]$, according to (3.1). An important property of the combinatorial part

FIGURE 3.11 – Mealy-machine abstraction of a WaveCore process-core (PCORE)

is that it does not contain state: it can be described as a set of one or more directed acyclic graph (DAG) [36] instances.

2. *One or more state registers*

The state register holds the current state $S[n]$ and updates this state each time that a synchronization event (i.e. firing event in case of a WaveCore-PCORE) takes place. Important to notice is that the amount of state $S[n]$ is static (i.e. time-invariant).

$$S[n] = F(S[n-1], X[n]) \quad (3.1)$$

$$Y[n] = G(S[n-1], X[n]) \quad (3.2)$$

The combinatorial functions $S[n]$ and $Y[n]$ can be modeled as one or more Directed Acyclic Graphs. A DAG (i.e. within the context of a Mealy-machine) is a data flow graph with the following properties:

1. Each edge in the DAG is *directed*. This means that token-flow through an edge can only be in one direction.
2. For every vertex V there is no path which starts from the outgoing edge of vertex, through other vertices within the DAG, back to an incoming edge of the vertex V . The DAG does not contain loops/cycles and hence is *acyclic*.
3. Within the context of a Mealy-machine, none of the vertices in the DAG contains state. In other words, all vertices in the DAG are *combinatorial*³.

3.2.3 WAVECORE PROCESS PARTITIONING

Computationally complex PCORES (e.g. detailed physical model implementations) may be too large to fit on a single processor core. Therefore, it is required that a PCORE can be partitioned into one or more WPPs.

³We add this property to the DAG, because we strictly consider DAGs in the context of Mealy-machines. The logic within Mealy-machines, being the functions F and G in eq. 3.1 and eq. 3.2, do per definition not contain state and hence the DAG(s) F and G are *combinatorial*

FIGURE 3.12 – PCORE partitioning

We introduced the Mealy-machine abstraction of a PCORE, and said that the combinatorial part of this structure, which may belong to more than one Mealy-machine, can be modeled as a set of DAGs. The WaveCore processor, as we will describe in section 3.3, consists of a configurable number of processing units. Each PCORE partition (WPP) is mapped on one such a processing unit, and the communication between the WPPs is mapped on an interconnection infrastructure. The PCORE partitioning strategy that we chose is optimized to the locality principle. From a locality perspective, it is required to have the operand data as close as possible to the arithmetic hardware which performs the actual computations. We observed that a DAG does not contain state. This implies that, if we optimize to maximum locality, then all the DAG input data is local (i.e. within a WPP). This leads to the PCORE partitioning, as depicted in fig. 3.12, and explained as follows:

A WPP contains one or more DAGs. The registers (i.e. state variables) which outputs are DAG inputs are located in the same WPP. The remaining DAG inputs originate from inbound token data (i.e. $X[n]$ or a subset of $X[n]$). Now, it can be the case that a particular WPP₁ contributes to the state variables of WPP_m, as depicted in fig. 3.12. In that case, this "remote state" information $S_r[n]$ needs to be moved from one WPP to the other WPP(s) in which the remote state information is required. The exchange of remote state $S_r[n]$ between WPPs is mapped on mentioned interconnection infrastructure within the WaveCore.

Partitioning of a PCORE into multiple WPPs is based on the allocation of indi-

FIGURE 3.13 – Example of partitioning of a parent-WPP into two child-WPPs

FIGURE 3.14 – Space/time schedule of a mapped PCORE

vidual DAGs within the PCORE onto processing units within the WaveCore. This process, called *spatial mapping*, will be explained in section 3.4.2. A partitioning example is given in fig. 3.13. The PCORE of the example WaveCore process initially consists of one WPP, being the *parent WPP*, WPP_p . This WPP_p consists of two DAGs (i.e. DAG1 and DAG2), and two state registers A and B. Partitioning of the WPP_p leads to two *child WPPs*, being WPP_{c1} (containing DAG1) and WPP_{c2} (containing DAG2). As can be seen, this partitioning leads to replication of the state register A. This replication is required because the register A output feeds into DAG1 (and hence needs to be kept local within WPP_{c1}) and feeds into DAG2 (and hence needs to be located in WPP_{c2} as well). After partitioning, the replicated register A_r and register B have become *remote-state* registers.

The proposed strategy of DAG allocation to a set of WPPs yields a space/time schedule, like the one depicted in fig. 3.14. The "space" dimension in this context

implies the number of WPPs which are executed in parallel, combined with the parallelism within the processing unit on which the WPP is mapped. The "time" dimension implies that firing a DAG is according to a temporal schedule: a number of processor clock-cycles is required in order to execute the mapped DAG. The duration of firing a DAG depends on the parallelism and data dependencies in the DAG itself (see also the example SDF in table 3.1) and the parallelism in the processing unit on which the DAG is mapped. The example space/time schedule of the PCORE in fig. 3.14 contains 7 DAGs, which are allocated to three different WPPs, and subsequently mapped on three processing units. After receiving a fire event, the three units start firing the DAGs according to a compile-time derived schedule. After firing all DAGs, the remote state $S_r[n]$ is updated.⁴

As can be seen in the example schedule, the WPP mapping process needs to be such that the real-time deadline (being the fire-period T_{fire}) is met. The example schedule also shows that the computational complexity of the physical model (i.e. the number and complexity of the DAGs) and the fire-period (i.e. the real-time deadline) are the constraints which should be used to find the optimal space/time mapping solution (i.e. the number of WPPs which are used to allocate the DAGs). We will further elaborate on the mapping process in section 3.4.

3.2.4 THE PRIMITIVE ACTOR

We modeled the PCORE of a WaveCore process as a Mealy machine. The rationale for this is that it forms a foundation for a space/time balanced mapping strategy. We introduced this strategy in the previous section and will further elaborate on it in section 3.4. This strategy results in a partitioned PCORE, where each partition is modeled as a WPP, and interaction between WPPs is modeled as remote-state updates.

We concluded that a WPP consists of one or more DAG(s) and state registers. The actual implementation of this structure consists of interconnected Primitive Actor (PA) instances. The definition of the PA is driven by the requirements from both the SDF characteristics, as well as physical modeling aspects (functional), as we will explain in this section. A WPP in itself is a SDF-like graph. The vertices within this graph are PAs. The PA is the atomic arithmetic entity within the programming model. The PA (see fig. 3.15) has three incoming edges: $x_1[n]$, $x_2[n]$, $\tau[n]$ and one output edge $y[n - \lambda[n]]$. Next to that the PA has three parameters, which are derived at compile-time: function f , parameter p and delay-line length Λ . Upon firing, the PA consumes one PT per incoming edge and produces one PT through its outgoing edge. The PA consists of two parts:

(1) A combinatorial part.

The combinatorial part of the PA has two incoming edges $x_1[n]$ and $x_2[n]$ and

⁴The actual temporal schedule for each WPP can be such that the execution of the DAGs and associated remote-state updates is interleaved.

FIGURE 3.15 – Primitive Actor

outgoing edge $y[n]$. This combinatorial part computes a token PT, according to the generic equation (3.3). The function f determines the operation (e.g. addition, multiplication). Parameter p is used in the arithmetic operation as well (for certain functions f , like multiplication with a constant p).

(2) *A sequential part (optional).*

The sequential part of the PA consists of an optional, and configurable delay-line. The delay-line has an incoming edge $y[n]$ (i.e. the outgoing edge of the combinatorial part of the PA), and a second incoming edge $\tau[n]$. The outgoing edge of the delay-line emits a delayed token PT. The delay-line has a maximum length Λ , which is statically defined at compile-time. If Λ equals zero, then the sequential part of the PA is not present, and hence the outgoing edge $y[n]$ is in that case the outgoing edge of the combinatorial part of the PA. If Λ is equal to 1, then the length of the delay is 1, independent of τ . For values of $\Lambda > 1$, the *actual* length $\lambda[n]$ of the delay-line depends on tokens from incoming edge $\tau[n]$, according to eq.(3.4). This means that the delay-line length $\lambda[n]$ can be dynamically varied (i.e. data-dependent delay-length).

The computation of $\lambda[n]$ by equation 3.4 is explained as follows. A fraction $\lambda[n]$ of the maximum delay-length Λ is computed by multiplying Λ with a scaling factor $0 < (\tau[n] - 1) < 1$. The result is a number which ranges between zero and the maximum delay-length Λ . The result of the computation needs to be casted to a positive integer number because the delay-length $\lambda[n]$ is not supported to be fractional. This type-casting is expressed by the truncation operator $\lfloor x \rfloor$ ⁵.

$$y[n] = f(x_1[n], x_2[n], p) \quad (3.3)$$

⁵The actual implementation in HW is based on fixed-point arithmetic. This is the reason that the floating-point number $\tau[n]$ is required to be within the range [1:2]. Casting to a fixed-point number within that range does not require HW resources, since the 23-bit mantissa of the floating point number linearly represents the range [0:1] in that case.

$$\lambda[n] = \begin{cases} \lfloor \Lambda \cdot (\tau[n] - 1) \rfloor & \text{if } \Lambda \geq 2; \text{ with } 1 < \tau[n] < 2 \\ \Lambda & \text{if } \Lambda \leq 1 \end{cases} \quad (3.4)$$

The firing condition for the PA depends on whether it is a combinatorial PA (i.e. $\Lambda = 0$), or a sequential PA (i.e. $\Lambda > 0$):

- » The firing condition for a *combinatorial* PA is met, when all required input tokens are available.
- » The firing condition for a *sequential* PA is met, when all required input tokens are available, *and* its outbound token is consumed by all connected PAs.

The motivation for the inclusion of a delay-line as an elementary PA-attribute comes from the observation that the signal delay is a fundamental element within physical models in ReMap applications, like described in section 2. The motivation for variable, data-dependent delay-line length is based on ReMap applications where acoustical wave propagation characteristics are not constant. An example is the Flanger effect ([109] or section 4.2), which is based on multi-path acoustical wave interference, where the path-lengths themselves are time-varying. Another example is the Leslie speaker [109]. This sound effect is based on a rotating speaker in a closed cabinet. This speaker rotation results in interfering Doppler effects, which can be mathematically modelled with varying and interfering wave propagation along different paths.

The type of a PA is defined statically at compile-time through function-type f . A limited number of functions, which are derived from the most common elementary arithmetic operations within digital signal processing algorithms, are defined in table 3.2. An example WPP of a so-called transposed direct-form II biquadratic

TABLE 3.2 – PA types

PA type	Mnem	Function
Constant/connector	CON	$y[n] = p$ (or inbound PT)
Adder	ADD	$y[n] = x_1[n] + x_2[n]$
Comparator	CMP	$y[n] == (x_1[n] > x_2[n])$
Divider	DIV	$y[n] = x_1[n]/x_2[n]$
Logic NAND	NND	$y[n] = \text{NAND}(x_1[n], x_2[n])$
Multiplier	MUL	$y[n] = x_1[n] \cdot x_2[n]$
Mul/Add	MAD	$y[n] = p \cdot x_1[n] \cdot x_2[n]$
Amplifier	AMP	$y[n] = p \cdot x_1[n]$
Noise Gen	RND	$y[n] = p \cdot \text{rnd} < 0 : 1 >$

filter [80] is depicted in fig. 3.16. The biquadratic filter implements an elementary second order difference equation, and is widely applied in audio processing algorithms (e.g. graphic equalizers). This infinite impulse response (IIR) filter type can be configured as lowpass, highpass, bandpass, band-rejection or allpass filter

FIGURE 3.16 – Biquad filter, represented as a WaveCore process

depending on its coefficients. The difference equation for the biquadratic filter is defined in (3.5).

$$y[n] = \text{Gain} \cdot (b_0x[n] + b_1x[n-1] + b_2x[n-2] - a_1y[n-1] - a_2y[n-2]) \quad (3.5)$$

The associated WPP for this filter is depicted in fig. 3.16. Note that $\tau[n]$ edges for the PA instances are not explicitly defined in fig. 3.16, because none of the specified delay-line lengths are larger than one: λ does in that case not depend on τ , as is defined in (3.4).

3.2.5 WAVECORE PROGRAMMING LANGUAGE

The WaveCore programming language, called WaveSlang, is a textual representation of the programming model that is introduced in this chapter. The language consists of three main constructs, which directly reflect the associated programming model constructs: a construct to describe a WPG, a process and a WPP.

WPG description

The language construct to describe a WPG is defined in listing 3.1. In compliance to the programming methodology, the language allows to describe a WPG by means of the ability to instantiate and define the scheduler, and an arbitrary number of arbitrarily interconnected processes.

- » The *Scheduler* is instantiated by means of the "Scheduler" construct. The name of the scheduler is defined with the character string "SchedulerID".
- » The "Frequency" parameter defines the clock frequency of the WaveCore processor (e.g. "Frequency 860160000" implies that the processor runs at

```

1 // Instantiation of the Scheduler:
2 Scheduler <SchedulerID> {
3     // Clock frequency:
4     Frequency <IntegerNumber>
5     // One or more FireLink(s):
6     FireLink <FireLinkID> <ClockDivision>
7 }
8 // Instantiation of one or more processes:
9 Process <ProcessID> {
10    // Synchronization through a link to the scheduler:
11    FireLink <SchedulerID> <FireLinkID>
12    // Process source-file reference:
13    Source <SourceFileName>
14    // Zero or more connected inbound-edge(s):
15    InEdge <EdgeID> <PortID>
16    // Zero or more connected outbound-edge(s):
17    OutEdge <EdgeID> <PortID>
18 }

```

LISTING 3.1 – Process graph syntax

86.016 MHz).

The scheduler may have more than one firelinks. Each firelink is defined with the construct "Firelink". The fire-rate is derived as follows:

fire-rate=Frequency/ClockDivision. Example: "FireLink AudioRate 1792" creates a fire-link called "AudioRate" which submits fire events at a frequency of 48 kHz to all connected processes.

A process is instantiated by means of the *Process* construct.

- » The name of the process is defined with the character string "ProcessID". This implies that a process should be defined, which name should match with "ProcessID" (see process description below). Note that more than one identical processes can be instantiated.
- » The "FireLink" statement links the process instance to the scheduler with name "SchedulerID" and firelink with name "FireLink".
- » The "Source" statement is a reference to the file, with name in character string "SourceFileName", which describes the process (see process description below).
- » The process may have one or more incoming edges. For each edge a connection between the edge-port, called "PortID", and the edge with name in character string "EdgeID" is established. Both "PortID" and "EdgeID" should be unique character strings.
- » Likewise, the process may have one or more outgoing edges. For each edge a connection between the edge-port, with name in character string "PortID", and the edge with name in character string "EdgeID" is established. Both "PortID" and "EdgeID" are unique strings.

```

1 ProcessEntity <ProcessID> {
2     // Process type:
3     ProcType [WaveCore,Foreign,Control]
4     // Declaration of one or more PCORE-partition(s):
5     // (only applicable to WaveCore process-type)
6     Wpp
7         WppID      <WppID>
8         WppSource  <SourceFileName>
9     EndWpp
10    // Declaration of 'remote-state':
11    // (only applicable to WaveCore process-type with partitioned PCORE)
12    RemState
13        // One or more 'Route' statement(s):
14        Route <ProducerWppID> <ConsumerWppID> <OutSignalName> <InSignalName>
15    EndRemState
16    // One or more edge-port declaration(s):
17    Port <PortID>
18        Direction [InBound,OutBound]
19        // Declaration of one or more signal(s):
20        Signal <WppID> <SignalName> <InitialValue>
21        // Declaration of zero or more port-attributes:
22        Attribute [Attr] <AttributeValue>
23    EndPort
24 }

```

LISTING 3.2 – Process entity syntax

Process description

The description of a *Process* is given in listing 3.2. A process with a unique name "ProcessID" is defined by means of the *ProcessEntity* language construct.

- » The *ProcType* statement defines the process type: either "WaveCore", "Foreign" or "Control".
- » Only in case of a WaveCore process, at least one WPP should be present. For each WPP, an associated construct "Wpp" specifies a reference to the actual WPP. The statement "WppID" gives a unique name "WppID" to the instantiation of the WPP. Therefore, more than one identical WPP can be instantiated in the PCORE of the WaveCore process.
- » The *RemState* statement describes the remote state signals between the several instances of WPPs. Hence, this statement is only applicable to PCORES with more than one WPP.
- » A remote-state connection between two WPPs is described with the *Route* statement. The WPP instance, with the name defined in character string "ProducerWppID", drives a remote-state edge with name defined in character string "OutSignalName". The WPP instance, with the name defined in character string "ConsumerWppID", receives the remote-state in an incoming edge with name defined in character string "InSignalName".

```

1 BeginWPP
2 // Zero or more connected PA instantiations:
3 <f> [<x1[n],nc>] [<x2[n]>,nc] [ $\tau$ [n],nc] <y[n-lambda[n-1]]> <p> <Lambda>
4 EndWPP

```

LISTING 3.3 – Process partition syntax

- » Each process has one or more ports. The description of each port, with the name defined in character string "PortID" is given with the statement *Port*. For each port the "Direction" is described: either "Inbound" or "Outbound". For each port, the mapping of the token that is passed through to/from the PCORE must be explicitly defined. This is done by means of one or more *Signal* statements. A signal with a name defined in character string "SignalName", within WPP with a name defined in character string "WppID" is connected to the port with the name defined in character string "PortID". An attribute can be linked to a port. This attribute is a directive for the WaveCore simulator, and has no further relevance at this point.

The token-buffer sizes cannot be specified in the language. These buffer sizes depend on whether producer and consumer processes are synchronous to the same fire-link, or not, like explained in chapter 3.1.1. When producer and consumer process share the same fire-link, then the buffer-size is assumed to be 2. If consumer and producer process share a different fire-link, then the buffer-size is assumed to be 1.

WPP description

The language construct for the definition of a WPP is described in listing 3.3. A WPP consists of a number of interconnected PAs in a directed graph. Within the WaveSlang language, each text line instantiates one PA in tabular format, according to listing 3.3 as follows:

- » The keyword *f* specifies the function of the PA, according to the table 3.2
- » The second column specifies the name of the incoming edge $x1[n]$. If this edge does not apply to the PA, then this column should indicate this by being specified as "nc"
- » The third column specifies the name of the incoming edge $x2[n]$. If this edge does not apply to the PA, then this column should indicate this by being specified as "nc".
- » The fourth column specifies the name of the incoming edge $\tau[n]$. If the delay-length $\Lambda < 2$, or the delay-length is static, then this edge can be kept unconnected, indicated with "nc" (not connected)
- » The fifth column specifies the name of the outgoing edge $y[n]$ of the PA.

- » The sixth column specifies the value of parameter p . If the PA does not have a parameter, then this value is ignored by the compiler but still should be specified (dummy parameter).
- » The seventh column specifies the delay-length Λ .

A WaveSlang code example of a WPG, which describes the BiQuad filter application, can be found in appendix A.

3.3 WAVECORE PROCESSOR ARCHITECTURE

The WaveCore processor architecture is derived from the requirements which are defined in introduction of this chapter. Next to that, the architecture can be seen as a logical consequence of the programming model which is defined in the previous section.

3.3.1 SCALABLE CLUSTER OF PROCESSING UNITS

According to Flynn's taxonomy [23] the WaveCore processor can be classified as a MIMD architecture. The motivation for a MIMD architecture originates from three aspects:

- 1) The computational complexity of detailed physical models lead to a scalability requirement for the processor. In section 2.3 we observed that a many-core architecture meets this scalability requirement well.
- 2) The programming model that we introduced in section 3.2 implies a space/time mapping which is illustrated in fig. 3.14. Each WPP implies a different execution thread, yielding different data/instructions. This results in the fact that mapping this schedule on a MIMD architecture is much easier compared to a SIMD approach.
- 3) The typical fire-rates, which are associated to ReMap physical models, are most often relatively low compared to feasible clock frequencies of digital circuitry. The ratio between feasible clock frequency and fire-rates is typically three orders of magnitude. This is a strong motivation for the choice of a programmable processor concept (computation in time), instead of processing in space completely.

At top-level, the WaveCore architecture is a scalable and interconnected cluster of processing unit (*PU*) instances, scheduler, external memory interface and host-processor interface (*HPI*). The architecture is depicted in fig. 3.17. The *PU* embodies an optimized RISC processor. The objective is that the physical model within the WPG, which we introduced in chapter 3.2, can be automatically mapped onto this cluster of *PU* instances by the WaveCore compiler. The *scheduler* is basically a programmable timer, and generates fire events to the *PU* cluster. This scheduler is a straightforward implementation of the scheduler within a WPG (i.e. a programmable timer). The WaveCore acts like a slave processor to a host, according

FIGURE 3.17 – WaveCore processor block diagram

to the conceptual ReMap processor block diagram in fig. 3.1. The HPI acts like a host/WaveCore processor interface. The PU cluster and scheduler can be initialized through this HPI. The PU cluster has access to a shared memory through the *external memory interface*, in accordance to the ReMap processor architecture in fig. 3.1. The PUs are connected in a ring-topology. The rationale for a ring topology is to keep the interconnection infrastructure as simple (and hence cheap in terms of area) as possible.

The organization of this section is as follows:

In section 3.3.2 we explain the relation between the programming model and the WaveCore processor.

In section 3.3.3 we specify the PU architecture. The PU consists of two major parts: the graph iteration unit (*GIU*) and the load/store unit (*LSU*). In section 3.3.4 we will elaborate on the GIU architecture, and in section 3.3.5 we will explain the LSU architecture.

Each WPP in the WPG is entirely mapped on one particular PU. Given the fact that a PU is a programmable RISC processor, the WPP is sequentially executed in a thread that is synchronized to periodic fire tokens. Sequential execution means that each PA within the mapped PCORE is executed in a compile-time derived static temporal schedule. Spatial parallelism is obtained through mapping of different WPP instances on different PU instances. The remote-state $S_r[n]$ is transmitted through a NoC topology called graph partitioning network (GPN). Each WPP may have a dependency with inbound or outbound ports and hence may be related to the transportation of tokens/token-parts. These tokens are allocated in external memory. Likewise, all delay-line buffers are allocated in external memory. This implies that each PU has access to this external memory. The scheduler process is mapped on the "Scheduler" within fig. 3.17. This scheduler is a programmable timer which broadcasts periodic firing events for each logical firelink to the PU cluster. Each individual PU can be programmed to synchronize to a unique fire event, and hence be logically linked to a periodic time instant to which the mapped WPP is synchronized. Fire events are broadcasted by the central scheduler to all PUs in the cluster. A fire event is associated with an ID (fire-link ID). This fire-link ID can be programmed in each individual PU. If a broadcasted fire-link ID matches with the programmed ID within the PU(s), and the scheduler 'fires' this fire-link, then the PU(s) start executing. Linkage between WaveCore processes and non-WaveCore processes (i.e. control and foreign) logically goes through process edges and hence physically goes through token exchange within the shared external memory. An externally connected GPP loads compiled program code to the program memories within each PU through the HPI.

Network-on-Chip

The NoC through which data exchange between PU instances takes place is a ring network, called GPN (see fig. 3.18 and fig. 3.17). The rationale for a simple ring-network is based on the observation that the amount of remote-state exchange through the GPN network is usually modest for many of the target applications (this will be shown in chapter 4 of this thesis). This justifies a relative simple NoC architecture, like a ring-network. Despite the fact that the primary objective of the GPN is to exchange remote state (associated to partitioned PCORE), this GPN is generic in the sense that every PU has the capability to issue remote-write transactions through this network. A GPN write-transaction consists of a data-element (PT) and an associated address (the target-PU and the local memory location within this PU). The PT and associated target address are combined in a packet which is generated upon a GPN-write transaction issue. Each PU evaluates passing packets: upon an address match, the packet is taken from the GPN ring and forwarded to the destination memory location within the matched-PU. If the address of a passing packet does not match, then the PU forwards this packet to the next PU in the ring network. Only write transactions are supported within the GPN network. The

FIGURE 3.18 – Graph Partitioning Network

motivation for write-only originates from the fact that remote-state $s_r[n]$, which is transmitted through the GPN network, can either be fetched from a remote PU by the other PU which consumes this remote state, or written by the producer PU to the consumer PU. We chose for the write option for two reasons:

- (1) Writes can be posted. This means that a producer PU can submit a write transaction onto the GPN ring, and proceed ("fire-and-forget").
- (2) Read capability is more expensive in terms of area: it requires a separate request channel and data channel (including data-tag to indicate the destination PU), while for a write-only capability only one write-request bus suffices in which write data and address can be combined.

The network is buffered (ring-buffer and transaction buffer per PU like indicated in fig. 3.18), in the sense that packet pass-through and issuing new transactions can happen simultaneously, provided that the buffering capability is not entirely occupied. Packet pass-through has a higher priority, compared to issuing new transactions, in order to prevent dead-lock. The amount of buffering capacity is a design parameter, and in practice only a few entries per buffer have shown to be sufficient for the application that we prototyped (chapters 4 and 6.3.3). In case of a full *transaction buffer* when a particular PU needs to issue a GPN transaction, the particular PU is stalled until the transaction can be issued. Deadlock is not possible because a stalled PU implies that the GIU pipeline is stalled, while it is still possible for the "Matcher" within the GPN port to write into the destination memory of the stalled PU. If a PU is not able to handle an incoming and matching GPN packet (i.e. because the target memory is being accessed at that moment, while the local ring-buffer is full) then the PU forwards the packet to the next PU in the ring (to

prevent deadlock of packet-passing through the ring): the packet travels an extra round through the ring. This mechanism implies that the packet-buffering capacity is evenly distributed within the GPN-network. The side-effect of this distributed buffering is that temporal scheduling of remotely issued GPN write transactions is determined by the GPN topology itself to a certain extent and hence not trivial to predict at compile-time. This is because the detailed clock-cycle behaviour of the GPN network needs to be known by the compiler. Furthermore, the exact moments at which GPN transactions are issued by each individual PU needs to be known at compile-time, as well as the exact buffering capacity. This "self-scheduling" and imprecise temporal behaviour aspect needs to be taken into account by the WaveCore compiler, as we will explain in section 3.4.

Memory architecture and locality of reference

We distinguish three locality levels within the WaveCore memory architecture. The objective is to minimize the amount of data-movements during execution of a WPG (exploitation of the locality of reference principle). These three levels are defined as follows:

1. *Locality level (1)*

All data storage which is associated to a single WPP (e.g. state information, and used memory to compute combinatorial functions) is located locally within the PU onto which the WPP is mapped. This means that each PU contains local data memory. The latency for this locality level is 1 clock cycle.

2. *Locality level (2)*

Data exchange between PU instances through the GPN network is defined as locality level 2. The latency between submission of a remote-write by the producer PU and the acceptance of the remote-write by the consumer PU depends on the distance between the producer and consumer PUs. Hence, the worst-case latency, in number of clock-cycles, equals the number of PUs plus a few additional pipeline stages within the GPN port.⁶

3. *Locality level (3)*

Token buffers and delay-line buffers are allocated to external memory, which is defined as locality level 3. The latency of this locality level very much depends on the external memory system topology. Typically, for a DDR based external memory this latency can easily be tens of clock cycles.

3.3.3 PROCESSING UNIT ARCHITECTURE

The heart of the PU (top-level depicted in fig. 3.19) is a pipelined RISC processor core, called Graph Iteration Unit (GIU). The GIU is optimized to sequentially execute all PA instances in the mapped WPP instance. The instruction set of the GIU is optimized to execute a PA within a single instruction. Given the fact that the

⁶The ring buffer within the GPN port in fig. 3.18 is a synchronous FIFO implementation. If the FIFO is empty, then the propagation of a transaction through this FIFO takes exactly one clock-cycle.

FIGURE 3.19 – Processing Unit block diagram

GIU is a pipelined processor, this implies that the GIU is designed to execute one PA per clock-cycle. The required ILP to achieve this yields the necessity for three independently accessible data memories ($Xmem$ and $Ymem$ for $x_1[n]$ and $x_2[n]$ PA-inbound PTs, and $Cmem$ for parameter p). The instructions for feeding the GIU pipeline are stored in local $Pmem$. The GIU is *loosely-coupled* with the Load/Store Unit (LSU) by means of the buffered inbound and outbound channels which implements mentioned coupling. The purpose of the LSU is to decouple the GIU from long external memory access latencies (e.g. fetching inbound token data or delay-line data). Therefore, the LSU embodies a programmable DMA controller. The instructions for the LSU, and required data (e.g. storage of pointers etc.) are located in $Pmem$ and $Cmem$. Hence, $Pmem$ and $Cmem$ are dual-ported where GIU and LSU can access these memories simultaneously through individually independent access ports. The rationale to share the physical $Pmem$ and $Cmem$ between GIU and LSU is to reduce area cost on the one hand, and the fact that FPGA memory resources (i.e. block-RAM) are usually dual-port memories. In the next subsections we will further outline the GIU and LSU architectures.

3.3.4 GRAPH ITERATION UNIT

The objective of the GIU is to sequentially execute all PAs within the mapped WPP, each time that the associated PCORE is fired. We define a GIU-thread as a compiled program which runs on the GIU. This GIU-thread is executed upon each linked fire-event. The instruction-set of the GIU is optimized, such that a PA is executed within a single instruction. A WPP is a representation of a SDF graph, where each vertex is a PA. We observed that every vertex in a SDF can be fired in a fixed, compile-time derived order. For the GIU this implies that a compiled WPP can be represented as a fixed list of instructions which can be sequenced in a linear order, each time a fire-event is received from the central scheduler. This implies that the instruction sequencer within the GIU can be kept very simple. *It is not required to support non-linear program flow (e.g. jumps, branches).* An incremental

FIGURE 3.20 – Graph Iteration Unit

program counter is sufficient to sequence through a compile-time derived thread in order to execute each PA within the WPP.

The GIU is a pipelined processor, such that it can execute one instruction per clock-cycle. Therefore, the thread-length (in number of instructions) for a given WPP-thread equals the number of PAs within the WPP (plus a few additional instructions for thread-initialization, as we will explain later) ⁷. If the pipeline is not stalled during thread execution, then the required number of clock-cycles to execute a WPP-thread is approximately the same as the number of PAs in the WPP.

GIU architecture

We mentioned that the instruction-set of the GIU is designed to execute a PA within a single instruction. The architecture of the GIU, which is depicted in fig. 3.20, is

⁷Additional "no-operation" instructions might be added, because of data-hazard, as we will explain in section 3.4.3

logically derived from this requirement. We explain the GIU architecture through an analysis on what is necessary to execute an arbitrary PA:

1. *Instruction fetch*

The type of the PA (see table 3.2), and the routing information of the PA (i.e. where the operands needs to be fetched from, and where the computed result should be sent to) is packed in one instruction. This instruction is fetched from *Pmem*, and the program counter is subsequently incremented.

2. *Operand fetch*

The PA needs at most two inbound PTs $x_1[n]$ and $x_2[n]$. The source of these PTs can either be PA outgoing edges which are stored locally in this PU, or token data from an inbound port of the associated WaveCore process. This implies that at most two operands need to be fetched from local memory, external memory (token data through inbound FIFO (*ibFIFO*)), or a combination. Given the requirement that it must be possible to fetch two operands from local memory, this local memory must be split into two independently accessible memories. Furthermore, these memories need to be dual-ported because operand fetching and writeback of computed result (pipelined processor) need to be possible simultaneously. These memories are called *Xmem* and *Ymem*.

3. *Parameter fetch*

The PA needs at most one parameter p . This implies that a third local memory, called *Cmem*, is necessary.

4. *The arithmetic operation*

The PA-function f determines the arithmetic operation, according to its definition in table 3.2. The actual arithmetic is performed by the ALU. The ALU operates on single-precision floating-point operands and is IEEE754 compliant.

5. *Writeback of computed result*

The destination of the computed result $y[n]$ from the ALU depends on the connection of the PA within the WPP. If the consumers of $y[n]$ are one or more PAs within the same WPP, then the destination is local data memory: either *Xmem* or *Ymem*, or both (replication of a PA output to both *Xmem* and *Ymem* is in some cases necessary, which will be explained in section 3.4). If the outgoing edge of the PA is linked to an outbound port of the associated process, then $y[n]$ is pushed into outbound FIFO (*obFIFO*). If the outgoing edge of the PA contributes to remote state, then $y[n]$ is forwarded to the GPN port.⁸ If the length of the delay-line is larger than 1, then the computed result $y[n]$ is pushed into *obFIFO* (i.e. delay-line input), and the delay-line output is fetched from *ibFIFO* and subsequently written into either *Xmem*, *Ymem*, or both. If the outgoing edge of the PA drives an incoming edge

⁸The associated GPN destination is fetched from either *Xmem* or *Ymem*. This implies that PAs which contribute to remote state can only have one connected incoming edge (e.g. AMP-type PA)

$\tau[n]$ of a different PA, then the computed result $y[n]$ is written to *Cmem* at a location which is determined by the address computation unit (*ACU*).

⁹ Note that this requires both a read-port and a write-port for the *Cmem* which introduces potential *Cmem* resource conflicts. A write-back buffer (*WBB*) resolves these conflicts through temporarily buffering of a *Cmem* write transaction, until an idle (i.e. PA without p dependency) *Cmem*-read cycle occurs.

The GIU is a pipelined processor. The following 15 pipeline stages are defined ¹⁰:

1. *Instruction fetch (FET)*

The program control unit (*PCU*) responds to a fire-event, which is received from the scheduler. This fire event initializes the program counter (*PC*) within the *PCU* to the location of the first instruction of the GIU-thread. The program memory *Pmem* is read by the *PCU* at location *PC* for every clock cycle until the end-of-thread (*EOT*) instruction which marks the end of the thread. The read instruction appears at the output of the *Pmem* in the next pipeline stage.

