Anja Niedermeier

A Fine-Grained Parallel Dataflow-Inspired Architecture for Streaming Applications

UNIVERSITY OF TWENTE.

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

CTIT

CTIT Ph.D. Thesis Series No. 14-322
Centre for Telematics and Information Technology
PO Box 217, 7500 AE Enschede, The Netherlands

STARS
Sensor Technology Applied in Reconfigurable Systems

This research is conducted as part of the Sensor Technology Applied in Reconfigurable systems for sustainable Security (STARS) project (www.starsproject.nl)

This thesis was typeset using LaTeX, Ti*k*Z, and GNU Emacs. This thesis was printed by Gildeprint, The Netherlands.

ISBN 978-90-365-3732-2
ISSN 1381-3617; CTIT Ph.D. Thesis Series No. 14-322
DOI 10.3990./1.9789036537322

A Fine-Grained Parallel Dataflow-Inspired Architecture for Streaming Applications

Dissertation

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
Prof. Dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Friday 29th August, 2014 at 12:45

by

Anja Niedermeier

born on October 25th, 1982
in Böblingen, Germany

ISBN 978-90-365-3732-2

Abstract

Data-driven streaming applications are quite common in modern multimedia and wireless applications, like for example video and audio processing. The main components of these applications are Digital Signal Processing (DSP) algorithms.

These algorithms are not extremely complex in terms of their structure and the operations that make up the algorithms are fairly simple (usually binary mathematical operations like addition and multiplication). What makes it challenging to implement and execute these algorithms efficiently is their large degree of fine-grained parallelism and the required throughput.

DSP algorithms can usually be described as dataflow graphs with nodes corresponding to operations and edges between the nodes expressing data dependencies. On the edges, data travels in the form of tokens. A node *fires* as soon as all required input data has arrived at its input edge(s). One firing consists of consuming the input data (i.e. input tokens), executing the desired operation, and producing the output data (i.e. output tokens). Usually, input data to the dataflow graph is provided as a stream of tokens. As a consequence, a well-behaved dataflow graph keeps executing as long as input data arrives.

To execute DSP algorithms efficiently while maintaining flexibility, coarse-grained reconfigurable arrays (CGRAs) can be used. CGRAs are composed of a set of small, reconfigurable cores, interconnected in e.g. a two dimensional array. Each core by itself is not very powerful, yet the complete array of cores forms an efficient architecture with a high throughput due to its ability to efficiently execute operations in parallel.

To program CGRAs, usually an architecture-specific subset of C is defined which is then used to specify and implement algorithms on the respective CGRA. However, the C programming paradigm was not developed to specify algorithms that contain a large degree of fine-grained parallelism. Instead, it was designed to implement sequential algorithms on single-core architectures.

In this thesis, we present a CGRA targeted at data-driven streaming DSP applications that contain a large degree of fine-grained parallelism, such as matrix manipulations or filter algorithms. Along with the architecture, also a programming language is presented that can directly describe DSP applications as dataflow graphs which are then automatically mapped and executed on the architecture.

In contrast to previously published work on CGRAs, the guiding principle and inspiration for the presented CGRA and its corresponding programming paradigm is the dataflow principle. Three main aspects can be named here:

1. A DSP algorithm is represented as a dataflow graph with nodes corresponding to operations and edges between the nodes corresponding to data dependencies.
2. The configuration and execution principles of the cores in the architecture are based on dataflow principles, i.e. a core starts its execution based on the availability of data (i.e. availability of input tokens).
3. Dataflow graphs can be explicitly expressed in the programming language.

The presented architecture is a CGRA with small, reconfigurable cores which communicate via point-to-point links. Each core is independent from its neighbours, i.e. there is no central entity controlling the complete array, instead, control is local to each core. The execution mechanism of the cores in the architecture is data-driven, i.e. it adopts the firing rule known from dataflow. A core starts its execution based on the availability of input data. Hence, no fixed schedules and no program counters are required, which makes the presented CGRA fundamentally different from previously presented CGRAs that rely on an imperative programming paradigm.

The architecture has been implemented using CλaSH, a hardware description language and compiler that can generate synthesisable VHDL code from a Haskell specification. Describing hardware with CλaSH enables a designer to describe hardware in terms of its structure.

The programming language for the presented architecture can describe a DSP algorithm as a dataflow graph. The grammar of the language resembles a dataflow structure, i.e. it contains constructs for dataflow nodes which are used to construct dataflow graphs. The language is implemented as an embedded language in Haskell. Therefore, Haskell's powerful features like recursion and higher order functions can be used. This is very beneficial for describing an algorithm in terms of its structure and data dependency, in particular for representing the fine-grained parallelism as present in the targeted application domain. By using the same design language for both the architecture and the programming language, no switching between environments is required and the same type definitions can be used.

The result of this work is a completely integrated framework targeted at streaming DSP algorithms, consisting of a CGRA, a programming language and a compiler. The complete system is based on dataflow principles, in particular the firing rule, i.e. execution is triggered by the availability of input data, not determined by a fixed schedule. We evaluate the framework by implementing a number of commonly used DSP algorithms, e.g. a FIR filter, a dot product and an FFT kernel, on the architecture using the presented programming language. We conclude that by using an architecture that is based on dataflow principles and a corresponding programming paradigm that can directly express dataflow graphs, DSP algorithms can be implemented in a very intuitive and straightforward manner.

Samenvatting

Data gedreven stromende applicaties zijn te vinden in moderne multimedia en draadloze toepassingen, zoals bijvoorbeeld bij video en audio verwerking. De grootste componenten binnen dergelijke applicaties zijn algoritmes voor digitale signaalverwerking (DSP, Eng: *Digital Signal Processing*).

Dit soort algoritmes is niet complex qua structuur en operaties binnen deze algoritmes zijn vrij simpel (meestal zijn dit binaire operatoren zoals optelling en vermenigvuldiging). De uitdaging bij het implementeren en efficiënt uitvoeren van dit soort algoritmes is het benutten van de hoge mate van fijnmazig parallellisme en het behalen van de vereiste doorvoersnelheid.

DSP-algoritmes kunnen meestal worden beschreven als *dataflow*-grafen waarbij *nodes* operaties voor stellen en de *edges* tussen de nodes de gegevensafhankelijkheden tussen operaties. Data worden doorgegeven via een edge in de vorm van *tokens*. Een node *vuurt* zodra alle vereiste invoer beschikbaar is op de inkomende edges. Een vuring bestaat uit het consumeren van de invoer (de inkomende tokens), het uitvoeren van de bijbehorende operatie en ten slotte het produceren van uitvoer (uitgaande tokens). De invoer voor een dataflow graaf bestaat uit een stroom van tokens. Als gevolg zal een correcte dataflow-graaf uitgevoerd worden zolang er invoer beschikbaar is.

Om DSP-algoritmes efficiënt te kunnen uitvoeren met behoud van flexibiliteit kunnen grofmazige herconfigureerbare *arrays* (CGRA, Eng: *Coarse-Grained Reconfigurable Arrays*) gebruikt worden. CGRA's bestaan uit kleine, herconfigureerbare rekenkernen welke aan elkaar verbonden zijn in bijvoorbeeld een tweedimensionale reeks. Alhoewel elke kern op zichzelf niet bijster krachtig is, vormt de complete reeks een efficiënte architectuur met een hoge doorvoersnelheid door operaties parallel uit te voeren.

Om CGRA's te programmeren wordt meestal een architectuur-specifieke deelverzameling van C gedefinieerd welke gebruikt kan worden om algoritmes voor de betreffende CGRA te implementeren. Echter, het programmeerparadigma van C is niet ontworpen voor het specificeren van algoritmes met een hoge mate van fijnmazig parallellisme maar voor sequentiële algoritmes voor architecturen met een enkele rekenkern.

In dit proefschrift presenteren we een CGRA voor data gedreven stromende DSP-applicaties met een hoge mate van fijnmazig parallellisme, zoals bij matrix bewerkingen of filter algoritmes. Behorende bij de architectuur presenteren we een programmeertaal voor het beschrijven van DSP-applicaties als dataflow-grafen welke automatisch afgebeeld en uitgevoerd kunnen worden.

In tegenstelling tot eerder gepubliceerde werken over CGRA's is het principe voor de gepresenteerde CGRA en bijbehorende programmeerparadigma gebaseerd op het dataflow principe. Drie hoofdaspecten zijn:

1. Een DSP-algoritme is beschreven als een dataflow-graaf waarbij nodes overeenkomen met operaties en de edges tussen nodes overeenkomen met gegevens-afhankelijkheden.
2. De principes bij configuratie en uitvoering op de rekenkernen in de architectuur zijn gebaseerd op dataflow-principes; een rekenkern begint met uitvoeren zodra alle benodigde invoer-tokens beschikbaar zijn.
3. Dataflow-grafen kunnen expliciet worden uitgedrukt in de programmeertaal.

De gepresenteerde architectuur is een CGRA met kleine, herconfigureerbare rekenkernen welke via punt-naar-punt verbindingen communiceren. Elke rekenkern is onafhankelijk van zijn buren; er is geen centrale besturing voor de hele reeks omdat elke rekenkern zelfstandig kan handelen. Het uitvoeringsmechanisme van de rekenkernen is data-gestuurd omdat de vuringsregels van dataflow worden gevolgd. Een rekenkern begint met zijn uitvoering zodra alle benodigde invoer beschikbaar is. Hierdoor is er geen vast schema voor de uitvoering van alle taken en zijn geen programma-stappentellers nodig waardoor de gepresenteerde CGRA fundamenteel anders is dan voorheen gepresenteerde CGRA's welke gebruik maken van een imperatief programmeerparadigma.

De architectuur is geïmplementeerd met CλaSH, een hardware beschrijvingstaal en vertaler die synthetiseerbare VHDL-broncode kan genereren van een Haskell specificatie. Het beschrijven van hardware met behulp van CλaSH geeft een ontwerper de mogelijkheid om hardware uit te drukken qua structuur.

De programmeertaal voor de gepresenteerde architectuur kan een DSP-algoritme beschrijven als een dataflow-graaf. De grammatica van de programmeertaal lijkt op een dataflow-structuur; het bevat constructies voor het beschrijven van dataflow nodes welke in de constructie van een graaf gebruikt worden. De programmeertaal is geïmplementeerd als een geëmbedde taal in Haskell. Hierdoor kunnen krachtige mogelijkheden uit Haskell zoals recursie en hogere-ordefuncties gebruikt worden. Dit is zeer gunstig voor het beschrijven van een algoritme qua structuur en gegevens afhankelijkheid, met name voor fijnkorrelig parallellisme zoals aanwezig in het beoogde toepassingsgebied. Door het gebruik van dezelfde ontwerptaal voor zowel de architectuur en de programmeertaal is geen omschakeling tussen omgevingen vereist en kunnen dezelfde type definities gebruikt worden.

Het resultaat van dit werk is een compleet geïntegreerd raamwerk voor stromende DSP-algoritmes bestaande uit een CGRA, programmeertaal en een compiler. Het hele systeem is gebaseerd op dataflow principes, met name de vuringsregel waarbij uitvoering wordt gestart bij de beschikbaarheid van invoergegevens en niet volgens een vast schema. We evalueren het raamwerk door middel van de implementatie van een aantal gangbare DSP-algoritmes zoals een FIR-filter, een *dot product* en een FFT-*kernel* op de gepresenteerde architectuur en met behulp van de gepresenteerde programmeertaal. We concluderen dat het gebruik van een architectuur, gebaseerd op dataflow-principes en bijbehorend programmeerparadigma voor het uitdrukken van dataflow-grafen, zorgt voor een intuïtieve en ongecompliceerde aanpak voor het implementeren van DSP-algoritmes.

Acknowledgements

Now that this thesis is almost finished with only a few last bits and pieces to be finished, it is finally time to write the acknowledgements. What a rewarding moment after more than four years of work!

Whenever people ask me how I ended up in Twente of all places, the story I have to tell them is not very straightforward. In 2002, I started to study Electrical Engineering at the University of Karlsruhe, and in 2006, I went as an Erasmus student to Trondheim, Norway, for a year. During that year I decided to stay in Trondheim and finish my studies there. At the end of the two-year masters in Trondheim, I got the opportunity to do my M.Sc. project at IMEC in Eindhoven. During my stay at IMEC, I decided that I would like to stay in research a little longer and try to pursue a PhD. My supervisor at IMEC heard of that and hinted that the CAES group at the University of Twente was looking for PhD students. I then sent an open application to Gerard Smit, who invited me over for an interview and on the next day I got the offer to start a PhD in Twente, which I was really happy about. And that is how I ended up in Twente.

My initial research topic in Twente was *off-chip communication*. But after a while it became clear to me that this was not really a topic I was particularly interested in so I decided to switch to something else. I discussed the matter with Gerard Smit and André Kokkeler and luckily they gave me the freedom to look for a new topic myself. After talking to a few people (especially Kenneth Rovers) and reading a bit I decided that I wanted to investigate dataflow related architectures, a topic which eventually resulted in this thesis. At this point I also want to thank Gerard for giving me the freedom to follow my interests during my PhD and investigate this really cool field of architecture design with dataflow principles.

While I was looking for a new research topic, Jan Kuper gently pointed me towards the wondrous world of functional programming. Even though I had heard about that programming paradigm at some point during my studies, I never actually worked with it. But since Jan (and a few more of the group) seemed to be very convinced of it I got curious. As a result, a great part of this thesis was written using Haskell, and, even though it sometimes gave me a headache, it was certainly fun to work with. So, at this point, a big thanks to Jan for showing me a new perspective to the world of programming and of course also thanks to the many discussions

we had, both on topic and off topic.

In 2010 I joined P-NUT, the PhD network of the University of Twente, for a few activities and eventually joined their board in 2011. Not only was this a lot of fun, but it was also a valuable experience and I met a lot of cool people in the board of P-NUT and during all our activities. I certainly miss being in the board and would like to thank all of you who I had the pleasure of organising stuff together or just having fun with in the last years!

In the beginning of 2012, I went back to Trondheim for three months. What happened between March 18 to 20 is still hard to grasp and even harder to accept. Nature can be cruel, without mercy and unpredictable and technology cannot always save you. Florian, even though you are not here anymore, I want to thank you for the time we had and all the inspiration you gave me. Without you I would not be who I am now.

I am very grateful for the enormous amount of support of friends and colleagues I received afterwards, and I really want to thank all of you. Yahya, Sven, Sylvie and Kata, your company meant a great deal to me in the few remaining days in Trondheim. Back in Twente, my colleagues from the CAES group were really nice and helpful, which is certainly not something to take for granted. In particular I want to thank Robert, Jan, Gerard, Marlous, Philip, Rinse, Koen and Berend. Besides my colleagues, I really want to thank Silja, Christina and Sarah for being there for me so many times. It would have been much more difficult without you.

But despite everything, I had a nice time doing my PhD in the CAES group, and two really nice office mates, Robert and Timon. I had the opportunity to travel to nice conferences and summer schools, and learned a lot about the academic life. I would like to thank Gerard Smit, André Kokkeler and Jan Kuper for their guidance and discussions during my PhD. And of course, a thesis cannot be written without support. So at this point, I would like to thank our secretaries for their help, and a big thanks to Christiaan for all the CλaSH support, to Philip for the help and loads of little hints concerning LaTeX and to Marco for all the little but very important tips for the last bits and pieces of the thesis itself. Also I would like to especially thank Silja and Robert for being my paranymphs during my defense, it means a lot to me.

And of course I would like to thank a very important person in my life, Berend. Thanks for your support in finishing up this thesis and for so many other things, not only for translating the abstract to Dutch :)

Schließlich möchte ich mich noch von ganzem Herzen bei meinen Eltern bedanken, die mich immer ermuntert und unterstützt haben, egal in welche Stadt oder welches Land ich ging. Ihr habt mir stets das Gefühl gegeben dass alles machbar ist und mir nie Steine in den Weg gelegt.

Anja Niedermeier
Enschede, August 2014

Contents

Chapter 1

Introduction

Data-driven streaming applications cover a broad range of applications and are nowadays ubiquitous. Common examples are video and audio streaming, which are being used by many people on a daily basis. In this thesis, we will develop a system (namely a programmable hardware architecture) for the efficient execution of data-driven streaming applications. We hereby consider three aspects of efficiency, namely *programmability*, *flexibility* and *energy efficiency* and select the type of architecture with the best balance of all three criteria.

In the term *programmability* we include all the required steps to map a desired algorithm onto a certain type of architecture. This includes the type of language that the architecture supports, the variety of supported languages, but also the availability of development tools and libraries.

By *flexibility* we mean how easy the architecture can be adapted to a different purpose or application. It might be only a matter of writing new software, but it might also involve a complete and cumbersome redesign requiring many verification steps.

Energy efficiency relates to the energy consumption to perform a certain task.

Since there is no such thing as *one ideal* architecture which is most suited for data-driven streaming applications, we will compare a number of architecture types. Each type of architecture has certain advantages, but also certain shortcomings.

Some important types of architectures that are currently available are:

» Application-Specific Integrated Circuits (ASIC),
» Field-Programmable Gate Arrays (FPGAs),
» Coarse-Grained Reconfigurable Arrays (CGRAs), and
» General Purpose Processors (GPP)

We compare the different types of architectures in terms of their respective advantages and shortcomings for the domain of data-driven applications in the remainder of this section. Besides the mentioned architectures more types exist, like e.g. **G**raphic **P**rocessing **U**nits (GPUs) or **A**pplication-**S**pecific **I**nstruction-Set **P**rocessors (ASIPs). However, it would be out of scope of this thesis to perform a full comparison of all available types of architectures.

In Figure 1.1, an illustration of the comparison of the four different types of architectures in terms of the above mentioned three criteria of efficiency is shown. A high value on an axis means that the respective architecture scores high for the respective criterion, a value close to the origin of the graph indicates a low score.

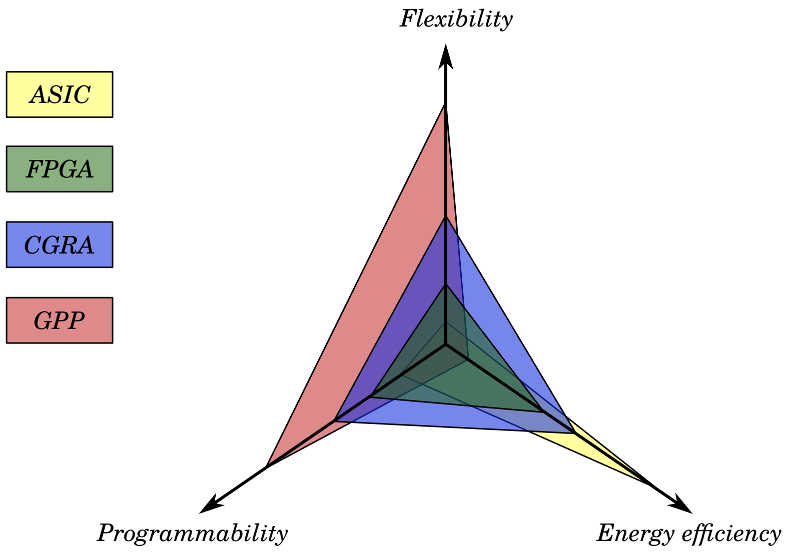


Figure 1.1 – Characteristics for different types of architectures

The illustration shows that *GPPs* are most flexible and easiest to program, but they are not very energy efficient. This is not surprising, since *GPPs* are, as the name suggests, designed for a great variety of tasks and application areas. On the other end of the spectrum are *ASICs*, which are neither flexible nor programmable, but very energy efficient since they are usually designed for a very specific purpose. *FPGAs* are more flexible than *ASICs*, but at the cost of energy efficiency since they can be reconfigured as they are mainly used for prototyping. *CGRAs* are usually targeted at a certain application domain, often signal processing, but are still programmable. This places them in the area between reconfigurable hardware and generic processors. Because *CGRAs* are at the same time flexible, energy-efficient and programmable, we focus this thesis on *CGRAs*.

Programmability is an important benefit of CGRAs. In Figure 1.2, we illustrate how the different types of architectures are programmed. The big circles indicate the support by high level programming languages, the small dots indicate support by low level programming languages.

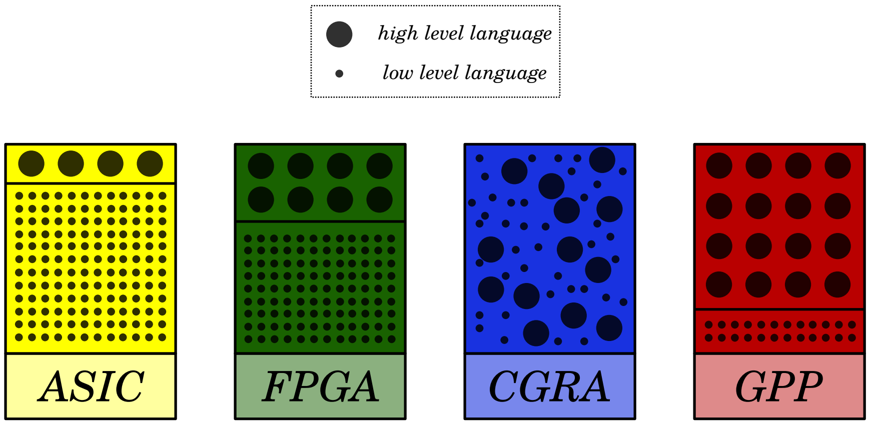


Figure 1.2 – Programmability of different types of architectures

ASICs are mainly programmed (or rather designed) using low level languages like VHDL or Verilog. Additionally, limited support for high level synthesis from high level languages like C is available, but not widespread used. Hence, a major part of the design process is performed using a low level language and is therefore cumbersome, time consuming and error prone.

Similarly to *ASICs*, *FPGAs* are mainly programmed using low level languages. However, the tool support for high level synthesis is more mature, since FPGA vendors can optimise the generated code for the respective FPGA.

GPPs on the other hand are almost exclusively programmed using high level languages. Many programming languages, compilers, libraries, and tools have been developed over the years. If required, it is also still possible to use low level languages like assembly to program *GPPs*.

The programmability of *CGRAs* cannot be as easily classified as for the other types of architectures. Mainly, because *the* CGRA does not exist. There are many different implementations of CGRAs, each with their own programming paradigm. In Chapter 2, we will elaborate on that further. In Figure 1.2, this is illustrated by a random distribution of high and low level languages.

1.1 Research goal

1.1.1 Architecture

Since we target data-driven streaming applications that contain fine-grained instruction level parallelism, the architecture itself also should be of a fine-grained parallel nature. We identified CGRAs to be a suitable class of architectures for our purposes.

CGRAs are composed of an array of small, configurable cores, often in combination with a general purpose processor for control operations. The cores in the CGRA usually contain an ALU and a small local memory. The control of the CGRA can be either centralised for the complete array (meaning there is a central control unit in the array), or local to each core (meaning there is a control unit in each core).

1.1.2 Target application domain

The presented system is targeted at data-driven streaming algorithms in the digital signal processing (DSP) domain. The algorithms for which the system is designed contain a large degree of fine-grained instruction-level parallelism. Those algorithms are commonly found in audio or video processing, for example in the form of matrix manipulations or filtering operations. The elementary operations in those algorithms are usually simple, e.g. additions and multiplications.

We consider DSP applications that have the structure of a dataflow graph, with nodes representing the operations, and arcs between the nodes representing communication between the nodes, i.e. the data dependencies. As DSP algorithms are usually stream based, the incoming and outgoing data is available as a stream of tokens.

The complete system is therefore inspired by the dataflow paradigm. That means the architecture, but also the programming language for that architecture should be based on dataflow principles to have a close relation between the target application domain and the actual system.

1.1.3 Programming of the system

We consider programmability (i.e. the development of an intuitive, easy to use programming paradigm) of CGRAs a crucial challenge. As we will present in Chapter 2, previously published CGRAs are either programmed using (an architecture specific subset of) C, or a low level assembly-like language.

Experience shows however that programming CGRAs (and multicore architectures in general) is known to be a tedious, difficult and error prone task. No satisfying programming paradigm has been developed yet. A lot of research is being put into the development of an efficient and at the same time easy and intuitive to use programming paradigm.

Programming an architecture should be tightly connected to the way the algorithms are composed and described. That means, the streaming and dataflow mechanisms should be supported by the programming paradigm that is close to the mathematics of DSP applications. The programming language should enable the designer to easily describe an algorithm in terms of its structure. Also, the language should both support low level instructions (like addition or multiplication) as well as higher-level instructions, i.e. to describe regular structure.

1.1.4 Design of the complete system

Designing a complete system consisting of a hardware architecture and a programming language and compiler usually involves the use of multiple design languages and environments. Mostly, hardware is designed using a hardware description language like VHDL or Verilog, whereas the programming language and compiler are usually designed in a completely different language, e.g. C/C++. If in addition also a simulation framework is required, yet another design language is used. This makes the integration of the various layers in the system a very complex task.

In this work, we will use an approach to system design which only involves *one* language. By using the same design environment for all parts of the design process, the same type definitions can be used in the hardware architecture and in the compiler. Also, the same design environment can be used to simulate the hardware, but also the software. By using one design environment, the hardware and the compiler are developed in cooperation instead of in two separated design processes.

1.2 Key requirements

Based on the previous analysis, we identified four key requirements to our system:

1. It should be highly *programmable*: That means, for maximum programmability and flexibility it should be possible to implement and map applications in a straightforward approach on the architecture. Hereby, the programming language should enable a user to express the operations and data dependencies present in data-driven streaming applications.
2. It should *support data-driven streaming applications*: That means, the execution mechanism and programming paradigm should be *data-driven* and should support operations on streams of data.
3. It should be an *efficient multicore architecture* for applications with a large degree of instruction-level parallelism: An interesting type of architecture for the target application domain are coarse-grained reconfigurable arrays (CGRAs), hence we will develop a CGRA fulfilling our requirements in the scope of this work.
4. It should be realised using *one* design environment, i.e. the specifications of all aspects of the full system, presenting a novel approach to system design. These aspects include the architecture and its synthesis, the programming model, the programming language, the compiler, and a simulation framework.

1.3 Structure of this thesis

In Chapter 2, the required background information for this work are presented. First, a short explanation to dataflow graphs is given, followed by an introduction to dataflow based programming languages. After that, dataflow machines are briefly introduced, followed by a more extensive introduction to CGRAs. Finally, we place our work in relation to existing work by a brief summary of the novel properties of our work compared to existing work.

In Chapter 3, a short introduction to Haskell and CλaSH is given, since we use them as design languages in our work. For illustration, we present the general syntax and give a number of examples.

In Chapter 4, the underlying conceptual basis for our complete work is presented. Since our work is based on dataflow principles, we present how both the architecture and the programming paradigm are inspired by dataflow.

In Chapter 5, the hardware architecture is presented. All components and their working principle, also in relation to the underlying dataflow principle, are presented.

In Chapter 6, we present the programming language and the compiler for our architecture. The grammar of the language and the relation to dataflow are illustrated by a number of examples.

In Chapter 7, we present an extensive case study to illustrate the working principle of our design flow. We also present the results of a number of further case studies.

In Chapter 8, we present the overall conclusion to our work where we relate our contributions to the key requirements. Furthermore, we give recommendations for future work.

1.4 Summary of our contributions

We designed and implemented a *coarse-grained reconfigurable array (CGRA)* consisting of simple, configurable cores. Each of the cores is *data-driven*, i.e. it follows the *firing rule* from dataflow. The cores each contain an ALU, local storage, a program memory and a control unit. The cores are interconnected using direct links to their respective neighbours.

Dataflow is the conceptual basis for the complete design process. The architecture is data-driven, each core is triggered by the availability of input data. As soon as the required number of data tokens has arrived, the operation is performed, the input tokens are consumed, and the required number of output tokens is produced. Also the programming paradigm is dataflow based. The programming language is used to express algorithms as a dataflow graph, hence fine-grained parallelism can be expressed in a straightforward way. The configuration principle of the architecture is a combination of dataflow and finite state machines.

The complete system, i.e. the architecture, the programming language and the compiler, was integrated into one framework which can be used to first simulate an algorithm in pure Haskell, then compile and map the algorithm onto the architecture, and then simulate the algorithm on the architecture. Since the complete design of the system was performed using Haskell, there is one complete, sound environment. We evaluated the presented system with a number of case studies. The case studies were DSP kernels, which are commonly used in streaming applications.

Chapter 2

Background

Abstract – *In this chapter, we will present the required background and related work. We will start with an introduction to general dataflow principles, dataflow graphs and briefly introduce synchronous dataflow models. Then, we will give an introduction and overview on dataflow languages. After that, we will present a brief summary on dataflow machines, and finally, we will give an introduction to coarse-grained reconfigurable arrays (CGRAs). We will conclude this chapter with a brief summary of the novel properties of the work presented in this thesis compared to previous work.*

2.1 Dataflow principles

In this section, the general principles of dataflow and dataflow graphs are presented.

2.1.1 Representing a program as dataflow graph

The basic principle of dataflow programming is to represent a program as a directed graph (in dataflow programming referred to as *dataflow graph*) [12] [25] [32] [54] [91]. This dataflow graph consists of *nodes* which are interconnected by *arcs*. The nodes represent *operations on data*, the arcs model the *dependencies* (or *channels*) between the nodes. Data is represented by *tokens* flowing between the nodes on the arcs.

In general, the granularity of the dataflow graph is determined by the nodes. When the nodes define operators, the granularity is on the operator level. When the nodes define complex macros (i.e. a collection of instructions or operations), the granularity of the dataflow graph is on the macro level.

Nodes

A node represents a certain operation or function in the graph. When the graph describes a mathematical algorithm, a node represents instructions such as arithmetic or comparison operations [51]. This operation is repeated indefinitely, as long as tokens arrive [25]. A node that produces a constant value regenerates this constant value as often as needed [25].

Arcs

The arcs connecting the nodes in the graph are directed. They represent data dependencies between the nodes [51]. An arc resembles a FIFO buffer [52] [53], which means that data items on an arc cannot overtake each other. Arcs going towards a node are *input arcs*, arcs that leave a node are the node's *output arcs*.

Tokens

A token is an instantiation of a data object flowing between nodes [25]. The token travels from producer to consumer [91] along the arcs [30]. Since the arcs resemble FIFO buffers, tokens cannot be interleaved on the arcs and as such have deterministic behaviour [25]. Besides data representation, tokens can also be used to represent iterations in cyclic dataflow graphs by using *initial tokens*. Tokens can even represent complex structures, as for example tuples, files, a function or complex data types [25].

Firing rule

The firing rule is a central principle in dataflow programming, since it triggers the execution of a certain node. Whenever a certain node has the required data on its input arcs, the node is said to be *fireable* [12] [25]. A fireable node is executed at some undefined time after it becomes fireable. As a result of a firing, the input data is removed from the input arcs, the operation is performed and the result is put on the output arc(s). Then, the node waits until it is fireable again [51].

2.1.2 Properties of the dataflow graph

According to [12, 25, 51, 52, 91], a dataflow graph has the following properties:

» **naturally concurrent**: each sub-part (hence also a single node) of a dataflow graph can be considered and executed independently (hence concurrently), the concurrency is fine-grained. The advantage is that more than one operation can be executed at once, hence it is inherently parallel and has the potential for massive parallelism. Dataflow has the potential to provide substantial speed improvement by utilising data dependencies to locate parallelism.

» **deterministic**: tokens cannot be interleaved on the arcs, and they are produced/ consumed in a fixed order.

» **composable**: each sub-part of the dataflow graph can be considered as a complete dataflow graph, hence simple dataflow graphs can be used to compose more complex dataflow graphs.

» **no global state**: everything is local to a node.

» **no current operation**: the concept of a *current time step* does not exist since firing of a dataflow node only depends on the availability of data and is not triggered by a clock.

2.1.3 Synchronous dataflow

Synchronous dataflow (SDF) [59] is a specific subset of dataflow, which can be used to model real-time streaming applications. For each node in an SDF graph, the number of tokens that are consumed and produced per firing is known at design time. Therefore, SDF can be used to analyse if an application, which is modelled as an SDF graph, meets all its Quality of Service (QoS) requirements [92]. SDF can, for example, be used to model the latency and rate characteristics of data streams over a predictable interconnect like a ring network [29].

A special case of SDF is Homogeneous SDF (HSDF), where all nodes consume and produce *one* token per arc and firing. Every SDF graph can be transformed into a corresponding HSDF graph [59].

Cyclo-static SDF (CSDF) [20] is an extension to pure SDF. In CSDF graphs, the nodes in the graph are modelled with periodic behaviour, i.e. the consumption and production rates of the nodes per firing follow a periodic scheme. CSDF graphs can model applications in a more compact form than pure SDF graphs. Recent work [27, 28] has shown that any CSDF graph can be transformed into an SDF graph with the same temporal behaviour, which is at most a linear factor larger.

