Marcel D. van de Burgwal

Interfacing Networks-on-Chip

Hardware meeting Software

Marcel D. van de Burgwal

Members of the dissertation committee:

prof. dr. ir.	G.J.M. Smit	University of Twente (promotor)
dr. ir.	A.B.J. Kokkeler	University of Twente (assistant promotor)
dr. ir.	J. Kuper	University of Twente (assistant promotor)
prof. dr.	J.L. Hurink	University of Twente
prof. dr. ir.	F.E. van Vliet	University of Twente / TNO
prof. dr. ir.	D. Stroobandt	Ghent University, Belgium
dr. ir.	H. Schurer	Thales Nederland B.V.
prof. dr. ir.	A.J. Mouthaan	University of Twente (chairman and secretary)

Parts of this research have been conducted within the CMOS Beamforming Techniques project (07620), supported by the Dutch Technology Foundation STW, applied science division of NWO and the Technology Program of the Ministry of Economic Affairs.

Parts of this research have been conducted within the Smart Chips for Smart Surroundings project (IST-001908) supported by the Sixth Framework Programme of the European Community.

Center for Telematics and Information Technology
P.O. Box 217
7500 AE Enschede
The Netherlands

Typeset with LaTeX.
Printed by Wöhrmann Print Service, Zutphen, The Netherlands

ISBN 978-90-365-3067-5
ISSN 1381-3617 (CTIT Ph.D.-thesis series No. 10-177)
DOI http://dx.doi.org/10.3990/1.9789036530675

INTERFACING NETWORKS-ON-CHIP
HARDWARE MEETING SOFTWARE

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 15 oktober 2010 om 15.00 uur

door

Marcel Dominicus van de Burgwal

geboren op 26 juni 1981
te Amersfoort

Dit proefschrift is goedgekeurd door:

prof.	dr. ir.	G.J.M. Smit	(promotor)
	dr. ir.	A.B.J. Kokkeler	(assistent promotor)
	dr. ir.	J. Kuper	(assistent promotor)

Abstract

Wireless communication is becoming more and more important in today's world. We rely on radio transmission for audio/video broadcasts, telecommunication, satellite navigation, security systems and many wireless sensor devices. These communication systems use reserved parts of the frequency spectrum, to ensure low interference between transmitters and receivers of different communication systems. Since spectrum is scarce, many advanced wireless standards have been proposed to efficiently use parts of the spectrum, by applying (often complex) digital processing to the signal to be transmitted. Another approach to efficiently use the spectrum is by applying spatial filtering, which can be done by using multiple antennas that are used one at a time (spatial diversity), or by using multiple antennas coherently (in a phased array).

An important aspect of wireless communication is battery lifetime. However, the digital processing algorithms, used to increase the spectrum utilization, require complex operations to be performed at high speeds by the hardware platform. Therefore, the ever increasing complexity of such algorithms poses tough requirements for next-generation hardware platforms. Instead of realizing a single complex processor with high transistor count, current single-chip architectures are based on multiple, less complex processors that work in parallel. They share a single memory space, which is accessible via a shared communication infrastructure. For small numbers of processors, a shared bus can be used efficiently and implementation costs are low. However, future architectures will consist of tens to hundreds of processors, which will be limited in performance when they have to share the bandwidth provided by a bus interconnect structure.

A Multi-processor System-on-Chip (MPSoC) with a Network-on-Chip (NoC) interconnect solves the problem of bus sharing. Each processor is connected to a local router, which on its turn is connected to a fixed number of other routers. Since the number of connections per router is independent of the number of routers and processors in the network, such a system can be scaled without losing interconnect efficiency. Processors can communicate via the NoC by using Virtual Channels (VCs), which are created by configuring the routers on the path from one processor to another, such that data received on the input port is forwarded to the correct output port. The arbitration protocol used by the routers is designed such that a minimum bandwidth guarantee and a maximum latency guarantee can be given for VCs mapped on the NoC.