2. *Instruction decode (DEC)*

The operand addresses are computed by the *ACU*, according to the instruction. This *ACU* maintains a number of pointers which are necessary for operand fetch. The computed addresses are passed to the *Xmem* and *Ymem*, and the requested operands become available at the outputs of these memories in the next pipeline stage.

3. *Operand fetch (OP)*

The required operands become available within the *OP*-stage. Possible sources are *Xmem*, *Ymem* and/or *ibFIFO*. The operands are routed to the operand input registers of the *ALU*. Furthermore, a *Cmem* read transaction is initiated within this stage. As a result, the parameter p will become available as well in the next pipeline stage at the output of the *Cmem*.

4. *Execute (EXE), 10 stages*

The majority of the pipeline stages are present in the *ALU*. The most complex computational path through the *ALU* is derived from the requirement that a multiply and add stage needs to be cascaded, in order to implement the *MAD*-type ($y[n] = p.x_1[n] + x_2[n]$) PA. This path yields a total *ALU* pipeline length of 10 stages for the *FPGA* type that we used. The *ALU* delivers its computational result to the next pipeline stage.

5. *Writeback 1 (WB1)*

Depending on the destination of the computed result, which is dictated by the instruction, the computed result is written to either *Xmem*, *Ymem*,

⁹Information passing from GIU to LSU: the LSU uses the *Cmem* as data memory.

¹⁰ The majority of the pipeline stages are in the *ALU*. The implementation of the *ALU* is to a large extent technology dependent, and so are the number of pipeline stages in the *ALU*. In our *FPGA* implementation the number of pipeline stages for the *ALU* has been optimized to 10.

$Xmem$ and $Ymem$, or obFIFO. When the intended destination is GPN, then a $Xmem$ or $Ymem$ read transaction is set up in order to retrieve the remote GPN address. This address, along with the computed result are passed to the next stage.

6. *Writeback 2 (WB2)*

The GPN address and associated data are combined in a packet and pushed into the transaction buffer within the GPN port (see fig. 3.18).

Instruction-set for the GIU

We explained that the GIU is designed to execute a WPP, and that all PAs in a WPP thread are executed in a fixed sequence. Furthermore, we explained that the GIU pipeline is designed to execute one PA per instruction/clock-cycle. This means that a GIU instruction should describe the execution of an arbitrary PA. The mapping of a WPG to a WaveCore PU implies that the PTs which are passed over internal edges of the WPG should be translated to variables which are allocated to local memory resources $Xmem$ and $Ymem$, and the parameters p should be allocated to $Cmem$. Furthermore, delay-lines are allocated to dedicated buffers in external memory, and WPG tokens are also mapped to buffers in external memory. The allocation of all data is static. This data-set is derived at compile-time.

A GIU instruction operates on this static data-set. This means that the following information is necessary within the GIU instruction:

- 1) The PA function, like specified in table 3.2.
- 2) The location of parameter p .
- 3) The location of the operands $x_1[n]$ and $x_2[n]$.
- 4) The write-back location of the computed PA output $y[n]$.
- 5) Whether the PA has a configured delay-line, with $\Lambda > 1$. In that specific case, the delay-line buffer is allocated in external memory.

We define the GIU instruction consisting of two bit-patterns, like indicated in the instruction format in fig. 3.21. The first bit-pattern is called the *function opcode*, and describes the PA function, which covers (1). The second bit-pattern, called *ACU instruction* is required by the Address Computation Unit (ACU). The *ACU instruction* describes the location of the associated operand(s) and write-back data. Hence, the *ACU instruction* bit-pattern covers (2),(3),(4) and (5).

The definition of the *opcode* bit-pattern is defined in table 3.3. As can be seen in table 3.3, there are three function codes, which do not describe PA-types: NOP, TAU and EOG. The NOP instruction describes a 'no-operation', and is inferred by the compiler to solve data-hazards in the pipeline.¹¹ The TAU instruction moves

¹¹ A data-hazard is a condition where an instruction A is pending in the pipeline, and a next instruction B depends on the computational result of A.

FIGURE 3.21 – GIU instruction format

TABLE 3.3 – GIU instruction: function opcode definition

Opcode	Mnemonic	Description	Remark
0x0	NOP	No-operation	-
0x1	RND	RND-type PA	Noise generator
0x2	AMP	AMP-type PA	The AMP instruction is also used to map the CON-type PA.
0x3	MAD	MAD-type PA	-
0x4	MUL	MUL-type PA	-
0x5	DIV	DIV-type PA	-
0x6	CMP	CMP-type PA	-
0x7	NND	NND-type PA	-
0x8	reserved	-	-
0x9	reserved	-	-
0xA	reserved	-	-
0xB	reserved	-	-
0xC	TAU	Graph connection	Moves x1 operand to Cmem, at location p
0xD	reserved	-	-
0xE	reserved	-	-
0xF	EOG	End-Of-Graph	Indicates the end of the GIU thread

the $x1$ operand (i.e. TAU) to a location in $Cmem$. As we will explain in section 3.3.5, the LSU uses TAU to compute a read-pointer to a delay-line buffer. This is to support the dynamic delay-line length $\lambda[n]$, as described in the PA definition in section 3.2.4. The End-Of-Graph instruction (EOG) terminates GIU execution through disabling the PCU at the end of the GIU-thread.

The definition of the *ACU instruction* (and the ACU itself) is the result of the following analysis of the required information which should be packed in the associated bit-pattern.

A rough analysis of the required information starts with the size-dimensioning of the local memories: consider that the maximum GIU-thread length equals the capacity of the $Pmem$, which we define as C . Then the required variable storage capacity also equals C , since each PA output needs to be stored. This implies that the capacity of both $Xmem$ and $Ymem$ can be set to $\frac{1}{2}(C)$, as we assume that the variables can be equally divided over both memories. The required storage capacity for parameters p equals C , since potentially each PA in the thread needs a parame-

FIGURE 3.22 – ACU instruction: default mode

ter. The MAD-type PA requires the maximally possible input/output data (i.e. two operands, a parameter p and a write-back value $y[n - \lambda[n]]$), so we analyze the required information to guide this PA-type through the pipeline:

1. The number of required bits to address two stored operands $x_1[n]$ and $x_2[n]$ equals $2(\log_2(C) + 1)$ (note that the operand can either be fetched from local memory or from ibFIFO, hence the extra bit '+1').
2. The number of required bits to fetch a parameter p from $Cmem$ equals $\log_2(C)$.
3. The number of required bits to write a computed result back to its destination ($Xmem$, $Ymem$, $Xmem$ and $Ymem$, obFIFO, or GPN) equals $\log_2(C) + 1$.

So in total, this straightforward analysis yields a required ACU instruction width ACU_{iw} as defined in (3.6).

$$ACU_{iw} = 3 + 4\log_2(C) \quad (3.6)$$

Now, suppose that we let the maximum GIU-thread size be $C = 4096$, then the required instruction width equals $iw = 55$ (function opcode included), which is significant. In order to minimize the area requirement, code-size and energy dissipation, we want to reduce the instruction width to 32 bits. Another motivation for 32 bits is that this equals the width of all other applied memories ($Xmem$, $Ymem$ and $Cmem$)¹². The resulting instruction format is defined in fig. 3.21.

Reduction of the required control bits is found in the utilization of temporal locality, which can be explained as follows. Each PA output implies the storage of a variable. Such a variable can either be a 'state-variable' (i.e. for sequential PAs) or a 'volatile variable' (i.e. for combinatorial PAs). As we already mentioned, the compiler statically allocates these variables to local memory ($Xmem$, $Ymem$) locations. Now, if the compiler takes spatial WPP topology information into account during the temporal scheduling process (i.e. computing the PA fire sequence), then the

¹²This holds for a single-precision floating point native datatype

FIGURE 3.23 – ACU instruction: pointer initialization mode

distance (i.e. memory address) between adjacent operands and writeback is usually relatively small, compared to the entire memory range. Having this in mind, we introduce differential addressing for operand fetching and result writeback. For parameter p fetching we introduce a $Cmem$ pointer that increments each time that a PA with p dependency is executed. Note that this inevitably results in parameter replication, if equal parameters p are used within different PAs). This reduction of the required control bits leads to the definition of the ACU.

The ACU, which is controlled with the 28-bits *ACU instruction* (fig. 3.21), maintains 7 independent pointers. Each pointer needs to span a memory range with C locations, so the required pointer width is $\log_2(C)$ bits. These 7 pointers are the following:

- » (a) Two pointers (X-op and Y-op) are dedicated to operand fetching.
- » (b) Two pointers (X-wb and Y-wb) are dedicated to writeback to $Xmem$ and $Ymem$.
- » (c) Two pointers ('PntrGPNx' and 'PntrGPny') are used to index GPN destination addresses which are stored in $Xmem$ and $Ymem$. These pointers increment each time that they are referenced.
- » (d) The $Cmem$ pointer keeps track of the location of parameters p during thread execution. This pointer increments each time that a parameter reference is required by the currently executed PA.

The ACU computes next-state pointer values, using the current-state values and a difference. For the operand and writeback pointers (a) and (b), this *difference* is coded in an 8-bit signed number (i.e. 7-bit 'a/s' field plus 1-bit 'pd' field). This implies that the distance between current and next-state for these pointers are within a range of $[-127 : +127]$ upon execution of every single instruction.

The ACU supports three modes of operation: *default*, *initialization* and *alt* (alternative).

Default ACU mode.

The majority of the PAs receive their operands from local memory $Xmem$ and

27	26	25	24	22	21	20	19	16	15	8	7	0
1	1	1	opRC	unused	wbRC					acuX/acuWa		acuY/acuWb

opRC	X1-source	X2-source
000	X-mem	ibFIFO
001	Y-mem	ibFIFO
010	ibFIFO	X-mem
011	ibFIFO	Y-mem
100	X-mem	Y-mem
101	Y-mem	X-mem
110	-	-
111	-	-

WbRC[3:0]	WB-destination
0000	Xmem (using acuWa)
0001	Ymem (using acuWa)
0010	XYmem (using acuWa)
0011	GPN (address from Xmem)
0100	Xmem (using acuWb)
0101	Ymem (using acuWb)
0110	XYmem (using acuWb)
0111	GPN (address from Ymem)
1--	obFIFO

FIGURE 3.24 – ACU instruction: alternative operand/writeback mode

Ymem, and write-back the computed result also to local memory. This mode of operation is called *default*. The ACU instruction for *default* mode is described in fig. 3.22. The *pd* (i.e. pointer distance), and *a/s* (i.e. add/subtract) fields in the instruction describe the next-state for operand and write-back pointers. The assignment of the memory resource *Xmem* and *Ymem* to operands $x_1[n]$ and $x_2[n]$ is controlled by the *oa* bit. The *dl* field indicates whether the PA has a configured delay-line with length $\Lambda > 2$. If that is the case, then the computed write-back value needs to be pushed in obFIFO, and the delay-line output needs to be popped from ibFIFO and written to local memory *Xmem* and/or *Ymem*. The *dest* field selects the write-back destination memory. One of the other ACU modes (i.e. either *initialization* or *alternative*) can be selected through setting the *dest* field to bit-pattern "11".

Initialization ACU mode.

Each pointer (except the *Cmem* pointer, which is reset after a fire-event) needs to be initialized at the start of a thread. Initialization for each pointer is done in *initialization* mode. The ACU instruction for the *initialization* mode is described in fig. 3.23. The selected pointer (i.e. *sel* field) is initialized with the the *imm* bit-pattern. The *functional opcode* should be set to NOP while initializing pointers, in order to avoid side-effects.

Alternative ACU mode.

If a PA is associated to a WPG edge through a WaveCore process port, or the PA drives remote-state, then the ACU should operate in *alternative* mode. The different permutations of operand sources and write-back destinations are coded in the *opRC* and *WbRC* bit-patterns. The ACU instruction for the *alternative* mode is described in fig. 3.24. PAs with delay-line are not supported in *alt* mode.

GIU stalls

Stalling means that the entire GIU pipeline state is frozen until the stall conditions are resolved. Basically there are three conditions that can lead to stalling the pipeline. The first stall-condition is the unavailability of an ibFIFO operand in the OP-stage of the pipeline. The underlying reason for this unavailability is that the LSU has not been able to pre-fetch the requested data from externally connected memory, and to push it into the ibFIFO in time. The second stall-condition is caused by the inability to push a computed result into the obFIFO. The underlying reason for this is that the LSU has not been able to move delay-line input data or outbound token data to external memory, resulting in an obFIFO that gets full. The third stall-condition is caused by the inability to push a GPN packet into the transaction buffer. The underlying reason for this is that the GPN ring network is fully loaded due to heavy GPN traffic. Obviously, stall conditions are unwanted and are a threat to the real-time execution (e.g. elapsed time between two adjacent fire events) and hence a threat to fault-free execution of the physical model within the WPG. However, the timing behaviour of the external memory heavily depends on many factors, e.g. how the WaveCore processor is embedded in the SoC, and the performance of the external memory. Hence this timing behaviour can be difficult to predict. The timing behaviour of packets through the GPN is also hard to predict precisely, as we have seen in a previous subsection. However, it can be predicted quite accurately whether the produced GPN traffic will cause congestion because of the static temporal schedule of mapped physical models.

In chapter 4 we analyzed, and mapped a number of ReMap application to an FPGA based WaveCore system. From the analysis we observed that at least for the given applications the stall behaviour has not yielded problems with real-time predictability of execution, but no guarantees can be given.

3.3.5 LOAD/STORE UNIT

The LSU serves as a gateway between the GIU and the external memory. The main objective is to hide the memory access latencies for the GIU, such that the GIU can execute at full speed. The LSU executes a load-thread and store-thread to support the GIU thread, as is displayed in fig. 3.25. All threads within the PU are synchronized to the fire-event that is sent out by the central scheduler. Immediately after receiving this fire event, the load-thread within the LSU starts pre-fetching required inbound token data and delay-line output data from external memory and pushes this data into the ibFIFO. The store-thread which also runs on the LSU is triggered upon the availability of obFIFO data (i.e. delay-line input and outbound token data). The LSU switches between load and store threads, depending on the status of ibFIFO and obFIFO:

- » The load-thread is active as long as the end-of-inbound-thread (EOT) state is not true, and there is sufficient ibFIFO space available.

FIGURE 3.25 – Interaction between GIU thread and LSU load/store threads

- » The store-thread is active when the obFIFO is not empty, and the end-of-outbound-thread (EOT) state is not true.
- » If both the load and store threads are active, then a round-robin arbitration mechanism switches between these threads.
- » There is no dependency between load and store threads, apart from the fact that they use the same external memory.

Given the fact that the GIU-thread sequentially executes each PA in the WPP according to a compile-time derived static temporal schedule, the ordering of the associated inbound/outbound data (i.e. delay-line and token data) must comply to this schedule as well. This implies that the external memory read and write transactions must also comply to a static schedule. These external memory transactions are initiated by the LSU.

LSU architecture

The heart of the LSU consists of a DMA engine. This engine moves data between external memory, through the 'ExtMem' interface, and the FIFOs. A block diagram of the LSU is depicted in fig. 3.26. A data-movement transaction (i.e. movement of a data-block between FIFOs and external memory) is controlled by means of a DMA-descriptor. Such a DMA-descriptor contains all the information that is necessary to move the data-block (e.g. direction, location in external memory, amount of data). The descriptor information is encoded using six registers (r_0 to r_5) within the DMA engine. Both the load-thread and store-thread which run on the LSU consist of a chain of DMA-descriptors, which is sequentially executed each time that a fire-event is received. The descriptors themselves are stored in *Cmem*, while the process of handling these descriptors is controlled by the thread control unit (TCU). The TCU is a simple processor pipeline with an instruction set that is only capable of moving data between *Cmem* and the register set within the DMA-engine. For each DMA-descriptor, the TCU moves the required descriptor data from *Cmem* to the register set and subsequently triggers the data-movement. The TCU maintains separate PCs for both the load and store threads. The 'Thread Select' unit switches

FIGURE 3.26 – Load/Store Unit

between load and store thread, according to the mechanism which is explained further in this chapter.

LSU programmers model

The TCU instruction format is defined in fig. 3.27. The instruction consists of four fields:

1. Three bits 'opcode', which is further defined in table 3.4.
2. *Cmem* address, or immediate (i.e. constant value), depending on the 'opcode'
3. Thread-switch enable 'arb'. If this bit is set then the TCU selects the next active thread. The objective is to only arbitrate/switch threads after loading/executing a DMA-descriptor. The TCU will keep on executing instructions from the selected thread while the 'arb' bit is not set. This implies that DMA-descriptors are executed atomically by the TCU (i.e. execution of a descriptor cannot be interrupted by the other thread).
4. Register address.

The definition of token and delay-line buffers in external memory, and associated descriptor definition is described in fig. 3.28.

Opcode	function
011	Move (imm $\rightarrow$ register)
100	Move ($Cmem \rightarrow$ register)
101	Move (register $\rightarrow Cmem$)
110	End-of-load-thread
111	End-of-store-thread
others	no-operation

The definition and allocation of edge-buffering and associated edge DMA descriptors is depicted in fig. 3.28. The number of tokens within the edge-buffer is defined by the compiler and depends on the WPG-topology, like explained in chapter 3.2. The 'TokenBase' pointer addresses the start-of-token location of the selected token within the edge-buffer, and the 'TokenLength' variable defines the size of the token in number of PTs. Individual PT-blocks can be randomly accessed by a single edge DMA-descriptor, where the size and location of these blocks (i.e. a number of consecutive PTs) are defined with variables 'TokenOffset' (i.e. the offset w.r.t.

FIGURE 3.28 – Relation between DMA descriptors, registers and memory-layout

```

1 void ReadDL() {
2   uint sclTau;
3   uint lambda;
4   uint rdAddr;
5   // Scale tau:
6   sclTau=(tau*dLen) >> 16;
7   // Compute lambda:
8   if(Offset < sclTau)
9     lambda=Offset-sclTau+dLen;
10  else
11    lambda=Offset-sclTau;
12  // Move data:
13  rdAddr=baseAddr+lambda;
14  PushIbFIFO(extMem[rdAddr]);
15 }
16
17 uint WriteDL() {
18   uint wrAddr;
19   // Move data:
20   wrAddr=Offset+baseAddr;
21   extMem[wrAddr]=PopObFIFO();
22   // Update offset
23   Offset++;
24   if(Offset==dLen)
25     // Return wrapped pointer
26     r5=dLen+0;
27   else
28     // Return incremented pointer
29     r5=dLen+(Offset << 16);
30   // Return updated write-pointer:
31   return(r5);
32 }

```

LISTING 3.4 – Execution of delay-line DMA descriptors

```

1 void UpdateTokenBase() {
2     uint nxtTBase;
3     // Compute return value
4     nxtTBase=tokenBase+tokenLength
5     if(nxtTBase==hiAddr)
6         // Return base address
7         r5=edgeBase
8     else
9         // Return pointer to next token
10        r5=tokenBase+tokenLength;
11 }
12
13 uint ReadTokenPart() {
14     uint rdPntr;
15     // Compute start address
16     rdPntr=tokenBase+tokenOffset;
17     // Move token-part
18     for(i=0;i<trnLen;i++)
19         PushTbFIFO(extMem[rdPntr++]);
20     // Compute return value r5:
21     UpdateTokenBase();
22     return(r5);
23 }
24
25 uint WriteTokenPart() {
26     uint wrPntr;
27     // Compute start address
28     wrPntr=tokenBase+tokenOffset;
29     // Move token-part
30     for(i=0;i<trnLen;i++)
31         extMem[wrPntr++]=PopObFIFO();
32     // Compute return value r5:
33     UpdateTokenBase();
34     return(r5);
35 }

```

LISTING 3.5 – Execution of token-part DMA descriptors

'TokenBase') and 'TrnLen' (i.e. the PT-block size). The detailed description of edge DMA descriptor execution is defined in listing 3.5.

3.4 WAVESLANG COMPILER

The WaveSlang compiler analyzes a WPG and subsequently maps the physical model (i.e. WaveCore processes) onto a target PU-cluster. This mapping process is bounded by the physical constraints that are associated to the PU cluster, like clock-frequency and PU memory capacity on the one hand and complexity of the physical model (i.e. the number of PAs, and the real-time requirements) on the other hand.

The main challenge for the compiler is to find an optimal mapping that efficiently exploits both the spatial dimension (i.e. utilize a minimal number of PUs) and the temporal dimension (i.e. maximize the individual PU load) of the given WaveCore processor architecture. This mapping is constrained by the real-time requirements of the algorithm, and the constraints of the WaveCore processor (i.e. number of PUs, PU capacity C). Only the physical model within the WPG (i.e. WaveCore processes) is mapped on the PU-cluster. The rest of the WPG is not translated by the compiler.

The mapping process yields a transformed and logically equivalent representation of the WPG which is written into a WaveCore object file (WOF). The difference between the WOF and WaveSlang representation of the WPG is mapping information for the physical model, which is described in the WOF:

- » The data-structures for WPG tokens and delay-lines are allocated to external

memory.

- » The data-structures within each WPP (i.e. parameters p for compiled PAs, and constants) are allocated to $Cmem$, $Xmem$, and $Ymem$ of the associated PU to which the WPP is allocated.
- » For each WPP the GIU and LSU instructions are described as bit-patterns, and allocated to the associated $Pmem$ of the PU to which the WPP is allocated.
- » For each WaveCore process, one or more PUs are allocated and the base addresses for the GIU and LSU threads are defined.

A WOF representation of a WPG is used for simulation, but can be mapped on a HW target platform (e.g. FPGA-board) as well as we will show.

The main challenge for the compiler is to use a minimal number of PUs in order to map the physical model within the WPG. Therefore, the objective for the compiler is to find an efficient space/time balanced schedule for each WaveCore process, like indicated in fig. 3.14 in chapter 3.2. There are five subsequent phases within the mapping process, which are depicted in fig.3.29:

1. Consistency checking.
The first phase is a consistency check of the WPG.
2. Spatial mapping.
Within this phase the physical model within the WPG is analyzed, partitioned and allocated to a minimum set of PUs. Another important objective of the spatial mapping process is to minimize the required level2 GPN traffic through minimization of remote-state, and to minimize level3 traffic through minimization of the number of process edges.
3. Temporal mapping.
For each occupied PU, the allocated WPG part (i.e. one WPP and associated interfaces) is mapped. This implies that each allocated WPP is mapped on the GIU processor pipeline. Temporal scheduling is the main task within this process.
4. Shared memory stream mapping.
During this phase the required delay-line and edge buffers are synthesized and allocated to shared memory. Furthermore, per PU the DMA descriptors are derived from the GIU schedules and subsequently mapped on the associated LSU.
5. Code generation.
The final mapping step is instruction synthesis and generation of $Pmem$, $Cmem$, $Xmem$, and $Ymem$ memory images for each involved PU. Moreover, non-WaveCore processes and the required structure for the central scheduler are generated in the WOF.

These four mapping phases are further explained in the following of this chapter.

FIGURE 3.29 – Compiler flow

3.4.1 WPG CONSISTENCY CHECKING

The first phase within the spatial mapping process is to parse the input WPG source files and to perform a consistency check on the following aspects:

- » Cycle detection within combinatorial PA graphs (i.e. DAG consistency checking). DAGs should be free of cycles.
- » Consistency of edges, process ports and port-maps: token-size, port-direction and signal connections.
- » Multiple drivers on PA edges. Each PA edge should have only one driver.
- » Unconnected PA incoming edges. All incoming edges of all PAs should be connected.

3.4.2 SPATIAL MAPPING

In principle, the physical model within the WPG which is described in WaveSlang can be transparently mapped on the WaveCore processor, provided that the described physical model meets the constraints of the WaveCore PU cluster. This mapping is as follows:

- » Each WPP is allocated to a single WaveCore PU.
- » The remote-state is mapped on the GPN network.
- » All process token buffers and delay-line buffers are allocated to external memory.

This implies that the input-WPG contains an initial spatial mapping of its physical model, which in fact may be optimal. If the WaveSlang programmer is aware of the target PU cluster (i.e. number of PUs, memory sizes, clock-frequency etc.), and has full control over the coding of a physical model, then an optimal spatial mapping can be directly coded in WaveSlang. As a matter of fact, all application examples in chapter 4 are coded in WaveSlang in this way. However, if a WPG in WaveSlang format is generated by a functional language compiler which is not aware of the PU cluster parameters (e.g. like in the experiment in chapter 5), then it is very useful (if not inevitable) to do the spatial mapping in an automated manner. The *WPG optimization* phase within the spatial mapping phase transforms the input-WPG to a functionally equivalent WPG, which is optimized to the structure of the WaveCore PU cluster. The transformed WPG utilizes a minimum set of PUs.

Therefore, the *WPG optimization* phase within the spatial mapping process is an optional compile step.¹³

WPG optimization

Like mentioned in the introduction of spatial mapping, the main objectives of WPG optimization are:

¹³The *WPG optimization* phase is currently not implemented in the WaveSlang compiler.

- » To enable the allocation of the WaveCore processes within the WPG to a minimal set of PUs
- » To optimize the locality of reference: maximize level₁ (i.e. intra-GIU) traffic and minimize level₂ (i.e. traffic over GPN network) and level₃ (i.e. DMA) traffic.

We propose an *WPG optimization* algorithm which consists of two steps:

The **first step** in the *WPG optimization* phase is to merge *isochronous* WaveCore processes. Two isochronous processes share an identical fire-link, and hence all sequential elements within these processes are synchronized to the same fire event. Merging of two isochronous processes is illustrated in fig. 3.30, and implies the following:

- » Edges between the merged processes are absorbed: i.e. Edge₁ and Edge₂ are replaced with a register which is synchronous to the fire-signal. Hence this implies that level₃ traffic is transformed to level₁ traffic.
- » The PCOREs of both processes are merged to one PCORE.
- » All WPPs in the merged processes are merged to one *parent WPP*. This implies that level₂ traffic (i.e. remote-state $S_r[n]$) is transformed to level₁.

After the merge-phase, each WaveCore process within the WPG has a different fire-link. Furthermore, each WaveCore process contains a PCORE which consists of only one *parent WPP*, and hence is free of remote-state. Note that the size of the parent-WPP, after the merge-phase, might exceed the capacity of a PU. Hence, the subsequent step is partitioning.

The **second step** in the *WPG optimization* phase partitions the *parent WPP* into one or more *child WPPs*, depending on the physical constraints of the PU cluster. In chapter 3.2.2 we explained that a PCORE of a WaveCore process can be modeled as a Mealy machine, and that this Mealy machine can be partitioned into one or more independent DAGs and registers. Next to that, we explained how a PCORE can be partitioned into one or more WPPs in chapter 3.2.3, and how this leads to the definition of *remote state* $S_r[n]$ and associated remote-state update through the GPN network.

In chapter 3.2.3 we illustrated in fig. 3.14 how a space/time mapped WaveCore process is distributed over the required PUs. Each PU executes one child-WPP, where each child-WPP consists of one or more DAGs and associated state registers.

Now, the most optimal spatial schedule implies that:

- » The size of each child-WPP (i.e. in terms of number of PAs) is optimized to the capacity C_m of a PU. The capacity C_m refers to the maximum number of PAs that can be executed on a PU within a thread. This capacity C_m is

FIGURE 3.30 – Merging of isochronous processes

smaller than the $Pmem$ size, because instructions for the LSU (which are also stored in $Pmem$) and temporal schedule inefficiencies (described further in this chapter) have to be taken into account.

- » The amount of remote-state $S_r[n]$ needs to be minimized. Remote-state implies level2 data traffic through the GPN network of the WaveCore processor. This traffic is associated with a certain unpredictability, because the ring-topology of the network has a limited throughput capacity.

Thus, an optimal spatial schedule can be derived by means of a bin-packing algorithm. Each bin in this algorithm represents a child-WPP and the objects which have to be packed are indivisible DAGs. Bin-packing is known to be a combinatorial NP-hard problem. The algorithmic solution that we propose is based on the following two heuristics:

- (1) Maximum utilization of capacity C_m .
- (2) Minimization of remote-state $S_r[n]$.

The goal is to utilize a minimum set of PUs with a minimum amount of remote-state, where each PU can be filled with a maximum of C_m PAs. A first estimation of the number of required PUs is found by dividing the sum of all PAs in the parent-WPP by the capacity C_m .

Now, the bin-packing algorithm needs to find an optimal solution, and is steered by heuristics. A possible method is based on simulated annealing [41]. We did not implement the actual algorithm. Hence a further definition is out of scope for the thesis, but an important topic for future work¹⁴.

PU allocation

Allocation of child-WPPs to PU can in principle be performed in an arbitrary way. However, these child-WPPs are connected by remote-state dependencies. Remote-state needs to be exchanged through the GPN network. The distance between PUs in the GPN ring determines the load on this ring. Therefore, it is beneficial to minimize the distance between producers and consumers of remote-state. Minimization of this distance can be used as a metric to allocated child-WPPs to PUs. We, however, did not take this into account.

During the PU allocation phase, it cannot be guaranteed that the spatial mapping will meet the timing requirements of the application. This becomes clear after the temporal scheduling phase. If the temporal schedule cannot be met, then the spatial mapping process has to be re-iterated with where less PU capacity C is taken into account.

¹⁴For all mapped applications which are described in this thesis, we developed optimized WPPs manually

FIGURE 3.31 – Operand resource conflict resolution through replication of variables

3.4.3 TEMPORAL MAPPING

The purpose of *temporal mapping* is to map each child-WPP onto the GIU pipeline to which it is allocated. The GIU executes a compiled WPP thread in a temporal fashion (i.e. one PA after another within single instructions): hence the name *temporal mapping*. The temporal mapping process consists of two subsequent phases:

1. GIU data memory allocation.
Each PA-output needs to be allocated to a data-memory location (i.e. a location in either *Xmem*, *Ymem* or both as we will explain later).
2. Temporal scheduling (GIU-pipeline scheduling).
The execution-order of each PA within the child-WPP needs to be derived.

The challenge is to schedule the GIU pipeline in such a way that a minimum number of stall/idle cycles (i.e. idle cycles due to data dependency conflicts) need to be introduced. These dependencies are taken into account as we will explain later in this chapter.

PU data memory allocation

Each PA-output variable which is consumed by one or more other PAs within the same child-WPP needs to be stored in local data memory *Xmem* and/or *Ymem*. The assignment of PA-output variables to local GIU memory is driven by two main requirements:

- » Requirement R_1 : memory resource allocation
The execution of a PA within a single instruction requires that at most two operand variables need to be read from local *Xmem* and *Ymem*. This implies that a PA-output variable is required to be assigned to either *Xmem*, *Ymem* or both (explained later) memory resource(s).

- » Requirement R_2 : memory location assignment
Once allocated to a memory resource, the variable needs to be assigned to a specific location (address) within the allocated memory.

Algorithm 1: Level1 memory-resource allocation

```

1 Global: Allocation of associated PA outbound variable to either Xmem,
   Ymem or replicated XYmem

2 Procedure: MemResourceAlloc(ChildWPP)
3 Input: ChildWPP: for all included PAs, allocate to either X/Y or XY
4 begin
5   for allPAs do
6     SelPA=SelectPA()
7     AllocateOperandSourcePAs(SelPA)

8 Procedure: AllocateOperandSourcePAs(SelPA)
9 Input: SelPA: allocate PAs which source SelPA, solve possible conflict, keep
   balanced X/Y
10 begin
11   if SelPA  $\rightarrow x_1 \neq \emptyset$  then
12     Allocate(SelPA  $\rightarrow x_1$ )
13   if SelPA  $\rightarrow x_2 \neq \emptyset$  then
14     Allocate(SelPA  $\rightarrow x_2$ )
15   ConflictResolve(SelPA)

16 Procedure: Allocate(PA)
17 begin
18   Input: PA: allocate PA <- output variable to either X/Y or XY memory
19   if paIsNotAlreadyAllocated then
20     MemSelect=SelectUponBalance(Xload,Yload)
21     PA  $\leftarrow$  MemSelect

```

 R_1 : memory resource allocation

Memory resource allocation (R_1) deals with the problem of assigning each PA-output variable in a conflict-free way to memory, and to evenly divide the required variables over the local memories *Xmem* and *Ymem*. The required operands for executing a PA with two incoming edges $x_1[n]$ and $x_2[n]$ need to be simultaneously read from local data memory resources *Xmem* and *Ymem*. This implies that the required allocation of the PA-output to local memories *Xmem* and *Ymem* needs to be done in such a way that memory resource conflicts, due to simultaneous operand fetching, are eliminated. The problem of memory resource conflicts is illustrated in fig. 3.31. PA(4), PA(5) and PA(6) require two operands each. This means that for each of these PAs, one operand needs to be stored in *Xmem* and the

other in *Ymem*. Now, in this particular example the assignment of variables cannot be done in a conflict-free manner. The operand, which is generated by PA(3), is required by PA(5) and PA(6). However, the other operand for both mentioned PAs are stored in opposite memory resources, in order to yield a conflict-free solution to the operand resource assignment for PA(4). Hence, the operand which is sourced by PA(3) needs to be allocated to *Xmem* while the same operand needs to be allocated to *Ymem*, in order to yield a conflict-free solution for PA(5) and PA(6). We solve this problem by a *replicated Xmem/Ymem write-back* during the execution of PA(3). The output of PA(3) is simultaneously written back to both *Xmem* and *Ymem* at the same memory location during the execution of this PA(3).