Besides the mentioned dataflow models, many more variants of SDF have been proposed, an extensive discussion is however out of the scope of this thesis. For more information on dataflow models see [20], [92].

2.2 Dataflow based programming languages

Dataflow languages are programming languages that are based on the *dataflow principles* introduced in the previous section. Dataflow programming is a not a new field, it has been used since the 1970s. In general, there is a strong mutual relationship between dataflow and functional languages. Also, there is a close relationship between dataflow languages and dataflow machines [91]. For a survey on the historic development of dataflow languages, the reader is referred to [25, 51, 91].

2.2.1 General properties of dataflow languages

According to [9] (and repeated by [91]), dataflow languages all have the following properties:

1. freedom from side effects
2. locality of effect
3. equivalence of instruction scheduling with data dependencies
4. single-assignment semantics
5. an unusual notation for iterations because of features 1 and 4
6. a lack of history sensitivity in procedures

Since these properties are important to understand the essence of dataflow programming, we will elaborate on each of them further in the following.

Freedom from side effects

Freedom from side effects means that it is impossible to define global side effects [25]. Also, the operation of each node is functional, i.e. existing data is never modified. The result of a node only depends on the value of the used input tokens, and *not* on a global state. Since there is no global data [51], there can be no side effects.

Locality of effect

In dataflow programming, there is no concept of a state of variables [91]. If required, a state can be modelled using a self-loop. Also, once a token has been generated, it is never modified.

Equivalence of instruction scheduling with data dependencies

Dataflow languages are applicative languages based solely on the notion of data flow [25]. Instead of describing the execution order, the data dependencies are defined by arcs between the nodes [25]. Also, there is no *current* operation [91]. In contrast to the *Von Neumann model*, where an execution is triggered by the program counter, operations are scheduled for execution as soon as their operands become available [51].

Single-assignment semantics

It is not allowed to change an existing value of a variable, for example, a statement like $x = x * 2$ is not allowed if both occurrences of x are assumed to refer to the same variable. Also, a statement like shown in Listing 2.1 is not allowed, since x is modified twice in the same iteration:

```
for(int=0;i<N;i++)
{
  x = 1;
  ...
  x = 2;
}
```

Listing 2.1 – Example for non single-assignment

An unusual notation for iterations because of features 1 and 4

Although iterations are not part of the pure dataflow model [91], they can be defined by using cyclic dataflow graphs with initial tokens [25]. A cyclic dataflow graph should be well-behaved. The initial token distribution will be restored after a few iterations.

A lack of history sensitivity in procedures

The nodes in the dataflow graph do not have a notion of state. That means, data is only relevant for the current *firing* of a node, after that, it is not stored for future firings. As a consequence, a node does not *remember* data from previous firings.

2.2.2 Advantages and disadvantages of dataflow languages

In [51, Sec. 6] and [38], an analysis is given concerning the advantages and disadvantages of dataflow programming.

Dataflow languages have the potential to express massive parallelism because of their inherent concurrency. Since dataflow languages describe the data dependencies, i.e. the *structure* of a program, parallelism can easily be located, and hence it is possible to speed up the execution of the program by exploiting this parallelism. Also, concurrency analysis is not required since it is already included in the dataflow graph description. Another advantage is that the pure dataflow model is deterministic. Because of the previously introduced *firing rule*, no static scheduling is required, since each operator executes when data arrives. Also, dataflow programming is free from side effects.

On the other hand, iterations are difficult to express in the pure dataflow model, a dataflow language which allows iterations has to provide some kind of specialised syntax. Data structures are incompatible with the pure dataflow model, since once a token is generated, it cannot be modified. The pure dataflow model does not allow for non-determinism. If a dataflow language features the expression of non-deterministic behaviour, special syntax has to be provided that does not concur with the pure dataflow model.

2.2.3 Concrete languages:

Over the years, a significant number of dataflow based programming languages has been developed and published. In [91], a good historic overview up to 1994 is given, [51] presents a survey on the history of dataflow languages up to 2004.

Early development

Dennis from MIT, a pioneer in the development of the dataflow field, published a paper in 1974 presenting a concrete dataflow language [30]. It was a generalisation of pure Lisp and designed to be a model for study of functional semantic constructs, and a guide for research in advanced computer architectures. The language is data-driven and contains the standard dataflow language constructors like operators and selectors.

Lucid [14] is a language that was developed around 1976. Originally, it was developed independently from the dataflow field, but the semantics were similar to languages required by dataflow machines [51]. The underlying execution model is a demand-driven model. A program in Lucid is a definition of a network of processors and communication channels, a variable represents an infinite stream of data. Originally, it was developed to be a language to write and prove the correctness of programs. The programming part follows dataflow principles, hence the order of statements is irrelevant. The proof part is designed to express mathematical principles. Lucid is meant to be one system for both programming and proving the correctness of the program.

Id [69], developed at Irvine between 1970 and 1980, is a very early example of a dataflow language. The semantics of Id has been influenced by Lisp [80] and Backus' functional programming notation [17]. Originally, Id was developed to design operating systems, but in the 1980s, the focus was shifted towards scientific problems. An important extension to the original Id language was the development of *I-structures* [69], which are parallel data structures to address the problem that dataflow languages cannot express complex data structures.

LUSTRE [45] is a synchronous dataflow language for programming reactive systems. It can also be used to describe hardware. The program structure of LUSTRE is based on block diagrams and networks of operators. The authors emphasise that, since LUSTRE is a synchronous language, it can be compiled into a sequential program.

Development in the 1980s and 1990s

According to [51], the common believe in the beginning of the 1980s was that dataflow languages would become the dominant type of language. However, research in dataflow languages even slowed down. The authors of [51] claim that dataflow languages required the support for a level of fine-grained parallelism in the hardware that was simply not viable at that time. It was not the dataflow idea that failed, but the hardware was not ready yet.

Nevertheless, there was some development in the field. CAJOLE [46] was presented in 1981, which was later used for structural programming tools for dataflow languages. VAL [62], published in 1982, is a language for the dataflow computers developed at MIT by Dennis. It is a language for expressing and identifying concurrency and translation of algorithms into dataflow graphs and designed for programming for a highly concurrent environment. The basic principles are implicit concurrency and assistance for programmers to design for a multiprocessor environment. The language has single assignment semantics.

SISAL [37], presented in 1983, is a language derived from VAL. It was designed as a platform for understanding and exploitation of parallelism in multiprocessor systems. It has a functional style and no side effects. It supports data structures. Its intermediate language, IF1, is a dataflow language that consists of acyclic graphs.

In the 1990s, the focus in dataflow languages shifted more towards experiments with different granularity [81]. Also, visual dataflow languages were developed. A well-known example is *Labview* [5], which has a dataflow language as its core.

Apart from the above mentioned languages, others were published. However, we will not discuss them in detail since it would be out of scope of this thesis to give a complete overview.

Development in the 2000s

In the 2000s, dataflow languages became more popular again.

StreamIt [42, 86], published in 2002, is a high-level, architecture independent dataflow language. The authors implement a compiler that compiles and maps code for the RAW processor [89]. The claim of the authors is that C is not suited for those kind of machines because C is not made for expressing parallelism and streams. The principle of StreamIt is based on *pipelines, splitjoin constructs* and *feedback loops*, all of them having stream structures. The basic computation unit is a *filter*. The syntax is based on Java.

CAL [34, 35] is a dataflow language presented in 2003. It consists of components (actors), that are interconnected by FIFOs. The execution of the actors is atomic, the actors follow the firing rule from dataflow. Xilinx has a front end for compiling CAL to VHDL. CAL has been chosen by the ISO/IEC standardisation organisation in the new MPEG standard called Reconfigurable Video Coding (RVC) [19]. In [73] and [11], two use cases are presented where CAL is used to implement an MPEG decoder. In [11], the authors claim that C fails for multicore platforms, whereas CAL might work. OpenDF [19], presented in 2009, is a dataflow toolset for reconfigurable hardware and multicore systems based on CAL.

Flextream [50], presented in 2009, is a dynamically adaptive streaming programming paradigm for multicore systems.

∑C [43], published in 2011, is a dataflow language for high-level programming. The syntax is C based. The language is a subset of the process network model, the executions are non-deterministic.

Besides the above mentioned languages, a number of languages were published which did not gain high popularity, and will not be discussed in the course of this thesis.

Functional languages, used for dataflow purposes

In 1978, John Backus gave a Turing lecture on functional languages and dataflow computing [17]. He points out that conventional languages are too large and awkward, hence they create unnecessary confusion in the way programmers think about programs. Furthermore, they are designed around Von Neumann Model and thus the design of alternative machine architectures is difficult.

2.3 Dataflow machines

Dataflow machines are machines that can execute dataflow graphs and are usually programmed using dataflow languages. In this thesis, we are not designing a classical dataflow machine, but a coarse-grained reconfigurable array (CGRA), which we will introduce in the next section. However, since our approach is dataflow inspired, we will give a short overview of the essence of dataflow machines.

Dataflow machines are all programmable computers of which the hardware is optimised for fine-grained data-driven parallel computing [87]. In general, a processing element of a classical dataflow machine is composed as follows. The nodes of a dataflow program are stored as templates containing a description of the node and space for input tokens. The description of the node consists of the operand code and a list of destination addresses. The unit that manages the storage of tokens is called the *enabling unit*. The token storage usually is separated from the node storage. The enabling unit is split into two stages: the matching and fetching unit. A dataflow multiprocessor is composed of a number of dataflow processing elements interconnected by a network. Communication in the network hereby can be either direct or packet oriented.

Common to all dataflow machines is the basic instruction cycle (although specific implementations might differ):

1. Detect when a certain node is enabled (this corresponds to the firing rule)
2. Fetch the instruction
3. Compute the result
4. Generate result token(s)

The token store mechanism can be either *static*, i.e. only one token per arc is allowed, or *dynamic*, i.e. multiple tokens can be present on one arc.

Figure 2.1 shows a general illustration of a static dataflow machine. Static dataflow machines were the first dataflow machines to be published. An important architecture is the static dataflow machine by MIT [32], which is the first published design of an actual dataflow machine. The oldest fully working dataflow machine is the DDM1 [26]. Another interesting architecture is presented in [55] which can execute Lisp programs, however, this architecture has not actually been implemented.

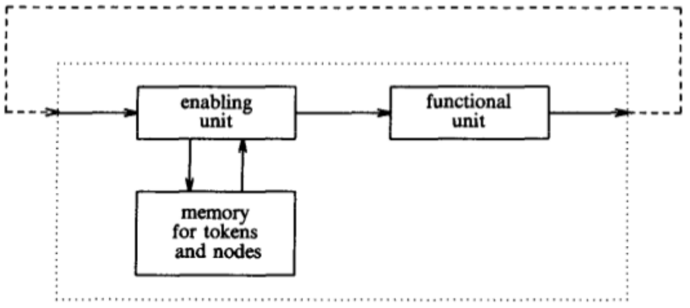


Figure 2.1 – Static dataflow machine, reprint from [87]

In Figure 2.2, a general illustration of a dynamic dataflow machine is shown. Dynamic dataflow machines allow, in contrast to static dataflow machines, multiple tokens per arc. This can be achieved by either code-copying or tagged tokens, for more details, see [87]. Dynamic dataflow machines potentially provide the highest level of parallelism [87].

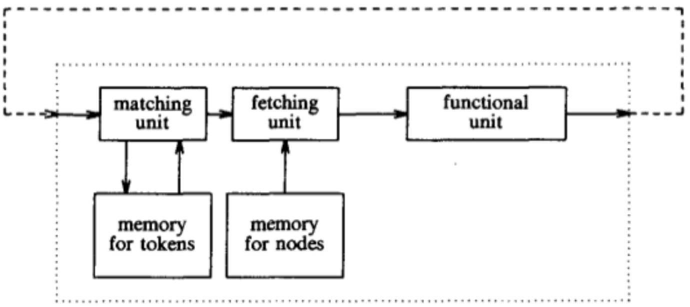


Figure 2.2 – Dynamic dataflow machine, reprint from [87]

The first detailed dynamic dataflow machine with code-copying is presented in [74] by Rumbaugh. The family of dynamic dataflow machines presented in [13] by MIT also uses code copying. It is an extension of the original static dataflow machine by MIT [32]. To program the machines, the language *Id* is used. The machines include special units to store data structures (I-structures). The *Manchester tagged token machine* is presented in [90] and [44]. It is the first dataflow machine that uses the principle of *tagged* tokens to allow several tokens per arc and is the basis for all the other tagged token machines. The *Monsoon* [70] is designed to be a general purpose multiprocessor. To support dynamic dataflow execution, it uses an *explicit token store* (ETS). The basic idea of ETS is that tokens are stored in dynamically allocated blocks, where the location within a block is determined at compile time. In [40], a fine-grained dataflow machine with local token tagging for functional languages is presented.

Another method to design dataflow machines is to combine Von Neumann and dataflow styles, i.e. to design a *hybrid* architecture. P-RISC [68] is a RISC architecture with dataflow elements. It is one of the important early papers on hybrid architectures. Also in Japan, research on hybrid architecture was performed. In [75], a dataflow machine with RISC-like processors is presented. WaveScalar [82, 83] is another dataflow machine with Von Neumann style programming. Unlike previous dataflow machines, WaveScalar can efficiently provide the sequential memory semantics that imperative languages require.

[87] is a good introduction to the dataflow domain. The authors give a good historic overview on the different dataflow machines and the developments. They also define the different kinds of dataflow machines, i.e. static and dynamic machines. Furthermore, they give a detailed graph and table on the different dataflow machines. Finally, they present the Manchester tagged token dataflow machine [90] in detail.

For further details on dataflow machines, the reader is referred to the surveys presented in [12, 31, 51].

2.4 Coarse-grained reconfigurable arrays (CGRAs)

2.4.1 General principle

Coarse-grained reconfigurable arrays (CGRAs) compose a class of architectures that consists of small, reconfigurable cores that are interconnected into an array, usually a mesh-configuration. The target applications of CGRAs are commonly DSP algorithms. The cores in the CGRAs usually contain an ALU, small local storage and a control unit. Good surveys on CGRAs can be found in [85], [24] and [48].

In this thesis we present a CGRA. In the remainder of this section, we will give an overview on the most important existing CGRAs.

2.4.2 Architectures

Over the years, many different CGRAs have been published. Even though they all belong to the general class of CGRAs, they greatly differ in their respective details like the number of cores, the type of interconnects or the functionality of each core. In the following sections, a more elaborate overview is given.

First, CGRAs that are closely related to the herein presented CGRA are presented. Following, CGRAs that are remotely related but are important to the general field of CGRAs are presented.

Closely related CGRAs

In this section, we will briefly present CGRAs that are closely related to the CGRA that will be presented in this thesis.

BilRC [15], published in 2013, is a 2D array of cores that operate on 16 bit data. In the array, there are three different kind of cores: ALUs, memory cores and multiplier cores. The computation model of the cores is not dataflow based. The proposed programming language, the *LRC* language, is a dataflow language with the ability to express loops. LRC is a *middle level language*, i.e. similar to assembly languages of microprocessors. Algorithms are mapped to the architecture using simulated annealing. They also present a SystemC cycle accurate simulator and a LRC to VHDL compiler which they use to compare results.

SmartCell [60], published in 2010, is composed of a 4x4 array of cells, where each cell consist of 4 processing elements (PEs) each including control and data switching fabric. That means, there are 64 PEs in total. The data width in the array is 8 bit. Each PE comprises an ALU, a logic unit, input and output registers and an instruction controller. The control is local to each PE. The connections within a cell are implemented via nearest neighbour links. The proposed programming scheme is called *SmartC*, but the authors remain vague about the actual implementation. The target application domains for SmartCell are multimedia and DSP applications.

Flora [58], published in 2009, consists of a RISC processor and a reconfigurable 2D array. The array contains 8x8 cores. The data width can be set to 8 bit or 24 bit. Each core comprises an ALU, a data manipulation unit, an 8 bit x 16-word register file, an 8 bit flip flop and a 16-depth instruction memory. The control is centralised, the mapping can either be spatial or temporal. As a special feature, Flora was designed to be able to perform floating point operations, to do so, PEs can be paired. Unfortunately, the authors do not enclose any details on the programming scheme.

MORA [57], presented in 2007, is a 2D array consisting of 4x4 quadrants with 2x2 cores each. The data width is 8 bit. Each core contains an internal RAM and an 8 bit ALU. Each core is a tiny *Processor-In-Memory* (PIM). Each core can be configured to perform one of four modes: feed-forward, feed-back, route-through single and route-through double output. The control is local on the cores. The cores are interconnected using unidirectional, nearest neighbour connections, furthermore, a number of longer connections are available. The target application domains of MORA are multimedia and streaming applications. The authors do not explain how the architecture is programmed.

DReam [10], published in 2000, is a 2D array which is scalable in size. The bit width is 8 bit. Each core consists of two dynamically reconfigurable 8 bit integer data paths, one spreading data path, one controller, two dual port RAMs and a communication protocol controller. There is one Configuration Memory Unit (CMU) per four cores. Per 2 CMUs and 4 global interconnect switching boxes (SWB), there is one communication switching unit (CSU). All CSUs communicate to one global communication unit (GCU). The cores are interconnected via nearest neighbour connections, segmented buses and reconfigurable local and global connections. The target application domain is next generation wireless applications and the programming of wireless devices. The authors do not provide details on the programming language.

[64] is an early example of a CGRA since it was already published in 1996. It is a 2D mesh consisting of 8 bit cores. Each core contains an 8 bit ALU, local memory and control logic. The cores are interconnected using direct links to their eight direct neighbours. Furthermore, connections of length four, and a number of global lines are available. The target area is general purpose computing. The programming is performed with an assembly level macro language.

Remotely related CGRAs

In this section, we will present publications on CGRAs that are not closely related to the herein presented CGRA, but are still relevant for the general field of CGRAs.

Trips [22], [77], [76], published in 2004, is not a real CGRA (the designers position Trips as *four ultra large cores* [77]. Nevertheless, Trips shares many similarities with CGRA architectures and is often cited in the same context. The architecture is a mesh consisting of two tiles with a grid of 4x4 configurable cores. The cores operate on 16 bit data. Each core contains an ALU, operand buffers, instruction buffers and a router. The control is global and follows the EDGE paradigm [22]. The principle of EDGE is to group chunks of code and map these chunks onto the array. In [79], a compiler is presented.

XPP [18] by Pact [6], published in 2004, is a regular array. The array is composed of three types of *processing array elements* (PAE): ALU-PAE, Function(FNC)-PAEs and RAM-PAEs. The ALU-PAE and RAM-PAE form a dataflow array. The FNC-PAEs build a VLIW-like processor kernel for control operations. For controlling the array, a global control tile per 4x4 grid is available. The connection is hierarchical. The programming is done with starting from C, which is compiled to a dataflow graph, from which assembly code is generated. Blocks of the code are mapped onto the grid and executed atomically. In the original XPP paper, a language called *NML* is presented, which is developed by Pact. In [47], a compiler is presented.

ADRES [63], published in 2003, is a reconfigurable grid that is closely coupled to a VLIW processor. The two parts are connected through shared memory. ADRES is designed to be an architecture template, the number of cores can be configured. The data width is 32 bit. The cores can also be configured. In [21], an instance of ADRES is presented. The target area of ADRES is next generation wireless applications. DRESC is the C-based programming language for ADRES.

MorphoSys [78], published in 2000, is an 8x8 array and operates on 8 or 16 bits. Each core contains an ALU, a multiplier, a register file and a 32 bit context word for configuration. For control of the grid, there is a general purpose RISC processor that controls the sequence of operations [85]. Context words are stored in a central context memory and are broadcast in a column or row-fashion, which makes MorphoSys a SIMD system. For programming, there is a SUIF-based compiler available, and a limited SAC compiler [88].

REMARC [65, 66], published in 1998, consists of a RISC processor and a 2D mesh containing 8x8 cores. The data width is 16 bits. Each core contains an ALU, a 16-entry RAM, an 8-entry register file, data input registers, data output registers and a 32 entry instruction RAM. The control is handled as follows: There is a global control unit that sends a common PC to the cores in each cycle. All cores thus receive the same PC. Since they all have their own instruction RAM, they can be configured to different operations, if necessary. The RISC processor is programmed using C, the array is programmed by adding assembly instructions to the C codes. The compiler generates assembly code for the RISC processor with the assembly code for the array included.

PADDI-2 [93], published in 1993, and PADDI [23], published in 1992, are early examples of CGRAs. While PADDI was designed as architecture for DSP applications, PADDI-2 was meant to be a platform for rapid prototyping of architectures for DSP applications. PADDI-2 also provided a toolbox with graphical support for signal flow graphs. Programming was done in assembly.

While in the previously mentioned CGRAs the cores are arranged in a 2D mesh, there were experiments with alternative topologies. Relevant examples are RaPID [33] and PipeRench [41], which consists of a 1D array and target mostly pipelined applications.

2.5 Conclusions

In this chapter, we gave a brief summary of dataflow in general, followed by an overview of dataflow programming languages. We then briefly introduced dataflow machines and finished with an introduction to coarse-grained reconfigurable arrays (CGRAs).

As mentioned in the previous chapter, the target application domain for the work presented in this thesis is data-driven streaming DSP applications that contain a large degree of fine-grained parallelism. As already presented in Chapter 1, we identified four key requirements for our work based on the target application domain: the system should be highly programmable, support streaming applications, an efficient multicore system and it should be realised using one single design environment.

The main focus in this work is on the first key requirement: the development of a novel programming paradigm to implement DSP streaming applications that contain a large degree of instruction-level parallelism on CGRAs. Most, if not all, previously published CGRAs (as presented in this chapter) are programmed using an architecture-specific subset of C or a low-level language. Since C does not support the expression of instruction-level parallelism or data dependencies, the burden of extracting the *structure* of an implemented algorithm lies in the compiler. We chose to not use C (or any other imperative programming paradigm), but instead start from a functional language, in particular Haskell. With Haskell, it is possible to describe an algorithm by its *structure* by using *higher order functions* or *recursion*. In our opinion, this is a much more intuitive approach to implement streaming DSP applications than relying on an imperative programming paradigm as previously presented CGRAs.

The second key requirement, i.e. the design of a system that supports streaming, is the main motivation to base our complete system on *dataflow principles*. That means, the architecture is data-driven, in the sense that all the cores in the architecture adhere to the (core-local) *dataflow firing rule*. As soon as the required data for a certain core arrives, it automatically starts the execution and produces the result which is then either used internally for the next firing of the core, or sent further to another core in the architecture. But not only the architecture is based on dataflow principle, also the programming language for the architecture is based on dataflow principles. We designed the programming language with the goal to be able to implement DSP algorithms as a *dataflow graph*. Usually, the specifications of a streaming DSP algorithm is available in the form of a task graph; by using a dataflow-based programming language, it is a straightforward step to implement this graph.

The third requirement, the design of an efficient multicore for streaming DSP applications, led to the development of a CGRA. The cores in the CGRA are small and simple, they contain an ALU for elementary binary operations, a small local memory for intermediate data and a program memory. The cores are interconnected in a 2D mesh by using direct links to the direct neighbours. We implemented the architecture using CλaSH, which is a hardware description language and compiler based on Haskell.

The fourth requirement, i.e. that the complete system should be development using *one* design environment, inspired us to use Haskell as a design language for all parts of the system. To the best of our knowledge, we are the first to present a complete system that is based on dataflow principles throughout both the architecture and the programming paradigm, which is based on a functional language for the design of the actual architecture, but also as a base for the programming language and compilation framework for the architecture.

Chapter 3

Design Methods and Tools

Abstract – *The functional programming language Haskell was used as design language in this thesis. The presented architecture was implemented using CλaSH, which is a hardware description language and compiler based on Haskell. Also the compiler and programming language for the architecture were both designed on the basis of Haskell. In this chapter, we will give a short introduction to both Haskell and CλaSH, to provide the reader with the required background.*

3.1 Introduction to Haskell

Functional programming, e.g. Haskell, is a different programming paradigm than the more known imperative programming approach, e.g. C. The main difference between the two is that in a functional language the execution mechanism is based on the evaluation of expressions instead of variable assignments. Furthermore, functional languages inherently have a notion of structure, since operations are described by their data dependencies. As such, these dependencies can be used to determine potential parallelism between operations on data. In the remainder of this chapter, we will demonstrate how Haskell can be used to describe algorithms in terms of their structure.

In the following sections, we will describe the elementary concepts and syntax of Haskell. We will focus on the concepts that are relevant for the work performed in this thesis. An extensive introduction to Haskell can be found in [61].

Haskell programs can be compiled using *GHC*, the Glasgow Haskell Compiler [3]. *GHC* also provides an interactive environment, *GHCi*. In *GHCi*, Haskell functions are directly interpreted.

3.1.1 Syntax

The general notation for definitions in Haskell is as follows: first the function name is specified, then a list of arguments. The arguments hereby are usually not enclosed within brackets. To specify a function `timesTwo` which is applied to an argument `x` and the result should be `x+x`, the following is specified:

```
timesTwo x = x+x
```

When the function should be applied to a specific value for `x`, for example 1, it is written down as follows:

```
timesTwo 1
```

The result will be 2.

3.1.2 Higher order functions

A key feature of functional languages is the possibility to use higher order functions. Higher order functions accept not only values, but also a function as argument. Three examples of higher order functions are explained in this section. These three higher order functions are used extensively in this thesis. For the three examples, we will first show a reference implementation in *C* for comparison, followed by the implementation in Haskell.

Element-wise application of a binary function to two lists

The first example is an element-wise application of a binary function `f` to two lists `xs` and `ys`. The implementation in C is as follows:

```
for(int=0;i<N;i++)
  zs[i] = f (xs[i] , ys[i]) ;
```

Listing 3.1 – Implementation in C

In Haskell, the higher order function **`zipWith`** can be used to apply a binary function to two lists. **`zipWith`** takes two lists and a function as arguments.

The function which was given as the first argument is applied to these lists as if they were “zipped” to form a list of tuples. The binary function `f` is applied to the two lists `xs` and `ys` as follows:

```
zipWith f xs ys
```

As an example, a graphical representation for adding two vectors of length four can be seen in Figure 3.1. It can be seen that the structure of adding the vectors element-wise can be directly expressed in Haskell-code without using for loops.

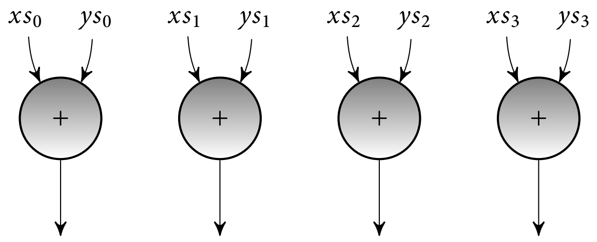


Figure 3.1 – **zipWith** (+) *xs ys*

Sequential application of a binary function to a list

The second example is a sequential application of a binary function *f* to a list *xs*, starting with an initial value *a*. The implementation in C can be done as follows:

```
s = a;
for(int i=0;i<N;i++)
  s = f (s , xs[i]);
```

Listing 3.2 – Implementation in C

In Haskell, another higher order function, **foldl**, can be used. With **foldl**, a function *f* is sequentially applied to a list, starting with an initial value. The arguments to **foldl** are a binary function, an initial value and a list. The function is first applied to the initial value and the first element of the list. Next, the function is applied to the result and the second element of the list and so on. The syntax to apply the binary function *f* together with the initial value *a* to the list *xs* is as follows:

foldl *f a xs*

As an example, a graphical representation for the sum of all elements using **foldl** of a list with four elements is shown in Figure 3.2.

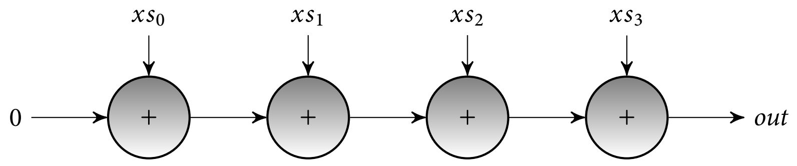


Figure 3.2 – **foldl** (+) 0 *xs*

Element-wise application of a unary function to one list

The third example is an element-wise application of a unary function *f* to a list *xs*. The corresponding implementation in C can be written as follows:

```
for(int i=0;i<N;i++)
  z[i] = f (xs[i]);
```

Listing 3.3 – Implementation in C

The higher order function **map** applies a unary function to each element in a list. It can for example be used to implement the element-wise increment of a list in Haskell. **map** is very similar to the previously presented **zipWith**, it accepts one function, but only one list instead of two as arguments. The function is then applied to each element in the list.

It is also possible to map a binary function *f* together with a fixed argument *a* which is applied during each function application to a list. While *f* is a binary function, *f a* becomes a unary function. To apply the binary function *f* with a fixed argument *a*, i.e. the unary function *f a* to a list *xs*, the following is used:

```
map (f a) xs
```

As an example, a graphical representation for a element-wise increment of a list with four elements is shown in Figure 3.3.

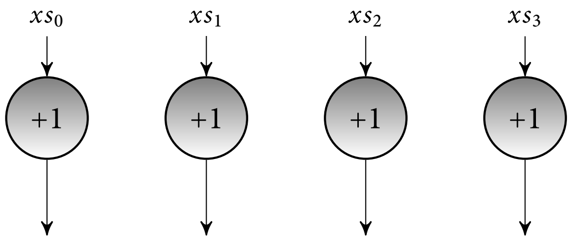


Figure 3.3 – **map** (+1) *xs*

3.1.3 Types

Haskell is a strongly typed language. In the following, we will explain this in more detail.

Function types

When defining a function, a type can be assigned. To define a simple function *foo* with one input argument of type integer and a result which is also an integer, the following is defined:

```
foo :: Int → Int
foo a = a
```

Listing 3.4 – Define a type

In line 1 of Listing 3.4, the type of *foo* is defined, in line 2 the functionality.

It is also possible to define a function without a specific type. In that case, the function is *polymorphic*. As soon as this function is applied to an argument, the type is automatically detected by the compiler. To illustrate this principle, a simple function *add* to add two numbers is defined. Furthermore, two arguments *x_int* (an integer) and *x_double* (a double) are defined:

```
add a b = a+b

x_int = 2::Int
x_double = 2.5::Double
```

Listing 3.5 – Defining concrete types for function arguments

In *GHCi*, the type of a function can be displayed by typing "`:t function_name`". The type of *add* is:

```
:t add
add :: Num a => a -> a -> a
```

That means, *add* expects two arguments of the same type (denoted *a*) and the result is of the same type. Furthermore, *a* is a number type (denoted by the first part of the type declaration **Num** *a*).

Applying *add* to the different arguments leads to the following types in *GHCi*:

```
:t (add (1::Int) 4)
(add (1::Int) 4) :: Int

:t (add (1::Double) 4)
(add (1::Double) 4) :: Double
```

This example shows that depending on the arguments to a certain function, the corresponding type is automatically derived.

If *add* is applied to two numbers of different types, e.g. an integer and a double, *GHCi* reports an error:

```
:t (add (1::Double) (4::Int))

<interactive>:1:19:
    Couldn’t match expected type ‘Double’ with actual type ‘Int’
    In the second argument of ‘add’, namely ‘(4 :: Int)’
    In the expression: (add (1 :: Double) (4 :: Int))
```

3.1.4 Algebraic datatypes

Haskell not only provides convenient methods to specify algorithms with regular structures, it also provides so-called algebraic datatypes. An algebraic datatype is a datatype which is constructed from other datatypes in combination with constructors.

```
data Choice = Yes Int | No
```

defines a datatype with name `Choice` that can have the values `Yes` followed by an integer, or simply `No`.

3.1.5 Data structures

Haskell also provides means to implement data structures, i.e. types that consist of more than a single element. Amongst others, the following two methods are available: *Tuples* and *Records*. In the following, we will introduce those two methods, since they will be used in the remainder of this thesis.

For the following examples, we will use a datatype that defines a circle. The circle is defined by an identifier, the coordinates of its centre (*x*,*y*) and the *radius*.

Tuples

A tuple combines a number of elements into one structure. To define the circle, we write the following:

```
type CircleTuple = (String, (Int,Int), Int)
```

Please note that not the keyword **data** was used to define the type, but **type**. This is because `CircleTuple` is a *type synonym* rather than a new datatype, since it does not introduce new constructors.