This thesis presents the design and analysis of the Hydra Network Interface

(NI), an efficient interface between worlds of computation (the processors) and communication (the NoC). It provides an abstraction mechanism for the application running on that processor, such that a VC can be used without knowledge of the routers that are traversed when a path through the NoC is taken. The characteristics and performance of the NI are evaluated to show that it is an efficient interface, for example because it introduces a minimal latency to communication streams and it does not limit the throughput bandwidth. A concrete realization of the Hydra NI was used in the Annabelle chip, a prototype multi-core chip developed in the EU Smart Chips for Smart Surroundings (4S) project.

Another advantage of using an MPSoC based architecture besides parallel processing, is concurrency in computation and communication. To utilize this concurrency efficiently, the NI should support this concurrency. The programming model for such an architecture differs from conventional single processor systems. Partitioning of the applications into multiple concurrent threads is important to obtain high utilization of the computational resources. The Synchronous Data Flow (SDF) model can be used to model and analyze an application as a set of independent kernels connected by communication channels. The kernels are mapped on the processors in the architecture, and the communication channels between these processors are mapped on the NoC that connects the processors. To verify the performance of an application mapped on a NoC based architecture, a simulation model is created containing information about both the application and architecture model. The application model is based on a functional programming language, which has a strong resemblance with mathematics such that the application can be gradually translated from a mathematical specification to a partitioned realization. Modifications can be performed to the obtained application model by applying transformations in the form of mathematical rewrite rules. Another advantage of the functional programming language is that functions are side-effect free, such that shared variables can only be used when explicitly modeled. In this thesis, the design flow that enables modeling of streaming applications is discussed. The design flow includes the mathematical description, partitioning and simulation of the application.

Although wireless communication should be energy-efficient, a certain minimum performance is required to guarantee correct reception and decoding of the signal. Two different examples are discussed in detail: a DRM receiver for handheld devices and a DVB-S satellite receiver for in-car infotainment. The first addresses a battery operated device, hence the receiver implementation should be energy-efficient. The latter example uses a phased array antenna, mounted on the roof of a car, to receive an audio/video broadcast transmitted by a satellite. Here, the large number of antenna streams determines the vast amount of processing required to coherently combine the signals received from individual antennas, such that an amplified and focused signal is obtained. Both applications are mapped on the same MPSoC architecture template to show the flexibility of the architecture. Special attention is given to the Montium Tile Processor (TP) and the mapping of the kernels of the DRM and DVB-S applications onto it. In this thesis, the performance of both applications is evaluated to show that the Hydra NI supports efficient processing.

Samenvatting

Draadloze communicatie wordt meer en meer gebruikt in de hedendaagse wereld. Voor veel toepassingen zijn we afhankelijk van radiotechnologie, zoals bij audio/video-uitzendingen, telecommunicatie, navigatiesystemen, beveiligingsapparatuur en voor sensorsystemen. Voor ieder van deze toepassingen is een deel van het frequentiespectrum gereserveerd, zodat er geen onderlinge verstoring plaats vindt. Aangezien het spectrum beperkt is, is het belangrijk dat het efficiënt wordt gebruikt door de verschillende toepassingen. Geavanceerde draadloze standaarden gebruiken delen van het spectrum zeer efficiënt voor radiocommunicatie. Hierbij wordt het te verzenden signaal eerst digitaal bewerkt, zodat het kan worden verzonden in een beperkter spectrum. Een andere techniek die populair begint te worden, is uitzenden het van signaal in een specifieke richting. Dit kan door gebruik te maken van richtingsgevoelige antennes, geconstrueerd uit een reeks van antennes.