The maximum number of PA-output variables which need to be allocated is equal to the capacity C of the PU (see also subsection 3.3.4, instruction set rationale). In the worst case, this means that C variables need to be allocated to data memory, where the total variable storage capacity equals C variables, while the individual data memory capacity equals $\frac{1}{2}(C)$. This implies that the required variables need to be allocated to *Xmem* and *Ymem* in a balanced way (even distribution).

The level1 resource allocation algorithm is summarized in alg. 1. The main procedure is called *MemResourceAlloc*, and it allocates memory resources for WPP, called *ChildWPP*. The main procedure iteratively selects an arbitrary PA, called *SelPA*, and subsequently assigns its operand-source PAs (i.e. the PAs which produce the inbound tokens for the selected PA) to *Xmem* and *Ymem* in the called procedure *AllocateOperandSourcePAs*. The procedure *AllocateOperandSourcePAs* checks if the PAs which produce inbound tokens for selected PA *SelPA* are already assigned to a memory resource. If not, then the assignment is done in procedure *Allocate*. After assignment a conflict-check is performed by procedure *ConflictResolve*. If a conflict is detected then this conflict is resolved. A first attempt to resolve the induced conflict is by swapping the memory resources for the PAs which produce the inbound tokens for selected PA *SelPA*. If the conflict persists, then one of the operand producing PA is assigned to both *Xmem* and *Ymem*.

R_2 : memory location assignment

Memory location assignment (R_2) deals with the problem of assigning variables to memory locations within the assigned memory instance(s). In principle this assignment can be done without restrictions. However, the accessibility of variables is controlled by the ACU within the GIU. In order to reduce the instruction width we introduced the concept of differential operand fetch and ALU writeback (see fig. 3.21). This implies that there exists a relationship between the temporal schedule (to be outlined in the next subsection) and the location of the related variables within data memory: if the distance in location exceeds the maximum distance, then a pointer update instruction needs to be inferred in the temporal schedule. In effect, an inferred pointer update instruction implies that the ALU cannot be used for computation. In order to reduce the number of inferred pointer updates we used the property of spatial locality within a PA-graph (child-WPP) to link memory

location assignment with the probable temporal schedule (note that the temporal schedule is not known within the phase of memory location assignment). The temporal scheduler, as we will outline in the next subsection, preferably schedules PAs which are spatially close to just-earlier scheduled PAs. This is a logical result from the fact that a PA within a DAG can only be scheduled when all its incoming edges carry valid (i.e. just-scheduled) PTs.

This observation has led to a memory location assignment strategy that assigns PAs that share a high spatial locality to subsequent (locally close) memory locations. Compile experiments for large, strongly connected graphs (like FDTD geometry schemes, which will be explained as application case in section 4.4), have shown that this strategy is effective.

Multi-threaded temporal scheduling

Algorithm 2: Temporal scheduler: main function

```

1 Global: Schedule all PAs within all child-WPPs within input WaveCore
   process WP

2 Procedure: DeriveTemporalScheduler(PCORE)
3 Input: Unscheduled WaveCore process PCORE
4 begin
5    $T_s = 1$ 
6   while  $\neg \text{AllScheduled}(\text{PCORE})$  do
7     for all childWPPs do
8       if  $\neg \text{IsScheduled}(\text{childWPP})$  then
9         CreateCandidateSet(childWPP)
10        if candidateSet  $\neq \{\emptyset\}$  then
11          ScheduleHighestPrioPA(childWPP)
12        else
13          if  $\neg \text{AllScheduled}(\text{childWPP})$  then
14            InferNOP()
15          else
16            InferEOG()
17            childWPP  $\rightarrow \text{IsScheduled} = \text{true}$ 
18         $T_s = T_s + 1$ 

```

The temporal scheduler derives the execution order of the PAs within a child-WPP. The actual scheduling algorithm basically simulates the data-flow process, which takes place within a PCORE. While simulating, the execution order of the PAs is derived. The PCORE is partitioned into a set of child-WPPs, and each child-WPP

is mapped on a PU. Given the fact that each PU executes a mapped child-WPP concurrently, this implies that all the child-WPPs need to be simulated concurrently in order to derive a valid temporal schedule for the entire PCORE.

The scheduler algorithm is based on a simulation of the propagation of tokens (i.e. PTs) through the WPPs. This propagation is dictated by the fire-rules of the PAs. The simulation start is equivalent to the moment when a fire-event takes place. The main function of the scheduler algorithm, called *DeriveTemporalSchedule*, is given in alg. 2, and explained as follows:

DeriveTemporalSchedule

The main function *DeriveTemporalSchedule* simulates the data-flow within the SDF-like graph, being the PCORE. The simulation is an iterative process. Within each iteration, one PA per child-WPP is selected and scheduled (if possible). Per iteration cycle, the following steps are taken for all child-WPPs within the PCORE:

- » Create list of schedulable PAs
Based on the current availability of tokens in the currently selected child-WPP, a list of schedulable PAs is created. This list is created by the function called *CreateCandidateSet*. For each PA in this list, a priority label is added. The concept of priority is explained later.
- » Allocate the schedule timeslot T_s
If the candidate list is not empty, then a PA with the highest priority is selected and scheduled within the function called *ScheduleHighestPrioPA*. From simulation point of view this means that the PA is pushed into an execution pipeline. This execution pipeline mimics the length of the GIU pipeline. While the PA is pending in this pipeline, its output token is not yet available for its consumer PA(s).
If the list of schedulable PAs is empty then, either a NOP is inferred, or the scheduler process is stopped. If not all PAs are scheduled, while the list of schedulable PAs is empty, then this implies that the progress is stalled due to a temporary cause (e.g. required operand is not available yet, because the producer-PA is pending in the GIU pipeline). In that case, a NOP is inferred in the schedule at the current timeslot T_s . If all PAs are already scheduled, then an end-of-graph (i.e. EOG) instruction is inferred, and the flag *childWPP.IsSchedule* is set to *true* (i.e. indicating that the child-WPP is scheduled).

CreateCandidateSet

The function *CreateCandidateSet* creates a list of *schedulable* PAs. A PA is schedulable when the firing-rule for that particular PA is met. For an arbitrary PA the *firing-rule* is defined as follows:

- » Firing-rule for a combinatorial PA
A combinatorial PA (i.e. a PA with delay-line length $\Lambda = 0$) is allowed to be

scheduled (i.e. executed) when all required inbound tokens are available.

» Firing-rule for a sequential PA

A sequential PA (i.e. a PA with delay-line length $\Lambda > 0$) is allowed to be scheduled (i.e. executed) when all required inbound tokens are available, *and* its outbound token is consumed by all consumer PAs which are connected to the outgoing edge of the sequential PA.

The tokens which are available at the start of the scheduling process are the following:

- » Tokens at outgoing edges of sequential PAs. These tokens are computed and stored during the previous fire-period (i.e. state).
- » Tokens at primary input signals $X[n]$, which originate from WaveCore process incoming edges.

After each iteration cycle, at most one new token per WPP becomes available because of previously fired PAs. Because of the fact that the GIU pipeline can only execute one PA per instruction, this results in the fact that only one new token per iteration cycle can become available. Hence, the production of a token is delayed by N iterations after firing the PA, where N equals the pipeline depth of the GIU.

ScheduleHighestPrioPA

The function *ScheduleHighestPrioPA* selects a PA from the list of schedulable PAs, and subsequently schedules this PA. The selection of the PA is based on a priority-label. Each PA within the list is labeled, according to the following priority ranking, where (1) is the highest priority:

1. *priority 1*: The PA consumes or produces remote-state $S_r[n]$.
The rationale for this priority level is to utilize the GPN network as efficient as possible. If the remote-state is consumed, through firing all consumer PAs of this remote-state, then it is allowed to schedule the producer-PA which is located in a different child-WPP. If such producer-PAs are allowed to be executed early in the thread, then the utilization of the GPN network can be spread over the duration of the thread. This leads to a lower probability of GPN network congestion.
2. *priority 2*: The PA consumes tokens from a combinatorial producer-PA.
The location of variables in $Xmem$ and $Ymem$ are derived by an algorithm which takes spatial locality of the PAs within the WPP into account. Therefore, the differential pointer updates, as explained in section 3.3.4, works most optimally (e.g. no or few pointer initialization instructions required) when the temporal scheduler is driven by the same locality principle.
3. *priority 3*: Any PA which does not meet the priority criteria which are described elsewhere in the list. Note that this class is not labeled with the lowest priority.

FIGURE 3.32 – Example PCORE, consisting of two WPPs

TABLE 3.5 – Temporal (pipeline) schedule of example PCORE

T	1	2	3	4	5	6	7	8	9	10
PU(A)										
FET	a1	a3	NOP	a4	a5	NOP	a2	a7	a6	EOG
DEC	-	a1	a3	NOP	a4	a5	NOP	a2	a7	a6
EXE	-	-	a1	a3	NOP	a4	a5	NOP	a2	a7
PU(B)										
FET	b1	b3	NOP	b4	b5	b6	b2	EOG	-	-
DEC	-	b1	b3	NOP	b4	b5	b6	b2	EOG	-
EXE	-	-	b1	b3	NOP	b4	b5	b6	b2	EOG

4. *priority 4*: The PA consumes tokens from ibFIFO.

The rationale for this lowest-level priority is to allow as much time as possible to the LSU to gather token or delay-line output data from external memory.

We apply the algorithm to the example PCORE in fig. 3.32. The example PCORE is partitioned into two child-WPPs: child-WPPa and child-WPPb. These WPPs are mapped on PUa and PUB. The associated schedule is described in table 3.5. The illustrative pipeline that we show in the example is three stages deep. Note that the actual GIU pipeline is 13 stages deep.

3.4.4 SHARED MEMORY TRANSACTION SYNTHESIS

The GIU is connected with the LSU by means of two FIFOs: ibFIFO and obFIFO. By definition (first-in, first-out) this implies that the temporal order in which PTs are written into ibFIFO or read from the obFIFO by the LSU is dictated by the temporal GIU schedule. This is illustrated in fig. 3.25. From this observation it becomes obvious that the DMA-descriptor chains for both the load and store LSU threads can be directly derived from the order in which the associated PAs are executed within the temporal GIU schedule. Hence, the synthesis of load and store LSU threads is a trivial process:

(1) Load-thread synthesis:

The temporal order in which each PA with delay-line (with $\Lambda > 1$) or inbound-edge association are executed, yields a DMA load descriptor chain. This descriptor chain describes how the required PTs must be (pre)fetches from external memory.

(2) Store-thread synthesis:

The temporal order in which each PA with delay-line (with $\Lambda > 1$) or outbound-edge association are executed, yields a DMA store descriptor chain. This descriptor chain describes how the produced PTs must be written to external memory.

Shared memory allocation

The allocation of delay-line buffers and edge-token buffers in shared memory can be done in an arbitrary way, if this shared memory is uniform (e.g. one DDR memory). If, however, the external memory is partitioned into for instance fast but relatively small on-chip memory and slower but big off-chip memory, then the allocation strategy needs to be adapted to the partitioning of the memory system.

DMA descriptor synthesis

DMA-descriptor synthesis is a process which logically follows from the temporal GIU schedule and shared memory allocation.

3.4.5 INSTRUCTION SYNTHESIS

After the mapping phase, the GIU and LSU instructions can be derived from the obtained schedules and DMA-descriptor chains. For the GIU-thread the instruction synthesis process implies that the instruction opcode and associated ACU bit-patterns are derived from the mapped thread. Similarly, the bit-patterns for the separate load and store threads for the LSU are derived from the DMA descriptor chains. This is a trivial process which does not need further explanation.

Code generation

The final step within the compilation process is the generation of the WOF file. For the WaveCore processes this means that the instruction and data memory images are generated for each PU. Furthermore, the rest of the WPG structure is also written into the WOF file. The generated WOF file describes a mapped WPG which is logically equivalent to the source WPG description which has been the input description for the compiler.

3.5 DEVELOPMENT TOOLS

The WOF which is generated by the WaveCore compiler can either be mapped on a virtual ReMap processor (i.e. for simulation) or on a HW platform. The

FIGURE 3.33 – WaveCore process graph simulator

development tools which enable this are a process graph simulator and a HW target interface.

3.5.1 PROCESS GRAPH SIMULATOR

The WPG simulator is an abstract model of the ReMap processor architecture as shown in fig. 3.33. Hence, the WOF which is generated by the WaveSlang compiler, can be executed by the simulator. The WaveCore processes are mapped on and executed by a WaveCore processor model, which is an important part of the simulator. Foreign processes are implemented as file-input/output (I/O) interfaces. Token streams through incoming edges of WaveCore processes are read from file (stimuli) during simulation, and token streams from outgoing edges of WaveCore processes are written to file (responses). Control processes within the WOF are not explicitly simulated. Control-tokens (i.e. tokens which are passed through edges from control processes to WaveCore processes) are initialized at the start of the simulation and not updated afterwards. A block diagram of the simulator is depicted in fig. 3.33.

The *WPG execution engine* includes a model of the central scheduler and hence simulates the timing behaviour of the WPG. This engine administrates time progression and generates fire events. Therefore, this engine embodies the overall execution control, in analogy with the defined WPG execution model. The foreign processes source or sink one token from/to file each time that they receive a fire event from the execution engine, and moves this token according to the process graph topology.

The *Processor model* embodies a detailed behavioural model of a scalable WaveCore PU cluster. The size of the model (i.e. the number of PUs and the PU capacity) is derived from the WOF and hence the configuration of the *processor model* is created during the WOF-parsing phase. The *processor model* contains all the architectural building blocks which are also defined in the WaveCore processor architecture (i.e. GIU building blocks, ibFIFO, obFIFO, memories, LSU and GPN). The *processor model* is abstract in the sense that it does not model pipeline behaviour. In response to fire events from the execution engine, the processor model acts as follows:

All PUs which are involved with the execution of the WaveCore process (i.e. identical fire-link) are executed in parallel. This is achieved by a virtual PU cluster clock signal. Each PU responds to this clock signal until GIU, LSU-load and LSU store threads are in EOT-state. This EOT state becomes active when the associated EOT instruction is executed. In response to each virtual clock-tick, each GIU executes one instruction (i.e. if the GIUs are not in EOT-state). Likewise, the TCU within each LSU executes one instruction per clock-tick, according to the active-state of the load and store threads.

3.5.2 HW TARGET INTERFACE

We explained that the compiled WOF is an equivalent representation of the source-WPG. The physical model within the WOF is mapped on the WaveCore processor, and utilizes one or more PUs. Furthermore, the real-time constraints (i.e. fire-rate, processor clock frequency) are defined in the WPG and translated to the WOF representation by the WaveCore compiler. This process graph representation is HW platform independent. The mapping of a WOF on a target HW platform requires specific information, which is specified within the board support package (*BSP*) in fig. 3.34, such as:

1. WaveCore cluster configuration (e.g. number of PUs).
2. PU configuration (e.g. capacity C).
3. Shared memory configuration.
4. Specification of HW entities on which foreign processes can be mapped.

The *HW target mapper* analyzes the match between WOF and BSP, maps the compiled process graph onto the target HW platform, and subsequently generates a binary image. The *board interface* loads this binary image to the target board and as a result initializes the SoC in which the PU cluster is embedded. Furthermore, the *board interface* allows the user to interact with the physical model, which runs on the WaveCore processor, at run-time (i.e. interaction model).

3.5.3 DEVELOPMENT BOARD

We used a commercially available FPGA board (Atlys by Digilent [17]) as WaveCore application development platform. The Atlys FPGA development board is based on

FIGURE 3.34 – Hardware Target mapping and board interface

the Xilinx Spartan6 LX45 FPGA. Apart from the FPGA the board contains analog audio I/O, an AC97 compliant audio codec (i.e. audio ADC and DAC), a USB interface and 128MByte DDR2 memory, and a flash memory for FPGA configuration. The flash memory contains the FPGA configuration code and is loaded to the FPGA immediately after power-up. We designed a WaveCore system which complies to the ReMap processor architecture which we introduced in section 3.1:

1. The GPP is embedded in an externally connected (through USB) personal computer.
2. The rest of the architecture is on the Atlys board. The block diagram of this part is depicted in fig. 3.35.

The WaveCore PU cluster is mapped on the FPGA. The cluster contains 6 PUs and runs at a clock frequency of 86 MHz. The AC97 audio codec streams autonomously to/from the DDR2 memory. This is enabled by the *ADC/DAC stream actors* within the *Digital Audio Interface* unit. These stream actors are fired by the central scheduler and move tokens (i.e. a token in this context implies two PTs, each one representing an audio channel) from audio codec to DDR2 memory (ADC) or vice versa (DAC). These stream actors are HW implementations of foreign processes.

The GPP has full control access capability through the USB interface: initialization of the level1 PU memories, the central scheduler and the AC97 codec. Next to initialization the GPP has full access to the shared memory (DDR) which enables the mapping of control processes on the GPP. These control processes are enabled to move control tokens from/to shared memory.

FIGURE 3.35 – WaveCore development platform on FPGA board

3.6 DISCUSSION AND CONCLUSIONS

Based on the problems with state-of-the-art ReMap applications, which we identified in chapter 1, we proposed a heterogeneous processor architecture. This architecture consists of a GPP (i.e. for execution of interaction models) and the WaveCore processor (i.e. for execution of real-time physical models).

We summarize how these problems are addressed, from WaveCore and WaveSlang perspective:

(1) Processing latencies

Latency is a critical parameter in DAFx applications, and should be kept small. Many programmable processors, which are used in the state-of-the-art of ReMap applications, apply buffering as a means to increase time-predictability (see chapter 2) at the cost of an increased latency. We addressed this problem as follows, from WaveSlang and WaveCore perspective:

- » WaveSlang

The processing latencies within the physical model can be precisely derived from a WaveSlang description of the WPG. Furthermore, the latency is kept to a bare minimum through minimization of token buffering.

- » WaveCore

Time-predictability of execution is the prerequisite to allow for a minimum amount of token buffering.¹⁵

(2) *Time predictability of execution*

ReMap applications are very sensitive to imperfections in audio production. Even one missing sample in a generated audio stream is audible and perceived as disturbing. Imperfections in audio production are the result of missing real-time deadlines in execution. Missing deadlines, in turn, are the result of a lack of time-predictability of execution. We addressed this problem as follows, from WaveSlang and WaveCore perspective:

- » WaveSlang

The centralized scheduler, combined with the SDF classification of the physical model within the WPG, yields a very predictable execution from programming model perspective.

- » WaveCore

From WaveCore processor perspective, the problem of time-predictability is addressed through a separation of global and local data traffic. This is done by means of a separate GIU/LSU design which are coupled by ibFIFO and obFIFO. This separation yields a high level of time-predictability of execution, but as there are shared memory resources the timing cannot be fully guaranteed.

(3) *Scalability*

Computational complex physical models within ReMap applications (e.g. detailed models of musical instruments) need to be partitioned, because they usually do not fit on a single processor core. As a result, the processor on which the model is mapped needs to be scalable (e.g. a multi-core architecture). For complex physical models we did not find a common state-of-the-art solution to this problem (see chapter 2). We addressed this problem as follows, from WaveSlang and WaveCore perspective:

- » WaveSlang

We described the physical model within a WPG as one or more Mealy-machines, which can be partitioned into DAGs and registers. This partitioning is the basis for scalability, in the sense that DAGs and registers can be packed into a set of child-WPPs.

¹⁵Note that the induced latency is also dependent on the structure of the physical model. In chapter 4 we will show what the overall latency is for a number of application cases.

- » WaveCore

The WaveCore is a scalable processor, consisting of a network of PUs. The number of PUs can be configured. Each PU executes one child-WPP. This scalability is reflected in WaveSlang.

(4) Programmability

We observed a discrepancy between the abstraction level of ReMap algorithms and the abstraction levels of commonly used programming languages within the ReMap domain (i.e. imperative programming languages like C, which are commonly used for programmable architectures like GPP and DSP). Next to that we observed that domain-specific functional languages are usually translated to imperative languages in order to ultimately compile these algorithms to state-of-the-art programmable cores. The result is a lack of transparency with respect to real-time constraints from a programmers perspective. We address this problem through the definition of WaveSlang, which matches closely to the data-flow nature of ReMap algorithms:

- » WaveSlang

WaveSlang addresses the heterogeneous nature of ReMap algorithms (i.e. interaction models and physical models). A WPG enables the programmer to describe interaction models, physical models, and the interaction between these.

- » WaveCore

The programming model of the WaveCore processor matches transparently with the semantics in the WaveSlang language.

At this stage it is difficult to claim that the proposed ReMap architecture, with WaveCore as a processor to execute physical models, is a competitive solution with respect to state-of-the-art solutions. Therefore, we need to obtain evidence on the following aspects:

- » How efficient can physical models within ReMap application cases be mapped on the WaveCore architecture? We will elaborate on this in chapter 4.
- » How efficient is the WaveCore processor architecture, compared to other cores which are applied in the ReMap domain? We will elaborate on this in chapter 6.
- » Is it possible to compile domain-specific functional languages to WaveSlang, while keeping a high degree of control over real-time constraints? We will elaborate on this in chapter 5.

ABSTRACT – We have analyzed the mapping of physical modeling applications on the WaveCore processor. Within the application domain we identified three sub-domains. The first sub-domain is virtual analog modeling, which aims to model analog circuit behaviour. We designed WaveCore models for the "phaser", "flanger" and "AutoWah" and showed how these models are derived from the associated analog circuit abstractions. The second sub-domain is digital waveguide which finds its application in reverberation modeling. A key element within digital waveguide is the delay-line which is an elementary feature in the processor concept, like we introduced in chapter 3. Within the digital waveguide sub-domain, we focussed on a so-called physically informed reverberation algorithm, called "FreeVerb" for which we also analyzed the WaveCore mapping. The third sub-domain is detailed physical modeling, based on FDTD. Applications like detailed musical instrument modeling are relevant for this sub-domain. We focussed on the mapping of FDTD schemes for different geometries like strings, membranes and plates and combinations. The goal of this research has been to explore the capabilities of the processor concept in relation to the targeted application domain¹.

Physical modeling within the domain of musical applications can roughly be classified in three domains:

- ¹Major parts of this chapter have been published in [MJW:6] and [MJW:5]

versatility. Examples of such models are guitar effects (e.g. phaser, flanger, chorus, equalizer), or analog synthesizers (e.g. Moog-filters, modular synthesizers).

2. *Digital Waveguide synthesis*

The digital waveguide is an abstraction of a medium through which an acoustical wave propagates. Digital waveguide synthesis focusses on abstract physical models of for instance acoustical spaces. The most important driving force behind the development of this domain is reduction of computational requirements which are often associated to detailed mathematical models of objects through which acoustical waves propagate. Perhaps the most important example of an application within this domain is acoustical reverberation: real-time simulation of acoustical spaces to add a spatial illusion to a musical instrument (e.g. as if it is positioned in a concert hall).

3. *Detailed physical modeling*

Detailed physical modeling is mostly based on finite element methods. The objective is to get a detailed realistic behaviour of the modeled object. Examples of modeled objects are musical instruments themselves (e.g. a detailed model of a guitar or a brass instrument), or reverberation objects. Such models can actually render acoustical signals from the model, depending on a stimulation method, in a realistic way. Application examples are instrument development tools, advanced synthesizers or realistic sound rendering in advanced computer games. Obviously, the required computational resources for particularly real-time applications can be extremely demanding.

In this chapter we probe into the three mentioned application domains. Per application domain we analyze one or more examples and analyze how the derived models can be mapped on the WaveCore processor.

4.2 VIRTUAL ANALOG MODELING

Virtual analog modeling deals with the translation of analog (e.g. electronic circuit) behaviour to a discrete-time model which can be executed in real-time. Within the domain of virtual analog modeling we focus on basic guitar effects.

The implementation of a generic virtual analog model within a WPG is defined in fig. 4.1. The "Virtual Analog Process" implements the actual processing core of the virtual analog model (i.e. the physical model). This process receives input tokens from the "ADC" process, which is an abstract model (foreign process) of an ADC. Likewise, the "Virtual Analog Model" emits its processed outgoing tokens into the "Resp" edge which is connected to an abstract model (foreign process) of a DAC. The "Virtual Analog Process" is pseudo-statically (i.e. based on events) controlled through the "Param" edge. The "Control Process" (i.e. the interaction model) runs on a GPP. The idea is that the "Control Process" allows to change settings of the modeled analog device (e.g. "bass", "treble", etc.). The processing chain

FIGURE 4.1 – Virtual Analog model, captured in WaveCore process graph

is synchronized by means of a single fire-link, called "AudioRate" (e.g. 48kHz). The WPG in fig. 4.1 can be compiled to our Atlys development board (see fig. 3.35), in combination with a connected computer (e.g. notebook) on which the "Control Process" is mapped on the GPP. The "ADC" and "DAC" processes are mapped on the AC97 audio codec on the Atlys board.

We start this section with an overview of the basic modeling techniques that play a central role in virtual analog modeling. Next to that we give a few examples on how such models can be derived from an analog reference point, and how these virtual analog models can be translated to WaveCore processes.

4.2.1 BASIC FUNCTIONS

Linear Time-Invariant systems

A wide range of common signal-processing functions can be classified as an LTI-system [57]. Examples of such functions are linear filters, harmonic oscillators and signal delays. Mathematically, an LTI-system can be described as a linear differential equation with constant coefficients. The general form of this continuous-time differential equation is expressed in (4.1).

$$y(t) = \frac{1}{\alpha_0} \left(\sum_{j=0}^M \beta_j \frac{d^j x(t)}{dt^j} - \sum_{i=1}^N \alpha_i \frac{d^i y(t)}{dt^i} \right) \quad (4.1)$$

An important property of a LTI system is that it can be characterized by its impulse response (i.e. the response to a Dirac pulse [57]). The output of an LTI system can be derived by the convolution of the input signal with the impulse response of the system, like defined in (4.2).

$$y(t) = x(t) * h(t) = \int_{-\infty}^{+\infty} x(\tau) h(t - \tau) d\tau \quad (4.2)$$

The transformation of a continuous-time LTI system to its discrete-time counterpart results in two important classes of discrete-time LTI systems:

FIGURE 4.2 – Transposed direct form FIR filter, represented as a WPP

Class 1: the FIR filter:

If the impulse response of the system is finite, or can be approximated with a finite response, then the system can be modeled with a so-called FIR filter. Such a FIR filter performs the discrete-time convolution operation on its input signal, like expressed in (4.3). Actually, the discrete-time formulation of the convolution in (4.3) is derived from the continuous time convolution in (4.2).

$$y[n] = \sum_{i=0}^{N-1} b_i x[n-i] \quad (4.3)$$

The coefficients b_i of this FIR filter represent the impulse response at discrete timestamps $[i]$. The implementation of an N-taps FIR filter as a WaveCore WPP is depicted in fig. 4.2. Our choice to apply the transposed direct form of the FIR filter is because of its efficiency. Within this form it is possible to combine the adder, delay and multiplier per filter-tap to a MAD-type PA. As a result, an N-taps FIR filter can be executed within N clock cycles (1 cycle/filter-tap) by the GIU. If N exceeds the capacity of a single PU, then the filter may be partitioned over multiple PUs.

$$H(z) = \sum_{i=0}^N b_i z^{-i} \quad (4.4)$$

The corresponding frequency response is defined in (4.4).

Class 2: the IIR filter:

If the impulse response is infinite w.r.t. time, and it cannot be accurately approximated within practical time-bounds, then an IIR structure is usually applied. The general form of an IIR filter is defined in (4.5).

$$y[n] = \frac{1}{a_0} \left(\sum_{i=0}^{N-1} b_i x[n-i] - \sum_{j=1}^M a_j y[n-j] \right) \quad (4.5)$$

Like the FIR filter, the IIR filter in (4.5) can also be directly implemented as a WaveCore WPP, as depicted in fig. 4.3. However, because of the deep recursive structure of (particularly high order N) the implementation of an IIR filter is usually a cascade of a first order and possibly more than one second order filter stages. This partitioned structure can only be derived if the numerator and denominator of the

FIGURE 4.3 – Transposed direct-form 2 IIR filter, represented as a WPP

system function can be written as a product of first and second order factors.

$$H(z) = \frac{\sum_{i=0}^P b_i z^{-i}}{1 + \sum_{j=1}^Q a_j z^{-j}} \quad (4.6)$$

The corresponding frequency response of the general IIR filter is defined in (4.6).

Low-Frequency Oscillator

Next to LTI-system models, another important function is the low-frequency oscillator (LFO) ². An important application of the LFO within ReMap applications, is that its output is often used to modulate parameters of the signal-processing part of the model. The waveforms that are generated by an LFO are often triangular, sinusoidal or randomly shaped and its frequency is very low (with very low bandwidth), often in the sub-Hz range. We will discuss two types of LFOs: one based on the Schmitt trigger [76], and the other one based on the Lorenz attractor [47].

LFO-type 1: Schmitt-trigger oscillator.

A common way to generate a triangular wave is by means of an accumulator, combined with a hysteresis function (Schmitt trigger). An example analog circuit, and equivalent WaveCore WPG is depicted in fig. 4.4. A brief explanation of the analog circuit operation is as follows. The capacitor C charges through a current that is

²The LFO is usually not an LTI system, because it is practically not feasible to generate a periodic signal within the sub-Hz frequency range, by means of a linear system

FIGURE 4.4 – Schmitt-trigger oscillator

delivered by the inverting Schmitt-trigger (74LS14 component), while the Schmitt trigger itself has a zero voltage at its output, and hence will switch when the input V_{tr} will rise above the positive threshold-level. When the capacitor is charged above this positive threshold level, the input of the 74LS14 will cause the Schmitt-trigger to switch to a high level, while the output of the component switches to its inverted (zero) value. This event will cause a polarity switch that will cause the capacitor to discharge through the resistor until the input of the 74LS14 drops below the negative threshold value and hence switches again in the opposite direction. The time constant $\tau = R.C$ determines the frequency $f_{ifo} = k.\tau$ of the generated triangular wave.³

The equivalent WaveCore WPP for the Schmitt-trigger LFO works in a similar way. The hysteresis function symmetrically switches between -1 and 1. Suppose that the output of the hyst-function is -1. Then the input of the accumulator (the adder/delay combination) subsequently (synchronous to the associated fire-link) adds a small number *Rate*, until the output of the accumulator reaches 1. At that particular moment, the hyst-function switches from -1 to 1. As a result, the polarity of the input of the accumulator switches to $-Rate$, and hence the accumulator starts to subtract the small number *Rate*, until the output of the accumulator reaches -1 again. The result is the generation of a periodic triangular wave. Note that the *Rate* input has to be negative. A "WaveShaper" can be added to shape the triangular input to for instance an approximated sine wave.

The hysteresis function is defined in listing 4.1. As can be seen, the output of the function "Hyst" is fed back to its input at every firing event, and hence the hysteresis function is synchronous to a fire-link. In order to map the "Hyst" function onto a WPG, the "if else" structure needs to be translated into a dataflow graph representation. An "if else" structure can be represented as a multiplexer, like shown in fig. 4.5.

³The actual generated waveform of the analog Schmitt-trigger oscillator approximates a triangular wave, where the accuracy of the approximation depends on the values of R, C and the switching levels of the Schmitt trigger

```

1  pt Hyst(pt x[n], y[n]) {
2      pt y[n+1];
3      if (y[n] < 0) {
4          if (x[n] > 1)
5              y[n+1]=1;
6          else
7              y[n+1]=-1;
8      }
9      else {
10         if (x[n] < -1)
11             y[n+1]=-1;
12         else
13             y[n+1]=1;
14     }
15     return(y[n+1]);
16 }
17 // Execute Hyst function on fire-event:
18 y[n+1]=Hyst(x[n],y[n]) @ fireEvent;

```

LISTING 4.1 – Hysteresis function

FIGURE 4.5 – Multiplexer function

When we apply this multiplexer to the hysteresis behaviour, as listed in 4.1, we can express the hysteresis function as a dataflow graph (i.e. a WaveCore WPG) like depicted in fig. 4.6. Note that, unlike an imperative execution of the reference code of the hysteresis function, all branches in the dataflow graph are always executed upon each fire event. This clearly is a disadvantage w.r.t. conditional execution on a traditional von Neuman processor.

LFO-type 2: The Lorenz oscillator.

Edward Lorenz studied atmospherical convection [47] and for this purpose he found a relatively simple set of ordinary differential equations, which became well known as the Lorenz equations. The solution of the Lorenz equations [47] can be interpreted as the orbit of an object, circling around two adjacent "attractors". The interesting aspect about this orbit is that it behaves chaotically, though stable: despite the fact that the Lorenz equations can be solved accurately in a numeric

FIGURE 4.6 – Hysteresis function

way, the orbit is completely different for even very small deviations in initial conditions, or slightly different computational accuracy.⁴ This phenomenon has become well known as the "butterfly effect": theoretically the flap of the butterfly's wings in southern America can cause a tornado in the Philippines.

This chaotic behaviour can be used to randomly control a parameter of the virtual analog model. Particularly for a musical application this randomness is an attractive property because it is per definition not correlated to the rhythmic aspect of music and hence leads to the perception that a randomly controlled parameter is embedded in the overall sound production in a subtle way.