To define a concrete circle with the name "Bob", the centre at (1, 2) and a radius of 3, we write

```
myCircleTuple = ("Bob",(1,2),3)
```

To access the elements of `CircleTuple`, we implemented the following helper functions shown in Listing 3.6.

```
getName   ( name , _     , _      ) = name
getX      ( _    , (x,_) , _      ) = x
getY      ( _    , (_,y) , _      ) = y
getRadius ( _    , _     , radius ) = radius
```

Listing 3.6 – Access the elements of CircleTuple

To display the name and the radius of the previously defined circle `myCircleTuple`, we write the following in *GHCi*:

```
getName myCircleTuple
"Bob"

getRadius myCircleTuple
3
```

Records

To define the same circle using the *record* syntax, the code shown in Listing 3.7 is implemented.

```haskell
data CircleRecord = CircleRecord { name   :: String
                                 , x      :: Int
                                 , y      :: Int
                                 , radius :: Int
                                 }
```

Listing 3.7 – Define the circle using records

To define a concrete circle with the same properties as `myCircleTuple`, we write the following:

```haskell
myCircleRecord =CircleRecord {name="Bob", x=1 ,y=2 ,radius=3}
```

To access a certain element of the circle, for example the *name*, we simply write `name myCircleRecord`, to access the radius, we write `radius myCircleRecord` and so on:

```
name myCircleRecord
"Bob"

radius myCircleRecord
3
```

Note that this syntax differs from other languages, where the syntax would be `myCircleRecord.radius`. In Haskell, field names in records are functions.

The difference between tuples and records is that in a tuple, elements are accessed by their *position* in the type definition, whereas in a record elements are accessed by their *identifier*.

3.1.6 Choice

Haskell has a number of *choice* constructions available.

If then else

The syntax for a standard *if then else* construction is shown in Listing 3.8.

```
foo x y = if x>y then x else y
```

Listing 3.8 – If then else

The function `foo` has two input arguments *x* and *y*, the result is the bigger value of those two.

Guards

The similar functionality can be implemented using *guards* (|) as shown in Listing 3.9.

```
foo x y | x>y       = x
        | otherwise = y
```

Listing 3.9 – Guards

Pattern matching and case construct

It is also possible to match certain patterns. This can be, for example, matching a certain number or a character, or a certain constructor. There are two possible methods to match on patterns: pattern matching and case constructs. We will demonstrate both in the following.

The code in Listing 3.10 shows two methods to match on a certain number, in this case the number 5. The first function, `foo`, uses standard pattern matching. The second function, `bar`, uses the case operator. The underscore in lines 3 and 7 represent a *don't care*, i.e. the argument is irrelevant. In pattern matching, the conditions are evaluated line by line. This means, first it is checked if the argument equals the number 5. If that is not the case, the next line in this example, i.e. the *don't care*, always evaluates to *true*. It can be seen as the *default* case.

```
foo :: Int → String
foo 5 = "You guessed the correct number!"
foo _ = "Sorry, you guessed the wrong number"

bar :: Int → String
bar x = case x of 5 → "You guessed the correct number!"
                  _ → "Sorry, you guessed the wrong number"
```

Listing 3.10 – Match on a certain number

An example execution in *GHCi*:

```
foo 3
"Sorry, you guessed the wrong number"
foo 5
"You guessed the correct number!"
```

As already mentioned before, pattern matching can not only be used to match on numbers, but also on constructors in data types. Assume, the datatype *Person* is defined as follows:

```
data Person = Name String | Age Int
```

Then, pattern matching can be used to output the person's details. Again, *foo* is implemented using pattern matching, *bar* is implemented using the case operator, as shown in Listing 3.11.

```
foo :: Person → String
foo (Name x) = "Your name is " ++ x
foo (Age  1) = "You are 1 year old."
foo (Age  x) = "You are " ++ (show x) ++ " years old."

bar :: Person → String
bar x = case x of (Name x) → "Your name is " ++ x
                  (Age  1) → "You are 1 year old."
                  (Age  x) → "You are " ++ (show x) ++ " years old."
```

Listing 3.11 – Match on a constructor

An example execution in *GHCi*:

```
foo (Name "Bob")
"Your name is Bob"

foo (Age 4)
"You are 4 years old."

bar (Name "Alice")
"Your name is Alice"

bar (Age 1)
"You are 1 year old."
```

3.1.7 Lambda expressions

Lambda expressions (or λ expressions) are *anonymous functions*, commonly used in combination with the previously introduced *higher order functions*. In Listing 3.12, we show two implementations of the same functionality, first (the function `foo`) the implementation without a lambda expression, and then (the function `fooLambda`), the implementation using a lambda expression. The implemented functionality operates on two vectors `xs` and `ys` which are added element-wise, then multiplied by 2 and then 1 is subtracted. The example demonstrates that by using lambda expression, a very compact description of a function can be achieved.

```haskell
foo xs ys = zipWith bar xs ys
  where
    bar x y = (x+y)*2-1

fooLambda xs ys = zipWith (λx y → (x+y)*2-1) xs ys
```

Listing 3.12 – Lambda expressions

3.2 CλaSH

In the previous sections, we gave a short introduction to Haskell. In the following, we will briefly introduce CλaSH, which is the hardware description language used in this thesis. CλaSH is a hardware description language and compiler based on Haskell. This means, with CλaSH it is possible to design and implement hardware using Haskell. CλaSH can be seen as a compiler to convert (a subset of) Haskell code to synthesisable VHDL code. Since CλaSH is directly based on Haskell, the standard Haskell compiler *GHC(i)* can be used. More information on CλaSH can be found on the CλaSH website [1] and in [16, 56].

Combinatorial hardware can be viewed as a block of combinatorial logic, with a set of input signals and a set of output signals. The same is true for Haskell. A function consists of its functionality, and a set of inputs and outputs. Hence, when designing hardware using CλaSH, the structure of the hardware can directly be specified in Haskell. Each function in the CλaSH code corresponds to one block (or module) in the resulting hardware.

In contrast to dedicated hardware specification languages, e.g. VHDL and Verilog, Haskell does not have a notion of state. In CλaSH, the state is handled explicitly as part of the function, thus forming a mealy machine. We will explain this in more detail in Section 3.2.2.

The toolflow when designing hardware with CλaSH is as follows:

1. The desired functionality is implemented and verified using Haskell
2. The Haskell code is modified to be compatible with the CλaSH compiler
3. VHDL code and a test bench are automatically generated
4. The generated VHDL code can be simulated and synthesised using standard design tooling

3.2.1 Differences between CλaSH and pure Haskell

Compared with Haskell, a number of differences exist in CλaSH. For example, special types for numbers and lists are required, since number types and lists of unknown size are not supported. Also, recursion is (currently) not possible. In the following, a detailed description of the differences is given.

Number types

In CλaSH, the bit width of number types has to be defined. CλaSH offers two types of numbers: signed and unsigned. To define a signed number with 16 bits, the following is written:

```
type WordS16 = Signed D16
```

An unsigned number with 4 bits is defined as follows:

```
type WordU4 = Unsigned D4
```

Lists

While Haskell operates on infinite lists, this is not possible in CλaSH. The reason is that hardware cannot be “generated” at runtime.

In CλaSH, *vectors* are used instead of lists. Vectors are lists of a defined datatype with a defined length. To define a vector with eight elements of type *WordS*16, the following is written:

```
type V8WordS16 = Vector D8 WordS16
```

Since the higher order functions that are available in Haskell cannot be applied to vectors, specific higher order functions are available in CλaSH. They are indicated by a leading **v** in the function name. The corresponding vector function to **`foldl`** is **`vfoldl`**, for **`map`** **`vmap`** and so on.

On the project website of CλaSH [1], documentation is available with a more elaborate list on the specific CλaSH datatypes and functions.

3.2.2 State

Since Haskell itself does not have a notion of state, a specific notation in CλaSH is required to represent a stateful function. In CλaSH, the state is handled as part of the function.

While a stateless function only has one input and one output (if multiple signals are required, they are grouped into a tuple or record), a function with state has an additional input and output for the state, as shown in Figure 3.4. The state hereby is an explicit argument of the function, as shown in Listing 3.13. The logic in the function takes care of computing the new state value, which is then fed back to the function for the next clock cycle. The new state value is determined by the current state and the input signals, which corresponds to a *mealy machine*.

```
function (State s) i = (State s', o)
    where
      (s',o) = logic  s i
```

Listing 3.13 – Handling of state

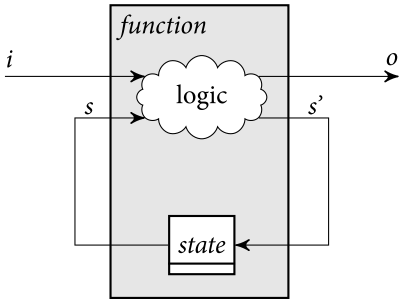


Figure 3.4 – State handling in CλaSH

Figure 3.5 illustrates the corresponding dataflow graph to handle state as feedback signal to a node. The state hereby is represented as token on the feedback arc, an initial value of the state has to be provided before the first firing of the node.

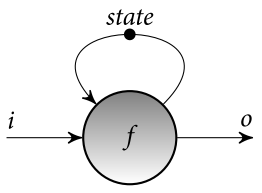


Figure 3.5 – Resemblance to dataflow handling of state

3.2.3 Define a component

The CλaSH compiler can generate synthesisable VHDL code from a design that was implemented using CλaSH. For that, *components* have to be defined. To define a component, a stateful function is provided with an initial state. To provide an initial state, *automata arrows* [71] are used as presented in [39]. As shown in Listing 3.14, the previously defined `function` is provided with an `initialState` by using the syntax ⥣ in line 2, the type of the component is defined in line 1. Hereby, the keyword `Comp` indicates that a component is defined, `FunctionInputType` is the input type for the component and `FunctionOutputType` is the output type of the component.

```
functionArrow :: Comp FunctionInputType FunctionOutputType
functionArrow = function ⥣ initialState
```

Listing 3.14 – Define a stateful component

It is also possible to define a component from a stateless function. In that case, the keyword `arr` is used as shown below:

```
stateless_functionArrow = arr stateless_function
```

Listing 3.15 – Define a stateless component

3.2.4 Composition of components

In the previous section, we explained how components are defined in CλaSH. Components can be used to compose more complex designs. By using components, the handling of the internal state of a stateful component is hidden from the user during composition, as we will show in this section. The composition of components is also done using the previously presented *arrow* notation. Both stateful and stateless components can be used during component composition.

Let's assume, two components `f1A` and `f2A` are to be combined in a pipelined fashion, as shown in Figure 3.6. The code in Listing 3.16 shows how to implement this pipeline. In lines 1 and 2, the components `f1A` and `f2A` are defined and provided with an initial state to their respective functions *f*1 and *f*2. Here, the same syntax is used as introduced in the previous section. *f*1*A* and *f*2*A* thus contain their respective functions *f*1 and *f*2 and also their initial states *iS*1 and *iS*2. The components *f*1*A* and *f*2*A* can now be used to compose a new module, the user does not have to take care of the handling of the initial state.

In lines 4 to 7, the pipeline is implemented. The syntax is read as follows: In line 4, the name of the module is defined (*topA*), and the input signal (*i*). Then, the two previously defined components *f*1*A* and *f*2*A* are used to compose *topA* to form the structure shown in Figure 3.6. In line 5, the input *i* is sent to *f*1*A*, its output is denoted *f*1'. In line 6, this output is used as input to *f*2*A*. The output from *f*2*A* is denoted *f*2'. Finally, in line 7, the output of the top module *topA* is defined to be *f*2', which was the output of *f*2*A*. The handling of the state is completely hidden from the user during the composition step.

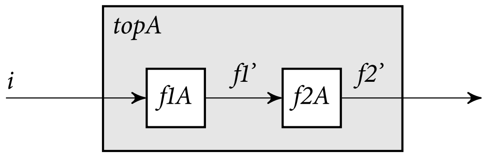


Figure 3.6 – Compose modules using arrows

```
1  f1A = f1 ⇑ iS1
2  f2A = f2 ⇑ iS2
3
4  topA = proc i → do
5      rec  f1' ← f1A ⤙ i
6           f2' ← f2A ⤙ f1'
7      returnA ⤙ f2'
```

Listing 3.16 – Composition of modules

3.2.5 Examples

In this section, we will show a number of small examples to demonstrate the working principle of CλaSH. We start with plain functions, and finish the section with a demonstration how components are generated from these functions.

Simple adder

A simple adder is defined as shown in Listing 3.17. The function name *add* and its argument, a tuple of two numbers (x, y) and the name of the output o are defined in line 1. In line 3, the actual computation is defined which is simply an addition of x and y. An illustration is shown in Figure 3.7.

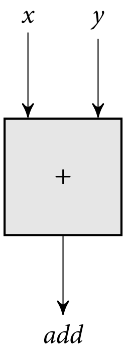


Figure 3.7 – Simple adder

```
add (x,y) = o
  where
    o = x + y
```

Listing 3.17 – Implementation of a simple adder

Compare two numbers

To compare two numbers and output the bigger of the two, the code in Listing 3.18 can be used. The bigger number is determined by using the *guard* operator as explained in Section 3.1.6. A graphical representation is shown in Figure 3.8.

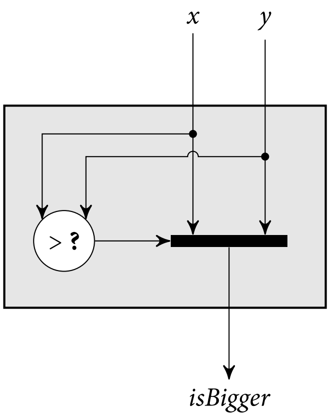


Figure 3.8 – Compare two numbers

```
isBigger (x,y) | x>y       = x
               | otherwise = y
```

Listing 3.18 – Compare two numbers

ALU

To define an ALU with a configurable operation, the code presented in Listing 3.19 can be used. The input of the ALU consists of a three-element tuple - an opcode and two operands. By using *pattern matching*, as explained in Section 3.1.6, the correct operation is chosen. Line 1 defines an addition, line 2 a subtraction and line 3 a multiplication. Figure 3.9 shows a schematic view.

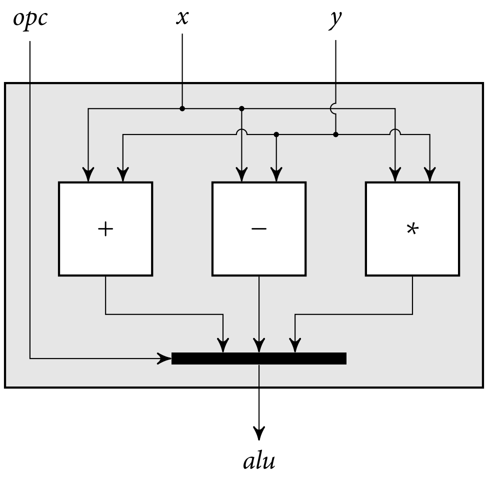


Figure 3.9 – ALU

```
alu (ADD, x, y) = x + y
alu (SUB, x, y) = x - y
alu (MUL, x, y) = x * y
```

Listing 3.19 – ALU

Find biggest number in a vector of numbers

To find the biggest number in a vector of numbers, the code shown in Listing 3.20 is used. To implement it, the *higher order function* **vfoldl** is used. **vfoldl** has the same functionality for vectors in CλaSH as **foldl** has for lists in Haskell (as introduced in Section 3.1.2). The function to compare two elements in the vector is implemented in line 3 using the λ notation as explained in Section 3.1.7 and is indicated by > in Figure 3.10. **vhead** and **vtail** are the equivalents in CλaSH for vectors to **head** and **tail** for lists in Haskell. They produce the first element and the vector of the remaining elements, respectively. A graphical illustration is shown in Figure 3.10.

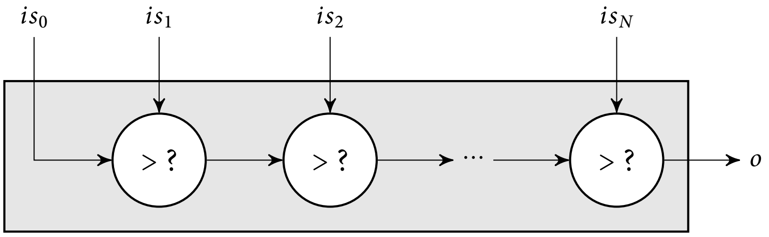


Figure 3.10 – Biggest number in a vector

```
biggestFromVector is = o
  where
    o = vfoldl (λx y → if (x>y) then x else y ) (vhead is) (vtail is)
```

Listing 3.20 – Find biggest number in a vector of numbers

Accumulator

The next example is a simple accumulator as shown in Figure 3.11. The implementation in CλaSH is presented in Listing 3.21. In contrast to the previous examples, the accumulator is *stateful*. This is also visible in the function definition in line 1, where the keyword `state` indicates the name of the internal state variable `s`.

The accumulator operates on an (infinite) stream of tokens that are sent to the accumulator via the input `i`. In each clock cycle, the incoming token is added to the value stored in the internal state `s` of the accumulator. Then, the result is used to update the internal state (denoted `s'`) and the output `o`. That means, the value stored in the internal state is the sum of all tokens that have been sent to the accumulator until that clock cycle. The result in each clock cycle is also the output of the accumulator.

```
acc (State s) i = (State s',o)                                  1
  where                                                         2
    o  = s + i                                                  3
    s' = o                                                      4
```

Listing 3.21 – Accumulator

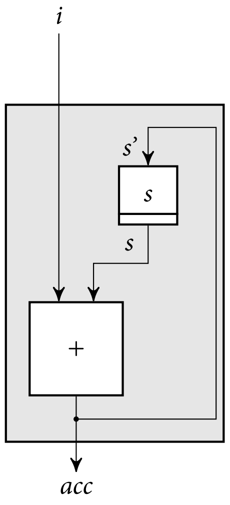


Figure 3.11 – Accumulator

Multiplexer with configurable selector

The final example is slightly more complex. It is a multiplexer in which the selector `sel` is saved in the internal state. That means, the selector is configured to a certain value. During runtime, the selector can be reconfigured.

An illustration of the configurable multiplexer is shown in Figure 3.12. The main component, the actual multiplexer, is represented by the white box with the label *mux*. The *mux* has three inputs in total, two of them are the data inputs *x* and *y* at the top, the third is the selector at the left. The selector is stored in the box labelled *sel*, which is the internal state and can be configured.

The type definitions for the configurable multiplexer are shown in Listing 3.22. In line 1, the number type is defined to be a signed integer with 16 bits. Then, in line 2, the possible values for the selector are defined, in this example, either *`L`* or *`R`*. In line 3, the type for the input signal is defined. It can either be a new configuration (indicated by the constructor *`Config`*) followed by the new value for the selector, or a data input (indicated by the constructor *`Data`*), followed by a 2-tuple. Finally, the data type for the output is defined in line 4. It is a tuple consisting of a boolean, which indicates whether a valid data output is available, and the data value itself. If a new configuration is sent to the configurable multiplexer, the boolean at the output is set to **`False`**, since no data is sent to the multiplexer in that clock cycle.

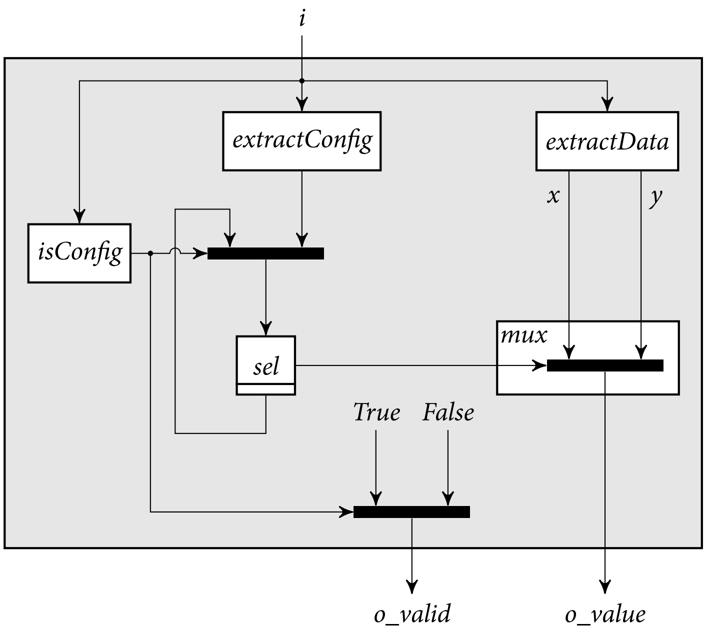


Figure 3.12 – Configurable multiplexer

```
type Word          = Signed D16
data Selector      = L | R
data ConfMuxInput  = Config Selector | Data (Word,Word)
type ConfMuxOutput = (Bool,Word)
```

Listing 3.22 – Type definitions for the configurable multiplexer

In Listing 3.23, the CλaSH implementation of the configurable multiplexer is shown. In line 1, the function name and the inputs and outputs are defined. Furthermore, the current state and the new state are defined, indicated by the `state` keyword. In lines 5 and 6, the new value for the selector is determined, which is either a new configuration (line 5), or the previous value (line 6). In line 8, the data is extracted from the input. In lines 12 and 13, the boolean value for the output is determined. In line 14, the data output determined by the multiplexer. In lines 16 to 22, helper functions are defined. Line 25 defines the actual multiplexer.

```
confMux (State sel) i = (State sel', (o_valid,o_value))
  where

    -- new state for the selector
    sel' | isConfig i = extractConfig i
         | otherwise  = sel

    -- data input to the mux
    (x,y) = extractData i

    -- output of the fixedMux
    o_valid | isConfig i = False
            | otherwise  = True
    o_value = mux sel (x,y)

    -- helper functions
    isConfig (Config _) = True
    isConfig _          = False
    extractConfig (Config sel_new) = sel_new
    extractConfig _                = L
    extractData   (Data   (x,y)) = (x,y)
    extractData   _              = (0,0)

    -- actual mux
    mux sel (x,y) = if (sel==L) then x else y
```

Listing 3.23 – Configurable multiplexer

Make components of the examples

Until now, the previously introduced examples are *polymorphic*, i.e. they do not have a specific datatype (as explained in Section 3.1.3). Also, they are not defined as components yet (as explained in Section 3.2.3.

The complete definition of concrete components for the presented examples including their types is shown in Listing 3.24.

```
1  addA,isBiggerA :: Comp (Word,Word) Word
2  addA = arr add
3  isBiggerA = arr isBigger
4
5  aluA :: Comp (Opcode,Word,Word) Word
6  aluA = arr alu
7
8  biggestFromVectorA :: Comp (Vector D8 Word) Word
9  biggestFromVectorA = arr biggestFromVector
10
11 accA :: Comp Word Word
12 accA = acc ⇑ (0::Word)
13
14 confMuxA :: Comp ConfMuxInput ConfMuxOutput
15 confMuxA = confMux ⇑ (L::Selector)
```

Listing 3.24 – Concrete modules for the examples

The functions `add` and `isBigger` have the same type, so they can be defined together. This is done in line 1. In lines 2 and 3, the *arrows*, i.e. the components, for `add` and `isBigger` are defined. Since both components are *stateless*, the `arr` keyword is used as introduced in Section 3.2.3.

The type and component definitions for the other two stateless functions, `alu` and `biggestFromVector` are done in lines 5-9.

For the stateful functions `acc` and `fixedMul`, an initial state has to be defined as explained in Section 3.2.3. This is shown in line 12 for the accumulator, where the initial state is the number 0. For the fixed multiplexer, the initial state is defined in line 15. Here, it is set to *L*, which means initially the left input of the multiplexer is chosen.

3.2.6 Simulation

To simulate a block with a list of test stimuli, CλaSH provides a simulation function called `simulate`. In this section, we will show two example simulations of previously defined components. First, the adder is simulated with a set of input stimuli, then the configurable multiplexer.

The stimuli for the adder are defined as

```
inputAdd = [(1,2),(3,4),(5,6),(7,8)]
```

In each step of the simulation, one tuple of the test stimuli is sent to the adder. That means, in the first step, `(1,2)` is sent to the adder, which should produce the result 3. In the next step, `(3,4)` is used, which should produce 7, and so on.

The expected result of the simulation is then: `[3,7,11,15]`.

Now, the adder can be simulated using the simulation function provided by CλaSH:

```
simulate addA inputAdd
[3,7,11,15]
```

The simulation result shows that the adder works as expected.

To simulate a stateful function, the same method is used. As example, we show a simulation of the configurable multiplexer. At the start of the simulation, the configurable multiplexer contains the initial state which was defined when the component was defined, as shown in Listing 3.24 in Section 3.2.5. The initial state of the configurable multiplexer is thus *L*.

The test stimuli are defined as follows:

```
confMuxInputs = [ Data (8,9)
                , Config L , Data (1,2) , Data (3,4) , Data (5,6)
                , Config R , Data (1,2) , Data (3,4) , Data (5,6)
                ]
```

Listing 3.25 – Simulate the fixed multiplexer

After the initial `Data (8,9)` input, a new configuration (*L*) is sent to the selector. Then, three data tuples are sent. After that, a new configuration is sent, this time containing the value *R*. Finally, three more data tuples are sent.

The expected output of the configurable multiplexer is `(False,0)` in the second and sixth step, since new configurations are sent to the component in those steps. Steps one and three to five should produce the left input, while steps seven to nine should produce the right input. The simulation of the configurable multiplexer shows the following result:

```
simulate confMuxA muxInputs
[(True,8)
,(False,0),(True,1),(True,3),(True,5)
,(False,0),(True,2),(True,4),(True,6)]
```

The results of the simulation hence show the expected output.

3.2.7 VHDL generation

Finally, the VHDL code can be generated by CλaSH.

This is done by first defining the name of the top module in the Haskell code, for the adder *addA* it would be

```
{-# ANN addA TopEntity #-}
```

CλaSH can also automatically generate a test bench, if that is desired, the name of the input stimuli function also has to be defined. In the case for the *addA*, it would be

```
{-# ANN inputAdd TestInput #-}
```

Finally, the command `:vhdl` executed in *GHCi* will trigger the VHDL generation:

```
:vhdl
Total number of transformations applied: 41

Total compilation took 1.285402s
```

During VHDL generation, a number of files is generated. One file contains type definitions and one file contains the definition for the top entity. Furthermore, VHDL file per block is generated. In the case of the adder, three files are generated, one for the type definitions, one for the top entity and one file that describes the actual adder.

The generated VHDL code for the internal behaviour of the adder is shown in Listing 3.26. All identifiers are autogenerated, which makes the resulting VHDL code difficult to read and debug. The VHDL code for the top level entity and the type definitions can be found in Appendix A.

```
entity addComponent_1 is
     port (param2046964804 : in Tuple2_0;
           o2046964827 : out signed_16;
           clock1 : in std_logic;
           resetn : in std_logic);
end entity addComponent_1;


architecture structural of addComponent_1 is
     signal y2046964829 : signed_16;
     signal x2046964828 : signed_16;
begin
     o2046964827 <= x2046964828 + y2046964829;

     y2046964829 <= param2046964804.AB;

     x2046964828 <= param2046964804.AA;
end architecture structural;
```

Listing 3.26 – Generated VHDL code for the adder

3.3 Conclusions

The design language used in this thesis is the functional programming language Haskell. Furthermore, the Haskell-based hardware description language and compiler CλaSH was used to implement the proposed hardware architecture.

To aid the reader to follow the implementation details provided in the following chapters, we presented the basic concepts for both Haskell and CλaSH in this chapter using a number of examples.

Chapter 4

Conceptual Basis for the Dataflow CGRA

Abstract – *In this chapter, we will present the underlying dataflow principles used to configure the CGRA. Dataflow is the key motivation and inspiration for the complete system presented in this thesis. The cores that compose the CGRA are based on dataflow principles, both in terms of their execution mechanism as well as their configuration principle. Also the programming principle, which we developed to implement and map algorithms to the architecture, is based on dataflow principles.*

4.1 Motivation

Figure 4.1 shows an abstract view of our proposed system. The algorithm on the left is a typical example of a desired target algorithm - a regular structure, simple operations and a large degree of instruction-level parallelism. The architecture on the right is an illustration of a CGRA with small, configurable cores interconnected in a 2D array.

In Chapter 2, we introduced CGRAs and their programming methods. Most existing CGRAs are programmed in an imperative approach, usually, a restricted subset of C is supported. Hereby, the CGRA-specific compiler is responsible to detect the parallelism of the implemented algorithm.

In our opinion, the reason for choosing C to program CGRAs is not evident. Since C has been designed as a sequential language, it lacks intuitive support to express fine-grained parallelism. There are no constructs available to describe an algorithm

Major parts of this chapter have been published in [AN:1, 2]

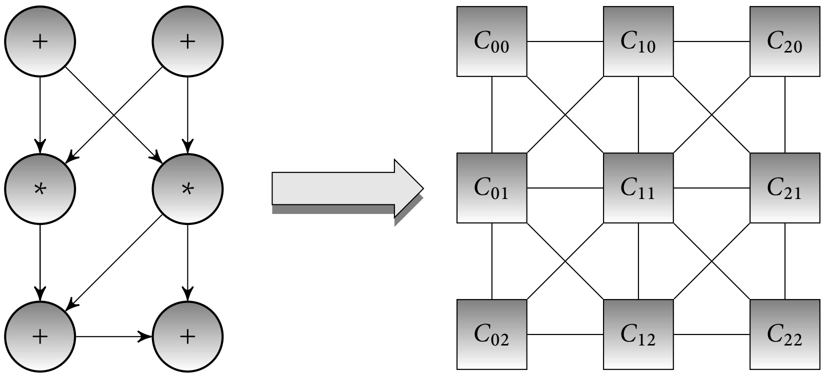


Figure 4.1 – Motivation

in terms of its structure. Furthermore, C is not a *single assignment language*, that means, variables can be modified multiple times, which makes automatic parallelisation of algorithms hard. Moreover, because of the frequent use of pointers in C, it is sometimes untraceable where and when a certain variable is updated.

Motivated by that, we developed a novel configuration paradigm for the presented system that has a close relation to the target application domain, both how algorithms are executed on the architecture, i.e. the *execution principle of the cores*, but also how algorithms are implemented, i.e. the *programming paradigm*.

The general idea which our approach is based on, is that algorithms of our target application domain – DSP algorithms with a large degree of fine-grained parallelism – resemble dataflow graphs. By this we mean that a DSP algorithm can be represented by a set of operators that consume and produce data tokens. The operators relate to each other in a certain way, i.e. they communicate. This principle can be seen as a dataflow graph, where the operators are represented by nodes, and the communication structure (i.e. dependencies) by arcs.

Our programming paradigm thus enables a designer to describe algorithms in terms of their structure, that means, describe the operations of the algorithm and their data dependencies.

4.2 Conceptual view on the algorithm

We consider a DSP algorithm as a dataflow graph, i.e. as a collection of communicating nodes. Hereby, the graph can be seen from two different views. One is the local behaviour of the nodes, and the other is the global communication pattern between the nodes. In our programming paradigm, we refer to those two views as the *local view* and the *global view*.

In the compiler, we use this principle to partition the code generation step into two parts, thus we decouple local behaviour from global communication. We will elaborate on that in chapter 6.

4.2.1 Local view

Figure 4.2 shows a high-level illustration of the local view.

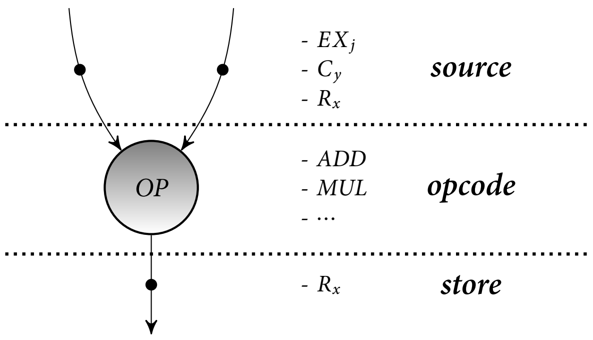


Figure 4.2 – Local view

The node in the dataflow graph is defined in terms of

» the ***source*** of each input (***EX***ternal input ***j***, a **C**onstant value ***y***, ***R***egister ***x***),

» the ***opcode*** defining the current operation (***ADD, MUL, ...***) and

» whether to ***store*** the result at the internal ***R***egister ***x*** or not

On the input arcs, a token indicates that input data (i.e. a data token) is required on that arc to trigger the execution. On the output arc, a token indicates whether output data (i.e. a data token) is produced to the external world, i.e. a token is sent out of the core, or if the result is only stored locally inside the core (then, no token would be produced on the output arc).

4.2.2 Extended local view

On the conceptual level, the distinction between local nodes and global communication makes sense. As soon as the graph is mapped to the architecture, however, the definition of the *local view* has to be extended. Since multiple dataflow nodes can be mapped to one physical core, the definition of the *extended local view* covers the local behaviour (i.e. configuration) of one *core* instead of one *node*.

Describing an algorithm as a dataflow graph is a very straightforward way, but only when the algorithm is very simple and regular. As soon as for example initial tokens, or loops are required, pure dataflow notation quickly reaches its limits [38].

Therefore, we decided to extend the pure dataflow notation with finite state machine (FSM) notation. The configuration of a single core is then described as a set of FSM states, and each FSM state is defined as a dataflow node. The transition conditions between the states are determined by the number of required iterations in the respective state. It is important to note here that the amount of iterations per state is determined at design time and thus fixed. This means that the amount of states of the extended local view is fixed at design time.