Bij draadloze communicatie is batterijduur van groot belang. De digitale bewerkingen die op het te verzenden signaal moeten worden toegepast om het signaal in een beperkt spectrum te kunnen verzenden, bestaan uit complexe operaties die door een geavanceerde rekeneenheid moeten worden uitgevoerd. Nieuwe communicatiestandaarden bieden een hogere kwaliteit van beeld en geluid, wat er voor zorgt dat er meer informatie moet worden verzonden. Omdat het beschikbare spectrum gelijk blijft, zijn complexe digitale berekeningen nodig om de informatie te kunnen verzenden in hetzelfde spectrum. De eisen aan beschikbare rekenkracht bepalen hierdoor de ontwikkeling van nieuwe generaties rekeneenheden. De huidige trend is het combineren van meerdere (simpele) rekeneenheden op een chip, in plaats van het toevoegen van extra rekenkracht aan een enkele rekeneenheid. Deze rekeneenheden wisselen informatie uit via een gedeeld geheugen, dat ze kunnen benaderen via een gedeelde verbinding. Zolang het aantal rekeneenheden klein blijft, kunnen ze om de beurt gebruik maken van deze verbinding. Echter, bij grotere aantallen rekeneenheden moeten de rekeneenheden lang op elkaar wachten, voordat ze bij het gedeelde geheugen kunnen.

Een multi-processor systeem-op-een-chip bestaat uit een aantal van zulke rekeneenheden die onderling verbonden zijn via een netwerk aan verbindingen. Iedere rekeneenheid kan met andere rekeneenheden communiceren via een lokaal routeringselement, dat verbonden is met een beperkt aantal routeringselementen van andere rekeneenheden. Op deze manier zijn alle rekeneenheden indirect met alle andere rekeneenheden verbonden. Bij een schaalvergroting van het totale systeem kunnen rekeneenheden worden toegevoegd zonder de efficiëntie van bestaande ver-

bindingen te beïnvloeden. De fysieke verbindingen tussen routeringselementen zijn opgedeeld in kleinere virtuele verbindingen, zodat een rekeneenheid tegelijkertijd één of meerdere verbindingen met andere rekeneenheden kan hebben. De routeringselementen zorgen ervoor, dat informatie wordt verzonden naar de gekozen bestemming, waarbij een minimale doorvoersnelheid van informatie kan worden gegarandeerd met een maximale gegarandeerde vertraging.

Dit proefschrift beschrijft het ontwerp en de implementatie van de Hydra-netwerkinterface, een efficiënte verbinding tussen de rekeneenheden en het netwerk. De interface verbergt details, die het netwerk met zich meebrengt, voor de rekeneenheid. Om te laten zien dat deze abstractie weinig invloed uitoefent op de doorvoersnelheid en vertraging van verstuurde informatie, worden de karakteristieken van de netwerkinterface geëvalueerd. Aan de hand van de Annabelle prototypechip die ontwikkeld is binnen het Smart Chips for Smart Surroundings (4S) project, wordt de realisatie van de interface getoond.

Naast de parallelle berekeningen die door verschillende rekeneenheden tegelijkertijd kunnen worden gedaan, biedt het gepresenteerde multi-processor-systeem-op-een-chip ook de mogelijkheid om rekeneenheden tegelijkertijd te laten rekenen en informatie te laten uitwisselen. De netwerkinterface ondersteunt deze mogelijkheid. Echter, voor het ontwerp van programmatuur voor een dergelijk systeem, moet rekening worden gehouden met de verdeling van berekeningen over meerdere rekeneenheden zodat alle rekeneenheden efficiënt kunnen worden ingezet. Door gebruik te maken van een model voor synchrone datastromen (SDF) kan een analyse worden gedaan op de programmatuur. Hierbij wordt de programmatuur opgedeeld in kleinere processen die onderling communiceren via kanalen, zodanig dat de processen worden uitgevoerd door rekeneenheden en de communicatiekanalen tussen rekeneenheden lopen via het netwerk. Door een simulatie te maken van de processen en onderlinge communicatie kan een inschatting worden gemaakt van de prestatie van de volledige programmatuur. Vanuit de wiskundige specificatie wordt de programmatuur stapsgewijs uitgewerkt in een functionele programmeertaal. Met behulp van herschrijfregels, die worden toegepast op de functionele programmeertaal, wordt de programmatuur voorbereid zodanig dat het uiteindelijk door één of meerdere rekeneenheden kan worden uitgevoerd. Omdat de programmeertaal geen impliciet gedeelde informatie tussen processen toestaat, moet alle communicatie tussen processen expliciet worden gemaakt. Dit proefschrift presenteert een aanpak voor het ontwerpen en simuleren van stroom-gebaseerde programmatuur, inclusief de wiskundige beschrijving, opsplitsing in kleinere processen en simulatie daarvan.