The orbit can be derived mathematically by solving the system of ordinary differential equation (ODE)s in (4.7), which describes the velocity vector of the orbiting object as a function of time.

$$\vec{v}(t) = \frac{dx(t)}{dt} \hat{i} + \frac{dy(t)}{dt} \hat{j} + \frac{dz(t)}{dt} \hat{k} = \begin{bmatrix} \sigma\{y(t) - x(t)\} \\ x(t)\{\rho - z(t)\} - y(t) \\ x(t) \cdot y(t) - \beta \cdot z(t) \end{bmatrix} \begin{bmatrix} \hat{i} \\ \hat{j} \\ \hat{k} \end{bmatrix} \quad (4.7)$$

The actual location of the moving object, as a function of time, is expressed in (4.8). The initial location of the object is $\vec{s}(0)$.

$$\vec{s}(t) = \int_0^t \vec{v}(\tau) d\tau + \vec{s}(0) \quad (4.8)$$

We use a linear approximation method to solve this ODE system, which yields the set of difference equations (i.e. "solver" equations) in (4.9).

$$\vec{\delta s}[n] = \begin{bmatrix} \sigma(y[n] - x[n]) \\ x[n](\rho - z[n]) - y[n] \\ x[n] \cdot y[n] - \beta \cdot z[n] \end{bmatrix} \begin{bmatrix} T_s \cdot \hat{i} \\ T_s \cdot \hat{j} \\ T_s \cdot \hat{k} \end{bmatrix} \quad (4.9)$$

⁴The chaotic behaviour only occurs for a certain combination of the parameters and initial conditions of the system

FIGURE 4.7 – Lorenz oscillator, implemented as WaveCore WPG

FIGURE 4.8 – Results of WaveCore simulation of the Lorenz oscillator

The solver equations computes the displacement vector $\vec{\delta}_s[n]$ of the orbiting object for each iteration period T_s . The actual location of the orbiting object is updated per iteration period, like expressed for the evolving location in space in eq.4.10.

$$\vec{s}[n+1] = \vec{s}[n] + \vec{\delta}_s[n] \quad (4.10)$$

The motion of the object (and also the computational accuracy) is proportional to the iteration period T_s , which directly follows from the solver equations. The WaveCore WPP structure of the Lorenz oscillator is depicted in fig. 4.7.

The result of a WaveCore simulation of the Lorenz oscillator is depicted in fig. 4.8. For this simulation we used the following parameters: $\sigma = 10$, $\rho = 28$, $\beta = 2.67$ and $T_s = 1/48k$. The simulation duration is 20 sec. Clearly visible are the two attractors around which the object orbits in a chaotic way.

Non-linear behaviour modeling

Non-linear behaviour in analog circuitry is usually minimized by design, because this is in many cases unwanted. However, within ReMap applications, non-linearities are sometimes the most important property. Examples are distortion effects or detailed analog (tube)-amplifier models. Similarly, the "WaveShaper" function, which we introduced in a previous subsection on LFO modeling, is a non-linear function. Implementations of non-linear functions as a WaveCore WPP may be one of the following:

1. *Polynomial approximation*
Polynomial approximations may be derived from Taylor approximations of the modeled function, and can be constructed by the supported PAs (i.e. adders, multipliers).
2. *Transcendental function approximation*
Transcendental functions, like $y[n] = \tanh[x]$ may be approximated piecewise, using if-else statements (like in the multiplexer example in fig. 4.5), combined with polynomial approximations of the "pieces".

4.2.2 ANALOG GUITAR EFFECTS

We analyze and map three example guitar effects on the WaveCore processor: the "Phaser", the "Flanger" and the "AutoWah". All these effects originate from analog implementations and are widely used in music electronics. Despite the fact that these effects can be implemented digitally, as we will see in this chapter, analog versions of these effects are still widely applied as of today.

The Phaser

The phaser finds its roots in another effect which is well known: the Leslie speaker [109]. The "Leslie effect" is based on a mechanically rotating speaker which is enclosed in a cabinet. The rotation causes multiple acoustical reflections within the closed cabinet, with associated Doppler shifts. The overall effect is a combination of tremolo (periodic variation in volume) and chorus (periodic variation of pitch). The analog phaser has been invented in an attempt to reproduce the Leslie effect in a cheap manner, and became popular in the late 1960s. The basic principle of the phaser is depicted in fig. 4.9. The heart of the phaser consists of a cascade of "allpass filters". The allpass filter has a "flat" magnitude response (i.e. the magnitude of each frequency is passed through the filter equally), while the filter modifies the phase of the input signal, depending on the frequency. Within fig. 4.9 there are 12 of such allpass filters chained within 3 clusters (each one consisting of 4 cascaded allpass filters). The phase shifted signal is selected from 3 taps, where either 4, 8, or 12 cascaded allpass sections can be selected by means of the associated switches. The phase-shifted input signal is added to the original input signal $x(t)$. This implies that frequencies which are phase-shifted with $\phi = (2n - 1)\pi$ cause complete cancellation of these frequencies in the output signal $y(t)$ (notches in the overall

FIGURE 4.9 – Functional block diagram of the Phaser

FIGURE 4.10 – Analog allpass filter section

frequency response). The typical phasing effect is caused by dynamically sweeping these mentioned notches through the frequency spectrum. This is achieved by a LFO. This LFO controls a frequency sensitive parameter for each allpass filter within the phase-shifting chain. Typically, the frequency of the (sinusoidal, or triangular) signal which is generated by the LFO is in the sub-Hz range (e.g. few seconds period time). Hence the notches in the frequency spectrum periodically move back and forth accordingly, which causes the typical phasing sound. A portion of the phase-shifted signal is fed back to the input of the chain. This feedback causes constructive interference (i.e. resonance) at frequencies with a phase-shift of $\phi = n.2\pi$.

As mentioned, the basic function within the phaser is the allpass filter. The schematic of this filter is depicted in fig. 4.10. This analog filter is built around an op-amp which is used in inverting mode (i.e. the output voltage of the op-amp inversely follows the difference between the inputs: $v_o = -A.(V_p - V_n)$). The fundamental property of the op-amp is that its gain A is very large, such that it can be approximated as infinitive. This implies that the voltage differences at the input terminals of the op-amp can be considered zero. If we apply this property to the analog allpass

filter in fig. 4.10, then we can derive the equations in (4.11).

$$v_p = v_n \text{ with } \begin{cases} v_p = \left(\frac{R_v}{R_v + \frac{1}{j\omega C}} \right) \cdot v_i \\ v_n = \frac{1}{2} \cdot (v_o - v_i) \end{cases} \quad (4.11)$$

The frequency response $H(j\omega) = \frac{v_o(j\omega)}{v_i(j\omega)}$ follows directly from (4.11) and is defined in (4.12).

$$H(j\omega) = \frac{j\omega\tau - 1}{j\omega\tau + 1} \text{ with } \tau = R_v \cdot C \quad (4.12)$$

We can rewrite equation (4.12) in exponential form, and hence make the magnitude and phase explicit. This yields equation (4.13).

$$H(j\omega) = |H(\omega)|e^{j\phi(\omega)} = \frac{\sqrt{1 + \omega^2\tau^2} \cdot e^{j\{\pi - \arctan(\omega\tau)\}}}{\sqrt{1 + \omega^2\tau^2} \cdot e^{j\arctan(\omega\tau)}} = e^{j\{\pi - 2\arctan(\omega\tau)\}} \quad (4.13)$$

From equation (4.13) it becomes obvious that the magnitude response of the allpass filter section indeed is 1 for all frequencies. The phase response $\phi(\omega)$ also follows directly from (4.13), like defined in (4.14).

$$\phi(\omega) = \pi - 2 \cdot \arctan(\omega \cdot \tau) \quad (4.14)$$

From this phase response it can be derived that the allpass filter inverts its input signal for frequencies which approximate zero ($H(j\omega) = e^{j\pi} = -1$ for $\omega \simeq 0$), and is transparent for high frequencies ($H(j\omega) = e^{j0} = 1$ for $\omega \rightarrow \infty$). Now, a cascade of n allpass filters yields a phase response which is defined in (4.15).

$$\phi(\omega) = n \cdot \pi - 2n \cdot \arctan(\omega \cdot \tau) \quad (4.15)$$

Subsequently, the overall phaser response can be derived by adding the input signal to the phase-shifted input signal (in our analysis we neglect the feedback path in the system). This yields the overall phaser response, as expressed in (4.16).

$$P(j\omega) = 1 + e^{j\phi(\omega)} \quad (4.16)$$

$$|P(\omega)| = \sqrt{2 + 2\cos[\phi(\omega)]} \quad (4.17)$$

The magnitude response of the overall phaser (with neglected feedback path) is defined in (4.17). The notches in the frequency response for an N -stage phaser are located at those frequencies where the phase-shift equals $(2k - 1)\pi$, like expressed in (4.18). For these frequencies ω_0 , the N allpass-filter stages yield an output signal which is exactly the inverted input signal.

$$(2k - 1)\pi = n\pi - 2n \arctan(\omega_0 \tau) \text{ with } (k, n) \in \mathbb{N}^+, 1 \leq k \leq \frac{1}{2}n \quad (4.18)$$

When we solve (4.18) for ω_0 we find the notch-frequencies as a function of τ , for each value of k , in (4.19).

$$\omega_{0,k}(\tau) = \frac{1}{\tau} \tan\left[\frac{\pi}{2n}(n + 1 - 2k)\right] \quad (4.19)$$

FIGURE 4.11 – Phase and magnitude response of an 8-stage Phaser

Similarly, we can derive the resonance frequencies ω_p which is expressed as a function of τ , for each value of k , in (4.20).

$$\omega_{p,k}(\tau) = \frac{1}{\tau} \tan\left[\frac{\pi}{2n}(n - 2k)\right] \quad (4.20)$$

The relation between the phase-shift and magnitude response of the overall phaser, for $\tau = 1$, is depicted in fig. 4.11, for an 8-stage ($n=8$) phaser. As can be seen, for this phaser with $n=8$ we find 4 notches and 4 resonance peaks. Clearly visible is that these notches and resonance peaks are not equally spaced w.r.t. frequency, since the phase response is non-linear.

When the parameter τ is varied according to a periodic (sinusoidal) function (i.e. generated by the voltage-controlled oscillator (VCO)), then the effect is that the magnitude response scales accordingly. This effect is visualized in the contourplot in fig. 4.12. The vertical axis in this plot represents the frequency, and the horizontal axis represents time. Light colours in the plot represent a high value of the magnitude, and dark colours represent low values of the magnitude.

A virtual analog model of the phaser which we can map onto the WaveCore implies that we need to transform the continuous-time frequency response $H(j\omega)$ to a discrete-time approximation $H(z)$. We use the bilinear transform to achieve this, which implies the substitution of $j\omega$ like defined in (4.21).

$$j\omega \leftarrow \frac{2}{T_s} \cdot \frac{z - 1}{z + 1} \quad (4.21)$$

Applying the bilinear transform [57] to (4.12) yields the frequency response $H(z)$ of the discrete-time model for the allpass-section, which is defined in (4.22).

$$H(z) = \frac{k + z^{-1}}{1 + k \cdot z^{-1}} \text{ with } k = \frac{T_s - 2 \cdot \tau}{T_s + 2 \cdot \tau} \quad (4.22)$$

FIGURE 4.12 – Magnitude response contour plot of the Phaser

From the frequency response $H(z)$ we can derive the corresponding difference equation through substituting $z^{-d}P(z)$ terms by discrete-time derivatives $p[n-d]$. This yields the difference equation of the allpass filter section, as defined in (4.23).

$$y[n] = k.x[n] + x[n-1] - k.y[n-1] \quad (4.23)$$

Finally, we can derive a graph representation of the discrete-time allpass section, which is depicted in fig. 4.13. Putting all pieces together (i.e. LFO, composition of allpass-filter sections, adders and multipliers) to one phaser-WPP is straightforward, and almost immediately follows from the functional block diagram in fig. 4.9. We instantiated the Lorenz LFO in the phaser model. The parameters "depth4", "depth8", "depth12", "resonance", "rate", "swing" and "centerfreq" are grouped together in one control-token definition and connected to the control port of the "Virtual Analog Process" (see fig. 4.1). The latency of the processing chain equals two firing periods ($2T_s$) ($1T_s$ for the "Stim" and "Resp" edges), plus additional latency in the AC97 codec (see fig.4.1). Note that the phaser process (i.e. "Virtual Analog Process") itself does not add latency since there exists a combinatorial path between input and output of the phaser.

FIGURE 4.13 – Bilinear transformed allpass filter section

TABLE 4.1 – Compiled Phaser

Compile results of algorithm: Phaser					
GIU utilization		Resource load		Mapping metrics	
Instructions	412	Pmem	456	Required PUs	1
Data-hazards	52.2%	Xmem	98	Total #PAs	172
Thread-sync	0.0%	Ymem	99	Total #DAGs	61
Pntr updates	0.0%	Cmem	106	Total #delay-lines	0
LSU utilization		Cycle-budget	23.0%	Longest path	28
Instructions	44	L2 GPN-load	0.0%	Firing rate (kHz)	48
DMA transactions	11	L3 bus load	0.6%	L3 mem (Bytes)	88

Compile results of the phaser are summarized in table 4.1. An explanation of the table can be found in appendix B. As can be seen, the phaser is a lightweight algorithm which utilizes 23% of a single PU. A remarkable property of this mapping is the high percentage of data-hazards. These data-hazards result in pipeline-bubbles (i.e. NOP instructions), as is explained in section 3.4.3. Approximately one out of two GIU instructions is a pipeline-bubble. The explanation of this high number is the relatively long combinatorial path through one of the DAGs. This combinatorial path starts at the input of the phaser and goes through all 12 all-pass sections. This path consists of a cascaded structure of 28 combinatorial PAs (i.e. 2 combinatorial PAs, PA₁ and PA₂ in fig. 4.13, per allpass section, plus 4 additional combinatorial PAs at top-level). The data dependencies between these PAs, combined with the relatively deep GIU pipeline and the relatively few remaining PA which can be scheduled between the mentioned data-dependent PAs, are the root-cause of the many data-hazards.

The Flanger

The Flanger effect finds its origin in Abbey Road studio back in 1966. A setup with two tape recorders like depicted in fig. 4.14 was used. The same track was recorded on both tape recorders and simultaneously played back. The output signal of both

FIGURE 4.14 – Flanging with two tape recorders at Abbey Road studio

recorders was added to one final output signal. Sound engineers found out that a typical modulation-like sound effect arose when one of the "flanges" was manually slowed down a bit and subsequently released. As a result, the playback speed of one of the recorders varies slightly with respect to the other one.

Mathematically this effect, called "flanging" can be modeled like depicted in the left schematic in fig. 4.15. The principle of the flanger is that its input signal is guided through a variable-length delay-line $y(t) = x(t - \Delta t)$ and subsequently added to the non-processed input signal. Typically, this delay is within the range 5ms - 25ms. In this subsection we will show that this implies a comb-filter: a number of equidistant notches in the frequency response of this comb-filter cancels out the corresponding frequencies from the input signal. A LFO causes dynamic variation of the distance between these notches, which causes the typical sound of the flanger effect. A feedback path from the output of the delay-line back to its input causes amplification of the frequencies between the notches which intensifies the effect.⁵ A more thorough background on the flanger effect can be found in literature: [22] [4] [82] [11] [16]

Originally, the flanger effect was built from analog electronics and is still being used despite the fact that digital flangers have become mainstream nowadays. The heart of the analog flanger is a bucket-brigade device (BBD) [22]. This is basically an analog shift register which samples its input and subsequently moves the charge quantity which is associated to each analog sample through the chain. In principle the BBD is a cascade of capacitors, each one holding an analog sample. Each analog sample is moved through the chained capacitors, synchronous to a clock signal.

⁵Note that the feedback from output of the Flanger, back to the input, is not present in the tape-recorder setup

FIGURE 4.15 – Principle of the Flanger effect, and its WaveCore WPP representation

The speed at which the analog samples move through the BBD is a metric for the signal delay. Hence, the delay length can simply be modulated by applying different clock frequencies to the BBD. This clock signal is generated by the combination of an analog Schmitt-trigger LFO, which output signal is an input to a VCO which finally generates the clock signal for the BBD.

We focus on the analog vintage MXR M-117 flanger [1]. The schematic on the left in fig. 4.15 is an abstraction of the electronics within this flanger. The LFO within this flanger generates a signal which directly represents the delay-line length $\Delta(t)$, like expressed in (4.24).

$$\Delta(t) = A \cdot \sin(\omega_0 \cdot t) + k \quad (4.24)$$

With:

1. *manual*
The "manual" parameter k controls the DC-offset of the LFO output.
2. *rate*
The "rate" parameter controls the frequency ω_0 of the LFO and hence the speed at which the distance between the notches in the frequency response decrease/increase.
3. *width*
The "width" parameter controls the amplitude A , and hence the variation of the distance between the notches in the frequency response of the flanger.

The WaveCore WPP representation of the MXR M-117 flanger is depicted as the schematic on the right hand in fig. 4.15. As can be seen, this representation is a straightforward translation of the abstract analog model within fig. 4.15. We apply the Schmitt-trigger LFO which we introduced in subsection 4.2.1. The BBD can be modeled with one variable-length delay line which is an intrinsic function within the PA concept (see section 3.2.4). Hence, the entire flanger core (i.e. everything except the LFO) can be captured with only three PAs:

1. *PA1*
The multiplier PA1 implements the regeneration parameter "regen" (β) through feeding back a portion of the output of the delay-line to the input of the flanger.
2. *PA2*
PA2 combines the adder in the feedback-path with the modulated delay-line. The full length of this delay-line equals $\Lambda = M$ samples. The value M is derived from the firing rate of the process in which the flanger-WPP is included. When we assume a firing rate of 48kHz, and a maximum delay of 25ms, then we choose $M = 1200$.
3. *PA3*
The PA3 adds the input signal of the flanger to the delay-line output, yielding the flanger output $y[n]$.

In our next analysis, where we will show that the flanger core is a variant of a comb-filter, we assume λ to be constant. The difference equations for the flanger-core (i.e. PA1, PA2 and PA3 in fig. 4.15) can be derived from fig. 4.15, and are given in equations 4.25 and 4.26.

$$p[n] = x[n] + \beta \cdot p[n - \lambda] \quad (4.25)$$

And

$$y[n] = x[n] + p[n - \lambda] \quad (4.26)$$

With:

$$\lambda[n + 1] = \lfloor M \cdot (\tau[n] - 1) \rfloor \quad (4.27)$$

Dynamic variation of the delay-line length, as is expressed in equation 4.27, is explained in section 3.2.4. After applying the z -transformation to the difference equations and merging the transformed equations, we find the transfer function in the z domain which is defined in equation 4.28

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1 + (1 - \beta)z^{-\lambda}}{1 - \beta \cdot z^{-\lambda}} \quad (4.28)$$

In order to analyze the magnitude response of the flanger-core, we replace z by $e^{j\theta}$, and subsequently derive the magnitude response which is defined in equation 4.29.

$$|H(e^{j\theta})| = \sqrt{\frac{1 + (1 - \beta)^2 + 2(1 - \beta)\cos(\lambda\theta)}{1 + \beta^2 - 2\beta\cos(\lambda\theta)}} \quad (4.29)$$

The magnitude response of the flanger is depicted in fig. 4.16, for a constant value of λ . The influence of the feedback parameter β is visualized in fig. 4.16 for two values. Clearly visible is the periodicity of the response function and the intensity of the resonance peaks for high feedback. The location of the peaks in the magni-

FIGURE 4.16 – Magnitude response of the Flanger

FIGURE 4.17 – Magnitude response contour plot of the Flanger

TABLE 4.2 – Compiled flanger

Compile results of algorithm: flanger					
GIU utilization		Resource load		Mapping metrics	
#instructions	119	Imem	162	Required PUs	1
Data-hazards	38.7%	Xmem	27	Total #PAs	50
Thread-sync	0.0%	Ymem	28	Total #DAGs	19
Pntr updates	0.0%	Cmem	26	Total #delay-lines	1
LSU utilization		Cycle-budget	6.6%	Longest path	7
Instructions	43	L2 GPN-load	0.0%	Firing rate (kHz)	48
DMA transactions	10	L3 bus load	0.6%	L3 mem (Bytes)	4864

tude response of the comb-filter follows from equation 4.29, and is as defined in equation 4.30.

$$\theta_{peak}(k) = k \frac{2\pi}{\lambda}, k = 0, 1, 2, \dots, \lambda - 1 \quad (4.30)$$

So, the number of equidistant notches and peaks in the magnitude response of the comb-filter is equal to λ and is directly related to $\tau[n]$ through the relation between $\tau[n]$ and λ in equation (4.27). The effect of the feedback path with the *regen* amplifier is that the peaks in the magnitude response get smaller and intenser for high *regen* values, which also follows from equation 4.29, and is visualized in fig. 4.16

Note that we did not take fractional delay-line length interpolation into account (to suppress "zipper-noise" due to the discrete-length model of the delay-line).

The magnitude response is scaled periodically like an accordion when the delay-line length λ is varied, as shown in the contourplot in fig. 4.17. The x-axis represents the time and the y-axis represents the frequency. Dark colours represent low values of the magnitude response and light colours represent high values of the magnitude response.

The WPP implementation of the flanger can be directly derived from the block diagram in fig. 4.15. The applied PAs in this figure are indicated by the round-edge rectangular boxes. We instantiated the Schmitt-trigger LFO in the flanger model. The parameters "rate", "manual" and "width" are grouped in one control-token definition and connected to the control port of the "Virtual Analog Process" (see fig. 4.1). The latency of the processing chain equals two firing periods ($2T_s$) ($1T_s$ for the "Stim" and "Resp" edges), plus additional latency in the AC97 codec (see fig. 4.1). Note that the flanger process (i.e. "Virtual Analog Process") itself does not add latency since there exists a combinatorial path between input and output of the flanger. Compile results of the flanger are summarized in table 4.2. An explanation of the table can be found in appendix B. Like the phaser, the flanger is lightweight algorithm, which utilizes only 6.6% of a single PU. Like within the phaser, the relatively large number of data-hazard pipeline bubbles are caused by a relatively long combinatorial path within a DAG that is associated to the hysteresis

function within the LFO, combined with relatively few PAs which can be scheduled instead of mentioned bubbles.

The AutoWah

The Wah-Wah effect originated in the 1920s. Back then trumpet or trombone players found out that they could produce an extra expression to their sound by moving a mute in the instrument's bell. This extra expression mimics properties of the human voice, hence it is called a "Wah" effect. Later in 1966 this effect was the inspiration for the invention of an electronic version: the WahWah pedal which was and still is applied to the electric guitar. The guitar player can control the effect by rocking the pedal with his/her foot. In this section we will discuss the "AutoWah". Unlike manual control by an instrument player, the control of the "AutoWah" is automatically derived from slowly varying properties of the electrical signal that is produced by the instrument.

The WahWah effect is based on a bandpass filter with center frequency ω_0 and quality-factor Q , where the center frequency marks the middle of the pass-band and quality-factor determines the bandwidth of the pass-band. The center frequency is dynamically varied within the WahWah effect. This can be done in various ways (e.g. expression pedal or controlled by input signal properties like envelope). In our example we chose an envelope controlled WahWah, which is sometimes referred to as 'AutoWah'. The continuous-time transfer function for a second order bandpass filter with ω_0 and Q parameters is defined in equation 4.31.

$$H(s) = \frac{\left(\frac{1}{Q \cdot \omega_0}\right) \cdot s^2}{\left(\frac{1}{\omega_0^2}\right) \cdot s^2 + \left(\frac{1}{Q \cdot \omega_0}\right) \cdot s + 1} \quad (4.31)$$

We need to translate the continuous-time transfer function to its discrete-time counterpart. For this purpose we use the bilinear transformation (like we did for the Phaser), which substitutes s with z , according to the substitution rule in equation 4.21. Application of the bilinear transformation yields the discrete-time transfer function $H(z)$ in equation 4.32, which embodies the required bandpass filter behavior with ω_0 and Q parameters.

$$H(z) = \frac{a(1 - z^{-2})}{(a + b + 1) + 2(1 - b)z^{-1} + (1 - a + b)z^{-2}} \quad (4.32)$$

With:

$$a = \frac{2}{Q \cdot \omega_0 \cdot T_s} \text{ and } b = \frac{4}{T_s^2 \cdot \omega_0^2} \quad (4.33)$$

The final step is to link the coefficients in the derived transfer function $H(z)$ to the coefficients in the general second order discrete-time transfer function (i.e. biquadratic or BiQuad function), which is defined in equation 4.34.

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \quad (4.34)$$

FIGURE 4.18 – AutoWah as a WaveCore process partition

The relation between the coefficients in the transfer function in equation 4.32 and the coefficients in the generic second order transfer function in equation 4.34 is defined in equations 4.35, 4.36 and 4.37.

$$b_0 = \frac{a}{d}, b_1 = 0, b_2 = \frac{-a}{d} \quad (4.35)$$

And:

$$a_1 = \frac{2(1-b)}{d}, a_2 = \frac{1-a+b}{d}, \quad (4.36)$$

With:

$$d = a + b + 1 \quad (4.37)$$

We map the generic transfer function in equation 4.34 on a direct-form II structure [57]. This structure is depicted in the "DF2-IIR" WP-partition in fig. 4.18. The transfer function in equation 4.34, and associated parameters in equations 4.35, 4.36 and 4.37 apply to this DF2-IIR structure (note that the parameter b_1 equals zero, and therefore is left out).

The recipe for computing the coefficients b_0 , b_2 , a_1 and a_2 from equations 4.35, 4.36 and 4.37 is implemented in the function called "Coefficient Computation" (obviously with inputs ω_0 and Q). The "Env-Follower" function in fig. 4.18 detects the envelope of the input signal $x[n]$ and uses this envelope to compute the center frequency ω_0 . Envelope detection is implemented by a single-phase rectification to the input signal, followed by lowpass filtering. The envelope signal is scaled and subsequently determines the frequency of the LFO (same LFO as used in the flanger in fig. 4.15). The sine-shaped signal output of the LFO is scaled with the *swing* parameter and added to the offset *freq*, and finally fed to the "Coefficient Computation" function.

TABLE 4.3 – Compiled AutoWah

Compile results of algorithm: AutoWah					
GIU utilization		Resource load		Mapping metrics	
#instructions	119	Imem	154	Required PUs	1
Data-hazards	5.9%	Xmem	52	Total #PAs	88
Thread-sync	0.0%	Ymem	52	Total #DAGs	26
Pntr updates	0.0%	Cmem	41	Total #delay-lines	0
LSU utilization		Cycle-budget	6.6%	Longest path	7
Instructions	35	L2 GPN-load	0.0%	Firing rate (kHz)	48
DMA transactions	8	L3 bus load	0.4%	L3 mem (Bytes)	64

The compile results of the AutoWah are summarized in table 4.3. An explanation of the table can be found in appendix B. The AutoWah WPP requires 88 WaveCore PAs. This implies 6.6% of the processing capacity of a single PU. The processing latency of the AutoWah algorithm is zero, since the shortest path from $x[n]$ to $y[n]$ does not contain delay-elements. Note that the GIU pipeline utilization is much better, compared to the phaser and flanger mappings.

4.3 DIGITAL WAVEGUIDE SYNTHESIS

Digital waveguide synthesis is a physical modeling methodology which focusses on the behaviour of travelling waves through the modeled object (e.g. a musical instrument model or an acoustical space model). As we will show, the delay-line plays a central role in this methodology. A Digital Wave Guide DWG model is a coarse abstraction of the physical object, where delay-lines model lossless propagation of acoustical waves and losses, dispersion and non-linearities are modeled at the end-points of each waveguide (e.g. junctions). DWG synthesis is known to be computationally efficient, compared to detailed physical modeling (e.g. based on FDTD) and finds applications in many musical instrument electronics like synthesizers.

4.3.1 THE ROOTS: KARPLUS-STRONG AND D’ALEMBERT

Alex Strong developed a sound production algorithm which is based on feedback wavetable synthesis [38]. The principle of this algorithm is that a wavetable (i.e. a sample buffer) is repeatedly read out while a filtered portion of the output signal is fed back to the input of the wavetable. Hence the wavetable content is continuously modified by the algorithm. It turned out that when the buffer is initially filled with noise (i.e. random numbers), the produced sound closely matches with the decaying sound of a vibrating guitar string. The injection of noise into the buffer can be seen as plucking the string. Kevin Karplus analyzed the algorithm and found out that the feedback wavetable can be seen as an abstract model of a one dimensional waveguide. The block diagram of this algorithm, which is called the Karplus-Strong algorithm [38], is depicted in fig. 4.19. The Karplus-Strong algorithm has become

FIGURE 4.19 – Karplus Strong algorithm

one of the foundations of digital waveguide modeling, which has been further developed by Smith et.al. [35].

A travelling wave along a string can be described with the one dimensional wave equation in (4.38). The parameter c represents the wave velocity and a solution $u(x, t)$ of this partial differential equation describes the deflection (i.e. deviation from equilibrium) of any string-position x at any time t .

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \quad (4.38)$$

$$u(x, t = 0) = f(x), \quad \frac{\partial u}{\partial t}(x, t = 0) = g(x) \quad (4.39)$$

The general solution $u(x, t)$ with boundary conditions, as defined in (4.39), are known as d'Alembert's formula (4.40). The essence of d'Alembert's formula is that it describes the superposition of travelling waves $f(x + ct) + f(x - ct)$ in opposite directions along the string. The term $g(x)$ adds additional deflection due to the initial forces along the string.

$$u(x, t) = \frac{1}{2} [f(x + ct) + f(x - ct)] + \frac{1}{2c} \int_{x-ct}^{x+ct} g(s) ds \quad (4.40)$$

Now, d'Alembert's formula can be modelled as a digital waveguide by means of two delay-lines in opposite directions, as is depicted in fig. 4.20. The example in fig. 4.20 consists of two lossless waveguides, *WaveGuide1* and *WaveGuide2*, which are connected with a scattering junction. Both waveguides *WaveGuide1* and *WaveGuide2* can be seen as two-port devices where each port of the waveguide represents the propagation of waves in both directions. Wave W_1^+ at *WaveGuide1* propagates through a delay-line with discrete length L (i.e. it requires L sampling periods for the propagation of wave W_1^+ to the scattering junction). Similarly, it requires the identical number of sampling periods L for the propagation of the wave in the opposite direction to W_1^- . The superposition of W_1^+ and W_1^- is the discrete-time representation of $u(t, x)$ at the left-hand side of *WaveGuide1*. A waveguide can be characterized with a wave impedance R . Waveguides can be connected by a scattering junction, as is also depicted in fig. 4.20. Connected waveguides with different wave impedances lead to transmission and reflection of the travelling wave at the junction of the waveguides. This is modeled with the scattering junction. The

FIGURE 4.20 – Digital waveguides W1 and W2 with scattering junction

relation between the wave impedances $R1$ and $R2$ and the scattering parameter k_1 is expressed in (4.41). If a waveguide is terminated in such a way that all the wave energy is reflected, then the digital waveguide model reduces to a single delay-line, as is conceptually the case in the Karplus-Strong algorithm.

$$k_1 = \frac{R_2 - R_1}{R_2 + R_1} \quad (4.41)$$

4.3.2 DIGITAL REVERBERATION

Reverberation is the acoustical impact that a physical space (e.g. a room or hall including objects within) has on a produced sound. Practically, this means that the acoustical waves, that are produced by the source, propagate from this source in a sphere-shaped way until the wavefronts are (partially) reflected by objects within or the boundaries of the enclosed physical space. The result is a complex combination of multiple reflections where parameters like distance, reflection properties and room shape play an important role. Depending on these parameters, the initiated wave ultimately will decay where the energy is absorbed by reflections and lossy propagation of the acoustical waves. Reverberation is widely used in musical applications. It adds the illusion that the played instrument is located in a virtual space.

Detailed modeling of reverberation can be achieved in a straightforward way through measuring the impulse response of the acoustical space. The assumption is that the acoustical space can be modeled as a linear system. Hence, the convolution of the obtained impulse response with the produced acoustical source-signal results into the observed "reverberated" sound. Note that such a model is static: any changes in the modeled setup (e.g. position of the observer/producer) results in a different impulse response. Now, the problem is that the duration of the impulse response is in practice long. Often the decay-time of reverberation is expressed as the time it takes to decay the produced acoustical signal with 60 dB. This decay-time, or reverberation time, is called t_{60} . If we assume $t_{60} = 2sec$ and a sampling rate of 50 kHz, then the required number of multiply-add operations per sample equals 100000, or 5 billion multiply-add operations per second. Obviously, this computational requirement is usually too demanding for practical implementations.

FIGURE 4.21 – Impulse response of FreeVerb

Another method to derive reverberation models is to model the acoustical space as a grid of finite elements. This can be achieved by means of digital waveguide mesh [53], or FDTD [8]. The grid density is directly related to the highest frequency which the model should be capable of to propagate: 20 kHz implies a wavelength of 1.27 cm. This implies that the number of required grid-points per m^3 equals 488189. A lossless 3D finite-difference model requires one multiply and six additions per grid-point per sampling period. If we assume a required sampling rate of 50 kHz, then the required computational complexity per $1m^3$ equals 170 billion operations per second. Obviously, this is far more demanding than impulse response modeling.

Rather than deriving a reverberation model from the physical aspects of the acoustical space, a computational inexpensive method can also be derived from the characteristics of a typical impulse response of the reverberation. If we take a closer look to a typical impulse response of reverberated acoustics, then we observe the following key characteristics:

1. *Early reflections*

The earliest part of the impulse response of reverberation are the so-called "early reflections". These are the first-reflected waves and their delay-times are directly linked to the size and geometry of the acoustical space.

2. *Late reverberations*

The second part of the impulse response is dominated by increasingly dense and diffused echos. The increase in density is caused by multiple reflections, which number exponentially add when time increases. Meanwhile the intensity of the response (the envelope) is characterized by an exponential decay.

An example of a typical reverberation impulse response is depicted in fig. 4.21.