Examples of the extended local view

In the following, we will present three examples to illustrate the principle of the extended local view.

The example state machine in Figure 4.3 shows a configuration consisting of two states. The condition to transit between the states is defined by the number of iterations per state, indicated by the variable i on the arcs of the state machine. The variable i counts the number of iterations *per state*, i.e. as soon as the FSM enters a new state, i is reset to zero. The FSM remains in the left state for two iterations, then transits to the right state, where it remains for only one iteration, i.e. it executes only once. Then it switches back to the left state and so on. One iteration is defined as one firing of the dataflow graph of the current state. A core with this configuration would first perform a multiplication on two incoming token pairs, and then an addition on the next incoming token pair. In each state, it would produce an output token. In the left state, i.e. after the multiplication, the result would not be stored locally, whereas the result of the right state, i.e. the result of the addition, would be stored at register 0.

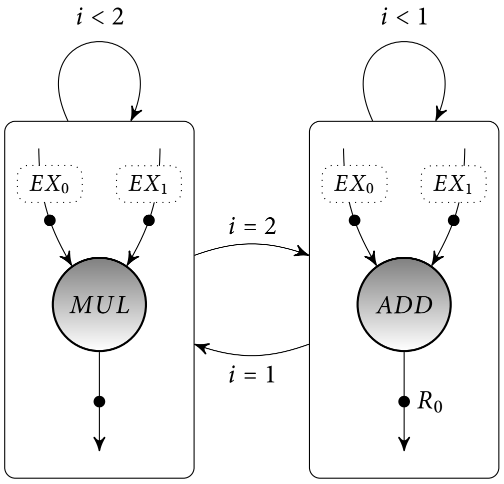


Figure 4.3 – Extended local view, first example

The second example, shown in Figure 4.4 shows a slightly more complex example. In total, three states are used to describe the behaviour. The configuration which is described by Figure 4.4 is as follows.

In the leftmost state, an addition is performed on two external inputs (EX_0 and EX_1), and one output token is produced. After two iterations (indicated by $i < 2$ at the self loop arc on the top of the state), the FSM transits to the next state, i.e. the middle one. In this state, a multiplication is performed on an external input (EX_0) and a constant factor 2 (indicated by C_2 on the input token on the right arc). The result is stored in register R_0, but no output token is produced. After one iteration, the FSM transits to the next state. In this state, which is represented by the rightmost state in the figure, a subtraction is performed of the value stored in register R_0 (that has been generated in the previous state) and a constant factor 1. A token is produced, containing the result of the operation.

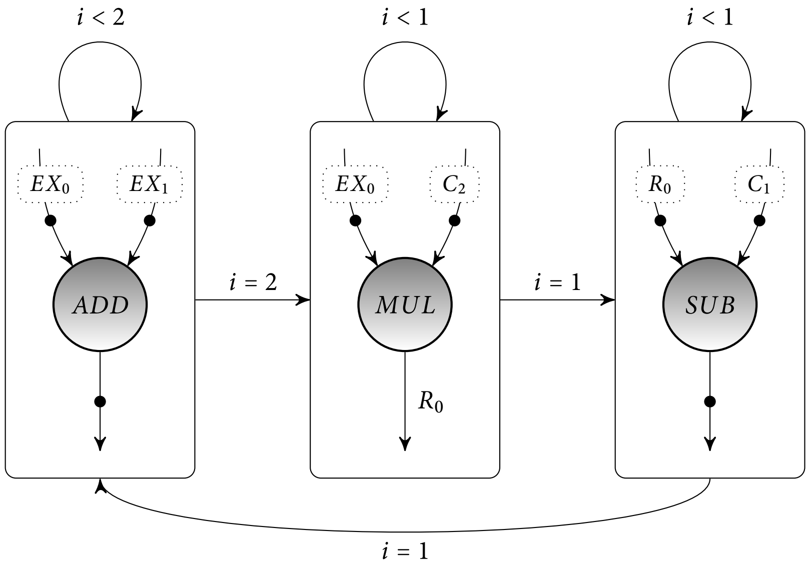


Figure 4.4 – Extended local view, second example

The third example shows an FSM that describes an initial state, followed by a state which is continuously repeated. The left state describes an addition on two external inputs (EX_0 and EX_1), which is executed twice (indicated by $i < 2$). After two iterations have been executed, the FSM transits to the next state and remains in this state for the rest of the runtime of the system. During this state, a multiplication is performed on one external input (EX_0) and a constant factor 2.

Relation to synchronous dataflow models

Our work is motivated by dataflow principles. Hence, it would seem a logical choice to use an existing dataflow formalism to model the behaviour of our dataflow CGRA. One candidate for a formal representation of the class of algorithms that we consider is *cyclo-static dataflow* (*CSDF*), which was mentioned in Section 2.1.3.

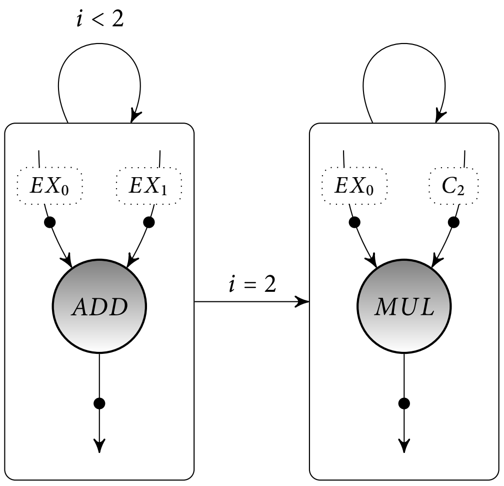


Figure 4.5 – Extended local view, third example

However, as CSDF is used to model dataflow graphs which can be analysed for e.g. throughput and timing behaviour, it is not suitable as our programming paradigm. Furthermore, CSDF does not actually describe the behaviour of the nodes, i.e. the operations within the nodes. For the analysis of CSDF graphs, the actual operation of a node is irrelevant, but it is essential to describe the actual behaviour of nodes for our programming paradigm.

Another reason why we chose a combination of state machines and dataflow notation, and not a CSDF notation, is that we wanted to be able to express initial states, which is not possible in CSDF. Using state machines gives us a much greater flexibility in expressing the behaviour of the dataflow CGRA.

In the following, we will illustrate the relationship of the extended local view to *CSDF* using the three examples. We will show how CSDF relates to our approach and also give one example which cannot be expressed using CSDF. The syntax we use to express the CSDF graphs is adopted from [92].

The first example, i.e. Figure 4.3, can be represented using a CSDF graph as shown in Figure 4.6. The CSDF actor labelled *OP* represents the operation. The internal behaviour of the node, i.e. the operation, cannot be directly specified in CSDF, however, the runtimes of the operation in each CSDF phase are specified above the node: $< \rho_*, \rho_*, \rho_+ >$. The two external inputs EX_0 and EX_1 are shown on the left, the annotations above the arc describes when a token is consumed or produced, e.g. $< 0, 0, 1 >$ on the bottom right means that only in the last phase a token is produced for R_0. In total, the operation node has three different phases, two multiplication phases and one addition phase. In each of the phases, it consumes on token of each of the external inputs. Also, in each phase, one output token is produced. The register R_0 is represented as a separate actor on the right side of the graph. On the arc going to the register, a token is only produced in the third phase, i.e. the addition step.

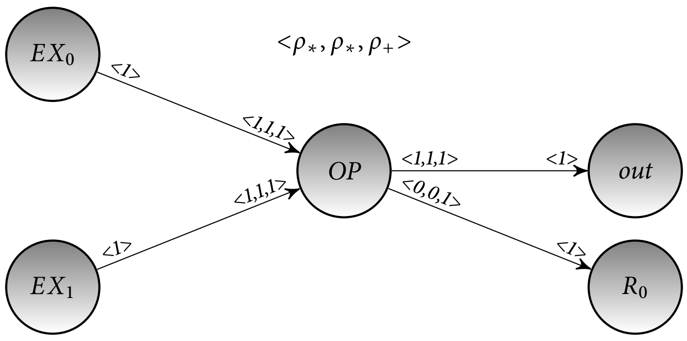


Figure 4.6 – CSDF representation of the first example

The CSDF representation of the second example, i.e. Figure 4.4, is shown in Figure 4.7. Again, the two external inputs EX_0 and EX_1 are shown on the left, but also two sources for the constant factors, represented by the actors C_2 and C_1. Furthermore, the register R_0 is shown at the right side of the graph. In contrast to the previous example, there is not only an arc going *from* the operation node to the register, but also from the register *back* to the operation node.

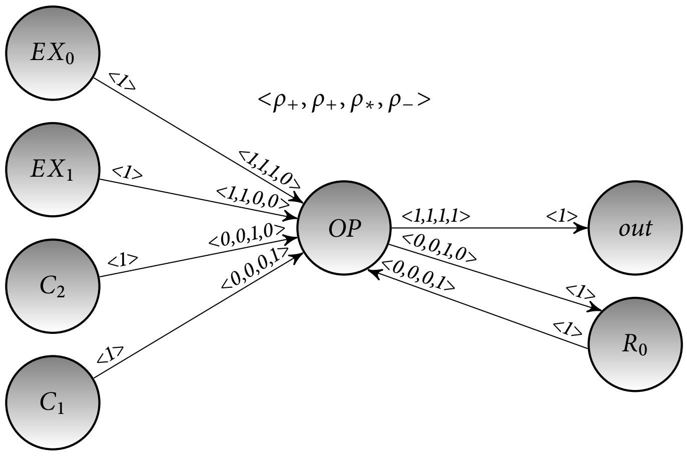


Figure 4.7 – CSDF representation of the second example

The previous two examples could be represented using CSDF notation by introducing specific actors for the inputs. However, the third example, i.e. Figure 4.5, cannot be represented using CSDF. The reason is, that the example contains an initial state which is only visited in the startup phase, and then never again. This non-repetitive behaviour cannot be modelled with CSDF, since CSDF can only describe repetitive execution patterns.

This last example illustrates the reason why we did not choose CSDF to describe the behaviour of our system. A limited subset of our presented configuration paradigm could be modelled using CSDF, however, as we show in the later chapters of this thesis, we in particular rely on initial states to configure our architecture, hence we cannot use CSDF.

4.2.3 Global view

While the *(extended) local view* defines everything that happens inside a core, the global flow of data is out of the core's scope. A core only has the notion that an input can come from an external source, e.g. another core or an external input, but precisely from where is irrelevant for the core. Consequently, for the flow of data a global dataflow scheme is required, i.e. the *global view*.

On the conceptual level, the *global view* defines the communication dependencies in the dataflow graph representing the DSP algorithm. On the architectural level, the *global view* defines the global flow of data within the array, i.e. which cores communicate with each other. The actual data transmissions are managed by the routing logic in the array.

The global view is illustrated in Figure 4.8. In this example, four cores (C_{00} to C_{11}) communicate using the destination core's relative position and the identifier of the destination core's inputs. To indicate the relative position, cardinal directions are used, i.e. if a core wants to send data to it's right neighbour, the direction would be east, to send data to the left neighbour, it would be west.

For example, C_{00} is sending a token to input 0 of C_{10} by annotating $(E, 0)$ to the token (since C_{10} is east of C_{00}) and C_{01} is sending a token to input 1 of C_{10} by annotating $(NE, 1)$ to the token since the relative position of C_{10} to C_{01} is north east.

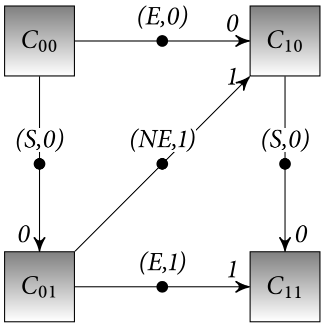


Figure 4.8 – Global view

4.3 Conclusions

In this chapter we presented the conceptual basis for the presented system. As explained in Chapter 1, a key motivation and requirement for our system was *programmability*. That means, our focus was on the development of a programming and configuration paradigm that can be used to implement DSP streaming applications containing a large degree of instruction-level parallelism.

We consider DSP applications to be composed of two views: The *(extended) local view*, which represents everything that happens within one core, and the *global view*, which represents the flow of data through the array. For our programming paradigm we adopted principles from dataflow and finite state machine (FSM) notations.

The FSM notation allows us to add control to the nodes and to iterate through successive states and hence (for example) define initial tokens or feedback loops. By extending pure dataflow notation with finite state machines, we can switch between dataflow graphs and hence also express initial states, which cannot be expressed by e.g. CSDF. The dataflow principle, especially the firing rule, allows us to look at each node in the graph individually without the need for an explicit global synchronisation mechanism.

Chapter 5

Architecture

Abstract – *In this chapter, we present the proposed architecture, developed to efficiently execute streaming algorithms. First, we motivate our design choices and give a list of requirements. Then, the separate components of the architecture are presented. We illustrate how the programming paradigm, presented in the previous chapter, is applied to the architecture. We finish the chapter with an example on how a concrete algorithm is executed on the architecture, using the proposed programming paradigm.*

5.1 Overview and goal

The presented architecture is targeted at streaming algorithms that contain a large degree of fine-grained parallelism. Those algorithms usually have a regular structure. Examples are matrix manipulations and filter operations that are common in audio and video processing. The parallelism available in those kind of algorithms is on a low level, i.e. on the instruction level. Usually, the elementary computations in those algorithms are simple, for example additions or multiplications.

The presented architecture belongs to the class of coarse-grained reconfigurable arrays (CGRA), that were already introduced in Chapter 2. A number of small, configurable cores are interconnected to form a reconfigurable array. Each core by itself has very limited functionality and processing power, but the complete array possesses a large amount of computing power that can be used to execute algorithms with a large degree of fine-grained parallelism. The array is regular and thus easily extendable and configurable in size.

Each of the cores in the architecture follows the dataflow principles. That means, the execution mechanism is inspired by dataflow firing rules, i.e. as soon as all the

Major parts of this chapter have been published in [AN:1, 2]

required operands for the current operation have arrived, the operation is executed and a result is produced. The programming paradigm for the cores follows the paradigm presented in Chapter 4.

5.2 Implementation

The complete architecture was implemented using CλaSH. By using CλaSH, the architecture was implemented on a high level of abstraction. Furthermore, the Haskell interpreter can be used as simulation environment which can speed up simulation times significantly compared to a simulation of pure VHDL code.

Not only the architecture, but also the remaining parts of the complete framework, i.e. the programming language, the compiler and the simulation environment, are implemented using Haskell. In the next chapter we will demonstrate the working principle and the implementation of the programming language and the compiler for our architecture.

By using one language for the complete design process, it is not necessary to switch between different design environments, or languages, which is common in the currently used design approaches. Additionally, the datatypes which we define for our architecture are the same as used by the compiler.

5.3 General principles

We base our architecture on streaming and dataflow principles. The target algorithms are streaming algorithms that contain a large degree of parallelism and simple elementary operations.

Furthermore, we assume that input data is provided as a stream of tokens. Therefore, the firing rule from dataflow is very well suited as execution mechanism of the cores in the architecture. In the cores, no program counter is required, since the execution is triggered by the arrival of data. When, for some reason, data is delayed, it does not affect the functionality of the core, since the firing rule can handle delayed data.

5.4 Architecture - hardware

Figure 5.1 shows the top level of the proposed architecture. The blocks denoted $C_{00} \cdots C_{33}$ represent simple reconfigurable cores. Each core includes a function unit, a local register file, and a small program memory. The cores are interconnected by means of local nearest neighbour connections. In addition to that, external data can be provided by either a broadcast input (denoted b) which is connected to each core, or by two local inputs per core (denoted e). The external inputs come from the external world, i.e. in an actual hardware implementation, they would be connected to pins. For our simulations, they are connected to the testbench. Finally, all cores have a configuration input (denoted c).

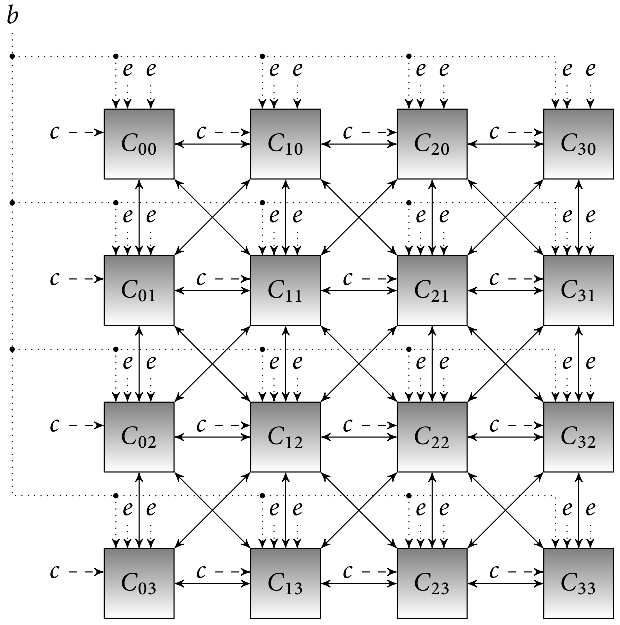


Figure 5.1 – Architecture

The complete architecture was implemented using CλaSH. In the following sections, we will explain all the parts of the architecture in detail. Furthermore, we will include a number of code snippets to illustrate the implementation in CλaSH.

5.4.1 Requirements

The presented architecture was designed to execute regular streaming DSP applications, like matrix manipulations or filtering operations. Based on the target application domain, certain requirements for the architecture were identified:

1. The target applications have a regular structure where data is sent from one operator to the next. Based on that, our architecture also has a regular structure. That means, the cores are interconnected with *nearest neighbour communications* using local links (although also other interconnection structures could be used).
2. Each core only needs *limited functionality*, i.e. simple mathematical operations like additions and multiplications are sufficient for the targeted application domain. More complex operations can be composed with combinations of these operations.
3. To quickly reconfigure the cores, each core is equipped with a *separate configuration input (c)*. This configuration input can be used to directly send a

new configuration to a core. In theory, (re)configuration could be done in parallel to normal execution, although this is not shown in this thesis.

4. In the target DSP applications, it is often required to broadcast input signals. Therefore, the architecture is equipped with a *broadcast input (b)* that can be used to send an input to (a subset of) all cores simultaneously.

5.4.2 Interconnect

Figure 5.2 shows a close-up of one core of the array shown in Figure 5.1 with a focus on the connectivity.

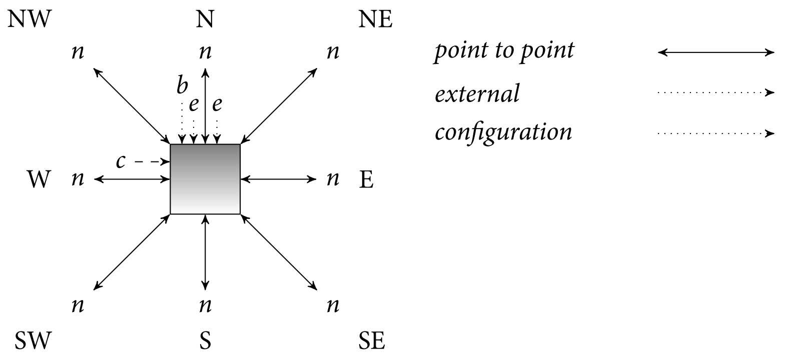


Figure 5.2 – Connections of one core

Each core is connected to its *direct neighbours* via *point-to-point links*, indicated by the continuous lines, labelled *n*. The relative position of the nearest neighbours is specified using compass directions, i.e. **N**, **NE**, **E** ... The nearest neighbour connections enable each core to directly communicate with its eight direct neighbours to support locality of reference, which is important for energy efficiency [72]. The reason to connect each core to its eight direct neighbours instead of only four is to provide greater flexibility when algorithms are mapped to the array.

Furthermore, *external input signals* are available to each core. Those inputs are represented by the dotted lines. Each core is connected to a *global broadcast* input, labelled *b*, that can be used to stream data to (a subset of) all cores simultaneously. Besides the broadcast input, each core has two additional external inputs, labelled *e*. These inputs are connected to the input pins of the cores and are used to provide a core with individual input data.

To program the cores, each core has a *configuration input*, represented by the dashed line, labelled *c*.

5.4.3 Number datatypes

As explained in Section 3.2.1, the bitwidth of numbers has to be defined in CλaSH. In Listing 5.1, we show the definitions for the number types in our architecture. First, we define in lines 1 and 2 that a *Word* contains 16 bits and is a signed integer. Since we support both integer and fixed point numbers in our architecture, a tag *NumType* to identify the number type is defined in line 4. *NumType* can either have the value *NUM*, which indicates an integer, or *FP*, which indicates a fixed point number. Line 5 defines a number type that includes the tag. Line 7 defines a type named *Operand* which is used as standard operand type in our architecture. In our architecture, the standard operand type is the previously defined *Number*, i.e. the number-type identifier followed by the actual data. Finally, line 9 defines an operand tokens that indicates whether a current operand contains valid data or not by using Haskell's **Maybe** datatype. An instance of type *OpToken* can either have the value **Just** *a* where *a* is of type operand and contains a valid operand, or **Nothing** which indicates that no valid data is available. The **Maybe** type corresponds to a *valid* bit.

```
type WordLength = D16                         1
type Word       = Signed WordLength           2
                                              3
data NumType    = FP | NUM                    4
type Number     = (NumType, Word)             5
                                              6
type Operand    = Number                      7
                                              8
type OpToken    = Maybe Operand               9
```

Listing 5.1 – Definition of important datatypes

5.4.4 Core

Figure 5.3 shows the internal details of one core. It consists of five main elements: the *ALU*, a small *register file* (*REG*), the *program memory* (*PMEM*), the internal *state* of the core (*CoreS*) and the block denoted *firing_rule*, which implements the previously introduced *firing rule* from dataflow, i.e. it checks whether the required operands for the current operations are available. Together with the program memory, the *firing rule* represents the local control of the core. Additionally, an *input buffer* is present at the input of the core, which manages the incoming data streams from other cores and the external inputs. A black line indicates a data line, a grey line represents a control signal. In the following sections, each of these elements will be presented in more detail.

The input signals follow the same scheme as used in Figure 5.2, i.e. the inputs from neighbouring cores are represented by continuous lines, the external inputs are represented by dotted lines, and the configuration input is represented by the dashed line.

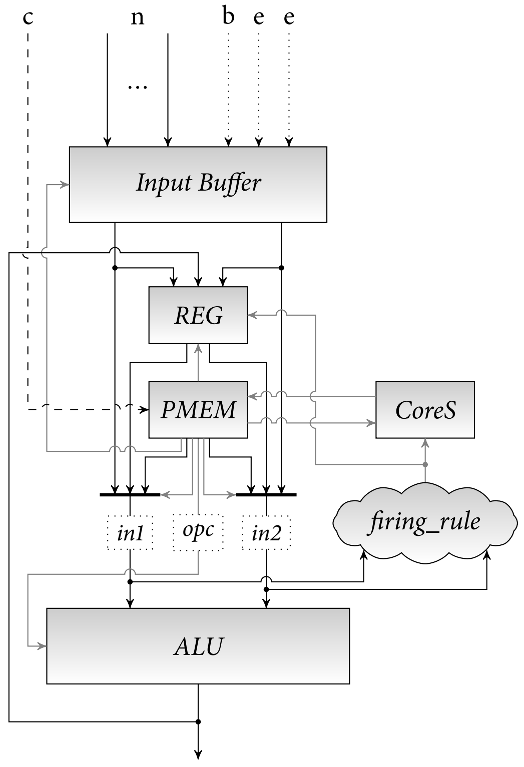


Figure 5.3 – Detailed core

Core state

The configuration principle of the presented architecture follows the programming paradigm that was presented in Chapter 4. Each core is configured using a finite state machine (FSM), where each state is defined by a dataflow actor defining the current behaviour of the core.

In the state of the core (`CoreS`), two properties are stored: The current FSM state which the core is in, and the current iteration of that state. The state of the core is used to control the program memory, as explained in the following section.

The definition of the state of the core is as follows:

```
type CoreS = ( SIndex , Iterations )
```

`SIndex` hereby refers to the current index of the program memory, i.e. the state of the FSM, `Iterations` denotes the number of iterations already executed in the current state. When a new state is entered, the iteration count is reset, the counting of iterations in the new state starts at zero.

Program memory

The program memory, labelled *PMEM*, stores the configuration of a core. It is implemented to support the programming paradigm presented in Section 4, i.e. that each core in the architecture is programmed using a finite state machine where each state is defined as a dataflow actor.

Figure 5.4 is an illustration of the working principle of the program memory. Its main element is the storage for the actual configurations. This storage represents the finite state machine which was explained in Section 4.2.2, i.e. the extended local view. Each entry in the storage pointed to by `SIndex` represents a state in the configuration state machine, i.e. the local view as explained in Section 4.2.1.

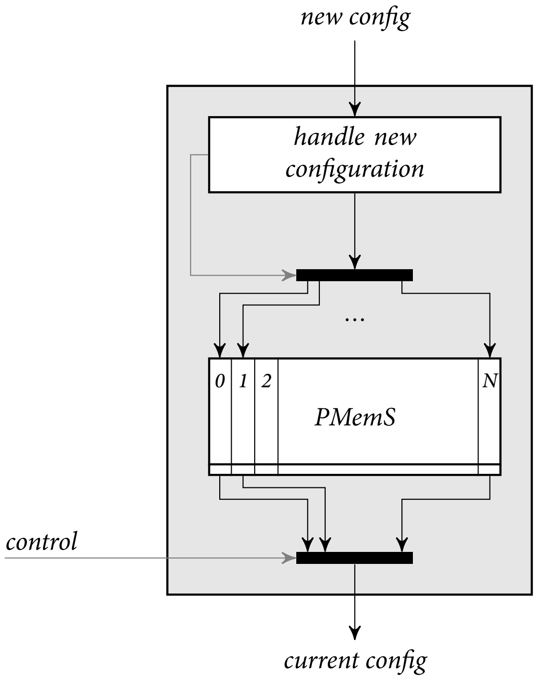


Figure 5.4 – Program Memory

The program memory has two inputs: One which carries new configuration data, denoted *new config* in Figure 5.4. In Figure 5.3, this input is represented by the dashed input denoted *c*. The other input to the program memory is a control input, denoted *control*. The control input is, as shown in Figure 5.3, the current state of the core *CoreS*. As explained in the previous section, *CoreS* contains the current state of the FSM *SIndex* and the current iteration *Iterations*. *PMemS* contains the actual configurations, i.e. the *local views*. The output of the program memory is the current configuration, i.e. the current dataflow actor (the local view), hence the configuration which the core is supposed to execute in the current clock cycle.

The program memory is implemented as Haskell datatype. The definition is as follows:

```
type PMemS = Vector PMemL PMemEntry
```

PMemS defines the content of the program memory, i.e. the storage for the configurations. *PMemS* is a vector of length *PMemL* of configuration entries of type *PMemEntry*, i.e. a vector of *local views* as introduced in Chapter 4.2.1.

PMemEntry defines the actual datatype for one configuration, i.e. for a *local view*. The implementation is shown in Listing 5.2. The definition of *PMemEntry* follows the scheme presented in Section 4.2. In Figure 5.5, the relation between the *local view* as presented and the definition of *PMemEntry* is shown

```
type PMemEntry =
  ( OpCode          -- defines opcode
  , Source          -- source of left input
  , Source          -- source of right input
  , Store           -- store result in regfile
  , OutToken        -- produce output token
  , Destinations    -- destinations of result token
  , Iterations      -- number of iterations in current state
  , SIndex          -- next state
  )
```

Listing 5.2 – Definition of a program memory entry

The left part of the image, i.e. the items *OpCode* to *Destinations* represent the *local view*, i.e. one stage in the configuration FSM. The right part of the image, i.e. the items *Iterations* and *SIndex* represent the *extended local view*, i.e. the transition conditions between the configuration FSM stages.

OpCode defines the opcode, i.e. the current operation. Possible values for the *OpCode* are defined by the functionality of the ALU. ***Source*** defines where the current input comes from. As explained in Chapter 4.2.1, the source can either be the ***EX***ternal input ***j***, a ***C***onstant value ***y***, or a value stored at ***R***egister ***x*** inside the *REG* module of the core. ***Store*** defines whether the result should be stored in the core's register file. The format is a boolean, indicating whether it should be stored, followed by the index of the register file. ***OutToken*** defines whether an output token is produced. The possible values are either *High*, indicating that a token is produced, or *Low*, indicating that no token is produced. ***Destinations*** defines the destination address(es) of the outgoing token. Hereby, the relative position of the destination core is used, i.e. the compass directions. ***Iterations*** defines the number of required iterations in the current state. ***SIndex*** denotes the index of the next state.

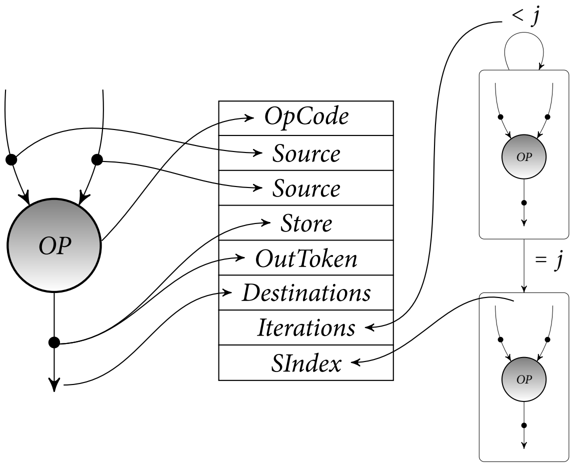


Figure 5.5 – Relation of the local view and the extended local view to the program memory

In Section 5.5, we will give an example configuration to illustrate the working principle of the program memory. In Table C.1 in Appendix C, a more elaborate explanation on the datatypes is given.

Listing 5.3 shows the implementation of the program memory in CλaSH. In line 1, the input and output signals are defined. `s` and `s'` are the current and new state, respectively. `i` is the input to the program memory and `out` is the output of the program memory. In line 3, the input is split into the separate input signals: `new_config` and `control`, as shown in Figure 5.4. In line 4, `new_config` is then further split into its elements, the tuple `(np,ni,p)`. `np` is a status bit indicating whether a new configuration is available, `ni` is the index of the new configuration and `p` contains the new configuration itself. `control` is the current state of the core, i.e. the current state of the state machine. In line 5, the output of the program memory is determined. Since the program memory itself is a vector of configurations, as explained before, the output is the vector element indexed by the current state of the core `s!control`, i.e. the current configuration. In lines 6 and 7, the state of the program memory is updated. The case in line 6 represents the case that no new configuration has arrived, the state thus remains the same. In line 7, the program memory is updated with a new configuration. Hereby, the CλaSH specific function **`vreplace`** is used to *replace* the vector `s` at index `ni` with the entry `p`.

```
pmem s i = (s',out)
  where
    (new_config,control) = i
    (np,ni,p)            = new_config
    out                  = s!control
    s' | np == Low = s
       | otherwise = vreplace s ni p
```

Listing 5.3 – Implementation of the program memory in CλaSH

During normal operation, in each clock cycle, the current configuration of the core is determined and extracted as follows:

```
(ps',curr_p) = pmem ps pi                                          1
(opc,source1,source2,store,outT,dest,it,sIndex) = curr_p           2
```

Listing 5.4 – Current configuration

In line 1, the current entry of the program memory is extracted as explained in the previous paragraph and Listing 5.3. In line 2, the separate elements of the current configuration are extracted. The current configuration is of type `PMemEntry`, the identifiers of its components resemble their respective function, which was shown in Figure 5.5. The first component in `curr_p` with the name `opc` hence represents the `OpCode`, the second and third components with the names `source1` and `source 2` correspond to the `Source` of the two inputs, and similarly for the remaining components.

Input buffer

At the input of the core, the incoming signals are connected to an *input buffer*. The input buffer consists of an array of FIFO buffers. The width of the array is configured during design time. In the current prototype of the architecture we defined the width to be four as a trade-off between flexibility and complexity. A detailed schematic of the input buffer is shown in Figure 5.6.

In total, the input buffer has 11 data inputs:

» (up to) eight inputs from the neighbouring cores (depending on where in the array the core is located)

» the broadcast input

» two external inputs

Each incoming data token which arrives at the input buffer carries information to which port, i.e. FIFO in the input buffer, it should be sent. This information is transferred to the multiplexer at the input of the buffer (represented by *in_ctrl* in

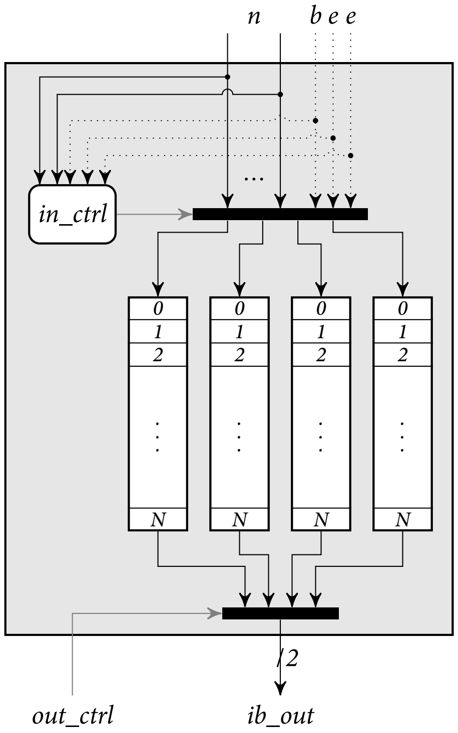


Figure 5.6 – Input Buffer

Figure 5.6) which then sends the data token to the respective FIFO. Since there is no explicit hardware support to resolve conflicts, i.e. when two inputs want to write to the same entry in the buffer in the same clock cycle, the compiler has to ensure that this case cannot occur.