Tot slot worden twee verschillende draadloze communicatie-ontvangers besproken, namelijk een mobiele ontvanger voor digitale radio (DRM) en een ontvanger voor satelliet uitzendingen (DVB-S), die op een auto wordt gemonteerd. Beide ontvangers dienen energie-efficiënt te zijn bij een minimale prestatie om het ontvangen radiosignaal goed te kunnen verwerken: de mobiele DRM ontvanger heeft een beperkte accucapaciteit en de DVB-S-ontvanger moet enorme informatiestromen verwerken die worden ontvangen door een groot aantal antennes. De implementatie van beide ontvangers op een multi-processor systeem-op-een-chip wordt in detail besproken, waarbij de prestatie van de ontvangers wordt gebruikt om de efficiëntie van de Hydra-netwerkinterface aan te tonen.

Dankwoord

In het vroege voorjaar van 2005, halverwege mijn afstudeerproject, liepen we op het Münsterplatz in Ulm (Duitsland), toen Gerard Smit mij vroeg: "Ken jij misschien nog mensen die interesse hebben in een AIO positie?". Ongeveer een jaar daarvoor had ik bij de vakgroep CAES aangeklopt, op zoek naar een begeleider voor mijn stage bij Lely Technologies. Gerard reageerde destijds met het antwoord "Dat wil ik zelf wel doen", dus toen Gerard in Ulm mij die ene vraag stelde, reageerde ik met datzelfde antwoord.

Mijn voorkeur voor Embedded Systems binnen de opleiding Technische Informatica werd al duidelijk bij het allereerste contact met de leerstoel CAES, bij het vak *(Basisbegrippen) Digitale Techniek*. Na mijn stage kwam ik via Gerard terecht op het CCU project, waarbij in korte periode een netwerkinterface voor de Montium moest worden ontwikkeld voor een prototypechip in samenwerking met Atmel Germany GmbH in Ulm. In die tijd begon Gerard met het huisvesten van afstudeerders tussen promovendi, zodat ze efficiënter konden meedraaien en er meer kennis overdracht kon plaatsvinden. Hierdoor kreeg ik de kans om de gang van zaken op een wetenschappelijke afdeling van binnenuit te bekijken, waarbij ik zelfs een zakentripje aangeboden kreeg naar Ulm. Ik wil Gerard ontzettend bedanken voor de kans die hij mij gaf om bij CAES te komen promoveren, voor de jarenlange begeleiding en alle motiverende discussies en brainstorm acties. Zonder de steun en feedback van Gerard zou dit proefschrift niet hebben bestaan.

Toen Gerard in 2007 werd aangesteld als leerstoelhouder van CAES, werd de rol van André Kokkeler als dagelijks begeleider een stuk groter. Het 4S project was in datzelfde jaar afgelopen en ik was juist overgestapt naar het CMOS Beamforming Techniques project. André's kennis over radiosystemen en signaalbewerking bleken ontzettend nuttig bij de begeleiding in dat project, want we hebben veel discussies gehad over specifieke operaties waarbij André vaak de basis kon uitleggen in een korte samenvatting. Jan Kuper raakte betrokken bij dit onderzoek toen bleek dat bestaande ontwerpmethoden te kort schoten in termen van formalisme. De overstap naar een functionele programmeermethode leidde, dankzij zijn onbeperkt enthousiasme en optimisme, tot nieuwe inzichten en mogelijkheden.