Schroeder found a computationally cheap algorithm which approximates the typical impulse response [79] [78]. The "Schroeder reverberator" basically consists of two stages. The first stage mimics the "early reflections" and is built of a number of filtered-feedback comb-filter (*FFCF*)s. A *FFCF* is basically a delay-line which output is fed back to its input. Within the feedback path a linear filter causes the signal to decay according to the characteristics of the feedback filter. Note that the Karplus-Strong algorithm (excluding the excitation part) is also a form of a *FFCF*. Within the Schroeder reverberator, a number of these *FFCF*s operate in parallel on the input signal, while the outputs of these filters are added, yielding the output of the early-reflections stage. The length of the delay-lines are mutually prime. This implies that the delay-times (i.e. echos) for each *FFCF* within this early-reflection stage which are distributed accordingly, which results in the maximization of different reflections of the input signal.

The second part of the Schroeder reverberator consists of a cascade of allpass-filter stages, where the unity delay element in the filter is replaced with a delay-line. Such an allpass filter causes echo diffusion. A cascade of these filters, with mutually prime delay-line lengths, intensifies the diffusion effect. Finally, the output of this "late reverberation" stage yields the output signal of the reverberation algorithm.

Now, the human ear is ultimately trained to recognise natural reverberation. Note that almost every acoustical source that we perceive is naturally reverberated. Hence, the human ear easily perceives the output of a Schroeder reverberator as artificial. Therefore, lots of variants on this reverberation concept have been proposed [96]. One of them is the "FreeVerb" algorithm [66]. The "FreeVerb" is a refinement of the Schroeder reverb, and its block diagram is depicted in fig. 4.22. Like the Schroeder reverberator, the FreeVerb consists of an early reflection stage (eight *FFCF* filters) and a late reverberation (diffusion) stage (four cascaded allpass stages). The transfer function in the Z-domain for the applied *FFCF* is defined in 4.42. The parameter f determines the gain of the filter. The feedback filter is a first order lowpass filter which transfer function is defined in 4.43. the parameter d determines the cut-off frequency of this filter.

$$FFCF_N^{f,d}(z) = \frac{1}{1 - H_{fb}(z)z^{-N}} \quad (4.42)$$

$$H_{fb}(z) = f \cdot \frac{1-d}{1-d \cdot z^{-1}} \quad (4.43)$$

The system function for the applied allpass filter is defined in 4.44.

$$AP_M^g(z) = \frac{z^{-M} - g}{1 - g \cdot z^{-M}} \quad (4.44)$$

Similar to the earlier discussed modeling of the virtual analog models, the system functions are transformed to the corresponding difference equations and from

FIGURE 4.22 – FreeVerb block diagram

FIGURE 4.23 – WaveCore representation of FFCF (left) and AP (right)

there mapped on a WaveCore WPP. The difference equations for the FFCF and allpass filter are defined in (4.45) and (4.46). The corresponding WPP graphs for the applied filters are depicted in fig. 4.23.

$$y[n] = x[n] - d \cdot x[n-1] + d \cdot y[n-1] + f \cdot (1-d) \cdot y[n-N] \quad (4.45)$$

$$y[n] = x[n-M] - g \cdot x[n] + g \cdot y[n-M] \quad (4.46)$$

The resulting WPP can be directly derived from the FreeVerb block diagram in fig. 4.22 and the difference equations. The parameters f and d are combined in one control token, and the model contains input $x[n]$ and output $y[n]$. The parameter f simulates the size of the modeled space. The parameter d models the reverberation time.

The compiler results for the FreeVerb algorithm is outlined in table 4.4. As can be seen, the FreeVerb algorithm is also a lightweight algorithm which easily fits

TABLE 4.4 – Compiled FreeVerb

Compile results of algorithm: FreeVerb					
GIU utilization		Resource load		Mapping metrics	
#instructions	327	Imem	452	Required PUs	1
Data-hazards	56.9%	Xmem	63	Total #PAs	120
Thread-sync	0.0%	Ymem	64	Total #DAGs	24
Pntr updates	0.0%	Cmem	70	Total #delay-lines	12
LSU utilization		Cycle-budget	18.2%	Longest path	22
Instructions	125	L2 GPN-load	0.0%	Firing rate (kHz)	48
DMA transactions	30	L3 bus load	1.7%	L3 mem (Bytes)	50396

within a single WaveCore PU. The high number of data hazards are mainly caused by one long combinatorial path through the entire structure from control-input to output $y[n]$.⁶ The majority of all combinatorial PAs within the entire algorithm implementation are packed within a single DAG. The latency of the algorithm implementation (i.e. the WPP) is zero fire-rate periods, because of the combinatorial path from input $x[n]$ to output $y[n]$.

4.4 FINITE-DIFFERENCE TIME DOMAIN MODELING

Finite Difference Time Domain (FDTD) modeling is a grid-based numerical approach to solve wave equations. FDTD modeling is widely applied in many application domains. One example is numerically solving Maxwell's equations, for the purpose of antenna design. The application of FDTD modeling within the acoustical domain is an emerging topic. FDTD Modeling of musical instruments and reverberation have much focus in many research studies. A historical overview and elaborate explanation of FDTD techniques within this domain can be found in [85], [8], and [109].

Mapping of real-time FDTD based physical models on processors is challenging because of the huge computational demand. Despite the impressive evolution of general purpose processors which are multi-core nowadays and widely applied in mainstream computer platforms, these processors can still not meet the computational requirements for solving FDTD models within reasonable timespans, let alone real-time. As a result, the focus is on different processor technologies, like General-Purpose computing on Graphics Processing Units (GPGPU) [10] and FPGA. Solutions based on GPGPU can achieve impressive acceleration and offer flexibility because of the abstraction of the associated programming model. Feasibility of real-time simulation of physical models on GPGPU has been investigated by Hsu et. al. [31]. They have been able to map square shaped 2D membrane

⁶ Pipelining of the FreeVerb graph, through the insertion of unity delays at some points in the graph is a method to partition the main DAG. We found that the percentage of data-hazards can be reduced with approximately 45%.

structures with grid sizes up to 81×81 points onto a GPGPU at 41.1 kHz real-time, and compared this against GPP performance. This, however, is not sufficient to simulate a complete instrument model in real-time.

FPGAs enable parallel computing to the extreme and can potentially meet the real-time computational requirements. In principle an FPGA is a fabric of primitive logic gates and memory resources that can be configured (interconnected) in an almost arbitrary way. The difficulty with FPGAs, however, is programmability. In chapter 2 we explained that the typical fire-rate (i.e. sampling frequency) of a physical model is typically three orders of magnitude lower, compared to the feasible clock frequencies within FPGAs. This implies that the implementation of a relatively complex physical model on FPGA requires the utilization of FPGA resources in both space (i.e. allocation of logical resources on FPGA) and time (i.e. temporal reuse of these logical resources). This implies that a feasible implementation of a detailed physical model on FPGA should be a many-core structure, where each core executes a partition of the physical model. Automated generation of such a structure, by means of EDA methodologies, from a high level behavioural description of the physical model is far from trivial. C λ aSH [40] is one example of an EDA methodology which aims at automated generation of a HDL description from a Haskell based mathematical description of the algorithm. A case study of the cochlea model [97] demonstrates that a mathematical description of a finite-element based model of the cochlea can be automatically translated to a HDL description, using C λ aSH. This HDL description, in turn, can be automatically mapped on FPGA. This study, however, also shows that the generated HDL model predominantly utilizes the "space" dimension of the FPGA (i.e. relatively little "folding"). Motuk et. al. [18],[19] designed FPGA-targeted HDL models for solving plate equations and obtained impressive efficiency results with this. Using these HDL models, however, makes it still not trivial to compose a complete instrument model. Pfeifle et. al. have to our knowledge been the first to design FPGA targeted HDL models that implement complete instrument models, based on the composition of different FDTD models [64].

Our conclusion is that FPGA technology has the ability to implement computational intensive real-time physical models of musical instruments, but the problem is to make this technology sufficiently flexible (i.e. to design HDL models which enable efficient and versatile model development) and accessible (i.e. without the steep learning curve to apply this technology).

In this section we will investigate how FDTD geometries can be mapped on the WaveCore processor. Furthermore we will show how different geometrical models can be combined in a WPG and as such show how a virtual instrument model can be composed.

Mathematical background

We focus on finite difference schemes that originate from the generic wave equation as defined in (4.47). Such schemes are elaborately explained in [8]. This equation models wave propagation in a medium, like a string (1-dimensional), plate (2-dimensional) or a 3D medium (acoustical space).

$$\lambda(s) \cdot \nabla_s^k u(s, t) = \epsilon(s) \frac{\partial^2 u(s, t)}{\partial t^2} + \mu(s) \frac{\partial u(s, t)}{\partial t} + f(s, t) \quad (4.47)$$

$$\nabla_s^k u(s, t) = \frac{\partial^k u(s, t)}{\partial x^k} + \frac{\partial^k u(s, t)}{\partial y^k} + \frac{\partial^k u(s, t)}{\partial z^k} \text{ with } s = (x, y, z) \quad (4.48)$$

Where s represents the spatial coordinates (i.e. $\langle x \rangle$ for a 1-dimensional geometry or $\langle x, y \rangle$ for a 2-dimensional geometry). The parameter k represents the order of the differential equation. The first-order partial time derivative term models wave propagation losses. The term $f(s, t)$ enables model stimulation (e.g. plucking a string or striking a snare-drum). In our analysis we will investigate 2nd and 4th partial differential equation (PDE) orders. We allow the parameters λ , ϵ and μ to be dependent on the spatial coordinates in our analysis. However, we assume these parameters to be time invariant. Physically this means that non-uniformity of the medium can be modeled. The first step is to transform the space/time continuous eq.(4.47) into its space/time discrete counterpart. Therefore we introduce a grid in space: $s = \sigma \Delta s$, and time: $t = n \Delta t$. Furthermore we apply a central difference approximation which yields the substitutions for the first and second partial time derivative in (4.49) and (4.50). For a thorough background on finite elements we refer to [5].

$$\frac{\partial u}{\partial t} \approx \frac{u_\sigma^{n-1} + u_\sigma^{n+1}}{2\Delta t} \quad (4.49)$$

$$\frac{\partial^2 u}{\partial t^2} \approx \frac{u_\sigma^{n-1} - 2u_\sigma^n + u_\sigma^{n+1}}{\Delta t^2} \quad (4.50)$$

We substitute (4.49) and (4.50) into (4.47) and subsequently solve for u_σ^{n+1} . This yields the iteration function, as defined in (4.51).⁷

$$u_\sigma^{n+1} = s_1(\sigma) \xi^n(\sigma) + p_1(\sigma) u_\sigma^n + p_2(\sigma) u_\sigma^{n-1} + p_3(\sigma) f_\sigma^n \quad (4.51)$$

Where we define $\xi^n(\sigma)$ as the spatial stencil computation function. This function accumulates the neighbourhood node values, including center node u_σ^n with discrete timestamp n . The definition of this spatial stencil depends on the dimensions of the models and the order of the differential equation. The general definition is given in equation (4.52).

$$\xi^n(\sigma) = \sum_k \alpha_k \cdot u_{\sigma_k}^n \quad (4.52)$$

⁷Within this subsection we use the discrete-time variable n notation as f^n , instead of the notation $f[n]$ which we used elsewhere in this thesis.

FIGURE 4.24 – 5-points stencil within membrane model

For a 2D geometry with 2nd order (membrane) we can expand this stencil computation function as expressed in (4.53).

$$\xi^n(i, j) = u_{i-1,j}^n + u_{i+1,j}^n - 4u_{i,j}^n + u_{i,j-1}^n + u_{i,j+1}^n \quad (4.53)$$

An illustration of a 2D node scheme with the stencil defined in (4.53) is given in fig. 4.24. Each dot in this scheme represents an instance of the iteration function 4.51. In particular, node $u_{i,j}^n$ in fig. 4.24 accumulates the black-coloured neighbour node values, according to the membrane stencil computation function in (4.53), and computes the next-timestamp value according to (4.51).

4.4.2 CAPTURING FINITE-DIFFERENCE SCHEMES INTO WAVECORE PROCESSES

A finite-difference scheme as in fig. 4.24, with node-computation defined in eq. (4.51) can be seen as a directed graph. The basic elements in such a graph are vertices (the nodes wherein the actual computation takes place) and edges. Data elements (i.e. tokens) are carried over the edges and consumed/produced by the vertices. A WPP can also be seen as a directed graph, where the vertices are represented as Primitive Actors (PA), which are arbitrarily connected by edges. Hence, a finite-difference scheme can be mapped on a WPP. The graph structure of a FD-scheme is very similar to the associated WPP structure. This is the reason why a WaveCore process transparently represents the mapped FD-scheme. There are basically three main requirements for mapping a finite-difference scheme onto a WaveCore process: (1) scalability, (2) the ability to move stimulus data(f_σ^n) to the model for each node and for each time-stamp n , and (3) the ability to move node-data from the model for each node u_σ^n and for each time-stamp n . A WaveCore PU executes compiled PAs sequentially in time, and the PU clock frequency is limited. This implies that in many cases only a limited part of a FD-scheme (depending on its size) fits on a single PU. Therefore a FD-scheme needs to be partitioned and each

FIGURE 4.25 – Plate computation scheme

FIGURE 4.26 – Plate node, represented as a PA graph

FIGURE 4.27 – WaveCore FD process

partition must be mapped on a PU. Partitioning of the FD-scheme and mapping of partitions on different PUs inherently means that stencil-computation for partition-boundary nodes cannot be done without gathering one or more node-values that belong to the stencil from "neighbour" PU. The GPN-network within the WaveCore PU-cluster is used for this purpose: partition boundary-node values are moved to neighbour partition(s) over this GPN network for every discrete timestamp. We call this phenomenon "radiation" since every partition radiates its boundary values to its direct environment. The radiation intensity depends on the geometry (1D, 2D, 3D) and spatial stencil complexity (order of the PDE). This is shown for a 2D model of a fourth order differential equation (plate equation) in fig.4.25. Radiation of node values is illustrated with arrows (e.g. node $u_{i,j+2}^n$ within partition A radiates to partition B). As can be seen, in the worst-case some nodes need to radiate to three neighbour partitions. The example node at position (i, j) depends on 7 radiated node inputs from partitions B,C and D and 6 local node values. Node $u_{i,j}^n$ itself radiates to partitions B,C and D. Note that "radiation" is an inevitable side-effect of partitioning, which has also been described within the context of physical model mapping on FPGA by Motu et.al [18]. The actual mapping of a FD-node onto a WPP is depicted in fig.4.26. The stencil computation is implemented with a multiply-add type PA tree structure (shown as a "black-box"). The inputs of this part of the WPP are C-type PAs, if applicable (GPN radiation entry-points) and outputs from different local (i.e. within the same partition) nodes. The output of this stencil computation part feeds into the temporal difference equation part. This part consists of four interconnected PAs (labeled 1,2,3,4). As can be seen, there is a transparent relationship between the WPP structure and the implemented difference equation (4.51). The example node (i, j) itself radiates to three neighbour partitions. The PAs 6,7 and 8 are included for this purpose. The execution of each of these PAs (6,7,8) initiate a GPN-write transaction. Moving stimulus data (function f_σ^n) from outside to the model is implemented with C-type PA(5). This PA is linked to the inbound port of the associated WaveCore process. The LSU of the PU

takes care that the required data is fetched from external memory through DMA and moved to the memory location that is allocated to PA(5). Moving node data from the model to outside is implemented with PA(9). This PA is linked to the out-bound port of the associated WaveCore process. The LSU of the PU takes care that the PA(9) output data is moved to, and the input to PA(5) is moved from external memory space via DMA. The abstract representation of a complete FD-scheme as a WaveCore process (including stimulation and response emission) is depicted in fig. 4.27.

4.5 COMPOSITION OF PHYSICAL GEOMETRY MODELS

The ability to capture finite-difference models in a scalable processor technology is an important requirement to build musical instrument models. This requirement on itself, however, is not sufficient. Other important requirements are (1) the ability to combine different geometry models where each model may run at a different sampling-rate, (2) the ability to move stimulus and response streaming data between the model and the external (e.g. analog or an externally connected computer that generates stimuli and post-processes generated model data), (3) real-time processing with low latency, (4) the ability to add processes which structures are not as regular as FD-schemes (e.g. traditional signal-processing functions or effects), (4) programming flexibility and processor architecture abstraction. In the following sections we will demonstrate by three experiments how these requirements fit on the WaveCore programming model and processor technology.

4.5.1 EXPERIMENT 1: PRIMITIVE SIX-STRING DEVICE IN AN ACOUSTIC ROOM

We combined a simple guitar model with a virtual guitar player in an acoustic room and described this in a WPG that is depicted in fig.4.28. The virtual guitar player is modeled as a process called "SixStringPlayer". The implementation is a low-frequency oscillator which output is connected to a tapped delay-line. As a result, this process emits a "downstrum" event at the LFO rate (sub-Hz frequency) by periodically generating a "pluck" token. This token consists of six PTs (one PT for each string). The guitar model is described in the "Fdtd_SixString" process. This process is composed of six 1D FD-scheme instances (each one coded as a WPP) that are connected to a "SixStringBody" WPP. The six string models are connected to the "pluck" port where each string receives one out of six PTs from the associated ingoing token. Each string is tuned in a way that the guitar model is tuned in an E-major chord. Each pluck event causes a raised-cosine shaped pluck at a fixed position in the associated string. The process partition called "SixStringBody" combines the outer edges of the six strings, combines these and routes these back to the same strings (simple interference model). Moreover, this "SixStringBody" extracts a stereo audio signal from the combined string vibrations and links this stereo signal to the "SixStringOut" port of the process. The "SixStringOut" port of the "Fdtd_SixString" process is connected to a reverberation process called "FDN_reverb". This process implements a feedback-delay network that models a

FIGURE 4.28 – WaveCore Process Graph of six-string model

TABLE 4.5 – Compiled SixString model

Compile results of algorithm SixString					
GIU utilization		Resource load		Mapping metrics	
#instructions	6854	Imem	7472	Required PUs	6
Data-hazards	3.3%	Xmem	3261	Total #PAs	5415
Thread-sync	11.9%	Ymem	3262	Total #DAGs	1807
Pntr updates	3.8%	Cmem	3489	Total #delay-lines	66
LSU utilization		Cycle-budget	94.2%	Longest path	25
Instructions	618	L2 GPN-load	0.1%	Firing rate (kHz)	48
DMA transactions	150	L3 bus load	8.4%	L3 mem (Bytes)	631024

3D space in an abstract way (zita_rev1, [81]). This is a simple model in terms of number of PAs, but fairly complex in terms of number of instantiated delay-lines (66 in total). This process yields a reverberated stereo output, called "DAC". Finally the "DAC" edge is connected to a process that is called "DAC". This is a foreign process: an abstract model of a DAC. All processes in the graph are synchronized to a single rate called "AudioRate" which is generated by the global scheduler model in the same process graph. The WaveCore compiler translates the entire process graph to a WOF. This WOF can either be mapped to a target WaveCore processor platform or simulated by the WaveCore simulator. For this process graph we did both and noticed no differences between the simulated model and the real-time (48kHz sampling rate) WaveCore processing on the FPGA board (Digilent Atlys [17]) that we used.

The mapping results of the six-string model are summarized in table 4.5. The modest data-hazard percentage shows that this model maps well on the required number of PUs. The thread-sync bubbles in the GIU pipelines are mainly caused

FIGURE 4.29 – WaveCore Process Graph of 2D plate model with stimulus generation/response capturing.

by the differences between the lengths of the different threads that are mapped on the different PUs in the WaveCore processor. Note that the GPN bus load is only a fraction of the capacity of the bus. The GPN traffic is caused by the interaction between the FDTD models of the 6 strings, and the guitar body model at which the strings are "attached". The L3 memory load is modest: the majority of this load is caused by the delay-lines within the reverberation model.

4.5.2 EXPERIMENT 2: SCALABLE PLATE MODEL

We implemented a 2D geometry model generator with 2nd and 4th order differential equations, based on equation (4.51). This generator produces a scalable and partitioned WPG which is depicted in fig. 4.29. The purpose of this process graph is to demonstrate the scalability of FD-schemes as partitioned WaveCore processes. Furthermore, we analyzed the mapping characteristics using this model. We applied square-shaped partitioning where we tried to optimize the partition size in such a way that a partition just fits on a targeted PU. The "PlateIO" process enables verification of the model: this process injects stimuli into the model and captures emitted responses. We used the WaveCore simulator to verify the functional correctness of the generated model. A simulation snapshot is depicted in fig.4.30 We observed that a FD-scheme is uniformly scalable. The WaveCore compiler turned out to be robust and fast: it took less than a minute to compile an FD-model with over 8000 nodes to a WOF.

FIGURE 4.30 – WaveCore simulation snapshot of 2D plate model.

The compile results of the scalable 2D geometry model are summarized in table 4.6. The results in this table refer to a rectangular 2D model (membrane, consisting of 8112 nodes) which consists of 48 partitions (configured in a 8x6 partition grid), where each partition consists of a grid of 13x13 nodes. The modest data-hazard bubble percentage reveals that the membrane model maps well on the GIU pipelines, because of the relatively shallow combinatorial structures of the DAGs. This implies that there is much inherent parallelism. The thread-sync bubble percentage is low as well, which implies that the compiler has been able to schedule the GPN transactions in an efficient way. The GPN load reveals the level of radiation (i.e. remote-state exchange as a side effect of scheme partitioning), and as can be seen this can be handled quite easily by the GPN bus. Note that the L3 bus is heavily overloaded. This is mainly because every node in the structure emits to "resp" outgoing token of the "Fddt_2D" process. This implies that either the "resp" token size needs to be reduced, and/or the L3 memory system needs to be enhanced (increased parallelism) in order to enable physical execution of the compiled model on a WaveCore processor with the required 48 PUs.

4.5.3 EXPERIMENT 3: REAL-TIME MEMBRANE MODEL ON FPGA

We generated a 2D membrane model (2nd order PDE) and integrated this as a partitioned WaveCore process in the WPG which is depicted in fig. 4.31. Within this process graph we connected the analog stereo input of the FPGA to two opposite diagonal corners of the 2D node-grid. Likewise, we connected the other two diagonal corner node outputs of the 2D node-grid to the stereo output of the FPGA board. These analog I/O connections are modeled in the WPG by the ADC and DAC processes. This model runs in real-time at 48kHz audio sampling rate and

TABLE 4.6 – Compiled rectangular membrane model

Compile results of algorithm: rectangular membrane model					
GIU utilization		Resource load		Mapping metrics	
#instructions	66421	Imem	91387	Required PUs	48
Data-hazards	0.4%	Xmem	36553	Total #PAs	63392
Thread-sync	1.8%	Ymem	36542	Total #DAGs	26108
Pntr updates	0.9%	Cmem	61236	Total #delay-lines	0
LSU utilization		Cycle-budget	78.0%	Longest path	5
Instructions	24966	L2 GPN-load	3.2%	Firing rate (kHz)	48
DMA transactions	8194	L3 bus load	457.3%	L3 mem (Bytes)	65552

FIGURE 4.31 – WaveCore Process Graph of a membrane model on FPGA, connected to ADC/DAC.

ultra low latency where we connect an audio source to the analog input of the board and a speaker to the analog output of the board. We also simulated this process graph using the WaveCore simulator. We used an audio sound bite that we linked to the ADC process, and played the produced (by the DAC process) audio with a media player. We observed a similar audible result, compared with the FPGA setup.⁸

Table 4.7 summarizes the compile results of the membrane model on the Atlys FPGA board. The compiled membrane model consists of 6 partitions, configured in a 2x3 topology. Each partition consists of 12x12 nodes. The table shows that the membrane model is compiled efficiently to the WaveCore processor.

⁸Based on the FPGA platform it is difficult to compare the response with the output data which is obtained with simulation, because of the inherent noise that is produced by the ADC. This noise causes minor vibrations in the membrane model, which cannot be exactly reproduced in simulation

TABLE 4.7 – Compiled rectangular membrane on Atlys

Compile results of algorithm: membrane on Atlys					
GIU utilization		Resource load		Mapping metrics	
#instructions	6772	Imem	6806	Required PUs	6
Data-hazards	1.0%	Xmem	4002	Total #PAs	6400
Thread-sync	0.6%	Ymem	4005	Total #DAGs	1902
Pntr updates	2.2%	Cmem	6224	Total #delay-lines	0
LSU utilization		Cycle-budget	63.8%	Longest path	5
Instructions	34	L2 GPN-load	2.5%	Firing rate (kHz)	48
DMA transactions	4	L3 bus load	0.2%	L3 mem (Bytes)	32

4.6 DISCUSSION

We investigated the target application domain through the analysis and mapping of application examples on the WaveCore processor. Our aim is to analyze the mapping characteristics w.r.t. different views, like mapping efficiency, programming transparency, scalability and finding the limitations of the WaveCore processor technology.

4.6.1 PROGRAMMING MODEL

The application examples show that the declarative programming model matches quite close to the mathematical formulation of the mapped application problems. We found a good match between the mathematical formulation of discretized problems and the associated SDF model on which the WaveCore programming model is based. Likewise, we found that the delay-line as a unique fundamental programming element has shown to be a construct that can be applied transparent and efficient, specifically within the domain of digital-waveguide modeling.

We showed that a WaveCore program actually is a description of a static and primitive data-flow graph (i.e. the declaration of interconnected computational primitives). In chapter 5 we will show that this implies a natural match with higher abstraction level functional programming.

We did not run into the limitations of SDF with the example applications. However, there are areas within the application domain which would have been interesting to analyze. One area is modeling of distortion effects [104] [106] which is of significant relevance for musical applications. A key mathematical structure within this domain is non-linear implicit differential equations [104]. Numerical solvers for this type of equations requires an iterative approach, which is steered by error minimization. These kind of solvers do not fit naturally in a SDF approach, since the number of iterations within these solvers are data dependent and hence unknown at compile-time. This implies that the resulting data-flow graph cannot be expressed statically unless a worst-case data pattern is assumed at compile-time.

A similar limitation of the SDF based programming model that we touched briefly in the LFO example is control-structures. We showed that an imperative "if-then-else" statement requires an unfolded multiplexer structure where all branches of the "code" need to be evaluated explicitly.

This weakness of the WaveCore processor (lack of efficient control processing) calls for heterogeneity of the processor system, where the cooperation between a GPP (interaction models) and WaveCore processor (physical models) is required. We will further elaborate on a heterogeneously mapped application on such a system in chapter 5.

4.6.2 APPLICATION SPECTRUM AND PROCESSOR SCALABILITY

The probed application space reveals that there is a significant difference between the various applications with respect to required computational capacity. We observed that the virtual analog models that we derived and mapped on WaveCore are lightweight: all of these only use one or a fraction of a single WaveCore PU, with a modest memory requirement. At the other side of the spectrum we find detailed physical modeling, which requires massive computational capacity and a far more demanding memory requirement (specifically when large tokens are necessary for detailed model stimulation or composition of multi-geometry models with different fire rates). This side of the spectrum requires a scalable processor concept, such as the WaveCore. In the middle-area of the spectrum we find for instance multi-effects processors (e.g. combination of different effects like reverberator, modulation, and distortion), synthesizer models (i.e. additive sound synthesis which requires a number of discrete-time oscillators, LFOs to modulate properties of additional effects), or physically-informed reverberation models. Typically, these applications require a relatively small PU cluster with a modest memory requirement.

The probed application examples have shown that the WaveCore processor concept is well scalable, and can be optimized to a certain application. For instance, a single-PU WaveCore processor may be applied as a reprogrammable guitar effects processing unit, which enables the most popular single-effects (e.g. Phaser, Flanger or WahWah). Likewise, a large PU cluster WaveCore processor instance may be optimized as a real-time physical modeling engine, which serves musical instrument development.

Apart from the investigation on how well algorithms from the target application domain map on the WaveCore processor, it is important to gather an understanding on how the processor technology relates to other, existing processor concepts. In chapter 6 we will elaborate on the scalability of the processor and the performance of the WaveCore processor, compared to other existing technologies. Likewise, it is interesting to elaborate on the compatibility of WaveSlang, related to existing

programming languages. We will investigate this aspect in chapter 5.

HETEROGENEOUS CODE GENERATION FOR A GRAPHIC EQUALIZER

ABSTRACT – *This chapter describes the implementation of a recently introduced Parallel Graphic Equalizer (PGE) in a heterogeneous way. The control and audio signal processing parts of the PGE are distributed to a PC (interaction model) and to the WaveCore (physical model), respectively. This arrangement is particularly suited to the algorithm in question, benefiting from the low-latency characteristics of the audio signal processor as well as general purpose computing power for the more demanding filter coefficient computation. The design is achieved in a high-level language called Kronos, which we have adapted for the purposes of heterogeneous code generation from one single program source¹.*

5.1 INTRODUCTION

A graphic equalizer is a signal processor with a variety of purposes, ranging from tonal correction of rooms and transducers to sound design and mastering [12]. One of the key benefits of the design is the immediate correspondence between the user interface and the frequency response. The quality and precision of this correspondence is key to a successful graphic equalizer design. In addition, many of its applications require low-latency processing—such as tonal correction of a loudspeaker system in a live situation. In this chapter, we present a recent parallel graphic equalizer design [71] and show how to adapt it to a heterogeneous system of a PC and a WaveCore signal processor, utilizing a high-level programming language called Kronos. Kronos and WaveCore share a declarative programming methodology, facilitating code generation. In addition, the ability of Kronos to automatically separate the signal-processing part (i.e. the physical model) from the user interface part (i.e. the interaction model) of a functionally coded algorithm is

¹The majority of this chapter has been published in [MJW:3]

FIGURE 5.1 – General structure of a digital Graphic Equalizer application

directly applicable. As such, heterogeneous code can be generated automatically.

This chapter is organized as follows. In Section 5.2, *Background*, we discuss the history and state of art of graphic equalization, and our equalization algorithm. The high-level programming method, *Kronos*, is presented in section 5.3. Section 5.4 details the results of the work, discussing the aspects specific to heterogeneous computing and the adaptation of the equalizer algorithm to the *WaveCore*, as well as avenues for future research. Section 5.5 concludes this chapter. The work which we present in this chapter is published in [MJW:3].

5.2 BACKGROUND

5.2.1 GRAPHIC EQUALIZATION

There are two types of equalizers, parametric and graphic ones [12]. In a parametric equalizer the user has access to the gain, center frequency, and bandwidth of an equalization filter. A graphic equalizer can be composed of several parametric equalizers or other similar filters, but their center frequencies and bandwidths are fixed. In the graphic equalizer, the user only controls each band gain by using a set of sliders, which form approximately the desired magnitude response [12, 30, 70, 71]. The general structure of a graphic equalizer application is depicted in fig. 5.1. It consists of a user-interface, consisting of a set of sliders, where each slider represents the gain of the associated frequency-band. The band-gains are usually called ‘command-gains’. The command-gain settings are translated to filter-coefficients by the *Coefficient Computation* function. Coefficient computation is an event-driven process (i.e. an interaction process): only when the user modifies slider positions, the process is called. The computed coefficients are passed to the real-time part

FIGURE 5.2 – Structure of the parallel filter (a) and second-order filter $H_k(z)$ (b)

of the algorithm, being the *Filter-bank*. This filter-bank (i.e. a physical model) is synchronous to a sampling frequency f_s , and processes the input signal $x[n]$.

The main challenge in designing a graphic equalizer is the interaction of neighbouring band filters. Problems arise when two adjacent command gains have a large difference. In this work we implement one of the recent graphic equalizers, which solves the interaction problem by optimizing the parameters with a least-squares technique [71]. For processing the input signal, this graphic equalizer uses a parallel IIR filter structure, which is well suited to parallel computing [7].

5.2.2 SPECIFIC GRAPHIC EQ ALGORITHM

In this work we implement a recently developed parallel graphic equalizer [71]. The filter itself is a parallel IIR structure, which is depicted in fig. 5.2 (a). Each filter block (i.e. $H_k(z)$) has a special second-order transfer function: a second-order denominator transfer function but a first-order numerator transfer function. The structure of this filter block is also depicted in fig. 5.2 (b). Additionally, there is a direct path with a real weight (i.e. d_0) from the input to the output. The overall transfer function of our graphic equalizer is defined in eq.5.1. In our application we configured the equalizer with 62 filter-blocks (i.e. $K = 62$).

$$H(z) = d_0 + \sum_{k=1}^K \frac{b_{k,0} + b_{k,1} \cdot z^{-1}}{1 + a_{k,1} \cdot z^{-1} + a_{k,2} \cdot z^{-2}} \quad (5.1)$$

The poles of the graphic equalizer are set in advance at pre-designed frequencies determined by the frequency resolution of the graphic equalizer. For our example with $K = 62$, when a third-octave graphic equalizer is designed, the poles go at 31 standard frequencies² between 20 Hz and 20 kHz (20 Hz, 25 Hz, 31.5 Hz, 40 Hz etc.). To obtain high accuracy, additional poles are assigned at 10 Hz and between each

²Command-gain frequencies are standardized in audio applications

```

1  Filter(x0 pole) {
2      y1 = z-1( (pole - pole) y0 )
3      ; y1 is initially zero, subsequently delayed y0.
4      ; The initial value of zero is expressed as 'pole-pole' to ensure
5      ; that the feedback path type matches the pole type.
6
7      y0 = x0 - pole * y1
8      ; Compute y0, the output.
9
10     Filter = y0
11     ; This is the filter output.
12 }

```

LISTING 5.1 – One-pole filter, coded in one Kronos function

standard center frequency, so that there will be altogether 62 poles (i.e. $K = 62$). The pole radii has chosen so that the magnitude responses associated with neighbouring poles meet at their -3 dB points. All of this is done off-line before running the filter within the *Coefficient Computation* function. The mathematics related to this design are detailed in [71]. During real-time operation, the user can adjust the command gains of the graphic EQ. To achieve a great accuracy, this EQ design uses least-squares optimization to adjust the numerator (feedforward) coefficients (i.e. the b_k coefficients), two per pole. This is similar to FIR filter design and only requires a matrix operation. However, a specific weighting function is needed to ensure that attenuation is implemented correctly. As a result, every time a command gain is changed, the matrix inversion needs to be executed and all feedforward coefficients of the graphic EQ updated [71], and subsequently these coefficients need to be updated to the *Filter-bank* process.