The output width of the input buffer, i.e. the number of inputs to the actual core, is set to two. This means that two words can be read from the input buffer simultaneously. This is because the ALU supports only binary operations. The multiplexer at the output of the input buffer is controlled by the signal *out_ctrl*, which is generated by the core the input buffer is connected to and depends on the current entry of the program memory.

ALU

The main computational element of the core is the *ALU*. The ALU is responsible for the mathematical operations of the core. It has three inputs: the two operands *in1* and *in2* and a control input *opc* which carries the *opcode*, i.e. defines the operation. The bitwidth of the input operands and the output is the same and is defined at design time, refer Section 5.4.3. The ALU can handle both fixed point and integer operations.

In Figure 5.7, a schematic view of the ALU is shown. At the top, the operands *in1* and *in2* are shown. At the left, the control input *opc* is provided. Listing 5.5 shows the actual implementation in CλaSH. The two operands are sent to the operational modules in the ALU, the opcode determines which function unit output is used as the result for the current operation.

Please note that the ALU does not have any status bits to check for error states like division by zero or overflow. This is a design choice since it is not desirable to propagate an error state throughout a dataflow program. It is assumed that the input DSP application graph is error-free, i.e. no faulty operations are executed in the ALU. For example, in most DSP algorithms an overflow leads to a result representing the minimum or maximum representable value of the integer or fixed point range. In streaming DSP algorithms, there is usually no time to handle exceptions as the next sample is coming in and cannot be delayed.

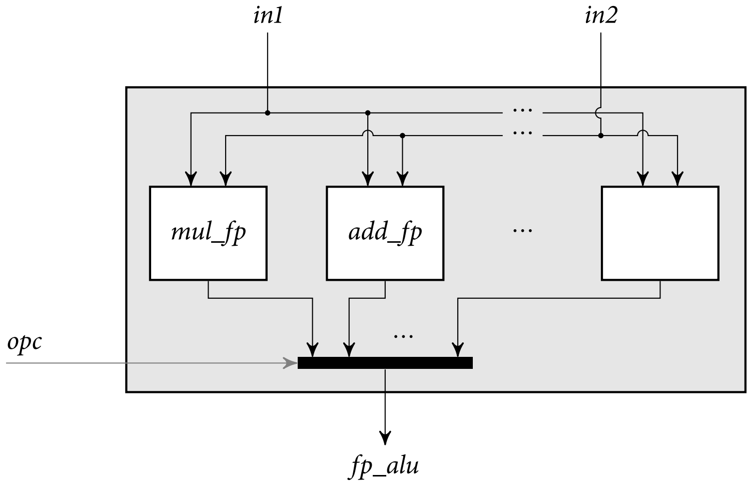


Figure 5.7 – ALU

```
fp_alu :: OpCode → Number → Number → Number
fp_alu opc in1 in2 = res
    where
        res = case opc of
               MUL → mul_fp in1 in2
               ADD → add_fp in1 in2
        ...
```

Listing 5.5 – Implementation of the ALU

Both integers and fixed point numbers are supported in the architecture. Each supported mathematical operation is implemented as a separate component, i.e. one adder, one multiplier, and so on. In each component, the respective mathematical operation is implemented for integer and fixed point numbers. The operations for fixed point numbers are implemented using standard mathematical operations and the required shift operations. As an example, the implementation of the fixed point adder and the fixed point multiplier are provided in Appendix B.

Local memory

In each core, a register file, denoted *REG* in Figure 5.3, is available to store intermediate data. The number of write and read ports is parameterised during design time. The current prototype of our architecture has one write port and two read ports.

The number of write ports was determined as follows: the ALU produces one (or none) result token per clock cycle. This token can be stored in the register file.

The number of read ports is based on the following analysis: The ALU operates on two tokens per clock cycle. Thus, a maximum of two tokens might be required from the register file per clock cycle. Hence, the number of read ports is set to two.

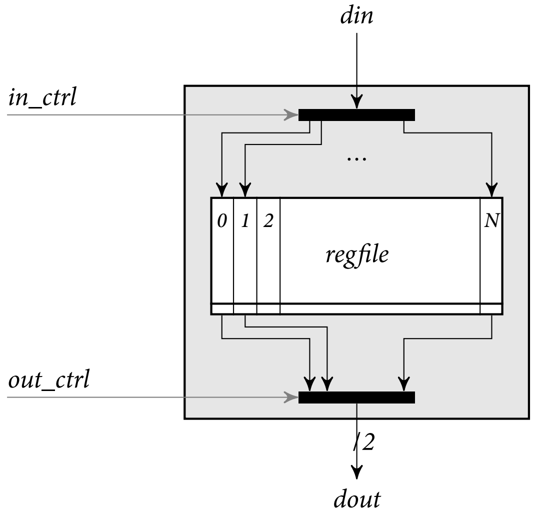


Figure 5.8 – Register File

Figure 5.8 shows a schematic view of the register file. On the left, the data inputs *din* are provided, and the control input *in_ctrl*. Depending on the control signal, the register file is updated with the input data. At the output of the register file, a control signal *out_ctrl* determines the current output signals of the register file. The control signals, i.e. *in_ctrl* and *out_ctrl* are generated by the core and depend on the current entry of the program memory.

Control

In Figure 5.9, a schematic view of the control to update the different parts of the internal core state is shown. The control of the different components in the core is implemented per component, and not as a central control instance. Hence, in Figure 5.3, no central control component is shown. The control logic presented in this section is thus the combined control logic of all components in the core.

In each clock cycle, the internal core state itself is updated, which consists of the following components:

» the core state `CoreS`, i.e. the current iteration and current state in the configuration FSM

» the content of the register file

» the content of the input buffer

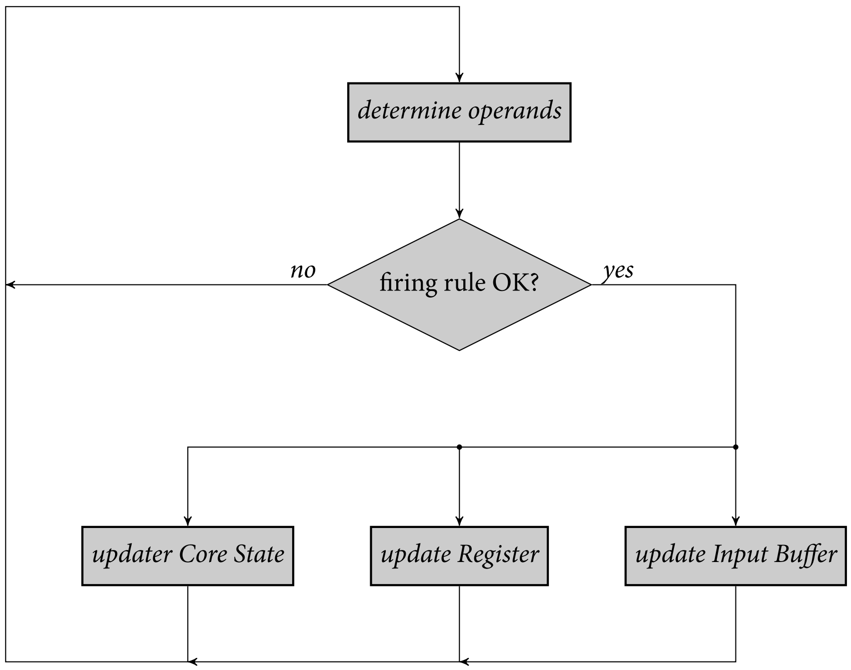


Figure 5.9 – Core Control

The first step in the control process is to determine the operands that are sent to the ALU together with the current opcode as defined in the current configuration. The implementation is shown in Listing 5.6.

In lines 1 to 3, the function *get_op* to determine the correct operand according to the current configuration is defined. The current operand is determined depending on whether it is an external input (line 1), a constant value from the program memory (line 2) or a value stored in the register file (line 3). The first argument *a* indicates whether it is the left or right input of the ALU. *ib_out* is the output of the input buffer, *ro* is the output of the register file.

In lines 5 and 6, the function *get_op* is called for the left (line 5) and right (line 6) input. *source*1 and *source*2 are the current configurations for the left and right input of the ALU, respectively, as shown in Listing 5.4

```
get_op a (EX _) = ib_out!a          1
get_op _ (C  x) = Just x            2
get_op a (R  _) = ro!a              3
                                    4
op1 = get_op 0 source1              5
op2 = get_op 1 source2              6
```

Listing 5.6 – Determine the current operands

As soon as the current operands are determined, the firing rule can be applied. That means, it has to be checked whether the current operands satisfy the firing rule, i.e. they contain a valid value and are not **Nothing**. In the current implementation of the architecture, each operation requires two operands, hence both operands have to contain a valid value for the firing rule to be satisfied. The implementation is shown in Listing 5.7.

```
firing_rule_satisfied = (op1 /= Nothing) && (op2 /= Nothing)
```

Listing 5.7 – Firing rule

The update of the content of the register file and the input buffer is performed in the register file and input buffer directly, as explained earlier in the respective sections.

The update of the core state, i.e. the current number of iterations *ii* and stage *is*, is implemented as shown in Listing 5.8. Three cases are possible: Either, the firing rule is not satisfied (line 2), in that case the core state remains unchanged. Or, the firing rule is satisfied, but the current number of iteration has not yet reached the maximum number of iterations required in the current state (line 3). Then, the number of iterations is increased by one, but the FSM state remains the same. Or, the number of iterations in the current FSM state has reached the required maximum, in that case, the current number of iterations is reset to zero and the FSM state is set to the next state *sIndex* according to the current configuration (line 4). *sIndex* is hereby the next state according to the current configuration as explained in Listing 5.4. That means, the core transits to the next state in its configuration FSM.

```
(ii',si')
  | not firing_rule_satisfied = ( ii   , si     )
  | ii < (it-1)               = ( ii+1 , si     )
  | otherwise                 = ( 0    , sIndex )
```

Listing 5.8 – Update Internal Core State

5.5 Example of a configuration

In this section, we will explain how the programming paradigm explained in Chapter 4 is applied to the presented architecture. In the following chapters, we will show more elaborate examples and also the automatic code generation.

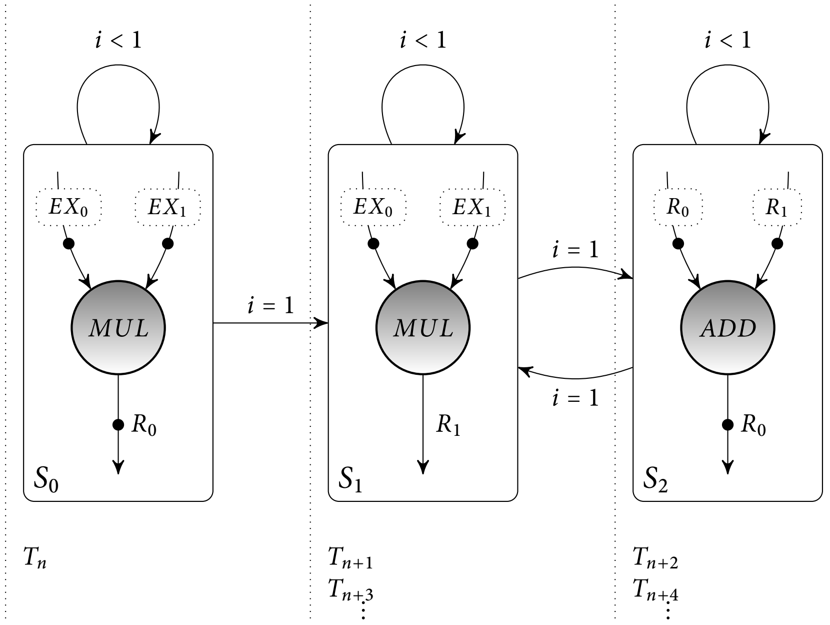
For illustration, we use the example of a pipelined multiply-accumulate (*mac*) operation on data streams. The *mac* operation on the streams **x** and **y** is defined as follows:

$$mac(\mathbf{x},\mathbf{y}) = \sum_{i=0}^{N} x_i y_i = x_0 y_0 + x_1 y_1 + x_2 y_2 + \cdots + x_N y_N \tag{5.1}$$

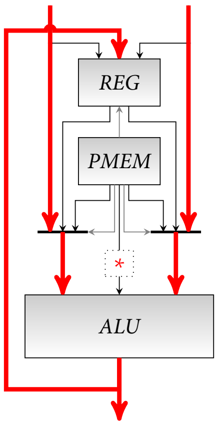
For illustration purposes the *mac* operation is implemented in a pipelined fashion on one core using separate stages for the multiplication and addition. The implementation of the complete *mac* operation requires three stages, of which the first one is an initial stage, i.e. only for the first input sample. In Figure 5.10(a), the configuration is shown, in Figures 5.10(b)-5.10(d), the corresponding execution on the core is shown.

The first stage is labelled S_0, which corresponds to Figure 5.10(b). Here, the two external inputs (x_0 and y_0 from Equation 5.1) are multiplied. The result is stored in the register file at R_0 and sent out of the core. Following, the stage S_1, which corresponds to Figure 5.10(c), is executed, which represents a multiplication of x_1 and y_1. The result of this multiplication is stored in R_1. The final stage S_2, shown in Figure 5.10(d), performs an addition on the results of the states S_0 and S_1 and stores the result in the register file at position R_0 and sends a result token out of the core. From here on, the core alternates between the stages S_1 and S_2.

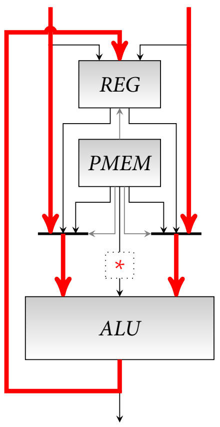
The thick red lines in Figures 5.10(b)-5.10(d) indicate the current configuration. That means, in the first stage, the external inputs are the data inputs to the ALU, and the result is stored in the register file, but also sent out of the core. In stage S_1, the external inputs are again the inputs to the ALU, the result is stored at position R_1 of the register file, but no token is produced at the output. In stage S_2, the data inputs to the ALU are sent from the register file (registers R_0 and R_1), the result is stored back in the register file at position R_0 and a token is produced on the output.



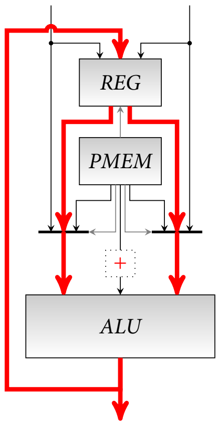
(a) Configuration



(b) S0



(c) S1



(d) S2

FIGURE 5.10 – Implementation of a *mac* operation on one core

The configuration code following our programming paradigm is shown in Table 5.1. The table entries represent the fields of the program memory as presented in Figure 5.5, the fields `Destinations` is omitted since the *mac* algorithm is executed on one core.

opCode	**source1**	**source2**	**store**	**outToken**	**iterations**	**sIndex**
MUL	EX_0	EX_1	*True* 0	*High*	1	1
MUL	EX_0	EX_1	*True* 1	*High*	1	2
ADD	R_0	R_1	*True* 0	*High*	1	1

Table 5.1 – Configuration of the MAC

For further illustration, we included Table 5.2 that shows a concrete execution of the *mac* algorithm on our architecture including concrete values. The two inputs x and y are stream inputs, i.e. they supply a continuous stream of input data. For illustration purposes, we only show the first seven cycles, and demonstrate what output is produced in which cycle, and also which values are stored in the register file. For input and output, the datatype *OpToken* as defined in 5.1 is used to distinguish between valid and invalid data. A – in Table 5.2 represents an invalid value, i.e. **`Nothing`**. A number x represents valid data, i.e. **`Just`** x.

The inputs are as follows:

$x = [1, 3, 5, 7, \cdots]$

$y = [2, 4, 6, 8, \cdots]$

The left part of the table shows the controls of the architecture and corresponds to the configuration as stored in the program memory (refer Figure 5.5), the right part of the table shows the actual values. The table is read as follows:

- » the column **time** indicates the current time step
- » the columns **s1** and **s2** represent the current input sources
- » the column **opC** the current opcode, i.e. the mathematical operation
- » the column **store** indicates where the result should be stored in the register file
- » the column **outT** determines whether an output token is produced
- » the column **currS** denotes the current stage
- » the columns **x** and **y** represent the current input *values*
- » the columns $\mathbf{R_0}$ and $\mathbf{R_1}$ show the current values in the register file
- » the column **res** shows the value of the current result token that is sent out of the core

time	s1	opC	s2	store	outT	currS	x	y	R_0	R_1	res
T_0	*x0*	*	*y0*	R_0	*True*	S_0	*1*	*2*	*2*	*0*	*2*
T_1	*x1*	*	*y1*	R_1	*False*	S_1	*3*	*4*	*2*	*12*	-
T_2	R_0	+	R_1	R_0	*True*	S_2	-	-	*14*	*12*	*14*
T_3	*x2*	*	*y2*	R_1	*False*	S_1	*5*	*6*	*14*	*30*	-
T_4	R_0	+	R_1	R_0	*True*	S_2	-	-	*44*	*30*	*44*
T_5	*x3*	*	*y3*	R_1	*False*	S_1	*7*	*8*	*44*	*56*	-
T_6	R_0	+	R_1	R_0	*True*	S_2	-	-	*100*	*56*	*100*

Table 5.2 – Execution of the *mac* on the architecture

5.6 Design decisions

In this section, we will present a summary on the important design decisions that we made during the implementation of the architecture.

The ALU in the cores has two inputs and one output. In the prototype implementation presented in this thesis, we only support binary operations, i.e. simple mathematical operations with two inputs. This is not a real restriction, since the ALU can be extended to support more inputs, this however would complicate the design at the cost of more silicon area.

The operands are currently 16 bit numbers, since most DSP algorithms operate on 16 bit numbers. However, the bit width of the architecture can easily be changed by modifying a parameter in the configuration of the architecture. Also, floating point numbers are theoretically possible, for that, the ALU has to be extended by a floating point module.

All operations in the ALU currently only take one clock cycle. This is because the operations that the ALU currently supports are simple enough to only take one clock cycle. However, the ALU could be extended to support operations that take multiple clock cycles, if desired.

The cores have a separate configuration input each. This was done to be able to configure all cores in parallel. If desired, the configuration input can be connected to a common configuration bus, and the configuration could be sent via the bus along with the address of the destination core.

The input buffer contains *four* FIFOs. This is a trade-off between flexibility and area. The more FIFOs, the more area is required. For our sample applications, the flexibility of four FIFOs was sufficient.

The cores are interconnected using nearest neighbour connections. This was easy to realise, and routing becomes very simple. If desired, other networks could be used as well, for example a network-on-chip.

5.7 Synthesis results

As a proof of concept, we performed a synthesis of the VHDL code that was automatically generated by CλaSH for an array of a 4x4 cores. We succeeded to synthesise the design for a Xilinx Virtex 7 FPGA [1]. Approximately 40 % of the Slice LUTs were used. The maximum clock frequency was 48 MHz. The detailed synthesis results can be found below.

```
Device utilization summary:
---------------------------


Selected Device : 7vx1140tflg1930-2



Slice Logic Utilization:
  Number of Slice Registers:          75456  out of  1424000     5%
  Number of Slice LUTs:               313681  out of  712000    44%
     Number used as Logic:            313681  out of  712000    44%

Slice Logic Distribution:
  Number of LUT Flip Flop pairs used:  389137
    Number with an unused Flip Flop:   313681  out of  389137    80%
    Number with an unused LUT:         75456  out of  389137    19%
    Number of fully used LUT-FF pairs:     0  out of  389137     0%
    Number of unique control sets:         1
```

Since this is a proof of concept design, the resulting hardware is not very efficient yet. Careful analysis of the synthesis results would be required to improve the generated hardware.

5.8 Conclusions

In this section, we presented our architecture: a reconfigurable array consisting of simple independent computing cores. The cores contain a functional unit, a register file, a program memory and control logic. The cores are interconnected using point-to-point links to the direct neighbours. The cores are designed according to the dataflow principle. The complete array is configured using a combination of dataflow, more specifically the firing rule, and finite state machine principles as presented in Chapter 4. We demonstrated the configuration principles using a multiply-accumulate (*mac*) operation and showed how the cores are configured and how the actual execution on the core is performed.

[1] 7vx1140tflg1930-2

Chapter 6

Programming Language and Compiler

Abstract – *In this chapter, we present the specification of the programming language targeted at the architecture introduced in Chapter 5. Similar to the architecture and the programming principle, we base the programming language on dataflow principles. The two main principles we use are the firing rule and the representation of a program as a dataflow graph. With the presented programming language, algorithms can be implemented as dataflow graphs, i.e. by describing the dependencies between operations. We demonstrate the complete compiler flow and illustrate the different steps during the compilation process.*

6.1 Introduction

Both the programming language and the compiler were designed using Haskell. With Haskell, it is possible for a programmer to describe an algorithm in terms of its dependencies, i.e. structure. That means, data dependencies and regular structures in algorithms can easily be expressed as we will demonstrate later in this chapter. Furthermore, it enables us to stay within *one* design environment for the complete design process. In Chapter 5, we showed that also the proposed architecture was implemented using Haskell, in this chapter we show the implementation of the programming language and the compiler using Haskell. By using one design environment for the complete design process, the same definitions for data types and control structures can be used in both the design of the architecture and the programming language and compiler.

Major parts of this chapter have been published in [AN:2, 3]

The programming language itself is designed as an *embedded domain specific language (EDSL)* inside Haskell. A *domain specific language (DSL)* is a language designed for a specific domain or purpose. An *embedded* DSL is then a DSL embedded in a general purpose host language, with the advantage that (a subset of) the features and syntax of the host language can be used.

The proposed language is designed to be used with the previously presented architecture. Algorithms can be specified using the herein presented language, and then the herein presented compiler generates a configuration for the architecture to execute the algorithm. To generate the correct configuration, a set of transformation rules has been defined. To map the algorithm on the architecture, we use *simulated annealing* [8], which is a commonly used optimisation heuristics for mapping algorithms [15, 49].

As already presented in Chapter 3, we use the Haskell interpreter *GHCi* as design environment.

6.2 The grammar

The proposed programming language was implemented as a recursive datatype in Haskell [36, 67]. Thus, the language is available as an Embedded Domain Specific Language (EDSL) in Haskell. Therefore, algorithms can be implemented directly using Haskell. The grammar for the EDSL is based on the operations which the cores in the architecture can execute, i.e. simple binary operations like multiplication and addition. The grammar can easily be extended to more elaborate operations, based on what the ALU in the architecture supports. Furthermore, a notation for delays and feedback loops to the same node is supported, which are common constructs in DSP algorithms.

We will be using the previously introduced number types to support both fixed point and integer numbers. The definition for a *Number*, as explained in Section 5.4.3 in Chapter 5, was as follows:

```
type Number = (NumType,Word)
```

NumType can have either the value *NUM*, which indicates an integer, or *FP*, which indicates a fixed point number.

Listing 6.1 shows the implementation of the EDSL. The constructors are explained in the remainder of this section in more detail.

```haskell
data Expr = Const Number
          | Input String
          | Op OpCode Expr Expr
          | DELAYED Expr
          | PREV_RES

data OpCode = ADD | MUL | SUB
```

Listing 6.1 – recursive EDSL definition for an expression

In this thesis, we target DSP algorithms that can be represented as a graph. Hence, implementing an algorithm using the proposed EDSL means constructing the corresponding graph using the constructors defined by the EDSL. In the following, we will explain which constructors the EDSL provides and how they can be used to implement graphs. Any graph that has been constructed using the EDSL, has the type *Expr*.

In order to implement a certain algorithm, a designer would have to use the constructors defined by the EDSL. For example, to implement an addition of two constant numbers 1 and 2, the following would be specified:

```haskell
Op ADD (Const (NUM,1)) (Const (NUM,2))
```

Since this is a cumbersome and non-intuitive way of specifying operations, the operations of the EDSL (i.e. the ones specified by the *OpCode*) are being defined for *normal* Haskell operations by making *Expr* an *instance* of the type class **Num**. The implementation is shown in Listing 6.2 in lines 2 to 4 where the definitions for addition, multiplication and subtraction are given. In line 5, the definitions how to convert an integer into the *Expr* is given, in lines 7 and 8, an instance for fractional numbers is created to handle fixed point representations.

```haskell
instance Num Expr where                                  1
  x + y = Op ADD x y                                     2
  x * y = Op MUL x y                                     3
  x - y = Op SUB x y                                     4
  fromInteger x = Const (NUM,fromIntegral x)             5
                                                         6
instance Fractional Expr where                           7
  fromRational x = Const (double2FP $ fromRational x)    8
```

Listing 6.2 – recursive EDSL definition for an expression

Now, a designer can specify the addition of two constant numbers 1 and 2 by simply writing down

```haskell
1 + 2
```

This expression is then *automatically* converted to

`Op ADD (Const (NUM,1)) (Const (NUM,2))`

6.2.1 The constructors of the EDSL

In this section, the constructors for the EDSL are explained in detail, each constructor is illustrated with an example.

In line 1 of Listing 6.1, the definition how to specify a **constant number** is given. First, the constructor `Const` is used, followed by the actual number, which is of type `Number`, i.e. a tag to identify if it is an integer or a fixed point followed by the actual value. In the example shown in Figure 6.1, the constant integer number 5 is defined.

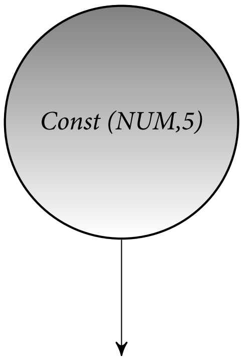


Figure 6.1 – `Const (NUM,5)`

Line 2 represents an input where the string denotes an input stream. In the example shown in Figure 6.2, an input stream with identifier "x" is defined. The identifier of the input stream is used later on by the compiler to map external input signals to input ports in the architecture.

Line 3 of Listing 6.1 defines an operation. *Op* is a data constructor in the type `Expr` and indicates an operation, and *OpCode* defines the opcode, i.e. the operation. The list of supported opcodes is determined by the operations that are supported by the ALU in the architecture (see Section 5.4.4). After the operation, two operands are defined that are themselves of type `Expr`. The example shown in Figure 6.3 shows a node that performs an addition on the element streamed through the `Input ''x''` and a constant value 5, i.e. every element in stream "x" is increased by 5. The implementation of this statement is as follows:

`5 + (Input ''x'')`

This is then automatically converted to a representation using the presented grammar:

`Op ADD (Const (NUM,5)) (Input ''x'')`

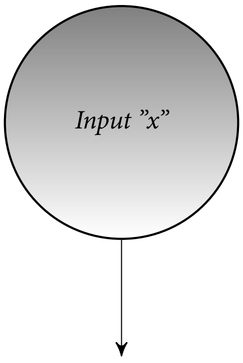


Figure 6.2 – `Input ''x''`

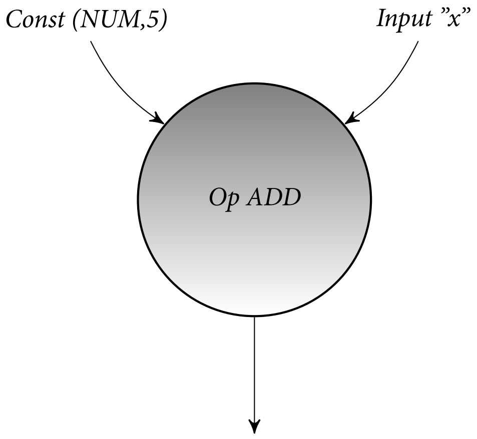


Figure 6.3 – `5 + (Input ''x'')`

Line 4 of Listing 6.1 specifies how a delay of one clock cycle is defined. The constructor *DELAYED* hereby indicates the delay, the following *expression* is then the expression which is delayed by one clock cycle. The example shown in Figure 6.4 shows an expression *x* which is delayed by one clock cycle.

The definition in line 5 of Listing 6.1 shows how the result from the previous clock cycle can be used (i.e. a feedback loop). This constructor cannot be used standalone, but only in combination with an operation node. The example shown in Figure 6.5 shows a node that performs an addition on the value provided through `Input '' x''` and the previous result.

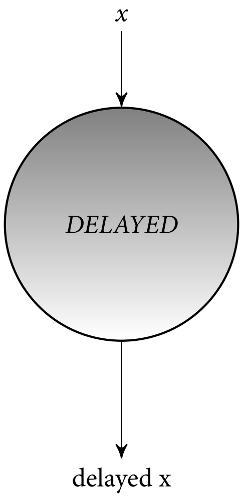


Figure 6.4 – `DELAYED x`

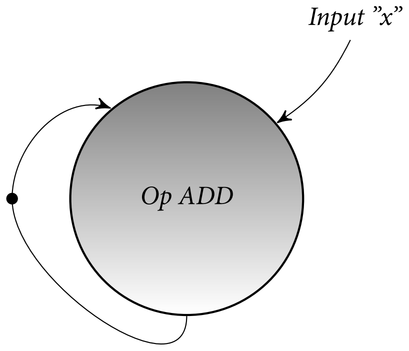


Figure 6.5 – `prev_res + (Input ''x'')`

6.2.2 Examples

In Chapter 4, we introduced the proposed programming scheme which is a combination of finite state machines and dataflow actors. Figure 6.6 is a reprint of the FSM shown in Chapter 4 and is an example of such a combination.

To implement the graph shown in the left state of the state machine in Figure 6.6, a designer would write:

```
(Input ''x'') * (Input ''y'')
```

which is automatically converted to:

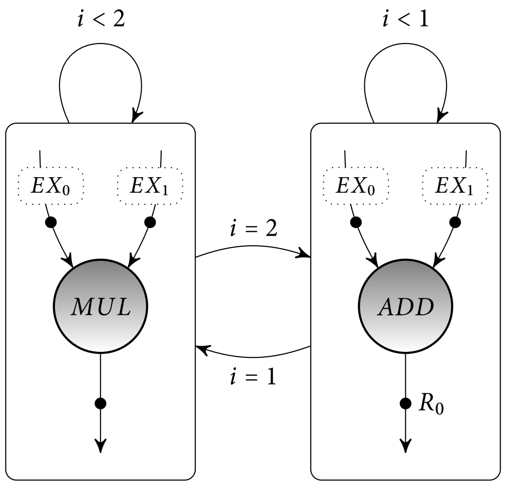


FIGURE 6.6 – Reprint from Chapter 4 - Extended local view

```
Op MUL (Input ''x'') (Input ''y'')
```

The right state is implemented as follows:

```
(Input ''x'') + (Input ''y'')
```

which is automatically converted to:

```
Op ADD (Input ''x'') (Input ''y'')
```

In the following section, we will illustrate in more detail how the proposed language can be used to construct more complex algorithms.

In particular for regular algorithms, *higher order functions* are very useful. To implement a simple sum of all elements in a vector, it can be written as follows:

```
sum_up x = foldl (+) 0 x
```

For the example shown in Figure 6.7, a vector of length three is used. For this, the function `sum_up` has to be applied to an input vector of length three.

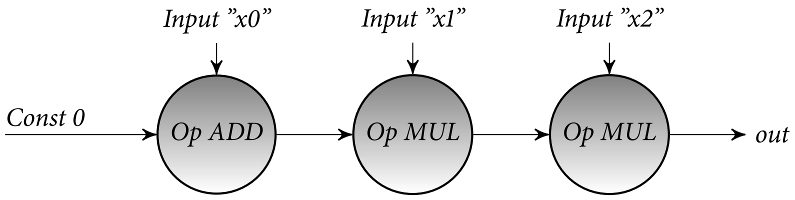


FIGURE 6.7 – `sum_up xs`

Note that the input *x* to `sum_up` is of type `[Expr]`, i.e. a list of expressions. The type of `sum_up x` itself is also `Expr`, i.e. it is an expression in our EDSL, as already explained in the beginning of this section. The call of **`foldl`** assigns an `Op ADD` to each element of the list of inputs *x*, the initial value is given by `Const 0`. This is the same principle that was used to describe the summation in Section 3.1.2.