Al op mijn eerste dag op de afdeling werd me duidelijk dat de sfeer binnen de CAES groep geweldig is en dat biedt een goede basis voor een promotietraject. Ik wil alle collega's van de CAES groep bedanken voor de geweldige samenwerking. Tijdens mijn studie, bij het vak *Ontwerpen van Digitale Systemen*, maakte ik voor het eerst kennis met Bert Molenkamp. De contacten die wij hadden in de jaren daarna leidden

er mede toe dat ik bij Gerard en Bert een afstudeeropdracht kwam doen. Het was ontzettend handig om een VHDL goeroe als begeleider én als buurman te hebben, in het kantoor om de hoek (zelfs als je vragen plaatst op de nieuwsgroep `comp.lang.vhdl`, waar Bert als eerste reageert). Voor de organisatorische zaken binnen de UT kun je altijd een beroep doen op de dames van het secretariaat: Marlous, Nicole en Thelma. In de tijd dat ik aan het 4S project werkte, heb ik veel over Networks-on-Chip geleerd van Pascal Wolkotte. De contacten met Paul Heysters, Gerard Rauwerda en Lodewijk Smit gaven mij erg veel inzicht in de Montium architectuur, die op dat moment centraal stond in de onderzoeksprojecten. Bij mijn overstap naar het CMOS Beamforming Techniques project ben ik gaan samenwerken met Kenneth Rovers. We hebben samen veel vruchtbare discussies gevoerd, waar ik enorm veel van heb geleerd. Samen hebben we ook een aantal afstudeerders begeleid, waaronder Koen Blom, wiens werk nuttig bleek bij de totstandkoming van dit proefschrift.

Naast mijn promotie heb ik de afgelopen jaren een flink deel van de avonden en weekenden doorgebracht met een groot aantal muzikanten in Enschede en omstreken. Ik wil in het bijzonder Bart Bijleveld noemen, ook wel de *maffiabaas van het oosten* genoemd vanwege zijn grote inzet voor en betrokkenheid bij de amateur-jazzmuziek in de regio Twente. Mede dankzij hem heb ik in de jazzmuziek de nodige afleiding gevonden om nieuwe energie en inspiratie op te doen voor mijn promotie.

Tijdens mijn lidmaatschap van D.B.V. Arriba en gedurende de periode daarna, waarin we huisgenoten waren, leerde ik Eelco Kuipers kennen. Samen met Eelco en zijn vriendin Jade Reinders hebben we de afgelopen jaren heel wat festivals en optredens bezocht. Ik wil jullie bedanken voor alle leuke tijden die we samen hebben gehad en hopelijk nog gaan krijgen.

Het jaar 2008 was een jaar dat voor mij in het teken van mijn familie stond. Door de ziekte van mijn vader realiseerde ik me hoe belangrijk je ouders zijn. Henrie en Agnes: tijdens mijn opleiding van ruwweg een decennium aan de UT heb ik nog steeds niet zoveel geleerd als wat jullie me bijbrachten en ik hoop nog lang en veel van jullie te kunnen blijven leren. Yolanda: jij bent degene die altijd als eerste klaarstaat, vooral als het gaat om het beschikbaar stellen van jouw organisatorisch vermogen voor welke aangelegenheid dan ook. Erik: bedankt dat je, samen met Kenneth, mij tijdens de aanloop van de promotie en gedurende de dag wilt ondersteunen als paranimf.

Ik sluit af door mijn allergrootste dank uit te spreken aan de belangrijkste persoon uit mijn leven: Tineke Klamer. Je staat altijd lijnrecht achter mij en de dingen die ik doe. Bedankt voor je steun in moeilijke tijden en dat je altijd voor mij klaarstaat.