5.3 KRONOS – A HIGH LEVEL MUSIC DSP LANGUAGE

FIGURE 5.3 – Schematic of the one-pole filter example

Kronos [55] [56] is a high level functional signal processing language designed for musical applications. Typical features, which makes the language domain-specific, are:

- » Constructs which support signal-delay, such as unity delay but also circular buffering (e.g. for describing delay-lines).

```

1 ; Routes a function output back to its first argument through a unity delay.
2 Recursive(sig binary-func) {
3     state = binary-func(z-l(sig - sig state) sig)
4     Recursive = state
5 }
6
7 Filter2(sig pole) {
8     ; Lambda arrow '=>' constructs an anonymous function: the arguments
9     ; are on the left hand side, and the body is on the right hand side.
10    dif-eq = (y1 x0) => x0 - pole * y1
11    ; onepole filter is a recursive composition of a simple
12    ; multiply-add expression
13    Filter2 = Recursive( sig dif-eq )
14 }

```

LISTING 5.2 – One-pole filter code, using higher-order function, polymorphism and recursion

```

1 Main() {
2     sr = #48000
3     pole = Control:Param("pole" 0.5)
4     (l r) = Audio:Input(0 0)
5     Main = Filter(l + r pole)
6 }

```

LISTING 5.3 – Main function, with event-driven control and instantiation of signal-path

- » Predefined domain specific functions, like audio input and output functions.
- » User interface functions

The language is based on pure functional semantics[33]. Properties like polymorphism, recursion and higher-order functions are supported in the language. For a more detailed description of the language we refer to [54]. Next to the language, the developer can also use a graphical programming interface. This interface allows the programmer to compose an application in a graphical (i.e. block-diagram) way.

We give an example of a simple application, described in the Kronos language, which includes many of the basic ingredients of the language. This application consists of a simple signal processor, the one-pole filter which block-diagram is depicted in fig. 5.3. The filter has a run-time controllable parameter, called *pole*, and is embedded in a real-time streaming topology. This application has the structure like depicted in fig. 4.1 in chapter 4, and hence contains all basic ingredients: physical model (i.e. the filter), interaction model (i.e. user interaction process, which controls the *pole* parameter), and real-time input and output.

The "main" function of the example application is given in the listing 5.3. A sampling rate *sr* is set to 48 kHz. The variable *pole* is returned from a function, called *Param*, from a standard function package called *Control*. This function embodies the in-

teraction model, where the user can modify the parameter *pole* during run-time of the application. The tuple (lr) is returned from the function *Input* within the standard function package *Audio*. This function embodies the connection between real-time stereo audio input and the mentioned tuple. This is similar to a *foreign* WaveCore process. The function called *Filter* embodies the one-pole filter, and can be compared with a WaveCore process (i.e. a physical model). This function has two arguments: $l + r$ is the signal input to the filter (within the function call, the left and right audio channels are added, resulting in a mono-input to the filter), and the parameter *pole*.

The implementation of the *Filter* function is given in the listings 5.1 and 5.2, where two alternative ways of coding are displayed. Both listings are taken from the journal paper [56].

The implementation in listing 5.1 is straightforward. The instantiation of a unity-delay is the only construct particular to Kronos. This unity delay is instantiated by the function $z^{-1}(A\ B)$, where *A* is the initial value of the unity-delay, and *B* the input for the unity-delay. The type of the arguments *A* and *B* are defined by the context in which the function is called. Hence the construct $(pole - pole)$ instead of 0. The implementation in listing 5.2 displays more advanced language features, like polymorphism, higher-order functions and recursion.

5.3.1 HETEROGENEOUS CODE GENERATION

The objective of the Kronos compiler, in the context of our graphic equalizer application, is to generate heterogeneous code which subsequently can run on both a GPP and the WaveCore simultaneously. The compile flow which performs this code generation is depicted in fig. 5.5. The compile target can be modeled as a WPG. For our graphic equalizer application this WPG is depicted in fig. 5.4. The compile process basically consists of the following three phases.

Phase (1): Dataflow Analysis

The Kronos program is analyzed with respect to data types and dataflow dependencies. During this process, a dataflow graph is composed and transformed to a graph which contains primitive vertices (i.e. Kronos primitives).

Phase (2): Clock-Domain Factorization

During this phase, the dataflow graph is partitioned into clock-domains. Each clock domain is synchronous to a unique sampling rate (similar to firelinks within a WaveCore WPG). Parts of the graph which are event-driven (i.e. interaction process) are also identified during this phase.

Phase (3): Object Generation

For each graph partition (i.e. clock-domain) its allocation (i.e. GPP of WaveCore) is determined. Control processes (i.e. equalizer control) and communication between control process and the physical model (i.e. the *Equalizer Kernel*) are allocated to

FIGURE 5.4 – Mapped equalizer application on heterogeneous ReMap platform

FIGURE 5.5 – Heterogeneous compile flow

the objects *Cntr Code* and *Interf* respectively. These objects are targeted to the GPP. All real-time graph partitions are mapped to either the ADC and DAC processes, or to a WaveCore process (i.e. the *Equalizer Kernel*). These real-time partitions are combined in a full WPG object description.

Despite the fact that both Kronos and WaveSlang have been two independent developments, we discovered significant similarities. The concept of a hybrid and static dataflow graph model, with event-driven part for interaction modeling and a synchronous dataflow part for physical modeling, is applied to both Kronos and WaveSlang. We also found a significant similarity between WaveCore Primitive Actors (i.e. PAs) and Kronos primitives. Therefore, the generation of a WaveCore process could be largely based on a simple idiom translation of Kronos primitives to WaveCore PAs. During this research we observed that most Kronos programs dealing with single precision floating point DSP can be mapped to the WaveCore, including the equalizer design discussed in this chapter.

The generated objects are mapped onto the target processors. This implies that the generated control object, and the object which implements the communication between the interaction model and the physical model (i.e. passing computed parameters *param* to the physical model) run on the PC. The WaveCore object (i.e. the generated WPG description) is translated to an executable (i.e. a WOF) by the WaveSlang compiler, and subsequently loaded to the WaveCore. Each time that the user changes slider settings of the example equalizer application, the control object translates these settings to filter coefficients, and subsequently sends these coefficients to the WaveCore board at run-time.

5.4 DISCUSSION

5.4.1 EQUALIZER RESPONSE

The presented equalizer algorithm offers a highly precise frequency response. Three command gain settings were used to find the maximum error in the actual magnitude response of the equalizer in comparison to the command gains. Table 5.1 lists the results of our algorithm (PGE) compared to the second order Regalia-Mitra EQ (MR) [73] and a higher order design (EQ4) [30]. The *All up* settings features all the command gains set to +12 dB. In *Zigzag*, the command gains alternate between +12 dB and 0 dB. The final setting of *Every 3rd up* features one band at +12 dB followed by two bands at 0 dB. The proposed algorithm features the smallest error maxima, staying within a decibel of the target curve on all settings. The target curve is defined by the settings of the command gains. The deviation between the target curve and the measured curve is the error. For a more detailed evaluation and comparison of the PGE algorithm, the reader is referred to [71].

TABLE 5.1 – Equalizer error maxima, decibels

Setting	RM	EQ4	PGE
All up	9.8	3.6	0.00
Zigzag	6.3	2.8	0.75
Every 3rd up	4.1	1.5	0.32

TABLE 5.2 – Equalizer performance summary

Equalizer Kernel per channel @ 44.1kHz	PAs	of PU	% of chip
Equalizer Control	avg	max	min
time per update	32ms	51ms	31ms

5.4.2 PERFORMANCE

As the described implementation is heterogeneous, we make a distinction between performance characteristics for the physical model on the WaveCore and the interaction model which runs in real-time on the PC.

For the physical model on the WaveCore, we focus on utilization of the processor and real-time constraints (i.e. latency and time predictability of execution). On the PC side, the computational complexity manifests as control latency – the delay between user interaction and the corresponding change in the equalizer response. This latency is dominated by the time required for coefficient computation. Table 5.2 summarizes the central performance characteristics of our solution. Our WaveCore chip is a cluster of six Processing Units running at 86MHz, achievable on the Digilent Atlys board with a Xilinx Spartan6 FPGA, as shown in fig. 3.35 in section 3.5. The equalizer control timings were measured on a Windows PC with an Intel Core i7 CPU running at 2.8GHz. Minimum, maximum and average control latency were collected from 1000 simulated coefficient updates. As shown in table 5.3, one audio channel can be equalized by one PU in hard real-time. This results in a throughput latency of two sample periods, excluding the latency of A/D/A conversion. Our WaveCore cluster is computationally capable of 6 channels of real time equalization, although the Atlys board is limited to stereo audio I/O.

Full detail of the mapping of the physical model on the WaveCore is given in table 5.3. An explanation of the table can be found in appendix B.

5.5 CONCLUSIONS

We demonstrated the similarity between the domain specific language Kronos, and WaveSlang by means of a graphic equalizer application. Despite the fact that Kronos and WaveCore have been developed independently, they have a lot in common. The concept of a hybrid and static dataflow graph model, with event-driven part for interaction modeling and a synchronous dataflow part for physical modeling,

TABLE 5.3 – Compiled Graphic EQ of one channel

Compile results of Kronos-generated Graphic EQ					
GIU utilization		Resource load		Mapping metrics	
#instructions	1342	Imem	1728	Required PUs	1
Data-hazards	46.9%	Xmem	364	Total #PAs	668
Thread-sync	0.0%	Ymem	364	Total #DAGs	125
Pntr updates	1.8%	Cmem	545	Total #delay-lines	0
LSU utilization		Cycle-budget	74.9%	Longest path	63
Instructions	386	L2 GPN-load	0.0%	Firing rate (kHz)	48
DMA transactions	125	L3 bus load	7.0%	L3 mem (Bytes)	1000

is applied to both Kronos and WaveSlang. Furthermore, the dataflow primitives in WaveSlang match closely with those in Kronos. This implies that a simple idiom translation from Kronos dataflow primitives to WaveSlang PAs has been sufficient to make Kronos a functional language which can be compiled automatically to our heterogeneous GPP/WaveCore processor platform. The mapped equalizer application runs in real-time with ultra low processing latency, and low latency in the control part (i.e. user interface) which leads to fast responsiveness to user events.

An interesting topic for future work is a further investigation of the automated mapping of more complex applications, like a modular synthesizer.

... and proce
application d-
...ore within p
...ate WaveCo
ologies Eval
...essor tech
in and pro
...ation dom
...within appl ...on domain an
...WaveCore within application do
Evaluate W
ologies Eval
...sor technol
and process
domain enc
plication de
...within app
...WaveCore
...uate Wav
...Evaluate
...ologies Evaluate Wav
......hologies...

EVALUATION

ABSTRACT – We have evaluated the WaveCore processor technology on three aspects. The first aspect involves the chip-area of the processor. We compared the WaveCore processor against two comparable many-core processor technologies, and found that the processor is efficient w.r.t. logic but less efficient w.r.t. level1 memory. The second aspect is execution efficiency. We used a benchmark application set to compare the mapping efficiency between WaveCore and a GPP with DSP extensions (ARM Cortex M4). We found an average of 4x better mapping performance in favour of the WaveCore. Better in this context means that a WaveCore PU requires 4x less clock cycles to execute a benchmark algorithm. Next to this mapping efficiency analysis, we compared the mapping of a detailed physical model (virtual Banjo model) on a dedicated processor on FPGA against an estimated mapping on the WaveCore (including mapping feasibility analysis). The third aspect is the evaluation of the design methodology of WaveCore. It is well possible to automatically generate an optimized WaveCore processor instance, together with compiled WaveSlang code, from an abstract functional description of a ReMap¹ algorithm.²

6.1 INTRODUCTION

Evaluation of a processor technology is based on performance indicators. There are many performance indicators, and the difficulty is that there are always different viewpoints and measurement conditions which can have a big influence on the outcome of the evaluation of the technology. Our evaluation aims to be as complete as possible, in the sense that the most important aspects of the technology are evaluated. In order to accomplish this, we evaluate the WaveCore processor technology on three different aspects.

¹ReMap is an acronym for REal-time Music/Acoustical signal synthesis and Processing
²Parts of this chapter have been published in [MJW:5]

The first aspect is related to physical properties of the WaveCore processor (i.e. the hardware). We compare the estimated chip-area of the WaveCore processor against the chip-area of other DSPs, mainly through an analysis of the required number of logic-gates. Another performance indicator that we use in this context is the relative size of that part of the processor in which the actual computations take place (i.e. the data computation unit (*DCU*), or data-path), compared to the overall size. Next to this area analysis, we investigate how a WaveCore processor instance can be derived (i.e. configured) from application requirements and physical constraints. This aspect is presented in section 6.2.2

The second aspect is related to the execution efficiency of the processor. We define a performance indicator which expresses the efficiency of the programming model, compiler and the ISA trinity. Subsequently we derive this performance indicator for a commercial GPP with DSP extensions, and a single WaveCore PU, using a representative benchmark algorithm set.

Next to this, we compare the mapping efficiency of a detailed physical model (i.e. FDTD based virtual banjo) on a dedicated FDTD processor on FPGA with an estimated mapping of the virtual Banjo on the WaveCore using the same FPGA. This aspect is discussed in section 6.3.3. The third aspect is the design methodology of WaveCore. It is possible to generate an optimized WaveCore processor instance and the instruction/data memory images from a functional code source (e.g. Kronos). We will describe WaveCore as a design methodology in more detail, and compare this to other existing design methodologies. This aspect is presented in section 6.4. We draw some conclusions in section 6.5.

6.2 PHYSICAL ANALYSIS OF THE WAVECORE PROCESSOR

We compare the area efficiency of the WaveCore processor with the area efficiency of two existing domain-specific processor architectures. Next to this we analyze how a WaveCore processor instance can be optimized, and describe its performance, related to space (i.e. the number of PUs) and time (i.e. temporal domain: clock frequency, sampling rate and resulting performance per sampling period).

Note: the problem of comparing chip areas of processors is that in general companies are not willing to provide details on area estimates of processors. Therefore, we compare our results with two processors, of which this area data *is* available. Moreover, area comparison is difficult because this is quite dependent on the used technology. To avoid these problems, we have compared the area through focussing on FPGA mapping results, where we used similar estimation conditions (i.e. FPGA type, synthesis conditions and processor configuration).

6.2.1 AREA EFFICIENCY

The ideal DSP architecture is a conceptual machine which executes a data-flow graph (*DFG*) ultimately efficient. This implies that the ideal DSP does all the re-

quired computations (i.e. the operations which are required by the vertices within the dataflow graph), and the data is assumed to automatically flow through the DFG without any additional effort, while real-time conditions are satisfied. When we apply this concept to a physical DSP, then we observe that a DSP is essentially a data-computation unit, which we call a DCU, plus additional "overhead" to move data from/to the DCU. Following on this observation, we can define the efficiency of a physical DSP through the introduction of the DCU-metric, like defined in (6.1), where we assume that the processor satisfies all computation and timing requirements. With A_{dcu} the area of the DCU and A_{total} the area of all the logic within the processor. A DSP with an DCU-metric of 100% implies that this processor complies to our description of the ideal conceptual machine ³.

$$\eta_{dsp} = \frac{A_{dcu}}{A_{total}} \cdot 100\% \quad (6.1)$$

The exploitation of the locality of reference principle is a key aspect to keep the "overhead" part of the processor as minimal as possible. Locality of reference consists of two aspects: a temporal aspect (i.e. *when* the DCU expects the required operands) and a spatial aspect (i.e. *which* DCUs to use in parallel). A good exploitation of the locality principle results in a high DCU-metric, and hence a relatively low amount of "overhead" logic. Additionally, the DCU should meet all computing and timing requirements. Note that the ideal conceptual machine presumably also is ultimately power efficient: the only energy consumption is within the arithmetic units. Hence, the DCU-metric primarily expresses the area efficiency of a processor, but it also indirectly expresses the power efficiency.

If we apply this idea to the WaveCore processor, we need to retrieve the physical area of the DCU within the GIU, and compare this against the total logic area of a single PU. By doing this we neglect the additional area that is necessary to integrate a number of PUs into a WaveCore processor (i.e. NoC), and the required area for external memory bus infrastructure.

The WaveCore processor is designed as a so-called softcore in VHDL. This softcore is designed in such a way that technology dependencies are almost completely abstracted away (i.e. apart from the specific floating-point arithmetic units, like multiplier and adder, there are no technology references in the code). This implies that this softcore is largely portable among different silicon technologies, like different FPGA families or ASIC technology. Processor "area", in terms of FPGA technology, can be interpreted as the required amount of primitive FPGA resources, like CLB, block-RAMs, or LUTs. We made an analysis of the resource breakdown for a WaveCore PU, which is depicted in fig. 6.1. In this analysis we focussed on a relative breakdown of the utilized logic resources (i.e. number of LUTs) for each processor building block. As can be seen, the majority of the PU resources are

³We implicitly assume here that the compiler and ISA are ideal as well, resulting in an optimally utilized DCU where all parallelism is maximally used

FIGURE 6.1 – WaveCore PU area decomposition

located in the GIU. Focussing on the GIU, we observe that the majority of the resources are located in the DCU. If all the PA-types (see table 3.2) are supported, then a floating-point divider is required for the DIV-type PA. The PU configuration for this analysis, however, does not contain a floating-point divider, and has a capacity (i.e. maximum thread-length in #PAs) $C=2048$. We removed this divider in our analysis, because the processors to which we compare the area also do not contain a divider in their DCUs. The actual DCU-metric of the WaveCore PU depends on whether the DSP48E macro [27] is used in the FPGA mapping of the PU⁴. Without DSP48E macro's the DCU metric for the WaveCore PU is 54% (i.e. 54% of all the utilized CLBs are allocated to the DCU of the PU). With DSP48E, this metric equals 41%.

The Montium Tile Processor

The Montium TP, which we introduced in the introduction chapter of this thesis [29], fits in a Coarse-Grained Reconfigurable Architecture (CGRA) concept. This processor concept is largely based on data-flow processing concepts, and exploits the locality of reference principle. From these viewpoints, the Montium processor tile is interesting to compare with a WaveCore PU because WaveCore is also a domain specific data-flow oriented processor. The area breakdown (130nm ASIC technology, area figures before place and route) of the Montium tile-processor is published in [29]. The relative area of the ALUs, with respect to the rest of the logic area (memories excluded), can be directly derived from these figures, and equals 24% (i.e. the DCU metric). The same publication also includes the results of an FPGA mapping experiment of a Montium TP, which we have used to compare the resource utilization with a WaveCore PU. The target device is a Xilinx Virtex2 device (XCV2000E). This device contains an array of 80x120 CLBs. The CLB topol-

⁴DSP48E macros are generic arithmetic hardmacro's which are available in most of the modern Xilinx FPGAs as primitive blocks. The DSP48E can be inserted within the floating-point units which are available as Xilinx IP blocks

FIGURE 6.2 – Virtex2 CLB topology

TABLE 6.1 – Processor area comparison between WaveCore and Recore Montium

	Montium	WaveCore	Ratio
#CLBs	9408	5375	57%
Memory (bits)	92160	122880	133%
DCU Metric	24%	54%	

ogy of this device is outlined in fig. 6.2. The Montium TP contains 5 independent ALUs, where each ALU is capable to compute a multiply-add operation in one clock cycle. Similar parallelism can be achieved with a WaveCore processor, consisting of 5 PUs. Therefore, for the comparison experiment we configured a WaveCore processor which consists of 5 PUs, we de-configured the floating-point divider, and configured a capacity $C=256$ (similar to the sequencer capacity in the Montium). We configured the floating point units within the WaveCore PU in such a way that they are fully implemented in CLBs, because we found that the Montium mapping results did not include the multiplier resources which are available in the Virtex2 technology. The mapping results are summarized in table 6.1. From table 6.1 we can conclude that the WaveCore processor requires 43% less logic, compared to the Montium TP. We also conclude that the DCU metric for the WaveCore is approximately a factor two better compared to the Montium TP. However, the Montium TP appears to be very efficient w.r.t. memory usage, compared to the WaveCore.

The Xentium VLIW processor

The Xentium processor is a VLIW DSP, developed by Recore Systems. Like the Montium TP, this processor is optimized to be integrated in a many-core topology [72]. Hence, the Xentium processor is interesting for comparison with the WaveCore processor. Area figures for the Xentium processor are published in [100]. Similar to the Montium TP, we find silicon area figures as well as FPGA mapping figures in this publication. We gather the DCU metric for the Xentium processor by dividing the explicitly published data-path area figure by the "logic-only" silicon area. This yields an indication of the DCU-metric, being 25%. The same publication also

FIGURE 6.3 – Virtex5 CLB topology

TABLE 6.2 – Processor area comparison between WaveCore (4 PU-cluster) and Recore Xentium

	Xentium	WaveCore	Ratio
CLB slices	6166	4231	69%
FlipFlops	4145	6524	157%
Block RAMs	13	24	185%
DSP48Es	8	24	300%
DCU Metric	25%	41%	

includes the results of an FPGA mapping experiment of a Xentium tile processor, which we have used to compare the resource utilization with a WaveCore PU. The target device is a Xilinx Virtex5 device (xc5vlx330t). This device contains an array of 240x108 CLBs. The CLB topology of this device is outlined in fig. 6.3. The parallelism within the Xentium datapath offers a peak performance of 4 multiply-add operations per clock cycle [72]. The equivalent parallelism for a WaveCore processor implies that 4 PUs are required (1 floating-point multiply-add per clock cycle, per PU, by means of the MAD-type PA). Hence, for this comparison experiment we have configured a WaveCore processor with four PUs, which matches the Xentium processor as close as possible. Furthermore, we de-configured the floating-point divider, and configured the PU with a capacity $C=1024$. We configured the floating point units within the WaveCore PU in such a way that they use DSP48E macro's, similar to the Xentium FPGA mapping experiment in [100]. The mapping results are summarized in table 6.2. The table shows that the WaveCore processor utilizes approximately 30% less logic resources, compared to the Xentium tile processor. Furthermore, we see (like we also saw in the Montium comparison) that WaveCore is relatively expensive in memory usage: 85% more memory, compared to the Xentium configuration (the Xentium instance has an instruction memory capacity of 8kByte and 16kByte data memory). The table also shows that the Xentium mapping uses relatively few DSP48E resources. The reason for this is, according to [100], that the Xentium HDL code is optimized to ASIC technology, and hence the FPGA

synthesis tool is not well capable in the detection of arithmetic structures in which the DSP48E macro can be inserted.

6.2.2 PROCESSOR INSTANCE OPTIMIZATION

165

As mentioned, the WaveCore processor is implemented as a softcore and described in a HDL. This softcore can be either mapped on a configurable fabric device (FPGA) or implemented in silicon (ASIC). This HDL implementation is configurable at compile-time (within this context "compilation" means the process of mapping the HDL description of the processor to the targeted technology). The most important configuration parameters are PU-cluster size (number of PU's) and PU memory configuration (i.e. instruction memory size per PU). The target technology (ASIC or FPGA), combined with application requirements dictates the optimal processor configuration. Typically the most important *constraints* within a given technology are silicon area (e.g. logic gates and amount of embedded memories), feasible clock frequency and power dissipation constraints. From the target application point of view, the main *requirements* are the processing capacity (i.e. an estimation of the required amount of operations (i.e. PAs) per data-flow node/actor (i.e. WPP)), and an estimation of the required fire-rate(s). The configuration of the WaveCore PU cluster is the result of bringing together the mentioned constraints and requirements.

The execution of a PU-thread is always linear: branches, loops or jumps are not supported in the program flow. This means that the number of instructions that can be executed during one thread period can be expressed by eq.(6.2).

$$C_p(f_t) = \begin{cases} N_{pu} \cdot C_{imem} & \text{if } f_t \leq f_d \\ N_{pu} \cdot f_{clk} / f_t & \text{if } f_t > f_d \end{cases} \quad (6.2)$$

With $C_p(f_s)$ the PU-cluster capacity as a function of the thread execution frequency, N_{pu} the number of PUs in the cluster, f_{clk} the PU clock frequency, f_t the thread execution frequency (i.e. fire-rate) and C_{imem} the instruction memory capacity. The maximum fire-rate at which the PU-cluster capacity C_p can be entirely utilized is expressed as f_d (i.e. drop-off fire-rate). The hyperbolic relation between PU-cluster capacity and thread execution frequency reveals the following:

1. *Low thread execution frequencies*

For low thread execution frequencies ($f_t < f_d$) the maximum thread size is bounded by the PU capacity (i.e. instruction memory size), because the WaveCore processor neither supports instruction caching, nor non-linear program flow.

2. *High thread execution frequencies*

For high thread execution frequencies ($f_t > f_d$), only a fraction of the instruction memory can be used by the running thread because of the limited thread-time, related to the processor clock frequency.

FIGURE 6.4 – WaveCore PU cluster capacity on Atlys FPGA board

The computational capacity as a function of thread-execution frequency for the WaveCore processor on our Atlys board is depicted in fig. 6.4. This WaveCore processor instance consists of 6 PUs where each PU has a maximum capacity of 2048 PAs within a thread. The processor runs at 86.016MHz. The maximum ILP for the supported set of PAs is associated to the multiply-add type PA, being 2 floating-point operations per instruction. This implies that the maximum performance in terms of floating-point operation (*FLOP*)s per thread is obtained when all the PUs are entirely loaded with mentioned multiply-add PAs (this corresponds to a 12288-taps FIR filter, where each tap is computed by means of a multiply-add PA). This leads to a maximum performance of $2 \times 2048 \times 6 = 24576$ FLOPs per thread. The capacity profile in fig. 6.4 reveals that this implies the theoretical maximum capacity that can be obtained for thread execution frequencies up to 42kHz. For frequencies above 42kHz the capacity drops according to (6.2).

If a given application implies a multi-rate process-graph (as is the case with detailed physical modeling as we will show in section 6.3), then it might be beneficial to configure a WaveCore processor as a multi-cluster, where each cluster is optimized to a class of threads which are required to run on a certain frequency range. Optimization in this context means that the capacity C (i.e. the instruction memory size) is aligned to the target thread frequency. For example: if a class of threads are required to be executed with a frequency of 84kHz, and the processor clock frequency is 86.016MHz, then the optimal capacity C equals 1024.

In the next section 6.3 we will apply the capacity profile for the WaveCore processor, in order to evaluate and rank the processor performance (i.e. ISA and compiler performance) for a few application examples.

6.3 PERFORMANCE ANALYSIS

In this section we analyze the performance of the WaveCore processor by means of a comparison of the mapping of a few applications on the WaveCore, and different processor technologies. We start with a performance analysis of virtual analog models on the WaveCore and a GPP. The objective is to compare the WaveCore ISA and compiler, against an ARM GPP. Next to this we focus on the comparison of the mapping efficiency of a virtual instrument model on an optimized FPGA design, and on the WaveCore using the same model.

167

6.3.1 ISA AND COMPILER PERFORMANCE ANALYSIS

The execution efficiency of a running application on a given processor can be expressed by the utilization level of the DCU within that processor. If the DCU is fully utilized during each clock cycle then the ultimate execution efficiency is obtained. An idle, or partially idle DCU means that it either cannot execute because of the unavailability of operands, or parts of the DCU resources are not active because the operation could not be merged with other operations in one instruction which keeps the DCU fully utilized. The execution efficiency depends on the ISA and the compiler performance. We analyze the efficiency of the ISA and compiler by expressing the mapping efficiency of a set of algorithms in terms of the ILP (instruction-level parallelism). We define the ILP metric as a relative number (in %), which expresses the averaged percentage of a PA execution per processor clock cycle. Example: an ILP figure of 25% implies that it takes four processor clock cycles to execute one PA. Furthermore, we focus on algorithms which are mapped on a single PU. The ILP number as such can be seen as an abstraction of the efficiency of both the compiler and the ISA (i.e. the efficiency of mapping a WPP onto the ISA). Basically, we derive the number of clock-cycles which are required to execute (i.e. fire) the thread which runs for each fire-period. We do this for the WaveCore and for the processor to which we compare the WaveCore. Then, we can express the ILP as the average number of clock-cycles that are required for executing one PA, since we know how many PAs are present in the compiled thread. We compare the WaveCore PU with an ARM processor. This ARM processor is integrated in a microcontroller chip, the STM32F405ZGT6 processor [88]. This chip is used within the "Owl Pedal" [89]. This Owl pedal is an open programmable guitar effects processor. The objective of this pedal is that an open-source community develops so-called "patches" (guitar effect programs), which are published and shared. The Owl pedal has the following properties:

1. *System*
Programmable open-source effect pedal: the Owl pedal [89].
2. *Processor*
168MHz 32bit ARM Cortex M4, with integrated DSP and floating-point unit (FPU).

TABLE 6.3 – ILP comparison between WaveCore and ARM Cortex M4

Algorithm	#PAs	ILP, ARM	ILP, WaveCore	Performance ratio
CryBaby	82	20%	70%	4
Lorenz Osc.	28	27%	58%	2
Phaser (6-stage)	115	17%	76%	4
FreeVerb	120	19%	77%	4
Octave Down	57	27%	72%	3
Overdrive	26	15%	56%	4
Flanger	48	10%	41%	4

3. *Memory*

192Kb RAM, 1Mb Flash memory.
1Mb external SRAM.

4. *Peripherals*

16/24 bit audio codec, stereo in/out, USB MIDI, 4 assignable control knobs, input for an expression pedal⁵

5. *Supported sampling rates*

8kHz to 96kHz sampling rate

A few important remarks apply to the ILP comparison that we make:

1. *Source code*

The source code for the mapped algorithms is not identical for WaveCore and the ARM, though *similar*. We analyzed the source code (C or Faust [58]) for the ARM-mapped algorithm and translated the algorithm manually to WaveCore code. Next to that, we used the number of required clock-cycles per thread to derive the ILP indicator.

2. *Physical boundary conditions*

The ARM processes in a block-based mode. This implies that an algorithm thread is repetitively executed on a block of input samples. WaveCore operates on elementary tokens without any buffering. The efficiency of the ARM depends on the block size, and hence might not be optimal for the given set of algorithms. Moreover, the clock frequency and memory architecture is different. Note, however, that the clock frequency is irrelevant in this analysis, because we focus on the ILP metric.

The set of algorithms, and associated ILP numbers and complexity in number of PAs, is summarized in table 6.3. As can be observed from the ILP benchmark table 6.3, there is a considerable deviation in the computed ILP figures. This can be explained as follows. A WaveCore thread consists of three parts: initialization (e.g. pointers), the actual algorithm part, and a number of "tail-NOPs" which are inferred to make sure that the GIU pipeline is entirely flushed at the end of the thread. This

⁵An expression pedal allows the musician to change a characteristic of a guitar effect while playing.

overhead (init and tail-NOPs) becomes relatively dominant for small algorithms, which can be seen in the table. A comparable overhead is present in the ARM, and therefore the performance ratio figures do not deviate so much accordingly. It can be concluded that the WaveCore ISA and associated compiler is better optimized to the characteristics of the mapped algorithms, compared to the ARM. In average, the performance ratio between WaveCore and the ARM (the right column in table 6.3) is a factor of four, which means that a single-PU WaveCore processor uses in average approximately 75% less clock cycles, compared to the ARM, to execute a similar implementation of the benchmark algorithms. This can be explained by the fact that the execution of an entire WaveCore PA is "packed" in a single instruction. Obviously, the ARM is more efficient in control processing (e.g. processing of musical instrument digital interface (*MIDI*) control streams).

6.3.2 GEOMETRY MAPPING PERFORMANCE

In chapter 4 we discussed the importance of geometry models, implemented as FDTD schemes, as building blocks for building virtual instruments or reverberating objects (e.g. virtual model of a spring or plate reverberator). In the same chapter we discussed how such objects can be linked together in order to compose more complex objects, like a virtual instrument. In the next section 6.3.3 we will discuss the mapping efficiency of a virtual Banjo on WaveCore. As we will show, such a virtual instrument model can be built from basic geometry objects (e.g. strings, membranes, plates and air spaces). In this section we will elaborate on the characteristics of the mapped basic geometry models, implemented as FDTD schemes. We will apply these derived characteristics to analyze the mapping efficiency of a complete instrument model on the WaveCore.

Based on the wave equation in (4.47) we derived FDTD schemes for strings, membranes and plates and we translated these partitioned schemes to a scalable set of WPPs. Now, in order to analyze the feasibility of mapping a complete virtual instrument to the WaveCore processor, we want to derive generic expressions of the mapping characteristics of 1d, 2d, and 3d FDTD schemes. For this purpose we want to express the processor capacity, as we have defined this in section 6.2, in terms of number of scheme-nodes⁶ as a function of thread execution frequency. Based on the compile results, as presented in section 4.5, we have derived the average number of scheme-nodes for different geometries and orders, as a function of the thread execution frequency. For the WaveCore processor on the Atlys board, this performance profile is depicted in fig. 6.5:

- » The " $1D2^{nd} order$ " curve expresses the maximum geometry model size (in number of second order scheme-nodes) as a function of the thread execution frequency, for a one-dimensional FDTD model: a string.