6.3 Streaming notation

To implement DSP algorithms, a notation for specifying a chain of operations is convenient. This corresponds to a streaming pipeline. Consider the case shown in Figure 6.8. To the left, a stream *x* is streamed into the system. In the first stage, `kernel1` performs its computation on *x*, then `kernel2` executes on the output of `kernel1` and finally `kernel3` is applied to the output of `kernel2`. In our compiler, we implemented a function that supports this streaming notation:

```
a ▸ f = f a
```

An argument `a` is streamed to the function `f` by using the notation ▸. Then, the function `f` is applied to the argument `a`. A usecase example is shown in Listing 6.3 where a *digital down converter (DCC)* is implemented. In lines 1 to 3, the kernels are defined. `fir4` represents a 4 tap FIR filter, `dc` is a down converter and `fir16` is 16 tap FIR filter. In line 5, an implementation of the streaming pipeline shown in Figure 6.9 using the streaming notation is presented.

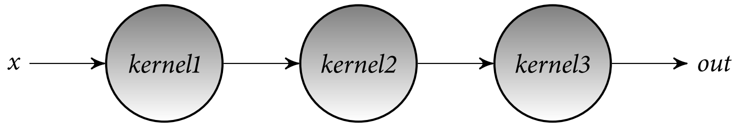


Figure 6.8 – Implementation of a streaming pipeline

```
kernel1 x = fir4 x                                 1
kernel2 x = dc x                                   2
kernel3 x = fir16 x                                3
                                                   4
stream1 x = x ▸ kernel1 ▸ kernel2 ▸ kernel3        5
```

Listing 6.3 – Implementation of a streaming pipeline

6.4 The abstract syntax tree

When we call the previously defined function `sum_up` with a concrete input vector in *GHCi*, the expression tree is then automatically displayed:

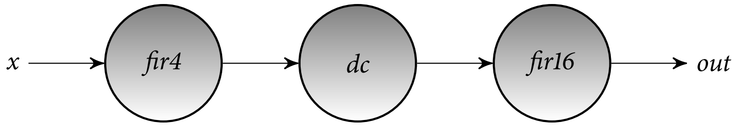


Figure 6.9 – Graphical representation of the implemented streaming pipeline

```
ghci> sum_up [Input "x0",Input "x1",Input "x2",Input "x3"]
ghci> Op ADD
          (Op ADD
                (Op ADD
                      (Op ADD (Const 0) (Input "x0"))
                      (Input "x1"))
                (Input "x2"))
          (Input "x3")
```

This demonstrates that when an EDSL is implemented in Haskell, the parser is "for free", meaning that a value of type *Expr* is already the abstract syntax tree (AST) of the expression that was specified.

However, one issue remains with this approach: when the algorithm contains feedback loops, an automatic extraction of the AST is not possible. Consider the expression in Listing 6.4, where a graphical representation is given in Figure 6.10.

```
floop x = add0
    where
      add0 = x + (DELAYED add1)
      add1 = add0 + 1
```

Listing 6.4 – Implementation of a simple feedback loop

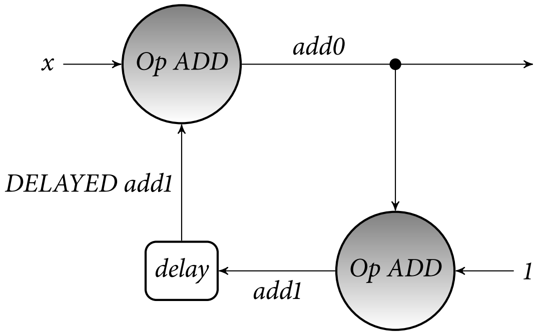


Figure 6.10 – Structure of *floop*

When the expression *floop* is called in the Haskell interpreter to display the AST, the following is observed:

```
ghci> floop (Input "x")
ghci> Op ADD (Input "x") (DELAYED (Op ADD (Op ADD (Input "x")
(DELAYED (Op ADD (Op ADD (Input "x") (DELAYED (Op ADD (Op ADD
(Input "x") (DELAYED (Op ADD (Op ADD (Input "x") ....
```

The Haskell interpreter tries to unroll the expression and thus goes into an infinite loop. That makes sense as `floop` contains a recursive feedback loop.

The infinite loop can be avoided by using the library *Reify* [7]. *Reify* provides methods to automatically convert any recursive data structure, in our case the expression, into a unique graph. The outcome of the conversion is a list of nodes, each containing a unique identifier, a constructor and pointer to the inputs.

In order to use Reify for our desired purpose, a number of definitions have to be provided, i.e. the transformation from the EDSL datatypes to unique Reify-datatypes. For each constructor in the EDSL, a corresponding constructor for Reify is defined. Furthermore, a transformation rule is defined. The definitions for the datatypes are given in Listing D.1, the definitions for the transformations in Listing D.2, both in Appendix D.

For the presented feedback loop in Listing 6.4, Reify gives us the following result, which is also displayed in Figure 6.11:

```
ghci> reifyGraph $ floop (Input "x")
ghci> [ (1,ExprOp ADD 2 3)
      , (3,ExprDelayed 4)
      , (4,ExprOp ADD 1 5)
      , (5,ExprConst 1)
      , (2,ExprInput "x")
      ]
```

The result is read as follows: The first entry is the unique identifier of the node, then the operation is specified. Next, the inputs to the node are specified using their respective identifier. The first entry of the reified graph has the identifier 1 and is an addition (*ExprOp ADD*). Its inputs are defined by the entries 2 and 3 which are the input (*ExprInput*) and the output of the delay element (*ExprDelayed*).

At this point, the expression has been converted to a unique graph, which is used in the following steps by the compiler.

6.5 Mapping to the architecture

To execute the expression on the architecture, each operation node in the AST is mapped to one core. This is performed using simulated annealing [8], which is a commonly used algorithm for mapping tasks to processor cores. It is an optimisation heuristic that finds a solution that in general is close to the global optimum of a given problem.

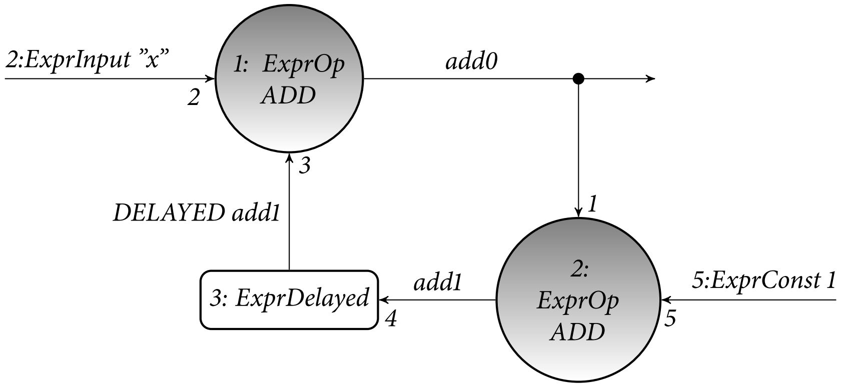


Figure 6.11 – Structure of `floop` after Reify

The idea behind simulated annealing is based on the physical process of annealing in metallurgy, which gave the algorithm its name. A material is first heated and then slowly cooled down, which leads to a structure of the material close to its thermodynamic optimum. The algorithm behind simulated annealing resembles that process by introducing a temperature factor, which is slowly decreased during the run-time of the algorithm.

The result of the simulated annealing algorithm is a mapping of each node in the algorithm graph to a core in the hardware architecture. The mapping information is used by the compiler in the next step, the generation of the configuration for each node.

The current implementation of the mapping maps *one* node to *one* core in the architecture. In case there are more nodes than cores, the simulated annealing algorithm does not find a solution. In that case, the algorithm can be split up manually in suitable chunks and each chunk can then be mapped to the architecture.

6.5.1 Simulated annealing

A schematic view of the simulated annealing algorithm is shown in Figure 6.12. The first step in the algorithm is the generation of an initial solution of the given problem, for example a complete random solution. For the mapping problem this corresponds to a random mapping. Then, the cost of this solution is computed. For the presented mapping, we only consider communication costs, i.e. the distance between communicating cores. The cost is calculated as follows:

$\mathbf{N}$ is the list of nodes in the algorithm graph, $\mathbf{C}$ is the list of cores in the architecture and $\mathbf{L}$ is the list of communicating node pairs. $\mathbf{M}$ is the current mapping of the nodes to the cores, i.e. $\mathbf{M} : \mathbf{N} \rightarrow \mathbf{C}$. The assigned mapping of a node s to a core is $\mathbf{M}(s)$.

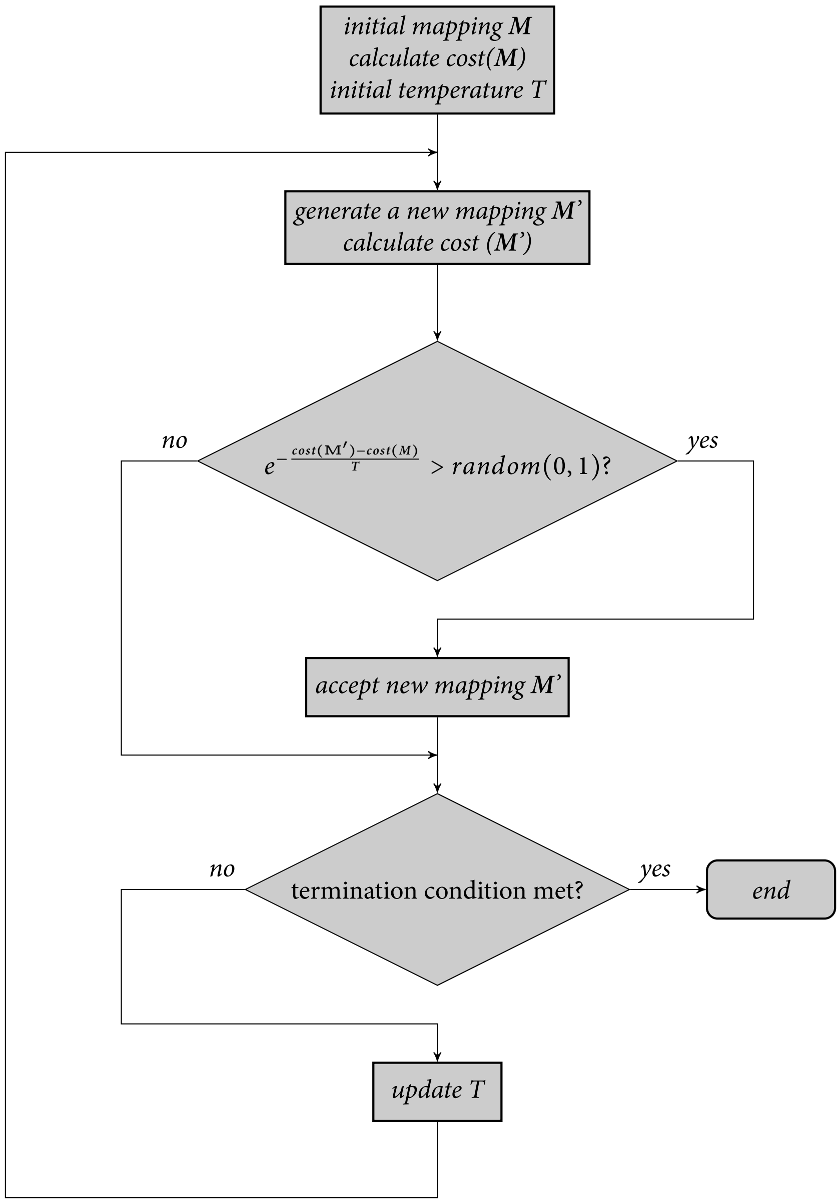


Figure 6.12 – Simulated annealing

The distance between two communicating nodes (s, d) is defined as follows:

$$dist_{\mathbf{M}(s),\mathbf{M}(d)} = \begin{cases} 1 & \text{if } \mathbf{M}\text{(s) and } \mathbf{M}\text{(d) are neighbours} \\ \text{manhattan}(\mathbf{M}(s), \mathbf{M}(d)) & \text{otherwise} \end{cases} \tag{6.1}$$

The cost of a mapping **M** is determined as follows:

$$cost(\mathbf{M}) = \sum_{(s,d)\in\mathbf{L}} dist^2_{\mathbf{M}(s),\mathbf{M}(d)} \tag{6.2}$$

Based on the initial solution, a neighbouring solution $\mathbf{M}'$ (i.e. a new, slightly adapted solution) is generated. In our implementation, we switched *assignments* of two adjacent cores. That means, if two adjacent cores each were assigned a node, these nodes were switched. If only one core of two adjacent cores is assigned a node, this node is moved to the other core. Then, the cost of this new solution is computed and compared to the previous cost. If the cost is less, i.e. the solution is better, the new solution is accepted. If the cost is higher, i.e. the solution is worse, the new solution is accepted with a certain probability which is dependent on the temperature T. By decreasing the temperature over time, the probability of accepting a worse solution is getting lower, but is never decreased to zero. This ensures that the algorithm can climb out of local minima. In the algorithm, this is implemented by using the formula $e^{-\frac{cost(\mathbf{M}')-cost(\mathbf{M})}{T}} > random(0,1)$.

6.6 Code generation

In order to execute an algorithm implemented using the proposed EDSL on the architecture, the AST of the expression has to be converted into a format that is understood by the architecture. That means generating a configuration for each core in the architecture following the programming paradigm presented in Chapter 4 and the assembly format introduced in Section 5.4.4, which is repeated in Listing 6.5.

```
type PMemEntry =
  ( OpCode          -- defines opcode
  , Source          -- source of left input
  , Source          -- source of right input
  , Store           -- store result in regfile
  , OutToken        -- produce output token
  , Destinations    -- destinations of result token
  , Iterations      -- number of iterations in current state
  , SIndex          -- next state
  )
```

Listing 6.5 – Definition of the configuration format

The separate elements of the program memory have been explained in detail in Section 5.4.4, a brief summary is given below as reminder:

`OpCode` defines the opcode of the node, `Source` defines the input source, `Store` defines whether the result should be stored in the register file, `OutToken` defines whether a token should be produced at the output, `Destinations` defines the destination addresses, `Iterations` defines the number of iterations in the current state and `SIndex` defines the next state. All information except the destination addresses is determined in the code generation step, the destination addresses are determined during the mapping of the expression to the architecture.

As mentioned before, each node in the AST is one of the five different possible cases given in Listing 6.1:

1. a *constant*,
2. a *delayed expression*,
3. an *operation*,
4. a pointer to the *previous result*, or
5. an *input*.

In order to generate code for the hardware architecture, the compiler converts the AST into a list of configurations that are mapped onto the architecture. Hereby, the compiler traverses through all nodes in the AST and generates the corresponding configuration code. Code is only directly generated for nodes that define an operation. All the other cases are used as inputs by the operation nodes and are handled there.

For each node in the AST, it is first determined whether the current node is an *operation* node, i.e. a node that defines an operation. If that is not the case, the compiler skips to the next node. If however the current node is an operation node, a configuration is generated by the compiler.

To generate a configuration for an operation node, all the entries specified in Listing 6.5 have to be determined. In Figure 6.13, an illustration is shown how the compiler determines each entry.

The code generation for an operation node can be split into two cases: 1. The simple case, where the expression is an operation on two incoming, non-delayed and non-feedback signals (the right branch in Figure 6.13), and 2. the complex case, where one or more of the inputs comprise a delay or a feedback loop (the left branch in Figure 6.13). For the simple case, an FSM with one state is sufficient, for the complex case, two stages are required.

In the following section, we will illustrate the code generation step using a number of examples, two of them will be also explained using Figure 6.13.

6.6.1 Code examples

In this section, we will demonstrate the code generation for a number of small examples. For each example, we will show the input to the compiler, and both a graphical representation of the configuration and the actual code generated by the

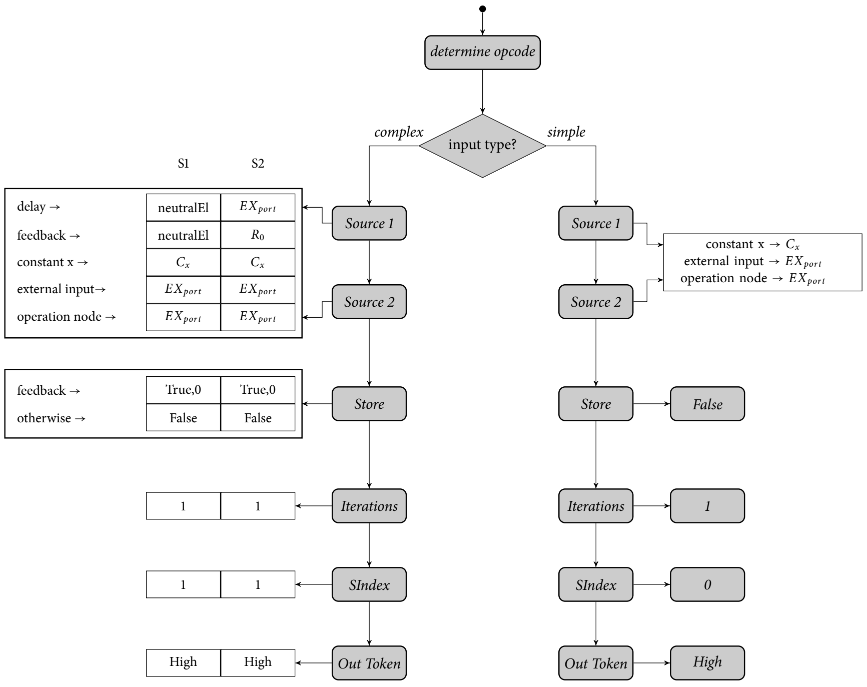


FIGURE 6.13 – Generate the configuration for an operation node

compiler for the architecture. For a better overview, all the generated code examples are shown in Table 6.1.

First, two examples for the simple case are shown. For the simple case, code generation is straight forward, since only one state is required and the dataflow actor can be obtained by defining the operator and the source of the inputs. In Figure 6.14, three examples are shown.

In Figure 6.14(a), the resulting graph is shown for a multiplication of an external input with a constant number 2. In Figure 6.14(b), a multiplication of two external inputs is shown. Finally, Figure 6.14(c) shows the addition of two external inputs. The generated code is shown in the corresponding rows in Table 6.1.

In Figure 6.15, the generation of the configuration according to the flow chart presented in the previous section is shown.

Next, we demonstrate the code generation for two complex examples, where the inputs are either delayed or form a feedback loop. For the complex case, the delay or the feedback has to be taken into account by providing an initial token for the first iteration and providing information where the data should be stored in the register file. Figure 6.16 shows the two complex examples.

Figure 6.16(a) is a graphical representation of the expression

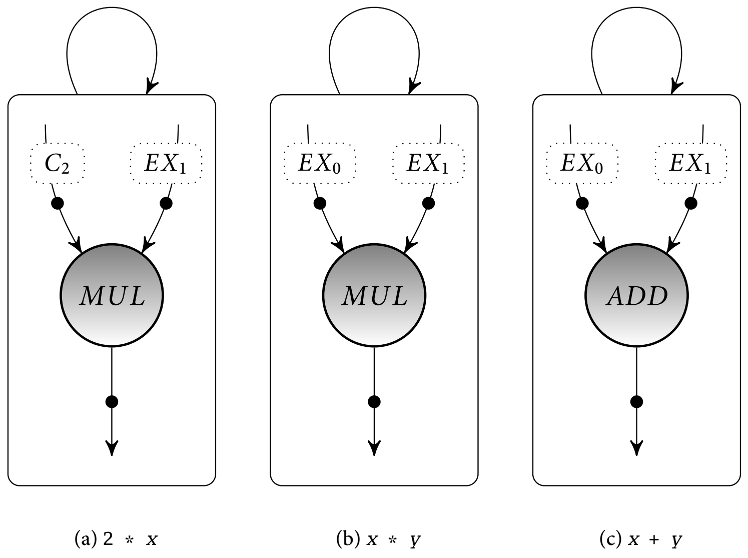


Figure 6.14 – Conversion of simple case nodes

Op ADD (DELAYED x) y

i.e. an addition with delayed input. In Figure 6.17, the corresponding dataflow actor following the scheme presented in Section 6.2 is shown. The generated code is shown in the corresponding rows in Table 6.1.

In Figure 6.18, the generation of the configuration according to the flow chart presented in the previous section is shown.

In Figure 6.16(b), the graphical representation of the expression

Op ADD PREV_RES x

is shown. Here, one of the operands is the previous result, thus forming a feedback loop. Figure 6.19 shows the corresponding dataflow actor. The generated code is shown in the corresponding rows in Table 6.1.

After the compiler has traversed through the complete AST, the configuration, i.e. the *extended local view*, of each core has been generated. The global view, i.e. the mapping, will be added in the next step and thus complete the configuration.

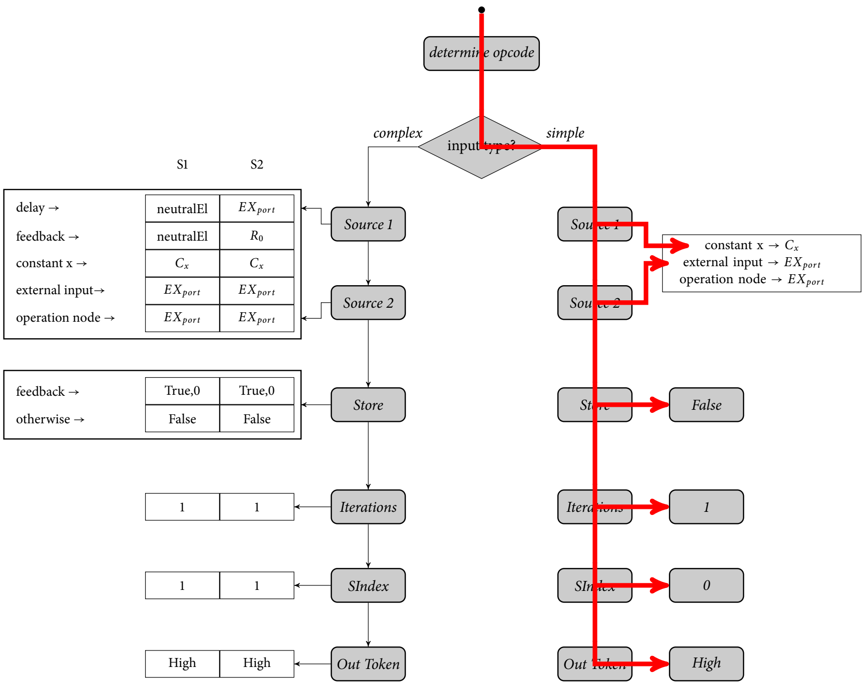


Figure 6.15 – Generate the configuration for Figure 6.14(a)

Table 6.1 – Generated Configurations

Graph	opCode	source1	source2	store	iter.	sInd.	outT.
6.14(a)	*MUL*	`C (NUM,2)`	`EX 1`	`False`	1	0	*High*
6.14(b)	*MUL*	`EX 0`	`EX 1`	`False`	1	0	*High*
6.14(c)	*ADD*	`EX 0`	`EX 1`	`False`	1	0	*High*
6.16(a)	*ADD*	`C (NUM,0)`	`EX 1`	`False`	1	1	*High*
	ADD	`EX 0`	`EX 1`	`False`	1	1	*High*
6.16(b)	*ADD*	`C (NUM,0)`	`EX 1`	`True 0`	1	1	*High*
	ADD	`R 0`	`EX 1`	`True 0`	1	1	*High*

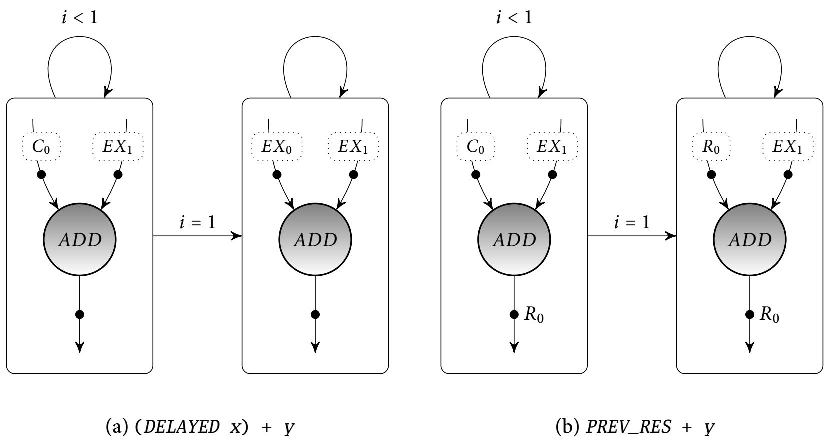


Figure 6.16 – Conversion of complex nodes

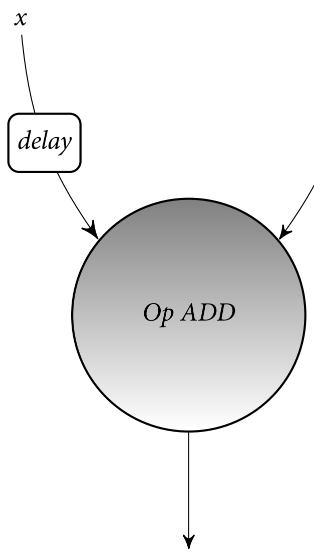


Figure 6.17 – Dataflow actor of (*DELAYED* *x*) + *y*

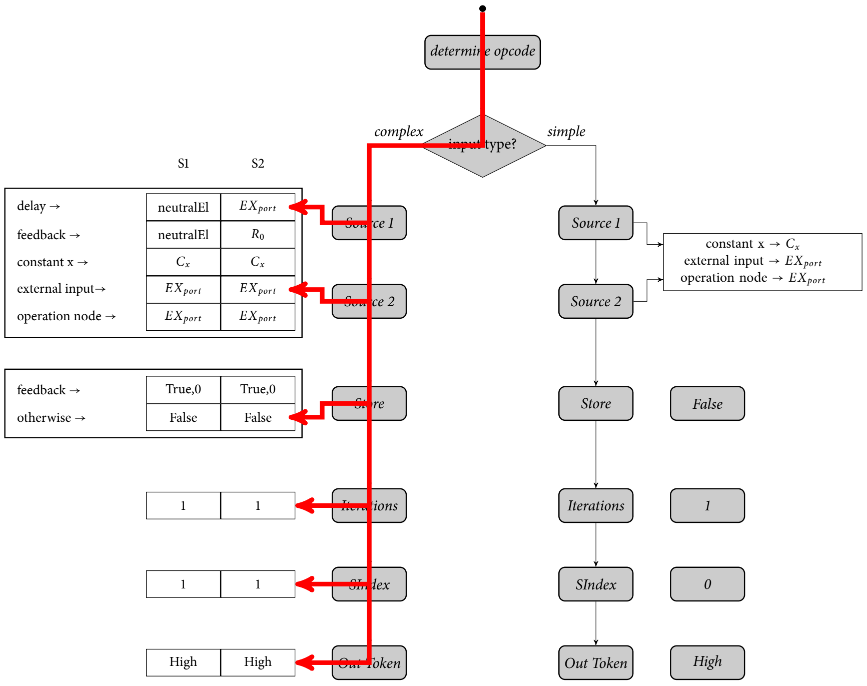


FIGURE 6.18 – Generate the configuration for Figure 6.16(a)

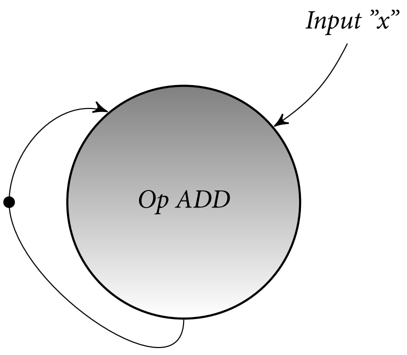


FIGURE 6.19 – Dataflow actor of *PREV_RES* + *x*

6.6.2 Adding the routing information to the configuration

As a final step, the mapping information, that has been generated in the previous step, is added to each core's configuration. This completes the configuration of the array.

In Figure 6.20, a core with its connections to neighbouring cores is shown. Each output port is labelled according to its *direction* on a virtual compass, as explained in Section 4.2.3 and Section 5.4.2. The port at the top is labelled *N* for *north*, the port at the lower left side is labelled *SW* for *south west* and so on.

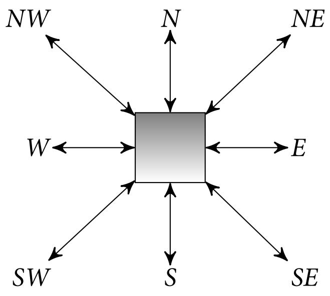


Figure 6.20 – Directions of one core

According to the determined mapping, the compiler can determine the direction of each outgoing packet from each core. Consider the example shown in Figure 6.21, which we already used in Chapter 4.

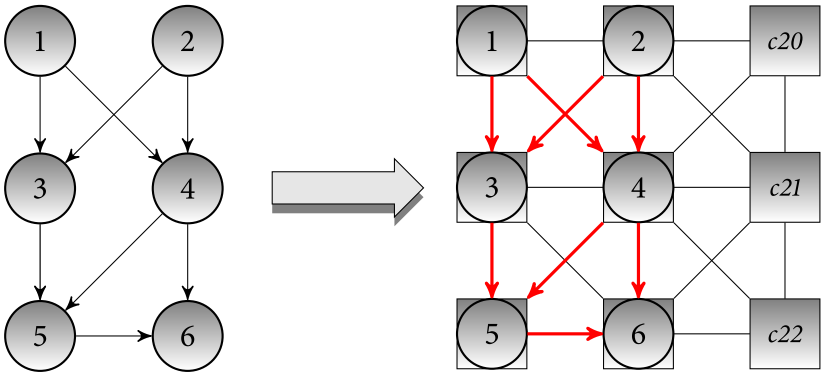


Figure 6.21 – Example mapping with respective directions

The example graph, consisting of six nodes labelled 1 to 6 is mapped to an array of three by three cores. The communication between the nodes and thus of the cores is indicated by the arrows. Node 1 sends out data to nodes 3 and 4, node 3 sends data to node 5 and so on. The resulting directions that are added to the configuration of the respective cores are shown in Table 6.2. In the following chapter, the integration of the directions into the final configuration will be shown using a more elaborate example.

node	direction(s)
1	S, SE
2	SW, S
3	S
4	SW, S
5	E

Table 6.2 – Directions for the example in Figure 6.21

6.7 The complete compilation flow

The main steps of the compiler are shown in Figure 6.22 :

In the following, we will briefly explain the separate steps. In the following Chapter, we will go through the steps in more details following an example.

Step 1: Preparation

The preparation contains two steps.

First, the expression has to be *reified* as explained in Section 6.4.

Then, the nodes in the AST are sorted by their indices, i.e. their unique identifiers as introduced in Section 6.4.

Graphical output

Next, a graphical output is generated to give the user feedback on the structure of the implemented expression. We chose to use the graph description language *dot* [4] which is a common format understood by many software tools. For our purposes, we used the Linux tool *dot* to generate a PDF file with the expression tree.

Step 2: Mapping

The next step in the compilation process is the actual mapping of the nodes onto the architecture.

First, the information required by the simulation annealing algorithm is generated and written to a csv file. In the csv file, the dependencies, i.e. the communication, between all the nodes in the expression are defined.

The mapping itself is performed using *simulated annealing*, as already explained. For performance reasons, the *simulated annealing* algorithm is implemented in C.