Marcel van de Burgwal
Enschede, September 2010

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Streaming DSP applications

Next generation multi-media appliances will communicate via wireless connections at any time and any place. Digital multimedia broadcast standards, such as Digital Radio Mondiale (DRM), Digital Audio Broadcast (DAB) and Digital Video Broadcast for Satellite (DVB-S), use encoded high-bandwidth streams of data to reconstruct the original high quality signal, at the cost of computation intensive processing. For battery powered portable devices this is quite challenging, as the energy source has limited capacity. By optimizing the computationally intensive kernels within an application, the energy consumption can be reduced significantly. Typically, the streaming multi-media applications mentioned have a regular communication scheme using connections that remain unchanged for a long period of time. Since they show strong temporal and spatial locality, these applications are quite suitable to be executed by a highly parallel Multi-processor System-on-Chip (MPSoC) platform [1]. For efficiency reasons, such Multi-Processor Systems-on-Chip are often designed as heterogeneous tiled architectures. These architectures consist of several types of tiles which are connected via a Network-on-Chip (NoC).

1.2 Low power versus high performance

With each new generation of processor architectures, the offered processing capacity is increased. This enables the design and execution of applications with a higher computational complexity. By using the hardware efficiently, the time between two processor generations can be increased. Depending on the type of application, such efficiency may either involve less energy consumption per execution or more executions per second. Energy consumption can be decreased by making the architecture suitable for *low power* operation, for example by adding accelerator blocks or by adding hardware building blocks that allow dynamic adaptation of the hardware to its changing environment. Another approach is to optimize the architecture for *high performance*, where the utilization of the processor capacity is increased, for example

in a multi-processor architecture running multiple applications in parallel such that each application has one or more processors at its disposal exclusively.

The architectures described in this thesis target both the low power and the high performance domain. The building blocks are designed for efficient processing, such that they can be employed for architectures optimized for energy efficiency (by running the cores at low clock frequencies) or for high-performance architectures (by using many processors in parallel connected by a high bandwidth network on chip).

This work has been performed in two projects: the EU FP6 project 4S [2] and the STW project CMOS Beamforming Techniques [3]. In the next section, the main objectives of these projects are presented.

1.2.1 Smart Chips for Smart Surroundings

The Smart Chips for Smart Surroundings (4S) project [2, 4] focused on energy efficient processing using both an efficient hardware platform and an efficient application design flow. Therefore, two objectives were proposed (cited from [2]):

1. *The design of a flexible reconfigurable platform based on heterogeneous building blocks such as analogue blocks, hardwired functions, fine and coarse grain reconfigurable tiles, DSPs and microprocessors that can adapt to several algorithms for ambient systems without the need for specialized ASICs. The concept is verified on hardware platforms. Furthermore, a digital MPSoC and an analog frontend IC will be designed. The DRM and MPEG-4 applications will be implemented on the platform in order to verify the flexibility of the platform.*

2. *To provide a design flow at compile time, which reduces development time and to provide functions that automatically allocate resources of the reconfigurable platform based on QoS, power and user demands. The DRM and MPEG-4 applications will verify the design flow.*

1.2.2 CMOS Beamforming Techniques

Another application for heterogeneous tiled architectures is the domain of computationally intensive applications. In this application domain, the processing requirements are very high due to the processing on high data rate signals or complex operations. A typical example of these applications is phased array processing, which is required for antenna systems consisting of hundreds to thousands of antenna elements. By combining the signals received by all individual elements, a beam is formed. Although the processing itself is relatively simple, data rates may become high (10 to 100 Msamples/s per antenna) and the maximum processing latency is limited. By dividing the processing needs over the analog front-end and the digital processing platform, data rates and the digital antenna processing requirement for forming a beam are lowered. The CMOS Beamforming Techniques project [3] aimed at a mixed-signal phased array receiver, which consists of a modular antenna system. This enables a multi-standard prepared phased array receiver that can be used for example for radar systems, radio astronomy, satellite communication systems and telecom base stations (see Figure 1.1).