⁶A scheme-node, or node, in the context of FDTD modeling performs a so-called stencil-computation. Examples can be found in section 4.4.2 of this thesis.

FIGURE 6.5 – Performance profile of Spartan6 based WaveCore system

- » The "*2D2nd order*" curve expresses the maximum geometry model size (in number of second order scheme-nodes) as a function of the thread execution frequency, for a two-dimensional FDTD model. Such a model is called a "membrane".
- » The "*2D4th order*" curve expresses the maximum geometry model size (in number of fourth order scheme-nodes) as a function of the thread execution frequency, for a two-dimensional FDTD model. Such a model is called a "plate".

The upper-limits in fig. 6.5 indicates the best possible mapping: no inferred no-operation (*NOP*)s. The performance profile should be interpreted as follows: it is possible to model a string (i.e. "*1D2nd order*" curve), consisting of 2977 nodes, at a sampling rate of up to 42kHz. Another example: a membrane model (i.e. "*2D2nd order*" curve), consisting of 1119 nodes at a sampling rate of 84kHz, fully utilizes the processor.

The Atlys FPGA board is very useful as a development platform for up to mid-range complexity systems, like within virtual analog or digital waveguide application domains. However, for detailed physical modeling the required computational capacity cannot be hosted by the embedded Spartan6 FPGA. In order to analyze the feasibility of larger scalable physical models, and the associated possibility to map virtual instrument models, we focussed on a larger FPGA system: the ML605 board. This board contains a high-end Virtex6 FPGA which contains approximately 10 times the Spartan6 fabric capacity, and can achieve higher clock rates. We optimized a WaveCore processor instance for the Virtex6 FPGA, which contains 2 PU clusters:

1. *PU cluster*:

This cluster consists of 23 PUs. Each PU has a capacity of 4096 instructions.

FIGURE 6.6 – Performance profile of Virtex6 based WaveCore system

2. PU cluster2:

This cluster consists of 32 PUs. Each PU has a capacity of 2048 instructions.

The rationale for 2 clusters comes from the requirement for both relatively high and low thread execution frequencies (i.e. multi-rate), which are dictated by an accurate modeling of physical properties of the required geometrical objects. The target clock frequency for the Virtex6 based WaveCore processor is 250MHz. In order to configure as many as possible PUs, we de-configured the relatively expensive floating-point divider (only floating-point addition and multiplication are supported for this WaveCore processor instance). The peak-performance of this WaveCore processor configuration on this ML605 board is $2(23 + 32) \times 250 \cdot 10^6 = 27.5$ GFLOPs/sec.

The resulting performance profile in terms of number of executable nodes as a function of the thread execution frequency is depicted in fig. 6.6. In the next section 6.3.3 we will use these performance profiles in order to estimate the mapping feasibility of a Banjo model, and to compare this estimation against an existing FPGA based system which is optimized for FDTD based musical instruments modeling.

6.3.3 FPGA BENCHMARK: THE BANJO

The Institut for Systematic Musicology in Hamburg focusses, amongst others, on physical modeling of musical instruments through finite elements methods. One of the achievements that Pfeifle and Bader have made is being the first to map complete virtual instruments, running in real-time, on FPGA [64]. An abstract model of the Banjo instrument is depicted in fig. 6.7. This musical instrument contains 5 strings, which are connected to the "peghead" of the instrument, on the right-hand side, and connected to the "bridge" of the instrument on the left-hand side. The

TABLE 6.4 – Banjo geometry decomposition

Instrument part	#Nodes	Order	Sample freq.
Strings	560	2,4	65kHz
Air	8192	2	65kHz
Membrane	2048	2,4	128kHz
Bridge	512	2	128kHz

FIGURE 6.7 – Banjo, as a composition of basic geometrical structures

peghead contains the tuning pegs. Note that this part of the instrument is not explicitly modeled, and hence not drawn in fig. 6.7. Instead, the strings are "rigidly terminated" at the "nut" of the instrument (i.e. the upper-end of the fretboard), which means that the travelling waves are reflected at this side. Like mentioned, the other end of the strings are connected to the bridge of the instrument. This bridge forms the interface between the strings and the circular shaped membrane. The bridge itself is a small, stiff plate which causes the vibrations at the end of each string to partially reflect, interfere, and admit part of the energy to the membrane in a complex way. Likewise, the 2 dimensional wave patterns in the membrane are transmitted to the underlying 3 dimensional enclosed air space which causes a complex wave scattering pattern which is partially fed back to the membrane, bridge and ultimately the strings.

In terms of physical modeling, the Banjo can be described as a composition of basic geometrical structures, like depicted in fig. 6.7. The geometrical structures from which the Banjo model is constructed are specified in table 6.4. Note that two different and relatively high sampling rates are required to iterate the model (64kHz and 128kHz). The abstract Banjo model has been mapped by Pfeifle on a XC6VLX240T FPGA [103], which is available on the ML605 evaluation board [102]. For this purpose, a dedicated and domain specific HW architecture has been developed which is depicted in fig. 6.8. For each geometry part a different computational unit is specifically developed (e.g. strings/air, membrane and bridge), where each unit computes a limited number of scheme-nodes in parallel. These units are repetitively executed by the central controller for each sample domain, for each sampling period (comparable to the central scheduler in the WaveCore architec-

FIGURE 6.8 – FPGA system for virtual instrument modeling, as described in [64]

ture). The model interacts with the user (player/observer) through a PCI express (*PCIe*) interface between a host computer and the ML605 board. This implementation of the Banjo model just fits within the XC6VLX240T FPGA, where effort had to be put in fitting the model using the available FPGA resources (e.g. timing optimization and resource optimization through optimization of arithmetic operators).

We analyzed the feasibility to map the same Banjo model on a WaveCore processor. The first step in this analysis is to derive a WPG for the model, which is depicted in fig. 6.9. This WPG consists of the following processes:

Interaction process "Stim"

This process mimics the Banjo player. In reality this is a complex process where the player causes vibrations to the instrument, not only by picking the strings, but also through interaction with different parts of the instrument. In theory this implies that every node in every string may be manipulated by the player. In our model, however, we limited the interaction through string interaction only. This implies that in theory, edge *S*₁ carries tokens with a width of up to 560 PTs in order to enable the player to manipulate any string at any position(s) along the fretboard⁷.

Process "64kHz process"

This process includes all parts of the Banjo which are required to be executed at a rate of 64kHz (i.e. the strings and the air). This process contains one incoming edge

⁷The fretboard, or fingerboard, is that part of a string instrument where the musician can press strings down in order to change the vibration length of the strings

FIGURE 6.9 – Banjo model, represented as a WaveCore Process Graph

S1 (for stimulation of the strings), one incoming/outgoing edge-pair (SB1, SB2) for interaction between the strings and the bridge, and one incoming/outgoing edge-pair (MA1, MA2) for interaction between the membrane and the air. Note that the associated interaction token-widths are considerable: the MA1/MA2 tokens carry interaction values for each scheme-node in the membrane in both directions, hence the token width MA1/MA2 is 2×2048 PTs. Note that the tokens along SB1/SB2 are relatively small: for each string one interaction value in both directions, resulting in a width of 10 PTs.

Process "128kHz process"

This process includes all parts of the Banjo which are required to be executed at a rate of 128kHz (i.e. the membrane and the bridge).

Process "Resp"

This process probes into the instrument model and transmits the probed node-values to the observer through the associated R1 edge. In reality the whole instrument vibrates and as such the audible acoustical signal which reaches the observer is a complex combination of interfering and reverberated waves which are induced by the entire instrument. As a result, all the nodes in the instrument model take part of the composition of the ultimate audible signal and as such, in theory the R1 edge needs to carry the entire state of the instrument (i.e. all 11312 nodes). In practice, however, a limited set of nodes is selected in order to compose an acoustical "output" signal of the instrument. In our case, we chose to probe into the membrane of the instrument, yielding a maximum R1 token width of 2048 PTs.

We used the capacity profile in fig. 6.6 (i.e. computational *constraints*) and the Banjo model *requirements* in table 6.4 to make an estimation of the feasibility to map a complete Banjo model onto the ML605-optimized WaveCore processor instance. This feasibility has been derived from a spread-sheet estimation model that we developed. The results of this analysis are summarized in table 6.5. This table gives an overview of how the different geometrical parts of the Banjo are allocated to the two PU clusters within the WaveCore processor. Based on the node-ILP fig-

TABLE 6.5 – Estimated WaveCore processor load: the Banjo

Part	Cluster Utilization			Average GIU load	Transferable Nodes	GPN load
Strings	Cluster1	4%	72%	93%	10%	3%
Air		68%		93%	1%	7%
Membrane	Cluster2	60%	75%	97%	1%	10%
Bridge		15%		97%	5%	10%

ures in the capacity profiles in fig. 6.6 the relative PU cluster utilization is computed (e.g. 4% of Cluster1 capacity is consumed by the "Strings" part of the Banjo). The "Average GIU load" indicates which percentage of the cycle-budget the associated GIUs are active. The remaining PU instruction memory capacity is available to implement node-transfer through the LSU within each PU. For example: if a GIU is loaded for 93%, then there is 7% of remaining instruction memory capacity which can be used to move node data from/to the model. This remaining instruction memory capacity results in the "Transferable nodes" figure (e.g. given the GIU load, it should be possible to transfer 10% of the string-nodes from/to the model). The GPN bus-load follows from the partitioning of the models (i.e. "radiation" see chapter 4.4.2). The aggregated cluster utilization figures (i.e. 72% for Cluster1 and 75% for Cluster2) indicate that it should be possible to map the Banjo on the WaveCore processor on the ML605 board. However, there are a number of assumptions and required optimizations to the processor technology (both the processor and the compiler) in order to enable this mapping:

1. *Processor clock frequency*

The relatively high sampling rates of the Banjo model leads to the 250 MHz clock frequency requirement for the Virtex6 FPGA. During mapping experiments we found a maximum PU clock frequency of approximately 230 MHz, which is clearly insufficient. Through an experimental deeper processor pipeline we found a theoretical upper limit of 380 MHz. Note, however, that a deeper pipeline most probably leads to more data-hazard stalls.

2. *Secondary process edge mapping*

From the Banjo WPG in fig. 6.9 we derived the large token sizes for the edges between the 64 kHz and 128 kHz processes. In our current compiler implementation all edges are mapped on external memory, by means of the LSU. Obviously, when the mentioned large tokens are to be mapped on external memory, then the external memory interface will certainly overload. Conceptually it is better to allocate these secondary process edges on the GPN network, instead of DMA.

3. *Automated partitioning of WPPs*

The current implementation of the compiler does not partition and allocate a large WPP, like we have proposed in chapter 3.4 of this thesis. However, for a well balanced and optimized partitioning it is required to do this in an automated way, in order to obtain a sufficiently high GIU utilization.

4. *Data memory usage*

The current implementation of data memory allocation within the compiler is greedy w.r.t. data memory usage. The allocation algorithm does not distinguish between volatile variables (i.e. these are variables which are temporarily used during computation of combinatorial parts of a DAG) and state variables. State variables need to be memorized, so there is no way to reduce data memory size for this. However, for volatile variables a "scratch memory" concept could be used, resulting in a smaller data memory footprint requirement. The current implementation implies a high risk of data memory overallocation for very large child-WPPs which just-fit a single PU.

5. *Burst-mode DMA*

In particular, for large primary tokens (e.g. stimulus to- and response data from the Banjo), there is a high risk of external memory bandwidth overload. Within the current LSU implementation, only single-beat DMA transactions are supported, which implies a non-optimal external memory bus utilization. Conceptually it is well possible to combine transactions in a burst transactions, and thereby reducing the external memory bus utilization.

6.4 WAVECORE DESIGN METHODOLOGY

In chapter 4 we analyzed the broad application range of physical modeling, ranging from relatively lightweight virtual analog models, to massively parallel detailed physical modeling. In principle, the WaveCore solutions for these modeling problems combine computation in space (i.e. partitioning of an algorithm and distributing the partitions over one or more PUs), and time (i.e. temporal scheduling of PAs within a child-WPP onto a GIU/LSU combination). To achieve this, the WaveCore compiler optimizes in space and time through the composition of indivisible DAGs to child-WPPs in such a way that the temporal scheduling of the resulting child-WPPs meet the required real-time requirements. Ideally this results in a solution which consists of a generated WaveCore processor which consists of a minimal set of PUs. Furthermore, the memory sizes can be minimized. Finally, the minimal clock frequency for the WaveCore processor can be derived as well, yielding a power optimal solution ⁸.

We propose a WaveCore design methodology, which flow is depicted in fig. 6.10. The proposed design flow serves three goals.

The *first goal* is to derive the ideal processor instance from a given set of application domain specific problems. If a set of representative application programs in the form of WPGs are available, then the WaveCore compiler can translate these into associated WOF descriptions. From this set of WOF descriptions, the optimized processor instance (i.e. required number of PUs, memory size, clock frequency)

⁸Note that the automation of the described design methodology is not developed. However, we see no fundamental issues which would make such a flow impossible

FIGURE 6.10 – WaveCore design flow

can be derived. Then, this optimized processor instance needs to be embedded in a SoC skeleton, and be mapped on a target technology platform (i.e. an ASIC or an FPGA). Note that this process has been carried out in a manual way, yielding the dual-cluster ML605 WaveCore processor instance from the virtual instrument mapping requirements, in section 6.3.2.

The *second goal* is to compile a program within the target application specific domain onto the processor. Examples are the mapping of virtual analog models onto a given processor instance, like the WaveCore processor on the Atlys FPGA board.

The *third goal* is to generate a "black-box" (i.e. fixed) DSP function. The implementation of this function is an optimized WaveCore processor with fixed program code. The design flow for generating such dedicated functions can be compared with for instance Simulink [39]. Simulink (by the Mathworks) has a graphical design entry, where library function blocks can be combined in a schematic. The composed design can then be compiled to RTL code which subsequently can be targeted to a technology (i.e. ASIC or FPGA). Another example of a design methodology which yields synthesizable HDL is Clash [40]. Clash is based on Haskell and allows the developer to design hardware from a functional, mathematical design entry. Often, design flows like mentioned lead to generated HW descriptions, where the focus is on space (i.e. much parallelism) and hence the generated HW unit is often optimized to operate at a relatively high sampling rate (i.e. the sampling rate is often identical or close to the clock frequency of the generated HW block). A methodology like WaveCore could fit in design methodologies like Modelsim or Clash, where sample-rates are typically much lower than the obtainable clock frequency of the processing block. Clash and the functional design methodology, based on WaveCore and Kronos like described in chapter 5, have the important similarity that the functional language compile process leads to a static data-flow graph. At this level, a translation can be made towards a WaveCore WPG, depending on whether the primitive functions in the graph can be mapped on WaveCore PA primitives.

6.5 DISCUSSION AND CONCLUSIONS

We have analyzed the efficiency of the WaveCore processor, the WaveSlang compiler and ISA, and development methodology. We did this by means of comparing the processor against two comparable (i.e. data-flow/DSP) Recore processors: the Montium and the Xentium. The efficiency of the ISA and compiler has been analyzed and compared with the ARM Cortex M4 processors through a benchmark algorithm set. We also compared a domain-specific FPGA solution, the virtual Banjo, with a WaveCore-based mapping feasibility study. From these analysis we can summarize the strengths and weaknesses of the WaveCore processor, and its design flow.

The strengths of the WaveCore processor and its design methodology can be summarized as follows:

1. *Abstraction*

The design entry for either an optimized WaveCore solution (i.e. a HW/SW combination, acting as a black-box domain specific accelerator), the optimization of a domain specific WaveCore processor instance, or the development of an application which is to be mapped on a WaveCore processor, originates from an abstract declarative formulation. This WPG description hardly contains any physical, and processor technology dependent constructs. Only the functional description and the real-time constraints (e.g. fire-rate(s) of the associated processes) need to be declared.

2. *Design effort*

Since the design flow is highly optimized and the design entry is abstract, the WaveCore application design effort can be limited. Developing an optimized application for the Owl (i.e. embedded ARM Cortex M4 programming) requires in-depth knowledge of the processor architecture (e.g. memory hierarchy and associated latencies, parallelism in the data-path and how to exploit this explicitly, allocation of heap) and/or the usage of standard optimized function libraries (e.g. FIR or FFT functions). Within the detailed physical modeling domain (e.g. detailed virtual instrument modeling) we compared an FPGA based design methodology with a WaveCore based design methodology. Here we also found that design abstraction can potentially reduce the required design effort significantly. Note, however, that this conclusion is based on a mapping estimation of the Banjo, and hence we did not actually perform a complete implementation.

3. *Flexibility*

We showed that the WaveCore processor can be optimized towards a specific area within the target application domain: small single-PU instances for many virtual analog applications, and large multi-cluster PU processors for detailed physical modeling. The programming entry, however, is identical: a target unaware WPG.

4. *Efficiency*

As shown in the benchmarks on the ISA, as well as on the physical aspects of the WaveCore processor (i.e. DCU metric and area comparison) the mapped DSP solutions are efficient, despite the abstraction level of the WPGs.

Weaknesses of the WaveCore processor and its design methodology are the following:

1. *Lack of control capabilities*

We showed that the WaveCore processor executes statically mapped data-flow algorithms. As a result, control structures, like "if then" are not necessary and therefore not implemented in the WaveCore. This implies that physical models which do not fit into this data-flow concept may be hard, if

not impossible, to map on the WaveCore. Additionally, it is in many cases inevitable to include a GPP next to the WaveCore processor, in order to map an interaction model.

180

2. *Inefficiencies with respect to thread execution frequencies*

Through the "capacity" curve we linked the processor capacity to the thread-frequency. This relation shows that:

(1) For low thread-frequencies, the capacity of the WaveCore processor is bounded by the capacity of the local instruction memory of the PUs. Here we miss the advantages of instruction caching and non-linear program flow, which are common for almost all GPPs and many DSP processors.

(2) For high thread-frequencies, the capacity of the WaveCore processor is limited by the clock frequency of the processor. In this area we observe that the DCU metric of the processor cannot compete with hardwired solutions (e.g. a "space"-type of mapping of a dedicated signal processing function).

CONCLUSIONS, SUMMARY, HETEROGENEOUS PROCESSOR ARCHITECTURE
, discussion and future work Conclude
future work Conclusions, claims, di
ussions, claims, discussion and fut
Con
claim
claim
, Con
, dis
ature
ation
ature
are we
n, cla
on and
k Con
claims,
and fr
achieve
discuss
ature wor
ussions.

CONCLUSIONS

7.1 SUMMARY

The subject of this thesis is a scalable architecture for real-time audio/acoustical signal processing, focussing on music signal processing applications. Within this application domain we distinguish two sub-domains: digital audio effects (DAFx), and audio synthesis applications (e.g. virtual music instruments). For both sub-domains we found that the related applications can be described with a physical model and an interaction model. The physical model executes the real-time signal processing, and can often be described by means of synchronous data flow graphs (SDF). The complexity (i.e. computational size) of physical models ranges significantly, from simple filter-based algorithms to complex finite-element based sound-rendering algorithms (e.g. virtual instrument modeling). An interaction model controls the physical model and implements the user interface, and executes in an event-driven fashion.

We observed a number of problems within the state-of-the-art approaches of music/acoustical signal processing. Problems, related to the real-time execution of physical models, are long processing latencies and difficulties in meeting real-time constraints (i.e. time-predictability of execution). Moreover, the computational complexity of, in particular, detailed physical models gives rise to scalability problems. We also identified a number of problems, related to programmability. Functional algorithm development methodologies are widely used in the application domain, whereas imperative programming languages (e.g. C++) are required in order to program the target architectures. This gives rise to a lack of transparency, with respect to real-time constraints, and scalability.

In order to address these problems, we proposed, developed and prototyped a domain-specific heterogeneous processor architecture. This architecture consists of a general-purpose processor (GPP) and our main contribution, the WaveCore: a programmable multi-core processor. The GPP executes interaction models, and the WaveCore executes physical models in real-time. Next to the heterogeneous

processor architecture and the WaveCore within, we developed a data-flow language called WaveSlang, a compiler and simulator. We developed a FPGA based development platform, to which we mapped the WaveCore processor. We analyzed various applications, described these in the WaveSlang language, and mapped these on the development platform. Next to this, we analyzed heterogeneous WaveSlang code generation for a Graphic Equalizer application, from a functional and domain specific language, called Kronos. We did a benchmark analysis in order to compare the WaveCore against other processors.

7.2 CONCLUSIONS

We have defined, developed and prototyped the WaveCore processor on a FPGA platform. Using this platform, we analyzed the WaveCore processor on different aspects like mapping of domain specific algorithms and comparison with existing processor solutions. Within the analysis we have attempted to cover the main aspects, being time-predictability of execution at ultra-low processing latency, scalability and programmability. Based on our work and contributions, we formulate a number of claims.

7.2.1 CLAIMS

Claim 1: data-flow model and application domain coverage.

The majority of real-time music/acoustical signal processing algorithms can be composed of a physical model and an interaction model. Furthermore, the majority of physical models can be described as synchronous data-flow graphs. Interaction models are event-driven and hence do not fit well in an synchronous data-flow paradigm. This implies that a hybrid data-flow approach is a good way to describe these algorithms. Through application cases we showed that this data-flow model covers the intended application domain to a large extend.

Claim 2: time-predictability and ultra-low latency.

The WaveSlang language, and the WaveCore processor have been developed concurrently. This results in transparent mapping of WaveSlang code onto the processor. Moreover, WaveSlang represents a physical model as a synchronous data-flow graph with a strict definition of real-time behaviour and processing latency. We have showed that this results in transparent mapping of WaveSlang code on the WaveCore processor, yielding a high degree of time-predictability and ultra-low processing latency.

Claim 3: high-level programming.

The declarative WaveSlang language is a natural compile-target for functional programming languages. We showed that it is possible to automatically compile a high-level functional description of a Graphical Equalizer application to WaveSlang. Subsequently, the WaveSlang code has been automatically mapped to the development platform, consisting of the WaveCore and a GPP. This implies that a

high-level functional description of an algorithm can be transparently, and automatically mapped with a high degree of control over real-time requirements (i.e. latency and time-predictability).

Claim 4: scalability.

The scalable WaveCore processor architecture, combined with the WaveSlang language, enable automated mapping of computationally complex physical models (e.g. virtual music instruments or reverberation models). We have showed that automated partitioning and allocation of such models is well possible, with a high degree of control over real-time constraints.

Claim 5: efficiency.

Our benchmark analysis has shown that the WaveCore processor is area-efficient. Furthermore, we found that the instruction-set of the WaveCore is approximately four times as powerful as the instruction set of an ARM processor, for the evaluated benchmark applications.

Claim 6: design methodology.

The WaveSlang language, combined with the proposed heterogeneous processing architecture with scalable WaveCore processor, enables an automated design methodology for real-time music/acoustical signal processing applications. Automated generation of an optimized WaveCore processor from a domain-specific functional language is shown to be possible.

7.3 DISCUSSION AND FUTURE WORK

We define the following three topics for future work.

(1) Optimization of the WaveCore for detailed physical modeling.

In chapter 4 and publication [MJW:5] we have showed that FDTD based physical geometry models, and the composition of these models to virtual instruments, are well supported within WaveSlang. Furthermore, we showed that such models can be automatically mapped on the WaveCore efficiently. In chapter 6, however, we showed that the mapping of a complete virtual instrument (e.g. the Banjo) is challenging. The feasibility analysis of mapping the virtual Banjo (discussed on and 6) revealed a number of issues that need to be further explored/enhanced:

» Spatial mapping

We experienced in our work on mapping FDTD models on the WaveCore that manual partitioning of geometry models is not trivial for two reasons:

(a) Interaction between partitions need to be taken into account. Making this interaction explicit in WPPs manually is tedious and error-prone.

(b) Manual optimization of partition-size (i.e. the geometry of the partition, and the number of FDTD nodes within the partition) for optimal utilization of PUs within the WaveCore is difficult.

In section 3.4 we explained that automated partitioning of a parent- WPP into one or more child- WPPs is possible, based on identification and clustering of DAGs. Research on an optimal algorithm for automated partitioning is required to solve the described problems.

» Further enhancements of locality

We identified that large tokens are required for interaction between the different geometry models. These tokens are exchanged through WPG edges, and these edges are mapped on level3 memory by means of DMA. This implies a high utilization of the external memory interface of the WaveCore. A solution to this problem is to map WPG edges between WaveCore processes on the GPN network.

» DMA enhancements

The WaveSlang compiler only supports DMA bursts with length 1. For large tokens this implies that the level3 memory interface can easily become overloaded. Reduction of level3 memory transactions can be obtained when larger bursts are supported.

» Speed-optimization of the WaveCore

WaveCore synthesis experiments for FPGA have showed that the feasible clock frequency of the PU is not sufficient to meet the performance requirement for the virtual Banjo. We found that a deeper GIU pipeline is necessary.

(2) Automated heterogeneous application mapping.

In chapter 5 we discussed the automated mapping of a graphic equalizer application to the heterogeneous processor architecture, where the application was described in the functional and domain-specific language Kronos. This application, however, requires the most basic interaction between interaction model and physical model, being a unidirectional interface where control tokens are channelled. Research on the mapping of a more demanding application, like a modular synthesizer, is a good follow-up on this research.

(3) Enhancements of the WaveCore for non-linear physical modeling.

Virtual amplifiers and distortion effects are very relevant in music signal processing. We only briefly touched non-linear modeling in chapter 4. Research on the computational requirements for non-linear modeling in the context of the WaveCore is necessary.


```

of the
and fig-
ne format
explains ti
this appendi
compute table t
at W-clang.com
fig- of the Wave
form
their
ppes
de 12'
relo
res of
mat and figures of the WaveSlang
and the format and figures of the Wa
add
e this
fig com
the W
and fig
ans the f
this appendix explai

```

WAVESLANG CODE EXAMPLE

The WaveCore process in fig. A.1 describes a typical DAFx application. The physical model in this example, being a BiQuad filter [80], receives its input from an ADC peripheral. Its output is routed to a DAC peripheral. The physical model can be controlled in run-time by the user, by means of the interaction model. This model is implemented with the control process. The scheduler synchronizes the entire processing chain to a firelink, called "AudioRate" (48 kHz).

The BiQuad filter itself is implemented in the PCORE within the physical model (Bi-Quad Filter Process). This PCORE contains one WPP, which is depicted in fig. A.2. The mapping of the edges on the I/O signals in the WPP is as follows:

1. The *Gain edge* is mapped on the Gain signal, and carries one PT.
2. The *ADC edge* is mapped on the $x[n]$ signal, and carries one PT.
3. The *DAC edge* is mapped on the $y[n]$ signal, and carries one PT.
4. The *AudioRate firelink* is mapped on the fire signal.

The WaveSlang code for the example application consists of 6 language constructs, being the descriptions of: WPG, physical model (WaveCore process), interaction model (control process), peripheral model (ADC/DAC foreign processes), and the PCORE of the physical model (WPP which describes the BiQuad filter). These descriptions are given in the following WaveSlang code listings.

FIGURE A.1 – ReMap application, described as WPG

FIGURE A.2 – Biquad filter, represented as a WPP

```

1  // Instantiation of the Scheduler:
2
3  Sscheduler MyScheduler {
4      Frequency 86016000          // Clk frequency of the WaveCore
5      FireLink AudioRate 1792    // 86016000/1792 = 48kHz fire-rate
6  }
7
8  // WaveCore process graph:
9
10 Process BiQuadControl {
11     Source    BiQuadControl.proc    // Reference
12     OutEdge   CntrEdge    ControlPort // Outgoing control edge
13 }
14
15 Process BiQuad {
16     Source    BiQuad.proc    // Reference
17     FireLink  MyScheduler AudioRate // Link to scheduler
18     InEdge   CntrEdge    ControlPort // BiQuad control port
19     InEdge   ADC          InPort    // Incoming edge
20     OutEdge  DAC          OutPort   // Outgoing edge
21 }
22
23 Process ADC_mono {
24     Source    ADC_mono.proc    // Reference
25     FireLink  MyScheduler AudioRate // Link to scheduler
26     OutEdge   ADC          ADCport // Output edge
27 }
28
29 Process DAC_mono {
30     Source    DAC_mono.proc    // Reference
31     FireLink  MyScheduler AudioRate // Link to scheduler
32     InEdge   DAC          DACport // Input edge
33 }

```

LISTING A.1 – WPG code for the BiQuad application

```

1 ProcessEntity BiQuad {
2     // Process type:
3     ProcType WaveCore
4     // WaveCore Partition #1 --> algorithm
5     Wpp
6         WppID      BiQuad
7         WppSource BiQuad.sfg // Reference to WPP description
8     EndWpp
9     // Incoming edge-port definition: signal input port
10    Port InPort
11        Direction InBound
12        Signal      BiQuad x[n]
13    EndPort
14    // Incoming edge-port definition: control port
15    Port ControlPort
16        Direction InBound
17        Signal      BiQuad .Gain
18    EndPort
19    // Outgoing edge-port definition
20    Port OutPort
21        Direction OutBound
22        Signal      BiQuad y[n]
23    EndPort
24 }

```

LISTING A.2 – WaveCore process description

```

1 ProcessEntity BiQuadControl {
2     // Process type:
3     ProcType Control
4     // Outbound port
5     Port ControlPort
6         Direction OutBound
7         Signal void Gain 0.02
8     EndPort
9 }

```

LISTING A.3 – Control process description

```

1 ProcessEntity ADC_mono {
2     // Process type:
3     ProcType Foreign
4     // Outbound port:
5     Port ADCport
6         Direction OutBound
7         Signal      void ADC_mono[n]
8         Attribute SimFileRef ClassicalGaz.wav
9         Attribute SimFormat WAV
10    EndPort
11 }

```

LISTING A.4 – ADC (foreign) process description

```

1 ProcessEntity DAC_mono {
2     // Process type:
3     ProcType Foreign
4     // Inbound port:
5     Port DACport
6         Direction InBound
7         Signal      void DAC_mono[n]
8         Attribute SimFileRef mono.wav
9         Attribute SimFormat WAV
10    EndPort
11 }

```

LISTING A.5 – DAC (foreign) process description

```

1 BeginWPP
2 // ADC edge connection to x[n] signal:
3 CON nc    nc    nc x[n]  0.00 1 // PA(6)
4 //
5 // Gain edge connection to .Gain signal:
6 CON nc    nc    nc .Gain 1.00 1 // PA(8)
7 //
8 // BiQuad filter implementation
9 MAD x[n]  pa2.y nc pa1.y  0.13 0 // PA(1)
10 MAD x[n]  pa4.y nc pa2.y  0.40 1 // PA(2)
11 AMP x[n]  nc    nc pa3.y  1.76 0 // PA(3)
12 MAD pa1.y pa5.y nc pa4.y -0.98 1 // PA(4)
13 MAD pa1.y pa3.y nc pa5.y -1.56 1 // PA(5)
14 //
15 // Connection of filter output to DAC edge:
16 MUL pa1.y .Gain nc y[n]   1.00 0 // PA(7)
17 //
18 EndWPP

```

LISTING A.6 – WaveSlang code of BiQuad filter WPP


```

e format and figures of the WaveSlang
-- explains the format and figures of the WaveSlang
explains the
appendix e
the table th
WaveSlang con
s of the WaveS
i and figure
the format
explains th
appendix explains the format
the table this appendix explains
WaveSlang con
s of the WaveS
i and figure
the format
explains th
appendix e
the table th
WaveSlang con
s of the WaveS
mat and figure
ndix explains the format and figures

```

WAVESLANG COMPILATION METRICS

For each compiled WaveSlang program in this thesis we have included a compile report. This report is formatted in a table, which is shown in B.1. An explanation of the entries in the compile table is as follows:

GIU utilization

This section of the table reports the utilization of the GIU pipeline for the compiled WaveSlang program.

- » *Instructions*
The total number of GIU instructions, including inferred pipeline bubbles (NOPs), and pointer update instructions. Pipeline bubbles are "no-operation" instructions which are inferred in the instruction stream because of data hazards. A data-hazard is a well-known problem in a pipelined processor. A data-hazard is a condition that occurs when the next instruction that can be scheduled depends on an operand which is not computed yet, and pending in the processor pipeline. If more than one PU is occupied, then this is an average number.
- » *Data-hazards*
The relative number of inferred NOPs in %, due to data-hazard. If more than one PU is occupied, then this is an average number.
- » *Thread-sync*
The relative number of inferred NOPs in %, due to thread-synchronization. Thread-synchronization NOPs need to be inferred when the thread is waiting on operands from a different PU which needs to send these operands through the GPN network. If more than one PU is occupied, then this is an average number.
- » *Pointer (pntr) updates*
Pointer updates are required when the address computation unit within the GIU cannot compute the required next-address from the current address and the offset w.r.t. the next address. In such a case, the required offset is

TABLE B.1 – Compile metrics of WaveSlang program

Compile results of algorithm: Phaser					
GIU utilization		Resource load		Mapping metrics	
Instructions	412	Pmem	456	Required PUs	1
Data-hazards	52.2%	Xmem	98	Total #PAs	172
Thread-sync	0.0%	Ymem	99	Total #DAGs	61
Pntr updates	0.0%	Cmem	106	Total #delay-lines	0
LSU utilization		Cycle-budget	23.0%	Longest path	28
Instructions	44	L2 GPN-load	0.0%	Firing rate (kHz)	48
DMA transactions	11	L3 bus load	0.6%	L3 mem (Bytes)	88

larger than the maximum offset. If more than one PU is occupied, then this is an average number.