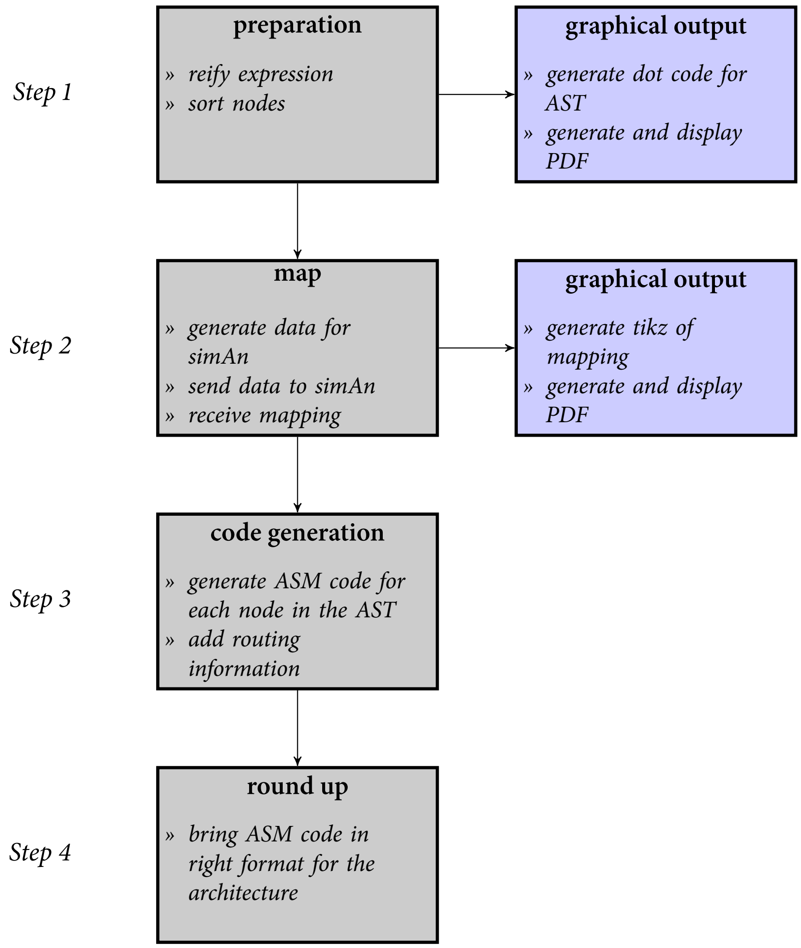

Step 1
preparation
» reify expression
» sort nodes
graphical output
» generate dot code for AST
» generate and display PDF
Step 2
map
» generate data for simAn
» send data to simAn
» receive mapping
graphical output
» generate tikz of mapping
» generate and display PDF
Step 3
code generation
» generate ASM code for each node in the AST
» add routing information
Step 4
round up
» bring ASM code in right format for the architecture

Figure 6.22 – Compiler flow

Graphical output

Next, again a graphical output is produced to give the user feedback about the mapping result. For this, the mapping information is combined with an abstract view of the architecture, and by using TikZ [84] a PDF file is generated and displayed.

Step 3: Code generation

The following step in the compilation process is the actual code generation. This means, for each node in the expression tree, the proper configuration is generated. Also, the routing information is included in the configuration of the nodes which was derived in the previous step.

Step 4: Round up

The final step is to round up the compilation process. At this point, all the code for the architecture is generated, it only has to be converted into the correct format to be sent as new configuration to the architecture. Also, the identifier of the output core(s) is provided which is helpful for the simulation of the expression on the architecture.

6.8 Design decisions

In this section, we will briefly motivate our design decision that were taken during the implementation of the herein presented programming language and the compiler.

Each operation has two inputs and one output. We chose for that since we only consider simple arithmetic operations like multiplication, addition and the like. However, this is not a principle restriction to our language and compiler. If desired, more complex operations with a different number of operands could be added. Also, different operations like for example *logic operations* could be added without any problem.

The opcode *PREV_RES* was added to support feedback loops to the same node, which is a quite common operation in DSP algorithms.

Each node in the application graph is mapped to one core in the architecture. In principle, this could be extended to a more complex mapping where multiple nodes are mapped to one core, but in the scope of this thesis, we restricted the mapping to a simple, straightforward implementation.

We chose *simulated annealing* as a heuristic during the mapping step since it is a commonly used algorithm for mapping algorithms to CGRAs. As the mapping step is not the main focus in this research, we chose simulated annealing since it served our purpose well.

6.9 Conclusions

In this chapter, we introduced our programming language and the corresponding compiler. The programming language follows the programming paradigm presented in Chapter 4. Both the programming language and the compiler were implemented using the functional programming language Haskell. The language was implemented as a recursive datatype, which enables a user to use the Haskell syntax. Especially the use of *higher order functions* is advantageous since it enables a user to describe regular structures in an intuitive way.

We showed the transformation rules that are the underlying principle for the compiler to generate configuration code out of an implemented algorithm using the proposed programming language. Furthermore, the steps of the compiler were described and illustrated in detail.

Chapter 7

Design Flow and Case Studies

ABSTRACT – *In the previous chapter, we presented the programming language and the design framework for the proposed architecture. In this chapter, we will present how the design framework is used to implement and simulate a concrete algorithm. The algorithm we will be using to illustrate our approach is the dot product, i.e. the multiplication of two vectors, a commonly used algorithm in the domain of digital signal processing. We will demonstrate all the required steps to implement the dot product using the presented design framework, and finally execute the algorithm with a set of stimuli on the architecture. Finally, we will present the results of a number of case studies.*

7.1 Introduction

In this section, we will describe the workflow of our system, i.e. all the required steps to implement an algorithm on our architecture using the framework presented in Chapter 6.

We see the workflow as three different parts: The first part is the input required from the user. The second part represents the automated steps performed by the compiler, as presented in the previous chapter in Section 6.7. The third part is the surrounding framework that integrates the inputs of the user and the results of the compiler to perform the final simulation and verification.

7.2 Showcase algorithm

As an illustrating example, we will use the *dot product*, i.e. the multiplication of two vectors, throughout this section. The dot product of two vectors **xs** and **ys** of

Major parts of this chapter have been published in [AN:3, 4]

length N is defined as

$$\mathbf{xs} \cdot \mathbf{ys} = \sum_{i=0}^{N-1} xs_i ys_i \tag{7.1}$$

A graphical representation of the dot product of two vectors **xs** and **ys** of length four is shown in Figure 7.1.

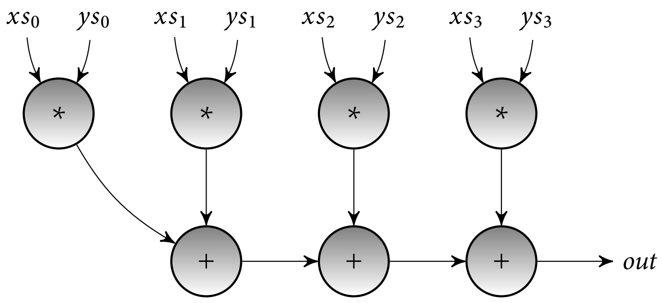


Figure 7.1 – Structure of the dot product

7.3 Implementation of the algorithm by the user

The first step in the design process is the actual implementation of the desired algorithm, here the dot product, by the user. Since the normal mathematical operations like addition and multiplication have been defined for our programming language, see Section 6.2 in Chapter 6, normal Haskell syntax is used for the implementation.

7.3.1 Implementing the algorithm in Haskell

The computation of the dot product can be formulated as two steps:

1. The two vectors have to be multiplied pair-wise, and
2. the results have to be accumulated.

A straightforward implementation of these two steps can be achieved by using the two higher order functions **`zipWith`** and **`foldl1`** as shown in Listing 7.1. The corresponding structure for two vectors **xs** and **ys** of length four is shown in Figure 7.1.

```
vxv xs ys = out                       1
    where                             2
      ms  = zipWith (*) xs ys         3
      out = foldl1  (+) ms            4
```

Listing 7.1 – Implementation of the dot product in Haskell

In line 1 of the code, the function name *vxv* and its arguments *xs* and *ys*, which are the two vectors to be multiplied, are defined, `out` is the resulting output. In line 3, the vectors are pair-wise multiplied which leads to the row of multiplications in Figure 7.1. Finally, in line 4, the results of the multiplications are accumulated, which leads to the row of additions in Figure 7.1.

An alternative method of implementing the *dot product* in Haskell is using *recursion*. In Listing 7.2, the implementation is shown.

```
vxv_recursion (x:[]) (y:[]) = x*y
vxv_recursion (x:xs) (y:ys) = x*y + vxv_recursion xs ys
```

Listing 7.2 – Implementation of the dot product in Haskell using recursion

In the remainder of this chapter, we will use the implementation using higher order functions (hence *vxv*). However, all presented steps are also valid for the implementation using recursion, since the resulting structure is the same.

7.4 Start the compilation process

Next, the user can start the automatic compilation process. For that, a concrete instance of *vxv* has to be defined that determines the length of the input vectors. To do so, first a helper function to generate the correct format for the input vectors (the constructor `Input` followed by a string to identify the input) is defined:

```
makeVxVIn prefix n = map (λn → Input (prefix++show n)) [1..n]
```

The argument `prefix` is a string which is attached in front of each number from 1 to *n* using the ++ function.

The command `makeVxVIn "x" 8` yields:

```
[Input "x1",Input "x2",Input "x3",Input "x4", ... ,Input "x8"]
```

which resembles the correct format for an input defined in the EDSL (refer Section 6.2).

Then, a concrete instance *vxv*8 of the dot product with input vectors of length eight is defined using the previously defined function `makeVxVIn`:

```
vxv8 = vxv (makeVxVIn "x" 8) (makeVxVIn "y" 8)
```

Now, the compilation process can be started. Therefore, we define a main function which, after the complete compilation is finished, contains the configuration for the dot product on the architecture. The main function is defined as follows:

```
compile_vxv8 = compile vxv8
```

By executing the command `compile_vxv8` in *GHCi*, the compilation is started.

7.4.1 Graphical output of the expression

As explained in Section 6.7, a graphical representation of the implemented expression is provided to the user in form of a PDF file. This is to give the user feedback to verify that the structure of the expression is as the user intended. For the dot product, the displayed expression is shown in Figure 7.2. It can be seen that the structure directly resembles the code shown in Listing 7.1 and the intended structure of Figure 7.1. The only difference is, that each operator node now is assigned a unique identifier, which is due to the *reify* step (as introduced in Section 6.4.

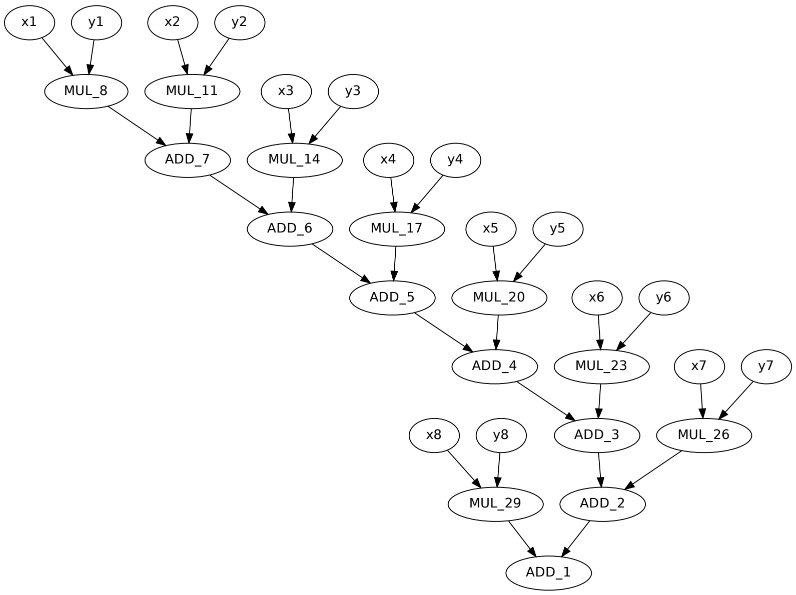


Figure 7.2 – Graphical representation of the compiled dot product

7.4.2 Mapping

The next step in the compilation process is the mapping of the nodes to the architecture. As explained in Section 6.7, *simulated annealing* is used. The resulting mapping for the *dot product* maps one operator node on one core each.

In the scope of this thesis, the mapping step always produces a one-to-one mapping, i.e. *one* node of the AST is mapped to *one* core in the architecture. In theory, multiple nodes could be mapped to one core, this is however not supported by the compiler yet. When mapping multiple nodes to one core, throughput would be lowered, but less cores would be used.

7.4.3 Graphical output of the mapping

After the mapping is completed, a graphical output is generated to give the user visual feedback, which is shown in Figure 7.3. The figure shows the resulting mapping for the dot product on the architecture.

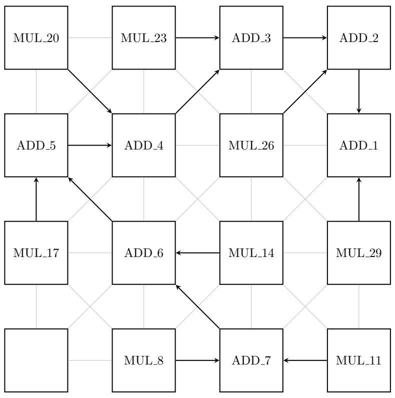


Figure 7.3 – Autogenerated graphical representation of the mapping of the dot product

7.4.4 Code generation

Next, the code is generated for all the operator nodes in *vxv*. In the AST of the *dot product* (refer Figure 7.2), the compiler encounters two different operator nodes: a multiplication of two external inputs, for example node *MUL_11*, and an addition of two external inputs, for example *ADD_7*. Note that by "external" we mean external to the core. This can mean either inputs to the expression tree (for the multiplication nodes) or inputs from another core (for the addition nodes).

The configuration generated for the multiplication nodes is shown in Figure 7.4.

The actual code generated for the architecture to execute the multiplication nodes is as follows:

```
[(MUL,EX 0,EX 1,False,1,0,High)]
```

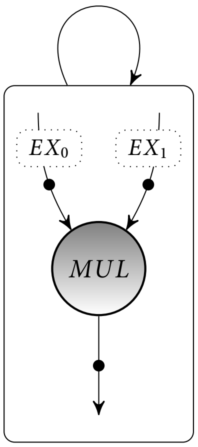


Figure 7.4 – Configuration for the multiplication nodes

The configuration for the addition nodes is shown in Figure 7.5.

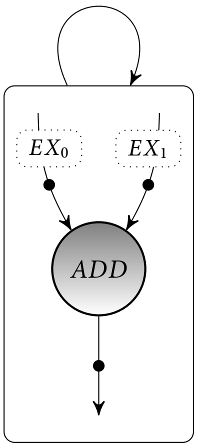


Figure 7.5 – Configuration for the addition nodes

The code generated for the architecture to execute the addition nodes is:

```
[(ADD,EX 0,EX 1,False,1,0,High)]
```

Since the mapping has already been performed in the previous step, also the destination addresses for the resulting tokens can be included in the code. Referring to the mapping shown in Figure 7.3, the node *MUL_11* sends its data to the west and to input port 1 (since it is the right input of the destination node *ADD_7* as seen in

Figure 7.2), the node *ADD_7* sends it to the north west and to input port 0 (since it is the left input of the destination node *ADD_6* as seen in Figure 7.2).

The final code for *MUL_11* is:

`[(MUL,EX 0,EX 1,False,1,0,High,<True,W,1>)]`

The final code for *ADD_7* is:

`[(ADD,EX 0,EX 1,False,1,0,High,<True,NW,0>)]`

For the other nodes, the configuration is done in a similar way, depending on where the data is sent to. The generated code for all nodes is shown in Table 7.1.

node	core	code
MUL_20	C00	`[(MUL,EX 0,EX 1,False,1,0,High,<True,SE,1>)]`
MUL_23	C10	`[(MUL,EX 0,EX 1,False,1,0,High,<True,E,1>)]`
ADD_3	C20	`[(ADD,EX 0,EX 1,False,1,0,High,<True,E,0>)]`
ADD_2	C30	`[(ADD,EX 0,EX 1,False,1,0,High,<True,S,0>)]`
ADD_5	C01	`[(ADD,EX 0,EX 1,False,1,0,High,<True,E,0>)]`
ADD_4	C11	`[(ADD,EX 0,EX 1,False,1,0,High,<True,NE,0>)]`
MUL_26	C21	`[(MUL,EX 0,EX 1,False,1,0,High,<True,NE,1>)]`
ADD_1	C31	`[(ADD,EX 0,EX 1,False,1,0,High,<False,N,3>)]`
MUL_17	C02	`[(MUL,EX 0,EX 1,False,1,0,High,<True,N,1>)]`
ADD_6	C12	`[(ADD,EX 0,EX 1,False,1,0,High,<True,NW,0>)]`
MUL_14	C22	`[(MUL,EX 0,EX 1,False,1,0,High,<True,W,1>)]`
MUL_29	C32	`[(MUL,EX 0,EX 1,False,1,0,High,<True,N,1>)]`
MUL_8	C13	`[(MUL,EX 0,EX 1,False,1,0,High,<True,E,1>)]`
ADD_7	C23	`[(ADD,EX 0,EX 1,False,1,0,High,<True,NW,0>)]`
MUL_11	C33	`[(MUL,EX 0,EX 1,False,1,0,High,<True,W,1>)]`

Table 7.1 – Generated code for all nodes in the dot product

7.5 Verification

The final step in the design process is the testing and verification of the implemented algorithm. For that, the following steps are automatically performed:

1. The configuration is converted to a format which can be sent to the architecture
2. Stimuli are generated based on a test set specified by the user
3. The algorithm is executed on the architecture using the specified stimuli

For the simulation of the *dot product*, we send a number of test vectors to the architecture. The input vectors **xs** and **ys** and the result of the dot product $\mathbf{x} \cdot \mathbf{y}$ are shown in Table 7.2.

$\mathbf{x}$	$\mathbf{y}$	$\mathbf{x} \cdot \mathbf{y}$
$[1, 1, \ldots, 1]$	$[1, 1, \ldots, 1]$	8
$[2, 2, \ldots, 2]$	$[2, 2, \ldots, 2]$	32
$[3, 3, \ldots, 3]$	$[3, 3, \ldots, 3]$	72
$[4, 4, \ldots, 4]$	$[4, 4, \ldots, 4]$	128
$[5, 5, \ldots, 5]$	$[5, 5, \ldots, 5]$	200
$[6, 6, \ldots, 6]$	$[6, 6, \ldots, 6]$	288
$[7, 7, \ldots, 7]$	$[7, 7, \ldots, 7]$	392
$[8, 8, \ldots, 8]$	$[8, 8, \ldots, 8]$	512
$[9, 9, \ldots, 9]$	$[9, 9, \ldots, 9]$	648
$[10, 10, \ldots, 10]$	$[10, 10, \ldots, 10]$	800

Table 7.2 – Stimuli for the dot product

These vectors were sent to the architecture with a delay of 10 clock cycles between the samples to have enough time between the samples to calculate the result. The output of the simulation is shown in Figure 7.6. The output of the simulation shows that the computed numbers are correct, a dot in the simulation means that no output is produced in the respective clock cycle.

7.6 Case studies

The previously presented usecase of the *dot product* was targeted towards a CGRA with 4x4 cores. Since our array is scalable, we implemented a number of test cases on a 4x4 array and on an 8x8 array to evaluate the usability of our programming language and the compiler.

On the 4x4 array, we implemented an 8-tap FIR filter, the 8x8 dot product which we used in this chapter as main example, and a 4 point FFT kernel.

On the 8x8 array, we implemented a 32-tap FIR filter, a 32x32 dot product, an 8 point FFT kernel, an 8 point DCT kernel, and 8 point autoregression filter kernel,

```
.....................8
.....................32
.....................72
.....................128
.....................200
.....................288
.....................392
.....................512
.....................648
.....................800
```

Figure 7.6 – Output of the simulation results

and an 8 point elliptic wave filter kernel. The latter two algorithms were obtained from the ExpressDFG benchmark set [2] from the ExPRESS research group of UC Santa Barbara.

Information on the mapping results are shown in Table 7.3. For each of the implemented algorithms, the number of used nodes (and hence required cores in the architecture) and connections are shown. All presented algorithms were successfully mapped in such a way that communicating nodes were allocated to neighbouring cores, i.e. communication via point-to-point links is sufficient for every test case. Simulating the algorithms with test stimuli showed correct behaviour.

Based on the presented results we can conclude that our presented system is usable to implement the class of algorithms the system was targeted at: small DSP kernels with a large degree of fine-grained parallelism and simple operations. The implemented algorithms were all implemented using the presented programming language without problem and could be mapped and executed on the architecture without manual input from the designer.

In Appendix E, the concrete implementations for the case studies can be found.

Table 7.3 – Test cases for the different array sizes

array size	Algorithm	nodes	connections
4x4	*FIR8*	15	14
	8x8 Dot Product	15	14
	FFT4	16	16
8x8	*FIR32*	63	62
	32x32 Dot Product	63	62
	FFT8	60	80
	DCT8	40	50
	ARF8	28	30
	EWF	34	47

7.7 Discussion

In this section, we will provide a brief discussion of our approach. We will mention the strong points, but also the weak points.

In our opinion, the choice of Haskell as a base language of our embedded programming language is very beneficial. Since Haskell by itself can express *structure*, i.e. data dependencies, DSP algorithms can be implemented with their structure in mind. Since many DSP algorithms are available as a graph, implementation is a straightforward task. Another advantage of using Haskell is that we could implement our programming language as an *recursive datatype*, hence each implemented algorithm by itself is already the *abstract syntax tree* of that algorithm. Therefore, no extra dependency analysis by the compiler is required.

In the current implementation of the compiler, one node in the algorithm graph is mapped to one core in the architecture. This leads to a high throughput, but in the case that there are more nodes in the graph than cores in the array, the algorithm cannot be mapped automatically. In that case, manual input of the user is required. However, since our architecture in principle supports multiple nodes per core, the compiler could in principle be extended to map more multiple nodes per core.

7.8 Conclusions

In this chapter, we gave a demonstration how the proposed design framework is used. The use case algorithm was the *dot product*, which is common in digital signal processing. The complete design framework can be seen in three parts: The first is the input of the user, i.e. the implementation of the actual algorithm. The second part contains the automated steps performed by the compiler, i.e. the code generation and mapping of the algorithm to the architecture. Finally, the third step is the simulation of the algorithm on the architecture using a set of stimuli, provided by the user. We explained the three steps in detail and also showed the result of a successful simulation of the *dot product* on the architecture. Apart from the detailed use case we also presented the results of a number of standard DSP kernels that were successfully implemented and mapped to our CGRA using the presented programming language and compiler.

Chapter 8

Conclusions

Dataflow is a powerful paradigm for expressing data-driven streaming algorithms. In this thesis, we showed how the principles of dataflow can be used as a base for a programming paradigm, but also as an execution mechanism for hardware.

We designed a complete system containing a hardware architecture, a programming language and a compiler that is targeted at data-driven streaming DSP algorithms that contain a large degree of fine-grained parallelism.

In Chapter 1, four key requirements for the complete system were presented:

1. *Highly programmable*
2. *Support for data-driven streaming applications*
3. *Efficient multicore architecture*
4. *Realised using one design environment*

In the following, we will present the key contributions of this thesis. Afterwards, we will relate them to the presented key requirements.

8.1 Key contributions

The three key contributions of this work are:

1. The design and development of a CGRA
2. The use of dataflow principles as conceptual basis for the complete system, i.e. for the software as well as for the hardware
3. A completely integrated framework, consisting of an architecture, a programming language and a compiler designed in a single functional programming environment

8.1.1 The design and development of a CGRA

The presented CGRA is targeted at data-driven streaming algorithms that have a regular, fine-grained structure, which can be found in filtering, matrix manipulations algorithms, and the like. The architecture consists of an array of reconfigurable cores. Each core adheres to the dataflow principles. It fires when sufficient input tokens are available, executes the required operation and produces the output token(s). The architecture was implemented using CλaSH, a hardware design language and compiler based on Haskell. CλaSH is a research tool developed in the Computer Architecture for Embedded Systems (CAES) chair at the University of Twente. The work performed in the course of this thesis is the first big hardware design project using CλaSH as a main design language.

8.1.2 The use of dataflow principles as conceptual basis

The execution mechanism of the cores in the architecture is data-driven, i.e. the cores adopt the concept of the *firing rule* known from dataflow. The programming principle which is the underlying basis for the programming language for the cores is a combination of finite state machines (FSM) and dataflow for enhanced flexibility. By using a programming paradigm based on dataflow, we show that fine-grained parallelism can be expressed in a straightforward and intuitive way. This is usually not the case for the programming environment for existing CGRAs.

8.1.3 A complete integrated framework in a single environment

The architecture was designed using CλaSH, a hardware description language based on Haskell. The programming language for the architecture was implemented as an embedded language in Haskell. The compiler also was implemented in Haskell. At no point the Haskell environment is left and the same datatype definitions are used for all the different parts in the framework. The same tooling (in our case the Haskell interpreter *GHCi*) is used to not only simulate the architecture and the programming language, but also for the implemented algorithms on the architecture. The real hardware can be generated automatically from the CλaSH specification by the CλaSH compiler. Everything is expressed in Haskell, hereby avoiding the burden of combining various different environments. We consider this an important achievement of our work.

8.2 Relation to key requirements

Key requirement 1: Programmability

Programming parallel architectures is a challenging task and much research is being conducted to find an efficient, yet easy to use programming paradigm. Related work on CGRAs suggests that the main focus on programming CGRAs is on the automatic parallelisation of C. Automatic parallelisation of sequential languages is known to be a serious challenge.

We chose a different approach. Instead of starting from C (or in fact any other imperative programming paradigm), we chose to use a functional programming approach to face the challenge of finding a good programming paradigm to program the herein presented CGRA.

We implemented our programming language as an Embedded Domain Specific Language (EDSL) in Haskell using a recursive datatype. This language is used to implement algorithms by constructing a graph representing the structure (i.e. dependencies between operations) of the algorithm. The grammar of the language is kept simple and straightforward. Implementing the grammar as a recursive datatype within Haskell has two major advantages: Firstly, Haskell's own syntax including higher order functions and recursion can be used to implement algorithms. Especially higher order functions enable a user to implement algorithms in a structural, straightforward way. Secondly, each algorithm that is implemented using the recursive datatype, *is* already the *abstract syntax tree* (AST) of the respective algorithm. Hence, no additional analysis by the compiler is required.

Key requirement 2: Support for streaming applications

In streaming applications, data arrives as a stream of tokens at the input of the system. As a result, the system has to cope with a continuous stream of data. Furthermore, it can occur that a token in the stream might be delayed, in that case, the system should wait until it arrives and not simply execute its operation without the actual input data being present.

In our architecture as well as in the programming paradigm, we used the firing rule concept of dataflow to support streaming applications. In dataflow, an operation is triggered by the availability of its required input tokens. The cores in our CGRA are data-driven, i.e. their execution is triggered as soon as the required input tokens have arrived. The programming language resembles a dataflow structure, i.e. a user specifies a certain algorithm as a dataflow graph using the dataflow constructors available in the programming language.

Key requirement 3: Efficient multicore architecture

In order to efficiently execute streaming applications that contain a large degree of instruction-level parallelism, a suitable hardware architecture is required. We developed a coarse-grained reconfigurable array (CGRA), since this is a promising class of architectures for that application domain.

The CGRA consists of an array of interconnected, small, configurable cores. Each core contains an ALU for binary mathematical operations, a local storage for intermediate results, a programming memory containing the configuration of the respective core and a control unit. The array can achieve a high throughput, since each core is based on dataflow principles and takes only one clock cycle to perform an operation. Moreover, it is energy efficient because the cores only process local data; there is no global memory in our approach.

Key requirement 4: Realised using one design environment

The design of a complex system consisting of several components, e.g. a hardware architecture, a programming language and compiler and a simulation framework, usually requires the use of several design languages and environments. In this thesis, we used *one* design environment for all parts of the system.

We succeeded in designing the complete system using Haskell. The architecture was implemented using CλaSH, the programming language was implemented as Embedded Domain Specific Language in Haskell, and the resulting system can be simulated using standard Haskell tooling. The advantage of using one language for the complete system is that a sound, complete system is generated.

8.3 Recommendations for future work

In this thesis we started to combine the world of computer architecture and functional programming. The initial results are very encouraging, but still a lot of open questions remain.

Possible improvements to key requirement 1: Highly programmable

The compiler currently maps one node to one core in the array. This ensures maximum throughput, but becomes a problem when the application graph contains more nodes than are available in the array. Hence, the compiler needs to be extended to map and schedule multiple nodes per core, which is already supported by our architecture.

Possible improvements to key requirement 3: Efficient multicore architecture

The current hardware implementation, i.e. the synthesised VHDL code, is a direct compilation of the CλaSH generated VHDL netlist to an FPGA. The synthesis results can be improved significantly when the synthesis results are carefully analysed and specific parts of the design are optimised.

The ALU in the cores currently only support operations with two inputs that take one clock cycle. In order to execute complex DSP applications more efficient, the ALU could be extended to more complex operations, e.g. multiply-add or even complete DSP kernels like FIR filters.

The cores in the CGRA are currently interconnected using point to point links to the direct neighbours. While it was sufficient for the presented case studies, it might be required to have a full Network-On-Chip (NoC) available.

Possible improvements to key requirement 4: A single design environment

By using Haskell as design environment for the complete system, the final system can easily be simulated using the interactive Haskell compiler. However, it would be desirable to have graphical feedback for the user to visualise the behaviour of applications, compiler and architecture.