(a) Radio Astronomy: EMBRACE array [5]

(b) Naval Radar: Thales APAR [6]

(c) Satellite receiver: TracVision ®A7 [7]

Figure 1.1 – Phased array antenna usage in several applications

A modular system that can be employed for multiple standards requires a flexible interconnection architecture to provide large communication bandwidths, as required by the high data rates. Moreover, since digital processing is distributed over multiple modules, centralized control of the system by a single host may cause timing problems and, therefore, will decrease the overall system performance. A thorough analysis of the application and the mapping on the underlying MPSoC architecture structure is important.

1.3 Problem definition

In multi-core systems the communication between processor cores is crucial. Any overhead in the communication will reduce the performance and efficiency of a multi-core system. In this thesis we focus on the interaction of multi-core systems: in particular we address (1) the Network Interface hardware between the core and the Network-on-Chip, and (2) the interaction between hardware and software.

State of the art hardware/software design methodologies usually are based on a top-down derivation of an efficient hardware architecture based on a certain application domain. For such approaches, the starting point is typically a reference program that has been implemented for a single processor with a single memory space in which its state is stored. The reference program is analyzed and profiled, and code fragments with high computational complexity are offloaded to other processors. Synchronization between processors is required for efficient communication and execution, hence memory consistency models are added to control the usage of the single memory space. To improve the overall performance, the code fragments with high computational complexity are implemented for specialized processor types. Finally, a composition of processors and interconnects is compiled and realized in a

CMOS circuit. The result of such a design flow is a specialized hardware architecture that performs well for the given applications. It is flexible within the application domain.

However, the disadvantage of such design methodologies appears immediately at its start, because the sequential code forming the reference program hides information that was available before writing the reference code. Instead of starting with sequential reference code, we advocate designing applications based on the mathematic definition, for example as specified by block schematics used in communication standards. Instead of deriving a processing architecture based on a typical application set, we assume a general purpose stream processing platform based on a MPSoC using a NoC infrastructure and map our applications to that architecture. To program such an architecture, a design flow is proposed that allows parallel code as input. By transforming and partitioning this code, the parallelism in the implementation is preserved. Using a simulation framework, the parallel code can be executed in order to test its behavior and to extract performance figures that are required to determine the expected behavior when executed at an embedded platform. After successful testing using the obtained performance figures, the application can be mapped on the general purpose stream processing platform. For two different applications, we will show the stepwise derivation of a block schematic to an implementation mapped onto the Montium TP architecture. A DRM receiver is implemented for a battery powered handheld device, hence a high utilization of the processor architecture is required to have an energy efficient solution. The other application includes a DVB-S receiver using a phased array antenna that allows for satellite signal reception in dynamic environments, like for example in-car infotainment. The energy budget for this mobile adaptive receiver is higher, but the phased array antenna requires considerably more processing and therefore, an efficient solution is desired also for this application.

1.4 Contributions

This thesis combines previous research and focuses on interfaces between existing building blocks and tools. We start at the hardware architecture level, where a general purpose stream processing architecture is composed from existing processors (for example, the Montium TP) and a NoC by adding an efficient Network Interface (NI). We proceed with the presentation of a design flow for such architectures, where a mathematical programming language is proposed that can be used to model, transform and simulate applications. Finally, we show how the design flow can be used to model the implementation of applications to the hardware architecture.

i The Hydra, a NI, is presented that can provide an abstraction layer for the processing tile by managing concurrent communication and synchronization (chapter 2). Using this interface, the programming model for communication between stream processors is demonstrated and we show how communication overhead (defined by the Communication to Computation (C/C) ratio) is reduced by supporting concurrent communication and computation.

ii We present a modeling technique strongly related to mathematics for modeling streaming applications. The applications are described in a functional programming language, for which transformations are defined that can be used to prepare the application for partitioning over multiple processors. Then, by specifying explicit communication and computation in the application, separate parts of the application can be simulated and executed in parallel. We introduce a Synchronous Data Flow (SDF) simulator that is used to execute the application and to analyze its real-time behavior with real data (chapter 3).

iii Using examples based on existing wireless communication applications, we show how a stream-based implementation of DSP kernels can benefit from our network interface and modeling techniques (chapter 4). Two mobile communication receivers are discussed to show that our generic stream processing platform is useful for both energy-efficient applications and computationally intensive applications.