LSU utilization

This section of the table reports the utilization of the LSU for the compiled WaveSlang program.

» *Instructions*

The total number of LSU instructions. If more than one PU is occupied, then this is an average number.

» *DMA transactions*

The total number of DMA transactions, being conducted for each thread execution. This figure is the sum of inbound and outbound transactions and incorporates DMA transactions to support token movement as well as delay-line data movement. If more than one PU is occupied, then this is an average number.

Resource load

This section of the table reports the occupation of the level-1 memories (i.e. Pmem, Xmem, Ymem and Cmem), the occupied cycle budget and the load of GPN and external memory busses.

» *Pmem*

Number of occupied 32-bit program memory locations. If more than one PU is occupied, then this is an average number.

» *Xmem*

Number of occupied 32-bit x data memory locations. Each 32-bit word represents a floating-point number. If more than one PU is occupied, then this is an average number.

» *Ymem*

Number of occupied 32-bit y data memory locations. Each 32-bit word represents a floating-point number. If more than one PU is occupied, then this is an average number.

» *Cmem*

Number of occupied 32-bit coef-memory locations. Each 32-bit word represents either a constant p , or pointer values which are administrated by the LSU. If more than one PU is occupied, then this is an average number.

» *Cycle budget*

This number represents the percentage of clock-cycles, relative to the maximum thread-time in #-clock-cycles, during which the PU is busy executing the thread. The maximum thread-time is related to the clock frequency of the PU and the fire-rate which applies to the thread. Unless mentioned otherwise, the fire-rate within the compile examples equals 48 kHz and the clock frequency of the PU equals 86.016 MHz, yielding a maximum thread-length of 1792 clock cycles.¹

» *L2 GPN load*

This number represents the percentage of clock-cycles, relative to the thread time, during which the GPN network is moving data.

» *L3 bus load*

This number represents the percentage of clock-cycles, relative to the thread time, during which the external memory interface is active. We assume a single external memory interface, like we implemented on the Atlys board.

Mapping metrics

This section of the table contains figures which are related to the mapping of the WPG.

» *Required PUs*

Total number of PUs which are occupied by the compiled WPG.

» *Total #PAs*

Total number of compiled PAs, including all occupied PUs.

» *Total #DAGs*

Total number of DAGs in the WPG. This metric is an indication for the amount of task-level parallelism within the WPG.

» *Total #delay-lines*

Total number of delay-lines within the WPG.

» *Longest path*

The longest path, in #combinatorial PAs, within the set of DAGs. The longest path defines the minimal thread-length in #-clock-cycles as follows:
 $\lambda_{tmin} = P.S$, with P the longest-path length, S the number of pipeline stages in the GIU and λ_{tmin} the minimal thread length in GIU-#clock-cycles.

» *Firing rate (kHz)*

The firing-rate of the processes within the WPG that are mapped to WaveCore

¹The clock frequency of the WaveCore processor on the Atlys board is a multiple of the audio clock. The processor clock is generated by an on-chip phase-locked loop (PLL)

PUs. This implies that all WPGs for which these tables represent the compile results has only one fire-link for the WaveCore processes.

» *L3 mem (Bytes)*

The total amount of allocated external memory in #Bytes.

ACRONYMS

201

A	ACU	address computation unit
	ADC	analog-to-digital converter
	ALU	arithmetic and logic unit
	ASIC	application-specific integrated circuit
B	BBD	bucket-brigade device
	BSP	board support package
C	CGRA	coarse-grained reconfigurable architecture
	CLB	configurable logic block
	CPU	central processing unit
D	DAC	digital-to-analog converter
	DAFx	Digital Audio Effects
	DAG	directed acyclic graph
	DCT	discrete cosine transform
	DCU	data computation unit
	DDR	double data rate SDRAM
	DFG	data-flow graph
	DLL	delay-locked loop
	DMA	direct memory access
	DSP	digital signal processor
	DWG	digital wave-guide
E	EDA	electronic design automation
	EOT	end-of-thread
F	FDTD	finite-difference time-domain
	FFCF	filtered-feedback comb-filter
	FFT	fast Fourier transform
	FIFO	first-in-first-out buffer
	FIR	finite impulse response
	FLOP	floating-point operation
	FPGA	field-programmable gate array
	FPU	floating-point unit

G	GALS	globally asynchronous locally synchronous
	GEQ	graphic equalizer
	GIU	graph iteration unit
	GPGPU	General-Purpose computing on Graphics Processing Units
	GPB	graph partitioning network
H	GPP	general purpose processor
	HDL	hardware description language
	HPI	host-processor interface
	HW	hardware
I	I/O	input/output
	ibFIFO	inbound FIFO
	IIR	infinite impulse response
	ILP	instruction-level parallelism
	IP	intellectual property
	ISA	instruction-set architecture
L		
	LFO	low-frequency oscillator
	LSU	load/store unit
	LTI	linear time-invariant
M	LUT	lookup table
	MAC	multiply-accumulate
	MIDI	musical instrument digital interface
	MIMD	multiple-instruction multiple-data
N		
	NoC	network-on-chip
O	NOP	no-operation
O	obFIFO	outbound FIFO
	ODE	ordinary differential equation
P		
	PA	primitive actor
	PC	program counter
	PCB	printed circuit board
	PCIe	PCI express
	PCORE	process-core
	PCU	program control unit
	PDE	partial differential equation
	PLL	phase-locked loop
	PT	primitive token
R	PU	processing unit
	RAM	random-access memory
	ReMap	REal-time Music/Acoustical signal synthesis and Processing
R	RISC	reduced instruction set computing
	RTL	register transfer level

S	SDF	synchronous dataflow
	SDRAM	synchronous dynamic RAM
	SIMD	single instruction, multiple data
	SoC	system-on-chip
	SW	software
T	TCU	thread control unit
	TP	tile processor
U	ugen	unit generator
	USB	universal serial bus
V	VCO	voltage-controlled oscillator
	VLIW	very long instruction word
W	WBB	write-back buffer
	WDF	wave digital filter
	WOF	WaveCore object file
	WPG	WaveCore process graph
	WPP	WaveCore PCORE partition

BIBLIOGRAPHY

205

- [1] Mxr m117 fanger, product description. URL http://www.jimdunlop.com/files/manuals/M117_man_WEB.pdf. (Cited on page 119).
- [2] R. Bader, J. Richter, M. Münster, and F. Pfeifle. Digital guitar workshop - a physical modeling software for instrument builders. In *Proceedings of the Third Vienna Talk on Music Acoustics*, Vienna, Austria, Sept 2015. (Cited on page 13).
- [3] B. Bank, V. Välimäki, L. Sujbert, and M. Karjalainen. Efficient physics-based sound synthesis of the piano using dsp methods. In *10th European Signal Processing Conference*, pages 1–4, Sept 2000. (Cited on page 23).
- [4] B. Bartlett. A scientific explanation of phasing (flanging). *Journal of the Audio Engineering Society*, 18:674–675, Jun 1970. (Cited on page 118).
- [5] K. Bathe. *Finite Element Procedures*. Prentice-Hall International Series in. Prentice Hall, 1996. ISBN 9780133014587. URL <https://books.google.nl/books?id=wKRRAAAAMAAJ>. (Cited on pages 5 and 133).
- [6] V. Baumgarte, G. Ehlers, F. May, A. Nüchel, M. Vorbach, and M. Weinhardt. Pact xpp a self-reconfigurable data processing architecture. *J. Supercomput.*, 26(2):167–184, September 2003. (Cited on page 28).
- [7] J. A. Belloch, B. Bank, L. Savioja, A. Gonzalez, and V. Välimäki. Multi-channel iir filtering of audio signals using a gpu. In *Proc. IEEE Int. Conf. Acoust. Speech and Signal Processing*, page 6692–6696, Florence, Italy, May 2014. (Cited on page 149).
- [8] S. Bilbao. *Numerical Sound Synthesis: Finite Difference Schemes and Simulation in Musical Acoustics*. Wiley, 1 edition, 2009. (Cited on pages 5, 13, 32, 128, 131, and 133).
- [9] S. Bilbao. Numerical simulation of spring reverberation. In *Proc. of the 16th Int. Conference on Digital Audio Effects (DAFx-13)*, Maynooth, Ireland, Sept 2013. (Cited on page 13).
- [10] S. Bilbao, A. Torin, P. Graham, J. Perry, and G. Delap. Modular physical modeling synthesis environments on gpu. In *Proceedings of the International Computer Music Conference*, 2014. (Cited on page 131).
- [11] H. Bode. History of electronic sound modification. *Journal of the Audio Engineering Society*, 32:730–739, Oct 1984. (Cited on page 118).
- [12] D. A. Bohn. Operator adjustable equalizers: An overview. In *Proc. AES 6th Int. Conf.*, volume 6, pages 369–381, Nashville, TN, May 2006. (Cited on pages 147 and 148).

- [13] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011. ISSN 0001-0782. doi: 10.1145/1941487.1941507. (Cited on page 19).
- [14] R. Boulanger. *The Csound Book*. MIT Press, 2000. (Cited on page 33).
- [15] S. D'Angelo and V. Välimäki. An improved virtual analog model of the moog ladder filter. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 729–733, May 2013. (Cited on page 12).
- [16] J. Dattorro. Effect design. *Journal of the Audio Engineering Society*, 45:660–684, Sept 1997. (Cited on page 118).
- [17] Digilent. Atlys spartan-6 fpga development board. URL <http://www.digilentinc.com/>. (Cited on pages 96 and 138).
- [18] R. W. E. Motuk and S. Bilbao. Fpga-based hardware for physical modeling sound synthesis by finite difference schemes. In *Proceedings of the IEEE field programmable technology conference*, pages 103–110, Singapore, 2005. (Cited on pages 4, 132, and 136).
- [19] R. W. E. Motuk and S. Bilbao. Implementation of finite difference schemes for the wave equation on fpga. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 237–240, Philadelphia, United States, 2005. (Cited on pages 4 and 132).
- [20] G. Evangelista and F. Eckerholm. Player-instrument interaction models for digital waveguide synthesis of guitar: Touch and collisions. *IEEE Transactions on Audio, Speech and Language Processing*, 18(4):822–832, May 2010. (Cited on page 16).
- [21] A. Fettweis. Wave digital filters: Theory and practice. *Proceedings of the IEEE*, 74: 276–327, Februari 1986. (Cited on page 14).
- [22] S. F.L.J and K.Teer. Bucket-brigade electronics - new possibilities for delay, time-axis conversion, and scanning. *IEEE Journal Solid-State Circuits*, SC-4:131–136, June 1969. (Cited on page 118).
- [23] M. Flynn. Flynn's taxonomy, 1966. URL https://en.wikipedia.org/wiki/Flynn's_taxonomy. (Cited on page 61).
- [24] L. Gabrielli, S. Zambon, and F. Fontana. Parallel digital signal processing for efficient piano synthesis. In *23rd European Signal Processing Conference (EUSIPCO)*, pages 2019–2022, Aug 2015. (Cited on page 23).
- [25] F. Gambini, M. Conti, S. Orcioni, F. Ripa, and M. Caldari. Physical modelling in systemc-wms and real time synthesis of electric guitar effects. In *Fifth Workshop on Intelligent Solutions in Embedded Systems*, pages 87–100, June 2007. (Cited on page 15).
- [26] R. Hartenstein. Coarse grain reconfigurable architecture (embedded tutorial). In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, pages 564–570. ACM, 2001. (Cited on page 28).

- [27] Hauck. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. (Cited on page 162).
- [28] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5 edition, 2011. (Cited on page 19).
- [29] P. Heysters. *Coarse-Grained Reconfigurable Processors - Flexibility meets Efficiency*. A, University of Twente, Enschede, September 2004. (Cited on pages 28 and 162).
- [30] M. Holters and U. Zölzer. Graphic equalizer design using higher-order recursive filters. In *Proc. Int. Conf. Digital Audio Effects*, pages 37–40, Montreal, Canada, Sept 2006. (Cited on pages 148 and 154).
- [31] B. Hsu and M. Sosnick-Pérez. Real-time gpu audio. *Commun. ACM*, 56(6):54–62, june 2013. doi: 10.1145/2461256.2461272. URL <http://doi.acm.org/10.1145/2461256.2461272>. (Cited on pages 5 and 131).
- [32] Y. H. Hu, editor. *Programmable Digital Signal Processors - Architecture, Programming and Applications*. CRC Press, 1 edition, December 2001. (Cited on page 22).
- [33] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989. (Cited on page 151).
- [34] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. In *ACM SIGPLAN Notices*, volume 32, pages 318–326, 1997. (Cited on page 33).
- [35] D. A. Jaffe and J. O. Smith. Extensions of the karplus-strong plucked-string algorithm. *Computer Music Journal*, 7(2):56–69, 1983. (Cited on page 126).
- [36] M. S. K. Thulasiraman. *Graphs: theory and algorithms*. Wiley, 1 edition, feb. 1992. (Cited on page 51).
- [37] M. Karjalainen. Efficient realization of wave digital components for physical modeling and sound synthesis. *IEEE Transactions on Audio, Speech, and Language Processing*, 16(5):947–956, July 2008. (Cited on page 14).
- [38] K. Karplus and A. Strong. Digital synthesis of plucked string and drum timbres. *Computer music journal*, 7(2):43–55, 1983. (Cited on pages 14 and 125).
- [39] A. Krukowski and I. Kale. Simulink/matlab-to-vhdl route for full-custom/fpga rapid prototyping of dsp algorithms. In *Proc. Matlab DSP Conf. (DSP'99)*, pages 1–10, 1999. (Cited on pages 33 and 178).
- [40] J. Kuper. Clash, from haskell to hardware, 2015. URL <https://http://www.clash-lang.org/>. accessed June 8. (Cited on pages 132 and 178).
- [41] P. J. M. Laarhoven and E. H. L. Aarts. *Simulated Annealing: Theory and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1987. (Cited on page 86).

- [42] E. Lambers, C. Moerman, P. Kievits, J. Walkier, and R. Woudsma. *Circuits and Systems for Wireless Communications*, chapter R.E.A.L. DSP: Reconfigurable Embedded DSP Architecture for Low-Power/Low-Cost Telecom Baseband Processing, pages 279–287. Springer US, Boston, MA, 2002. (Cited on page 29).
- [43] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proc. Int. Symp. Code Generation and Optimization (CGO 2004)*, pages 75–86, 2004. (Cited on page 33).
- [44] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept. 1987. (Cited on page 41).
- [45] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow program for digital signal processing. *IEEE transactions on computers*, 36(1):24–35, jan. 1987. (Cited on page 40).
- [46] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995. ISSN 0018-9219. doi: 10.1109/5.381846. (Cited on page 39).
- [47] E. A. Lorenz. Deterministic nonperiodic flow. *Journal of the atmospheric sciences*, 20:130–139, March 1963. (Cited on pages 107 and 109).
- [48] J. McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002. (Cited on page 33).
- [49] G. H. Mealy. Mealy machine definition. URL https://en.wikipedia.org/wiki/Mealy_machine. (Cited on page 50).
- [50] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In P. Y. K. Cheung and G. A. Constantinides, editors, *Field Programmable Logic and Application: 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003 Proceedings*, pages 61–70, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. (Cited on page 28).
- [51] J. A. Moorer. Signal processing aspects of computer music: A survey. *Proceedings of the IEEE*, 65(8):1108–1137, Aug 1977. (Cited on page 15).
- [52] E. Motuk, R. Woods, S. Bilbao, and J. McAllister. Design methodology for real-time fpga-based sound synthesis. *IEEE Transactions on Signal Processing*, 55(12):5833–5845, Dec 2007. (Cited on page 26).
- [53] D. Murphy. *Digital Waveguide Mesh Topologies in Room Acoustics Modelling*. University of York, 2000. URL <https://books.google.nl/books?id=D8vFHAACAAJ>. (Cited on pages 14 and 128).
- [54] V. Norilo. Introducing kronos—a novel approach to signal processing languages. In *Proc. Linux Audio Conf.*, pages 9–16, Maynooth, Ireland, 2011. (Cited on pages 4, 33, and 151).
- [55] V. Norilo. Recent developments in the kronos programming language. In *Proc. Int. Computer Music Conf.*, Perth, Australia, 2013. (Cited on pages 33 and 150).

- [56] V. Norilo. Kronos: A declarative metaprogramming language for digital signal processing. *Computer Music Journal*, 39(4):30–48, 2015. (Cited on pages 7, 150, and 152).
- [57] A. V. Oppenheim, A. S. Willsky, and S. H. Nawab. *Signals and Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2 edition, 1996. (Cited on pages 14, 15, 16, 44, 105, 115, and 124).
- [58] Y. Orlarey, D. Fober, and S. Letz. Faust (programming language). URL <https://faust.grame.fr>. accessed Dec. 8, 2013. (Cited on pages 4 and 168).
- [59] Y. Orlarey, D. Fober, and S. Letz. Syntactical and semantical aspects of faust. *Soft Computing*, 8(9):623–632, 2004. (Cited on page 33).
- [60] J. Pakarinen and M. Karjalainen. Enhanced wave digital triode model for real-time tube amplifier emulation. *IEEE Transactions on Audio, Speech, and Language Processing*, 18(4):738–746, May 2010. (Cited on page 14).
- [61] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2007. (Cited on pages 19 and 21).
- [62] J. Pekonen, V. Lazzarini, J. Timoney, J. Kleimola, and V. Välimäki. Discrete-time modelling of the moog sawtooth oscillator waveform. *EURASIP J. Adv. Signal Process.*, 2011:3:1–3:15, Jan. 2011. (Cited on page 12).
- [63] F. Pfeifle. Real-time physical modeling of large instrument geometries using coupled fpgas. In *Proceedings of the Third Vienna Talk on Music Acoustics*, page 267–267, Vienna, Austria, Sept. 2015. (Cited on page 3).
- [64] F. Pfeifle and R. Bader. Real-time finite-difference physical models of musical instruments on a field programmable array (fpga). In *Proceedings of the 15th int. Conference on Digital Audio Effects*, York, UK, 2012. DAFx 2012. (Cited on pages 4, 5, 13, 132, 171, and 173).
- [65] N. Plath, F. Pfeifle, C. Koehn, and R. Bader. Microphone array measurements of the grand piano. In A. Mayer, V. Chatziioannou, and W. Goebel, editors, *Proceedings of the Third Vienna Talk on Music Acoustics*, pages 161–161, Vienna, Austria, Sept 2015. (Cited on page 13).
- [66] Public. The freeverb reverberation algorithm. URL <https://ccrma.stanford.edu/~jos/pasp/Freeverb.html>. (Cited on pages 14 and 129).
- [67] M. Puckette. Pure data: another integrated computer music environment. In *Proc. 1996 Int. Computer Music Conf.*, pages 269–272, 1996. (Cited on page 33).
- [68] R. Rabenstein and L. Trautmann. Digital sound synthesis by physical modelling. In *Proceedings of the 2nd International Symposium on Image and Signal Processing and Analysis*, pages 12–23. ISPA 2001, 2001. (Cited on page 13).
- [69] R. Rabenstein, S. Petrausch, A. Sarti, G. D. Sanctis, C. Erkut, and M. Karjalainen. Blocked-based physical modeling for digital sound synthesis. *IEEE Signal Processing Magazine*, 24(2):42–54, March 2007. (Cited on page 13).

- [70] J. Rämö and V. Välimäki. Optimizing a high-order graphic equalizer for audio processing. In *IEEE Signal Process. Lett.*, volume 21, pages 301–305, Mar 2014. (Cited on page 148).
- [71] J. Rämö, V. Välimäki, and B. Bank. High-precision parallel graphic equalizer. In *IEEE/ACM Trans. Audio Speech Language Processing*, volume 22, pages 1894–1904, 2014. (Cited on pages 147, 148, 149, 150, and 154).
- [72] Recore-Systems. Why use the xentium dsp ip core? Webinar, 2013. URL <http://www.design-reuse.com/video/view/73/xentium-dsp-ip-core.html>. (Cited on pages 163 and 164).
- [73] P. A. Regalia and S. K. Mitra. Tunable digital frequency response equalization filters. *IEEE Trans. Acoustics, Speech and Signal Processing*, 35:118–120, Jan. 1987. (Cited on page 154).
- [74] C. Roads. *The Computer Music Tutorial*. MIT Press, Cambridge, MA, 1996. (Cited on page 33).
- [75] R. Saar, G. Levy, and J. Fainguelernt. Implementing physical models of musical instruments in the tms320c6748. In *European Embedded Design in Education and Research Conference (EDERC)*, volume 6, pages 114–118, Sept 2014. (Cited on page 15).
- [76] O. H. Schmitt. A thermionic trigger. *Journal of Scientific Instruments*, 15(1), 1938. (Cited on page 107).
- [77] B. Schottstaedt. Machine tongues xvii: Clm: Music v meets common lisp. *Computer Music Journal*, pages 30–37, 1994. (Cited on page 33).
- [78] M. R. Schroeder. Natural-sounding artificial reverberation. *Journal of the Audio Engineering Society*, 10(3):219 – 223, 1962. (Cited on pages 14 and 129).
- [79] M. R. Schroeder and B. F. Logan. Colorless artificial reverberation. *IRE Trans. on Audio*, AU-9:209–214, Nov 1961. (Cited on page 129).
- [80] J. O. Smith. The biquad section, . URL https://ccrma.stanford.edu/~jos/filters/BiQuad_Section.html. (Cited on pages 56 and 189).
- [81] J. O. Smith. Zita-rev1 reverberation, . URL https://ccrma.stanford.edu/~jos/pasp/Zita_Rev1.html. (Cited on pages 14 and 138).
- [82] J. O. Smith. An allpass approach to digital phasing and flanging. In *Proceedings of the 1984 International Computer Music Conference*, pages 103–109, Paris, 1984. (Cited on page 118).
- [83] J. O. Smith. A new approach to digital reverberation using closed waveguide networks. In *Proc. Int. Computer Music Conf.*, pages 47–53, Vancouver, Canada, Sept 1985. (Cited on page 14).
- [84] J. O. Smith. Physical modeling using digital waveguides. *Computer Music Journal*, 16 (4):74–91, Winter 1992 1992. Special issue: Physical Modeling of Musical Instruments, Part I. (Cited on pages 4 and 13).

- [85] J. O. Smith. *Physical Audio Signal Processing for virtual musical instruments*. W3K Publishing, 1 edition, 2010. (Cited on pages 3, 4, 12, 13, 16, and 131).
- [86] C. Sodsri, R. Yutipong, and T. Boonrat. A real-time khim sound synthesizer using iir resonator implemented with a dsp board. In *7th International Conference on Information Technology and Electrical Engineering (ICITEE)*, volume 7, pages 317–321. ICITEE, Oct 2015. (Cited on page 23).
- [87] G. L. Steele. *Common Lisp*, volume 2. Digital Press, 1984. (Cited on page 33).
- [88] STMicroelectronics. Stm32f405zgt6 datasheet. Internet. URL <http://www.st.com/web/en/resource/technical/document/datasheet/DM00037051.pdf>. (Cited on page 167).
- [89] O. team. Introducing the owl. Internet. URL <http://hoxtonowl.com/>. (Cited on pages 21 and 167).
- [90] R. Trausmuth, C. Dusek, and Y. Orlarey. Using faust for fpga programming. In *Proc. of the 9th Int. Conference on Digital Audio Effects (DAFx-06)*, pages 287–290, Sept. 2006. (Cited on page 27).
- [91] L. Trautmann and R. Rabenstein. *Digital Sound Synthesis by Physical Modeling Using the Functional Transformation Method*. Springer, 2003 edition, 2003. ISBN-10: 0306478757. (Cited on page 3).
- [92] J. Travis and J. Kring. *LabVIEW for Everyone: Graphical Programming Made Easy and Fun*. Prentice Hall PTR, 2006. (Cited on page 32).
- [93] V. Välimäki, M. Karjalainen, Z. Janosy, and U. Laine. A real-time dsp implementation of a flute model. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 249–252. ICASSP-92, March 1992. (Cited on page 23).
- [94] V. Välimäki, J. Pakarinen, C. Erkut, and M. Karjalainen. Discrete-time modelling of musical instruments. *Reports on progress in physics*, 69(1), 2005. (Cited on page 15).
- [95] V. Välimäki, S. Bilbao, J. O. Smith, J. S. Abel, J. Pakarinen, and D. Berners. *DAFX: Digital Audio Effects*, chapter Virtual analog effects, pages 473–522. Wiley, second edition edition, 2011. (Cited on pages 15 and 16).
- [96] V. Välimäki, J. Parker, L. Savioja, J. O. Smith, and J. S. Abel. Fifty years of artificial reverberation. *IEEE Trans. on Acoustics, Speech and Language Proc.*, 20(5):1421–1448, 2012. doi: 10.1109/TASL.2012.2189567. (Cited on pages 13, 14, 16, and 129).
- [97] F. Van Nee. To a new hardware design methodology: A case study of the cochlea model. Master's thesis, University of Twente, March 2014. (Cited on page 132).
- [98] A. Varghese, B. Edwards, G. Mitra, and A. P. Rendell. Programming the adapteva epiphany 64-core network-on-chip coprocessor. In *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 984–992, Washington, DC, USA, 2014. IEEE Computer Society. (Cited on page 29).

- [99] A. Varghese, B. Edwards, G. Mitra, and A. P. Rendell. Programming the adapteva epiphany 64-core network-on-chip coprocessor. *Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, pages 984 – 992, May 2014. ISBN:978-1-4799-4117-9. (Cited on pages 5 and 30).
- [100] K. Walters. *Versatile architectures for onboard payload signal processing*. PhD thesis, University of Twente, Enschede, October 2013. (Cited on pages 163 and 164).
- [101] K. J. Werner, J. S. Abel, and J. O. Smith. A physically-informed, circuit-bendable, digital model of the roland tr-808 bass drum circuit. In *Proc. of the 17th Int. Conference on Digital Audio Effects (DAFx-14)*, Erlangen, Germany, September 2014. (Cited on page 15).
- [102] Xilinx. Xdata ml605 evaluation board. Internet, . URL <http://www.xilinx.com/products/boards-and-kits/ek-v6-ml605-g.html>. (Cited on page 172).
- [103] Xilinx. Virtex6 datasheet. Internet, . URL http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf. (Cited on page 172).
- [104] D. Yeh. *Digital implementation of musical distortion circuits by analysis and simulation*. PhD thesis, Stanford University, CCRMA, June 2009. (Cited on pages 15 and 142).
- [105] D. T. Yeh. Automated physical modeling of nonlinear audio circuits for real-time audio effects; part ii: Bjt and vacuum tube examples. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(4):1207–1216, May 2012. (Cited on page 15).
- [106] D. T. Yeh, J. S. Abel, and J. O. Smith. Simplified physically-informed models of distortion and overdrive guitar effects pedals. In *Proc. of the 10th Int. Conference on Digital Audio Effects (DAFx-07)*, Bordeaux, France, September 2007. (Cited on pages 13, 15, and 142).
- [107] D. T. Yeh, J. S. Abel, and J. O. Smith. Automated physical modeling of nonlinear audio circuits for real-time audio effects part i: Theoretical development. *IEEE Transactions on Audio, Speech, and Language Processing*, 18(4):728–737, May 2010. (Cited on page 15).
- [108] Z. Yu, M. J. Meeuwsen, R. W. Apperson, O. Sattari, M. A. Lai, J. W. Webb, E. W. Work, T. Mohsenin, and B. M. Baas. Architecture and evaluation of an asynchronous array of simple processors. *Journal of Signal Processing Systems*, 53(3):243–259, 2008. (Cited on pages 28 and 30).
- [109] U. Zölzer. *DAFX: Digital Audio Effects*. Wiley, 2 edition, 2009. (Cited on pages 2, 56, 112, and 131).

LIST OF PUBLICATIONS

213

- [MJW:1] G.E.A. Lousberg and M.J.W. Verstraelen. Patent: Self-test for integrated memories, 1998.
- [MJW:2] C.M. Moerman and M. Verstraelen. Patent: Architecture for a multi-port cache memory, 2008.
- [MJW:3] Vesa Norilo, Math Verstraelen, and Vesa Välimäki. Implementing a low-latency parallel graphic equalizer with heterogeneous computing. In *Proc. of the 18th Int. Conference on Digital Audio Effects (DAFx-15)*, Trondheim, Norway, Nov 30 - Dec 3 2015. DAFx.
- [MJW:4] Manoj Sachdev and Math Verstraelen. Development of fault model and test algorithms for embedded drams. In *Proceedings of the IEEE International Test Conference on Designing, Testing, and Diagnostics*, pages 815–824. IEEE Computer Society, 1993.
- [MJW:5] M.J.W. Verstraelen, F. Pfeifle, and R. Bader. Feasibility analysis of real-time physical modeling using wavecore processor technology on fpga. In *Proceedings of the 3rd Vienna Talk on Music Acoustics*, pages 273–280, Sept. 2015. Invited paper.
- [MJW:6] M.J.W. Verstraelen, G.J.M. Smit, and J. Kuper. Declaratively programmable ultra-low latency audio effects processing on fpga. In *Proceedings of the 17th International Conference on Digital Audio Effects (DAFx-14)*, pages 263–270, 2014.

THIS THESIS

```
@phdthesis{verstraelen2017:thesis,  
  author={Verstraelen, Math},  
  title={The WaveCore - A Scalable Architecture for Real-time Audio  
    Processing},  
  school={University of Twente},  
  address={PO Box 217, 7500 AE Enschede, The Netherlands},  
  year={2017},  
  month=jan,  
  day={20},  
  isbn={978-90-365-4240-1},  
  doi={10.3990/1.9789036542401}  
}
```

INDEX

215

A

ACU, 69
ADC, 5
ALU, 18
ASIC, 17

B

BBD, 118
BibTeX of this thesis, 213
BSP, 96

C

CGRA, 19
CLB, 24
CPU, 18

D

DAC, 5
DAFx, 2
DAG, 51
DCT, 31
DCU, 160
DDR, *see also* SDRAM, 19
DFG, 160
DLL, 25
DMA, 29
DSP, 4
DWG, 13

E

EDA, 14
EOT, 69

F

FDTD, 5
FFCF, 129
FFT, 22
FIFO, 30
FIR, 22

FLOP, 166
FPGA, 4
FPU, 167

G

GALS, 31
GEQ, 12
GIU, 62
GPGPU, 131
GPN, 63
GPP, 4

H

HDL, 24
HPI, 61
HW, 22

I

I/O, 95
ibFIFO, 68
IIR, 56
ILP, 18
interconnect, *see also* NoC
IP, 25
ISA, 8

L

LFO, 107
LSU, 62
LTI, 14
LUT, 24

M

MAC, 22
MIDI, 169
MIMD, 37

N

NoC, *see also* interconnect, 28

NOP, 170

O

obFIFO, 68
ODE, 110

P

PA, 50
PC, 69
PCB, 24
PCIe, 173
PCORE, 48
PCU, 69
PDE, 133
PLL, 197
PMC
 annotations, *see* annotations
PT, 47
PU, 61

R

RAM, 24
ReMap, 2
RISC, 29
RTL, 25

S

SDF, 34
SDRAM, *see also* DDR
SIMD, 23
SoC, 24
SW, 23

T

TCU, 76
TP, 28

U

ugen, 33
USB, 25

V

VCO, 115
VLIW, 23

W

WBB, 69

WDF, 14
WOF, 80
WPG, 41
WPP, 50

ISBN 978-90-365-4240-1

9 789036 542401

STELLINGEN

behorende bij het proefschrift

The WaveCore - A Scalable Architecture for Real-time Audio Processing

door Math Verstraelen,
te verdedigen op vrijdag 20 januari 2017

- 1 — Het optimaliseren naar lokaliteit is een doeltreffende strategie voor de ontwikkeling van een real-time data-flow processor. *(Hoofdstuk 3)*
- 2 — De delay-lijn, gekoppeld aan een primitieve rekenfunctie, blijkt goed toepasbaar voor het audio applicatie domein. *(Hoofdstuk 3 en 4)*
- 3 — Een processor die vanuit goed gebruik van dataflow principes ontwikkeld is behoort een efficiënte compile target te zijn voor functionele programmeer talen. *(Hoofdstuk 5)*
- 4 — Het is relatief eenvoudig om een dataflow graaf af te beelden op een multi-core processor. Een sterk complicerende factor ontstaat wanneer hier real-time eisen aan gekoppeld worden, daar dit de vrijheidsgraden voor het afbeelden aanzienlijk beperkt. *(Hoofdstuk 3)*
- 5 — Omdat EDA tools over het algemeen niet goed in staat zijn om hardware resources temporeel te scheduleren, wordt de rekencapaciteit van FPGAs vaak niet optimaal benut bij gebruik van deze tools.
- 6 — Technologische evolutie is een bovenmenselijk proces.
- 7 — Audio bugs zijn onder te brengen in families. Voor iedere bug familie blijkt dat deze vaak een uniek herkenbaar stemgeluid hebben, en daardoor te identificeren zijn.
- 8 — De introductie van de rekentoets in het voortgezet onderwijs is het gevolg van structureel te weinig beta-mensen in de politiek.
- 9 — Kinderen met een vorm van autisme hebben vaak uitzonderlijk sterke eigenschappen, zoals analytisch en logisch denkvermogen. Het zou voor deze kinderen beter zijn om in ons leersysteem deze eigenschappen extra te stimuleren, in plaats van het corrigeren van hun afwijkend gedrag.
- 10 — Iemand die zelden valt kan waarschijnlijk niet goed lopen.