8.4 Main conclusions

Based on the findings presented in this thesis, we conclude that:

1. The combination of a functional language and dataflow principles makes a powerful programming paradigm
2. The principles of dataflow, in particular the firing rule concept, are a powerful basis when designing an architecture and programming language targeted at data-driven streaming applications
3. A CGRA based on dataflow principles is well-suited for the efficient execution of data-driven streaming applications
4. Haskell is a convenient design environment for a complete system

Appendix A

VHDL for the Adder

```
  -- Automatically generated VHDL
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;

package types is

     subtype tfvec_index is integer range -1 to integer'high;

     subtype signed_16 is signed (15 downto 0);

     type Tuple2_0 is
          record AA : signed_16;
                 AB : signed_16;
          end record;

     function show (s : std_logic;
                     paren : boolean)
                    return string;

     function show (b : boolean;
                     paren : boolean)
                    return string;

     function show (sint : signed;
                     paren : boolean)
                    return string;

     function show (uint : unsigned;
                   paren : boolean)
                 return string;

    function show (tup : Tuple2_0;
                   paren : boolean)
                 return string;

end package types;

package body types is

    function show (s : std_logic;
                   paren : boolean)
                 return string is
    begin
        if s = '1' then
            return "High";
        else
            return "Low";
        end if;
    end;

    function show (b : boolean;
                   paren : boolean)
                 return string is
    begin
        if b then
            return "True";
        else
            return "False";
        end if;
    end;

    function show (sint : signed;
                   paren : boolean)
                 return string is
    begin
        return integer'image(to_integer(sint));
    end;

    function show (uint : unsigned;
                   paren : boolean)
                 return string is
    begin
        return integer'image(to_integer(uint));
    end;

    function show (tup : Tuple2_0;
                   paren : boolean)
                  return string is
    begin
        return '(' & (show(tup.AA, false) & ',' & show(tup.AB, false
    )) & ')';
    end;

end package body types;
```

Listing A.1 – Generated VHDL code for the adder, type definitions

```
-- Automatically generated VHDL
use work.types.all;
use work.all;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;

entity fixedMuxAComponent_0 is
    port (param2046912649 : in MuxInput_1;
          res2046912655 : out Tuple2_2;
          clock1 : in std_logic;
          resetn : in std_logic);
end entity fixedMuxAComponent_0;

architecture structural of fixedMuxAComponent_0 is
begin
    comp_ins_res2046912655 : entity fixedMuxComponent_1
                                        port map (i2046912723 => param20469
    12649,
                                                  res2046912774 => res20469
    12655,
                                                  clock1 => clock1,
                                                  resetn => resetn);
end architecture structural;
```

Listing A.2 – Generated VHDL code for the adder, top level entity

Appendix B

Fixed Point Adder and Multiplier

```
add_fp  ( opta , a )  ( optb , b )  = ( t , res )
    where
      res = op1 + op2
      (t,op1,op2)
        | opta == NUM  &&  optb == NUM  =  ( NUM  , a   , b   )
        | opta == FP   &&  optb == FP   =  ( FP   , a   , b   )
        | opta == FP   &&  optb == NUM  =  ( FP   , a   , b'  )
        | otherwise                     =  ( FP   , a'  , b   )
      a' = (a<<<<dotPos)
      b' = (b<<<<dotPos)
```

Listing B.1 – Implementation of the fixed point adder

```
mul_fp  ( opta , a )  ( optb , b )  = ( t , res )
    where
      res
        | t == NUM  = mul_res
        | otherwise = mul_res >>>> dotPos
      mul_res = op1 * op2
      (t,op1,op2)
        | opta == NUM  && optb == NUM  = ( NUM , a   , b   )
        | opta == FP   && optb == FP   = ( FP  , a   , b   )
        | opta == FP   && optb == NUM  = ( FP  , a   , b'  )
        | otherwise                    = ( FP  , a'  , b   )
      a' = (a<<<<dotPos)
      b' = (b<<<<dotPos)
```

Listing B.2 – Implementation of the fixed point multiplier

Appendix C

Datatypes

name	purpose
OpCode	opcode for the current operation
Source	source of the input
IBWindex	Width of the input buffer
RIndex	Number of outputs of the register file
Store	defines whether the result, the left input or the right input should be stored in the register file
RInL	Number of inputs to the register file
Iterations	Number of iterations in the current state
SNext	Next state in the configuration FSM
PMemL	Number of entries in the PMem
OutToken	Defines whether an output token is produced
Destination	Defines the destination of a result
Destinations	A vector of destinations
DestinationsW	Maximum number of destinations
Direction	Direction towards the destination core

Table C.1 – Definition of the PMem Datatypes

Appendix D

Reify Definitions

```
data ExprGraph = ExprGraph [(Unique,ExprNode Unique)] Unique
data ExprNode s = ExprConst Number
                | ExprInput String
                | ExprOp OpCode s s
                | ExprDelayed s
                | ExprPREV_RES
```

Listing D.1 – Type definitions for Reify

```
instance MuRef Expr where
  type DeRef Expr = ExprNode
  mapDeRef f (Const x)       = pure $ ExprConst x
  mapDeRef f (Input x)       = pure $ ExprInput x
  mapDeRef f (Op opc a b)    = ExprOp opc <$> f a <*> f b
  mapDeRef f (DELAYED a)     = ExprDelayed <$> f a
  mapDeRef f PREV_RES        = pure ExprPREV_RES
```

Listing D.2 – Transformation rules for Reify

Appendix E

Implementations of the Case Studies

```
fir x c = out
  where
    ms  = map (*x) c
    out = foldl (λa b → delay (a+b)) (delay $ head ms) (tail ms)

fir8  = fir (Input "x") (firConcreteConstants 8)
fir32 = fir (Input "x") (firConcreteConstants 32)
```

Listing E.1 – FIR filter

```
fft4 x = y
  where

    x1r = x!!0
    x1i = x!!1
    x2r = x!!2
    x2i = x!!3
    x3r = x!!4
    x3i = x!!5
    x4r = x!!6
    x4i = x!!7

    a1r = x1r + x3r
    a1i = x1i + x3i
    a2r = x1r - x3r
    a2i = x1i - x3i
    a3r = x4r + x2r
    a3i = x4i + x2i
    a4r = x4i - x2i
    a4i = x2r - x4r

    y1r = a1r + a3r
    y1i = a1i + a3i
    y2r = a2r - a4r
    y2i = a2i - a4i
    y3r = a1r - a3r
    y3i = a1i - a3i
    y4r = a2r + a4r
    y4i = a2i + a4i

    y = [y1r,y1i,y2r,y2i,y3r,y3i,y4r,y4i]

fft4Concrete = ListOutput $ fft4 $ fftInputs 8
```

Listing E.2 – FIR filter

```
fft8 x_in = y
    where

      x = listToTuples x_in

      w = [(0.707,-0.707),(-0.707,-0.707)]

      t1 = concat [ radix2_fft8_w0 (x!!0) (x!!4)
                  , radix2_fft8_w0 (x!!2) (x!!6)
                  , radix2_fft8_w0 (x!!1) (x!!5)
                  , radix2_fft8_w0 (x!!3) (x!!7)
                  ]

      t2 = concat [ radix2_fft8_w0 (t1!!0) (t1!!2)
                  , radix2_fft8_w2 (t1!!1) (t1!!3)
                  , radix2_fft8_w0 (t1!!4) (t1!!6)
                  , radix2_fft8_w2 (t1!!5) (t1!!7)
                  ]

      t3 = concat [ radix2_fft8_w0 (t2!!0) (t2!!4)
                  , radix2         (t2!!2) (t2!!6) (w!!0)
                  , radix2_fft8_w2 (t2!!1) (t2!!5)
                  , radix2         (t2!!3) (t2!!7) (w!!1)
                  ]

      y = [(t3!!0),(t3!!2),(t3!!4),(t3!!6),(t3!!1),(t3!!3),(t3!!5),(t3
     !!7)]

```

```
radix2_fft8_w0 a b = [cadd a b,csub a b]

radix2_fft8_w2 a b = [a',b']
    where
      (ar,ai) = a
      (br,bi) = b
      a'      = (ar+bi,ai-br)
      b'      = (ar-bi,ai+br)

radix2 a b w = [a',b']
    where
      bw = cmul b w
      a' = cadd a bw
      b' = csub a bw

cadd (ar,ai) (br,bi) = (ar+br,ai+bi)
csub (ar,ai) (br,bi) = (ar-br,ai-bi)
cmul (ar,ai) (br,bi) = (ar*br-ai*bi,ai*br+ar*bi)

listToTuples (x0:x1:xs) = ((x0,x1):listToTuples xs)
listToTuples []         = []

fft8Concrete = ListOutput $ fft8 [ (Const (NUM,x)) | x ← [1..16] ]
```

Listing E.3 – FFT kernel, 8 point

```
dct_golden_fixed x = y
    where
      s1 = concat [ dct_radix (x!!0) (x!!7)
                  , dct_radix (x!!1) (x!!6)
                  , dct_radix (x!!2) (x!!5)
                  , dct_radix (x!!3) (x!!4)
                  ]

      s1' = [(s1!!0),(s1!!2),(s1!!4),(s1!!6),(s1!!7),(s1!!5),(s1!!3),(
    s1!!1)]

      s2 = concat [ dct_radix (s1'!!0) (s1'!!3)
                  , dct_radix (s1'!!1) (s1'!!2)
                  , dct_radix_c3 (s1'!!4) (s1'!!7)
                  , dct_radix_c1 (s1'!!5) (s1'!!6)
                  ]
    s2' = [(s2!!0),(s2!!2),(s2!!3),(s2!!1),(s2!!4),(s2!!6),(s2!!7),(
  s2!!5)]

    s3 = concat [ dct_radix (s2'!!0) (s2'!!1)
                , dct_radix_sqrt2c1  (s2'!!2) (s2'!!3)
                , dct_radix (s2'!!4) (s2'!!6)
                , dct_radix (s2'!!7) (s2'!!5)
                ]

    s3' = [(s3!!0),(s3!!1),(s3!!2),(s3!!3),(s3!!4),(s3!!7),(s3!!5),(
  s3!!6)]

    s4  = concat [ dct_radix (s3'!!7) (s3'!!4)
                 , dct_sqrt_line (s3'!!5)
                 , dct_sqrt_line (s3'!!6)
                 ]

    s4' = take 4 s3' ++ [(s4!!1),(s4!!2),(s4!!3),(s4!!0)]

    y = [(s4'!!0),(s4'!!7),(s4'!!2),(s4'!!5),(s4'!!1),(s4'!!6),(s4
  '!!3),(s4'!!4)]

dct_radix i0 i1 = [o0,o1]
  where
    o0 = i0+i1
    o1 = i0-i1

dct_radix_c3 i0 i1 = [o0,o1]
  where
    a = 0.8315
    b = 0.5556

    o0 =  i0*a + i1*b
    o1 =  i1*a - i0*b

dct_radix_c1 i0 i1 = [o0,o1]
  where
    a = 0.9808
    b = 0.1951

    o0 =  i0*a + i1*b
    o1 = -i0*b + i1*a

dct_radix_sqrt2c1 i0 i1 = [o0,o1]
  where
```

```
        a = 1.387
        b = 0.2759

        o0 =  i0*a + i1*b
        o1 = -i0*b + i1*a

dct_sqrt_line i = [1.4142*i]


dctConcrete = ListOutput $ dct [ (Input ("x"++ show x)) | x ← [1..8
     ]]
```

Listing E.4 – FIR filter

```
arf i = out
    where
        -- actual graph
        f3_1 = k3 [i!!0,i!!1]
        f1   = k1 [i!!2,i!!3]
        f2   = k1 [i!!4,i!!5]
        f3_2 = k3 [i!!6,i!!7]
        f4   = k4 [f2,f1,f2,f1]
        f5   = k5 [f3_1,f4!!1,f4!!0]
        f6   = k5 [f3_2,f4!!0,f4!!1]
        out  = ListOutput [f5,f6]
        -- out = f5

        -- kernels
        k1 k1_in = cadd (add (cmul (k1_in!!0)) (cmul (k1_in!!1) ))
        k3 k3_in = add (cmul (k3_in!!0)) (cmul (k3_in!!1))
        k4 k4_in = [k3 [(k4_in!!0),(k4_in!!1)],k3 [(k4_in!!2),(k4_in!!3)
     ]]
        k5 k5_in = add (k5_in!!0) ( add (cmul (k5_in!!1)) (cmul (k5_in!!
     2)) )

        -- helper
        cmul a = a * 2
        cadd a = a + 1

arfConcrete = arf (makeARFInputs "x" 8)
makeARFInputs prefix n = map (λn → Input (prefix++show n)) [0..(n-1)]
```

Listing E.5 – 8 point Autoregression filter kernel

```
ewf i = out
    where
      out = ListOutput [a13,a30,a33,a34,a29]
      a1  = cadd (i!!0)
      a2  = cadd a1
      a3  = cadd (i!!1)
      a4  = cadd a2
      a5  = add a4 a3
      m6  = cmul a5
      m7  = cmul a5
      a8  = add a2 m6
      a9  = add m7 a3
      a10 = add a8 a5
      a11 = add a2 a8
      a12 = add a9 a3
      a13 = add a10 a9
      m14 = cmul a11
      m15 = cmul a12
      a16 = add a1 m14
      a17 = cadd m15
      a18 = add a1 a16
      a19 = add a16 a8
      a20 = add a9 a17
      a21 = cadd a17
      m22 = cmul a18
      a23 = cadd a19
      a24 = cadd a20
      m25 = cmul a21
      a26 = cadd m22
      m27 = cmul a23
      m28 = cmul a24
      a29 = add m25 a17
      a30 = add a26 a16
      a31 = cadd m27
      a32 = cadd m28
      a33 = add a23 a31
      a34 = add a32 a24
      cmul a = a * 2
      cadd a = a + 1
ewfConcrete = ewf (makeARFInputs "x" 2)
```

Listing E.6 – Elliptic wave filter kernel

Bibliography

[1] http://clash.ewi.utwente.nl/, . (Cited on pages 32 and 34).

[2] http://express.ece.ucsb.edu/benchmark/, . (Cited on page 109).

[3] http://www.haskell.org/ghc/, . (Cited on page 23).

[4] http://www.graphviz.org/, . (Cited on page 97).

[5] http://www.ni.com/labview/, . (Cited on page 15).

[6] http://www.pactxpp.com/, . (Cited on page 20).

[7] http://hackage.haskell.org/package/data-reify/, . (Cited on page 86).

[8] E. Aarts and J. Korst. *Simulated annealing and Boltzmann machines*. New York, NY; John Wiley and Sons Inc., 1988. ISBN 978-0-471-92146-2. (Cited on pages 78 and 86).

[9] W. Ackerman. Data flow languages. *Computer*, 15(2):15–25, 1982. ISSN 0018-9162. doi: 10.1109/MC.1982.1653938. 00501. (Cited on page 12).

[10] A. Alsolaim, J. Becker, M. Glesner, and J. Starzyk. Architecture and application of a dynamically reconfigurable hardware array for future mobile communication systems. In *2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 205–214, 2000. doi: 10.1109/FPGA.2000.903407. (Cited on page 19).

[11] I. Amer, C. Lucarz, G. Roquier, M. Mattavelli, M. Raulet, J. F. Nezan, and O. Deforges. Reconfigurable video coding on multicore. *IEEE Signal Processing Magazine*, 26(6): 113–123, 2009. ISSN 1053-5888. doi: 10.1109/MSP.2009.934107. (Cited on page 15).

[12] Arvind and D. E. Culler. Dataflow architectures. *Annual Review of Computer Science*, 1(1):225–253, 1986. doi: 10.1146/annurev.cs.01.060186.001301. URL http://www.annualreviews.org/doi/abs/10.1146/annurev.cs.01.060186.001301. 00003. (Cited on pages 9, 10, and 18).

[13] Arvind and R. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, 1990. ISSN 0018-9340. doi: 10.1109/12.48862. (Cited on page 17).

[14] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, 1977. URL http://dl.acm.org/citation.cfm?id=359715. (Cited on page 14).

[15] O. Atak and A. Atalar. An efficient computation model for coarse grained reconfigurable architectures and its applications to a reconfigurable computer. In *2010 21st IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP)*, pages 289–292, 2010. doi: 10.1109/ASAP.2010.5541009. (Cited on pages 19 and 78).

[16] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards. CλaSH: Structural descriptions of synchronous hardware using haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD)*, pages 714–721, Sept. 2010. doi: 10.1109/DSD.2010.21. 00027. (Cited on page 32).

[17] J. Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978. URL `http://dl.acm.org/citation.cfm?id=359579`. (Cited on pages 14 and 16).

[18] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. Pact xpp - a self-reconfigurable data processing architecture. *The Journal of Supercomputing*, 26 (2):167–184, Sept. 2003. ISSN 0920-8542, 1573-0484. doi: 10.1023/A:1024499601571. (Cited on page 20).

[19] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet. OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems. *SIGARCH Comput. Archit. News*, 36(5):29–35, June 2009. ISSN 0163-5964. doi: 10.1145/1556444.1556449. URL `http://doi.acm.org/10.1145/1556444.1556449`. (Cited on page 15).

[20] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *Signal Processing, IEEE Transactions on*, 44(2):397–408, 1996. (Cited on page 11).

[21] B. Bougard, B. De Sutter, D. Verkest, L. Van der Perre, and R. Lauwereins. A coarse-grained array accelerator for software-defined radio baseband processing. *IEEE Micro*, 28(4):41–50, 2008. ISSN 0272-1732. doi: 10.1109/MM.2008.49. (Cited on page 20).

[22] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald, and W. Yoder. Scaling to the end of silicon with EDGE architectures. *Computer*, 37(7):44–55, 2004. ISSN 0018-9162. doi: 10.1109/MC.2004.65. (Cited on page 20).

[23] D. Chen and J. Rabaey. A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high-speed DSP data paths. *IEEE Journal of Solid-State Circuits*, 27(12):1895–1904, 1992. ISSN 0018-9200. doi: 10.1109/4.173120. (Cited on page 21).

[24] K. Choi. Coarse-grained reconfigurable array: Architecture and application mapping. *IPSJ Transactions on System LSI Design Methodology*, 4:31–46, 2011. 00008. (Cited on page 18).

[25] A. Davis and R. Keller. Data flow program graphs. *Computer*, 15(2):26–41, 1982. ISSN 0018-9162. doi: 10.1109/MC.1982.1653939. (Cited on pages 9, 10, 11, 12, and 13).

[26] A. L. Davis. The architecture and system method of DDM1: a recursively structured data driven machine. In *Proceedings of the 5th annual symposium on Computer architecture*, page 210–215, 1978. URL http://dl.acm.org/citation.cfm?id=803050. (Cited on page 17).

[27] R. de Groote, P. K. F. Hölzenspies, J. Kuper, and G. J. M. Smit. Multirate Equivalents of Cyclo-Static Synchronous Dataflow Graphs. In *Proceedings of the 14th International Conference on Application of Concurrency to System Design (ACSD'14)*, June 2014. (Cited on page 11).

[28] R. de Groote, P. K. F. Hölzenspies, J. Kuper, and G. J. M. Smit. Single-Rate Approximations of Cyclo-Static Synchronous Dataflow Graphs. In *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2014)*, June 2014. (Cited on page 11).

[29] B. H. J. Dekens, P. Wilmanns, M. J. G. Bekooij, and G. J. M. Smit. Low-cost guaranteed-throughput communication ring for real-time streaming MPSoCs. In *2013 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 239–246, Oct. 2013. 00002. (Cited on page 11).

[30] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium*, page 362–376, 1974. URL http://link.springer.com/chapter/10.1007/3-540-06859-7_145. (Cited on pages 10 and 14).

[31] J. B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, 1980. ISSN 0018-9162. doi: 10.1109/MC.1980.1653418. (Cited on page 18).

[32] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. *ACM SIGARCH Computer Architecture News*, 3(4):126–132, 1974. URL http://dl.acm.org/citation.cfm?id=642111. (Cited on pages 9 and 17).

[33] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD — reconfigurable pipelined datapath. In R. W. Hartenstein and M. Glesner, editors, *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, number 1142 in Lecture Notes in Computer Science, pages 126–135. Springer Berlin Heidelberg, Jan. 1996. ISBN 978-3-540-61730-3, 978-3-540-70670-0. URL http://link.springer.com/chapter/10.1007/3-540-61730-2_13. (Cited on page 21).

[34] J. Eker and J. Janneck. CAL language report. *University of California at Berkeley, Tech. Rep. UCB/ERL M*, 3, 2003. URL https://embedded.eecs.berkeley.edu/caltrop/docs/LanguageReport/CLR-1.0-r1.pdf. (Cited on page 15).

[35] J. Eker and J. W. Janneck. Dataflow programming in CAL - balancing expressiveness, analyzability, and implementability. In *Signals, Systems and Computers (ASILOMAR), 2012 Conference Record of the Forty Sixth Asilomar Conference on*, page 1120–1124, 2012. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6489194. (Cited on page 15).

[36] C. Elliott, S. Finne, and O. De Moor. Compiling embedded languages. *J. Funct. Program.*, 13(3):455–481, May 2003. ISSN 0956-7968. doi: 10.1017/S0956796802004574. URL http://dx.doi.org/10.1017/S0956796802004574. 00084. (Cited on page 78).

[37] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990. URL `http://www.sciencedirect.com/science/article/pii/074373159090035N`. (Cited on page 15).

[38] D. Gajski, D. Padua, D. Kuck, and R. H. Kuhn. A second opinion on data flow machines and languages. *Computer*, 15(2):58–69, 1982. ISSN 0018-9162. doi: 10.1109/MC.1982.1653942. (Cited on pages 13 and 49).

[39] M. Gerards, C. Baaij, J. Kuper, and M. Kooijman. Higher-order abstraction in hardware descriptions with CλaSH. In *2011 14th Euromicro Conference on Digital System Design (DSD)*, pages 495–502, 2011. doi: 10.1109/DSD.2011.69. (Cited on page 35).

[40] C. Giraud-Carrier. A reconfigurable dataflow machine for implementing functional programming languages. *SIGPLAN Not.*, 29(9):22–28, Sept. 1994. ISSN 0362-1340. doi: 10.1145/185009.185014. URL `http://doi.acm.org/10.1145/185009.185014`. (Cited on page 18).

[41] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor. PipeRench: a reconfigurable architecture and compiler. *Computer*, 33(4):70–77, 2000. ISSN 0018-9162. doi: 10.1109/2.839324. (Cited on page 21).

[42] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. *SIGOPS Oper. Syst. Rev.*, 36(5):291–303, Oct. 2002. ISSN 0163-5980. doi: 10.1145/635508.605428. URL `http://doi.acm.org/10.1145/635508.605428`. (Cited on page 15).

[43] T. Goubier, R. Sirdey, S. Louise, and V. David. ∑C: a programming model and language for embedded manycores. In *Algorithms and Architectures for Parallel Processing*, page 385–394. Springer, 2011. URL `http://link.springer.com/chapter/10.1007/978-3-642-24650-0_33`. (Cited on page 16).

[44] J. R. Gurd, C. C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Commun. ACM*, 28(1):34–52, Jan. 1985. ISSN 0001-0782. doi: 10.1145/2465.2468. URL `http://doi.acm.org/10.1145/2465.2468`. (Cited on page 17).

[45] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=97300`. (Cited on page 14).

[46] C. L. Hankin and H. W. Glaser. The data flow programming language CAJOLE - an informal introduction. *SIGPLAN Not.*, 16(7):35–44, July 1981. ISSN 0362-1340. doi: 10.1145/947864.947867. URL `http://doi.acm.org/10.1145/947864.947867`. (Cited on page 15).

[47] F. Hannig, H. Dutta, and J. Teich. Regular mapping for coarse-grained reconfigurable architectures. In *IEEE International Conference on Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04)*, volume 5, pages V–57–60 vol.5, 2004. doi: 10.1109/ICASSP.2004.1327046. (Cited on page 20).

[48] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '01, page 642–649, Piscataway, NJ, USA, 2001. IEEE Press. ISBN 0-7695-0993-2. URL `http://dl.acm.org/citation.cfm?id=367072.367839`. (Cited on page 18).

[49] R. Hartenstein and R. Kress. A datapath synthesis system for the reconfigurable datapath architecture. In *Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95., IFIP International Conference on Hardware Description Languages. IFIP International Conference on Very Large Scal*, pages 479–484, 1995. doi: 10.1109/ASPDAC.1995.486359. (Cited on page 78).

[50] A. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *18th International Conference on Parallel Architectures and Compilation Techniques, 2009. PACT '09*, pages 214–223, 2009. doi: 10.1109/PACT.2009.39. (Cited on page 15).

[51] W. M. Johnston, J. R. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 2004. URL `http://dl.acm.org/citation.cfm?id=1013209`. (Cited on pages 10, 11, 12, 13, 14, and 18).

[52] G. Kahn. The semantics of a simple language for parallel programming. In J. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, 1974. (Cited on page 10).

[53] G. Kahn and D. Macqueen. Coroutines and networks of parallel processes. 1976. URL `http://hal.inria.fr/inria-00306565`. (Cited on page 10).

[54] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6): 1390–1411, Nov. 1966. ISSN 0036-1399, 1095-712X. doi: 10.1137/0114108. URL `http://epubs.siam.org/doi/abs/10.1137/0114108`. (Cited on page 9).

[55] Keller. A loosely-coupled applicative multi-processing system, 1979. URL `http://www.computer.org/csdl/proceedings/afips/1979/5087/00/50870613-abs.html`. (Cited on page 17).

[56] J. Kuper, C. Baaij, M. Kooijman, and M. Gerards. Exercises in architecture specification using CλaSH. In *2010 Forum on Specification Design Languages (FDL 2010)*, pages 1–6, Sept. 2010. doi: 10.1049/ic.2010.0149. 00000. (Cited on page 32).

[57] M. Lanuzza, S. Perri, P. Corsonello, and M. Margala. A new reconfigurable coarse-grain architecture for multimedia applications. In *Second NASA/ESA Conference on Adaptive Hardware and Systems, 2007. AHS 2007*, pages 119–126, 2007. doi: 10.1109/AHS.2007.10. (Cited on page 19).

[58] D. Lee, M. Jo, K. Han, and K. Choi. FloRA: coarse-grained reconfigurable architecture with floating-point operation capability. In *International Conference on Field-Programmable Technology, 2009. FPT 2009*, pages 376–379, 2009. doi: 10.1109/FPT.2009.5377609. (Cited on page 19).

[59] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9): 1235–1245, 1987. (Cited on page 11).

[60] C. Liang and X. Huang. SmartCell: an energy efficient coarse-grained reconfigurable architecture for stream-based applications. *EURASIP Journal on Embedded Systems*, 2009(1):518659, June 2009. ISSN 1687-3963. doi: 10.1155/2009/518659. URL `http://jes.eurasipjournals.com/content/2009/1/518659/abstract`. 00012. (Cited on page 19).

[61] M. Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011. ISBN 1593272839, 9781593272838. (Cited on page 23).

[62] J. R. McGraw. The VAL language: Description and analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(1):44–82, 1982. URL `http://dl.acm.org/citation.cfm?id=357157`. (Cited on page 15).

[63] B. Mei, S. Vernalde, D. Verkest, H. D. 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*, number 2778 in Lecture Notes in Computer Science, pages 61–70. Springer Berlin Heidelberg, Jan. 2003. ISBN 978-3-540-40822-2, 978-3-540-45234-8. URL `http://link.springer.com/chapter/10.1007/978-3-540-45234-8_7`. (Cited on page 20).

[64] E. Mirsky and A. DeHon. MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *IEEE Symposium on FPGAs for Custom Computing Machines, 1996. Proceedings*, pages 157–166, 1996. doi: 10.1109/FPGA.1996.564808. (Cited on page 20).

[65] T. Miyamori and K. Olukotun. A quantitative analysis of reconfigurable coprocessors for multimedia applications. In *IEEE Symposium on FPGAs for Custom Computing Machines, 1998. Proceedings*, pages 2–11, 1998. doi: 10.1109/FPGA.1998.707876. (Cited on page 21).

[66] T. Miyamori and K. Olukotun. REMARC: Reconfigurable multimedia array coprocessor. *IEICE TRANSACTIONS on Information and Systems*, E82-D(2):389–397, Feb. 1999. ISSN , 0916-8532. URL `http://search.ieice.org/bin/summary.php?id=e82-d_2_389&category=D&year=1999&lang=E&abst=`. 00266. (Cited on page 21).

[67] N. Muniz, M. Alves, and S. D. M. Schneider. *Implementation of an Embedded Hardware Description Language Using Haskell.* 2003. 00004. (Cited on page 78).

[68] R. S. Nikhil. Can dataflow subsume von neumann computing? In *ACM SIGARCH Computer Architecture News*, volume 17, page 262–272, 1989. URL `http://dl.acm.org/citation.cfm?id=74955`. (Cited on page 18).

[69] R. S. Nikhil and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):598–632, 1989. URL `http://dl.acm.org/citation.cfm?id=69562`. (Cited on page 14).

[70] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. *SIGARCH Comput. Archit. News*, 18(3a):82–91, May 1990. ISSN 0163-5964. doi: 10.1145/325096.325117. URL `http://doi.acm.org/10.1145/325096.325117`. (Cited on page 18).

[71] R. Paterson. Arrows and computation. *The Fun of Programming*, page 201–222, 2003. URL `http://ipaper.googlecode.com/git-history/243b02cb56424d9e3931361122c5aa1c4bdcbbbd/Arrow/arrows-fop.pdf`. 00031. (Cited on page 35).

[72] J. Rabaey. *Low Power Design Essentials*. Springer Publishing Company, Incorporated, 1st edition, 2009. ISBN 0387717129, 9780387717128. 00215. (Cited on page 60).

[73] G. Roquier, E. Bezati, and M. Mattavelli. Hardware and software synthesis of heterogeneous systems from dataflow programs. *JECE*, 2012:2:2–2:2, Jan. 2012. ISSN 2090-0147. doi: 10.1155/2012/484962. URL `http://dx.doi.org/10.1155/2012/484962`. (Cited on page 15).

[74] J. Rumbaugh. A data flow multiprocessor. *Computers, IEEE Transactions on*, 100(2):138–146, 1977. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5009292`. (Cited on page 17).

[75] S. Sakai, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a dataflow single chip processor. In *ACM SIGARCH Computer Architecture News*, volume 17, page 46–53, 1989. URL `http://dl.acm.org/citation.cfm?id=74931`. (Cited on page 18).

[76] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. Keckler, and C. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings*, pages 422–433, 2003. doi: 10.1109/ISCA.2003.1207019. (Cited on page 20).

[77] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S. W. Keckler, R. G. McDonald, and C. R. Moore. TRIPS: a polymorphous architecture for exploiting ILP, TLP, and DLP. *ACM Trans. Archit. Code Optim.*, 1 (1):62–93, Mar. 2004. ISSN 1544-3566. doi: 10.1145/980152.980156. URL `http://doi.acm.org/10.1145/980152.980156`. (Cited on page 20).

[78] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, 2000. ISSN 0018-9340. doi: 10.1109/12.859540. (Cited on page 21).

[79] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. McKinley. Compiling for EDGE architectures. In *International Symposium on Code Generation and Optimization, 2006. CGO 2006*, pages 185–195, 2006. doi: 10.1109/CGO.2006.10. (Cited on page 20).

[80] G. L. Steele, Jr. and R. P. Gabriel. The evolution of Lisp. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, page 231–270, New York, NY, USA, 1993. ACM. ISBN 0-89791-570-4. doi: 10.1145/154766.155373. URL `http://doi.acm.org/10.1145/154766.155373`. 00099. (Cited on page 14).

[81] T. Sterling, J. Kuehn, M. Thistle, and T. Anastasis. Studies on optimal task granularity and random mapping. *Advanced Topics in Dataflow Computing and Multithreading*, pages 349–365, 1995. (Cited on page 15).

[82] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, page 291–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2043-X. URL `http://dl.acm.org/citation.cfm?id=956417.956546`. (Cited on page 18).

[83] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers. The WaveScalar architecture. *ACM Trans. Comput. Syst.*, 25(2):4:1–4:54, May 2007. ISSN 0734-2071. doi: 10.1145/1233307.1233308. URL `http://doi.acm.org/10.1145/1233307.1233308`. (Cited on page 18).

[84] T. Tantau. The TikZ and pgf packages. 2007. URL `http://elibrary.matf.bg.ac.rs/handle/123456789/2925`. 00024. (Cited on page 98).

[85] V. Tehre and R. Kshirsagar. Survey on coarse grained reconfigurable architectures. *International Journal of Computer Applications*, 48(16):1–7, June 2012. ISSN 09758887. doi: 10.5120/7429-0104. URL `http://adsabs.harvard.edu/abs/2012IJCA...48p...1T`. (Cited on pages 18 and 21).

[86] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: a language for streaming applications. In R. N. Horspool, editor, *Compiler Construction*, number 2304 in Lecture Notes in Computer Science, pages 179–196. Springer Berlin Heidelberg, Jan. 2002. ISBN 978-3-540-43369-9, 978-3-540-45937-8. URL `http://link.springer.com/chapter/10.1007/3-540-45937-5_14`. (Cited on page 15).

[87] A. H. Veen. Dataflow machine architecture. *ACM Computing Surveys (CSUR)*, 18 (4):365–396, 1986. URL `http://dl.acm.org/citation.cfm?id=28055`. (Cited on pages 16, 17, and 18).

[88] G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh, W. Bohm, and J. Hammes. Automatic compilation to a coarse-grained reconfigurable system-on-chip. *ACM Trans. Embed. Comput. Syst.*, 2(4):560–589, Nov. 2003. ISSN 1539-9087. doi: 10.1145/950162.950167. URL `http://doi.acm.org/10.1145/950162.950167`. (Cited on page 21).

[89] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, et al. Baring it all to software: Raw machines. *Computer*, 30(9): 86–93, 1997. (Cited on page 15).

[90] I. Watson and J. Gurd. A practical data flow computer. *Computer*, 15(2):51–57, 1982. ISSN 0018-9162. doi: 10.1109/MC.1982.1653941. (Cited on pages 17 and 18).

[91] P. G. Whiting and R. S. Pascoe. A history of data-flow languages. *Annals of the History of Computing, IEEE*, 16(4):38–59, 1994. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=329757`. (Cited on pages 9, 10, 11, 12, 13, and 14).

[92] M. H. Wiggers. *Aperiodic multiprocessor scheduling for real-time stream processing applications*. PhD thesis, Enschede, June 2009. URL `http://doc.utwente.nl/61568/`. (Cited on pages 11 and 52).

[93] A. Yeung and J. Rabaey. A reconfigurable data-driven multiprocessor architecture for rapid prototyping of high throughput DSP algorithms. In *Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences, 1993*, volume i, pages 169–178 vol.1, 1993. doi: 10.1109/HICSS.1993.270747. (Cited on page 21).

List of Publications

[AN:1] A. Niedermeier, J. Kuper, and G. J. M. Smit. Dataflow-based reconfigurable architecture for streaming applications. In *Proceedings of System on Chip (SoC), 2012 International Symposium on System-on-Chip, Tampere, Finland*, pages 1–4, USA, October 2012. IEEE Circuits & Systems Society.

[AN:2] A. Niedermeier, J. Kuper, and G. J. M. Smit. A haskell-based programming paradigm for coarse-grained reconfigurable arrays. In *Proceedings of the Work in Progress Session of the 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2012) and the 15th EUROMICRO Conference on Digital System Design (DSD 2012), Çeşme, Izmir, Turkey*, SEA-Publications, pages 17–18, Linz, Austria, September 2012. Institute for Systems Engineering and Automation, J. Kepler University Linz.

[AN:3] Anja Niedermeier, Jan Kuper, and Gerard Smit. A dataflow inspired programming paradigm for coarse-grained reconfigurable arrays. In *Reconfigurable Computing: Architectures, Tools, and Applications, 10th International Symposium, ARC 2014*, pages 275–282. Springer, April 2014.

[AN:4] A. Niedermeier, J. Kuper, and G. J. M. Smit. A dataflow-inspired CGRA for streaming applications. In *23rd International Conference on Field Programmable Logic and Applications, FPL 2013, Porto, Portugal*, pages 1–2. IEEE Circuits & Systems Society, September 2013.

[AN:5] A. Niedermeier, J. Kuper, and G. J. M. Smit. High level structural description of streaming applications. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on, Oslo, Norway*, pages 485–486, USA, August 2012. IEEE Circuits & Systems Society.

[AN:6] A. Niedermeier, R. Wester, C. P. R. Baaij, J. Kuper, and G. J. M. Smit. Comparing CλaSH and VHDL by implementing a dataflow processor. In *Proceedings of the Workshop on PROGram for Research on Embedded Systems and Software (PROGRESS 2010), Veldhoven, The Netherlands*, pages 216–221, Utrecht, November 2010. Technology Foundation STW.

[AN:7] A. Niedermeier, R. Wester, K. C. Rovers, C. P. R. Baaij, J. Kuper, and G. J. M. Smit. Designing a dataflow processor using CλaSH. In *28th Norchip Conference, NORCHIP 2010, Tampere, Finland*, page 69. IEEE Circuits and Systems Society, November 2010.

[AN:8] A. Niedermeier, K. Svarstad, F. Bouwens, J. Hulzink, and J. A. Huisken. The challenges of implementing fine-grained power gating. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI, Providence, Rhode Island, USA*, pages 361–364, New York, 2010. ACM.