1.5 Thesis Outline

The thesis is organized as follows. The MPSoC architecture, and in detail the Hydra NI, are presented in chapter 2. A new modeling technique is introduced in chapter 3, that enables modeling streaming applications and their execution on a multi-processor architecture. For two wireless communication applications, namely a DRM receiver and a mobile DVB-S receiver, the proposed modeling techniques are discussed in chapter 4 by mapping the applications to a reconfigurable processor architecture. The performance figures for these algorithms are used to show how the Hydra NI presented in chapter 2 contributes to shorter execution times. Finally, in chapter 5 the work is concluded.

Chapters 2 and 3 are divided into two parts. The first part gives an overview of the state of the art, and the second part presents the contributions in each of these topics. Chapter 4 consists of three parts: the first part discusses common Digital Signal Processing (DSP) kernels that are used in many DSP algorithms, the second part evaluates the performance of a DRM receiver and in the third part a DVB-S receiver is evaluated. Finally, chapter 5 presents the joint conclusion of the three topics in this thesis.

Chapter 2

Network Interfaces for a Reconfigurable Tiled Architecture

Abstract

Reconfigurable tiled architectures are used as a flexible platform for streaming DSP applications. Such architectures consist of different processor types, suitable for different applications, which are interconnected by a Network-on-Chip. Reconfigurable processors can be dynamically customized to perform parts of these applications very efficiently. This chapter presents an efficient network interface that connects such a reconfigurable processor, the Montium TP, to an on-chip network. The network interface enables concurrency in computation and communication between processors, such that processors can operate together efficiently. The performance of the network interface is evaluated and its area footprint is related to the Montium TP.

Continuous improvements in Complementary Metal Oxide Semiconductor (CMOS) process technology enable Very-Large-Scale Integration (VLSI), such that Integrated Circuits (ICs) can contain more and more transistors. With a larger number of transistors such circuits can integrate more functionality, resulting in better performance. However, although the number of transistors is increasing, efficient usage of the available transistors is important, as inefficiencies lead to higher energy consumption and to lower performance. Additionally, the design complexity grows with the number of transistors, which may lead to more design errors as it is hard to generate all possible test patterns and check the response of the circuit to these patterns. Therefore, in order to keep the circuits testable as well as efficient, circuits are often designed as multi-processor circuits consisting of multiple processor cores. Such an architecture is also called a Multi-processor System-on-Chip (MPSoC) [152, 153]. If all processors in the circuit are identical, the MPSoC is called *homogeneous*. Otherwise, such an architecture is called *heterogeneous*.

Parts of this chapter have been presented at the International Conference on Engineering of Reconfigurable Systems & Algorithms (ERSA'06) [151], at the Dynamically Reconfigurable Architectures workshop [152], at the Tenth International Workshop on System-Level Interconnect Prediction (SLIP 2008) [153] and was published in the EURASIP Journal on Embedded Systems [154]

The functionality of an application is divided over the cores such that each core is responsible for a part of the overall functionality. Intermediate results calculated by one of the cores have to be synchronized with and communicated to another core for further processing. Hence, the cores have to be connected to a communication medium.

2.1 State of the Art

Conventional MPSoC architectures have been built using a shared bus to connect multiple devices in the system, which can be either Input/Output (IO) devices or processor cores. Via the shared bus, any device can transmit data to any other device. A device that is allowed to initiate data transfers is called a master, and a device that is capable of responding to such a data transfer is called a slave. Such communication schemes work efficiently as long as only a few devices are connected to the bus, because a bus has a fixed bandwidth that is shared for all connections made.

The shared bus enables communication between any two connected cores. During a certain time slot, its wires are reserved for a transaction between two cores. For that time slot, because multiple cores may want to write to the bus simultaneously, the bus arbiter determines which cores can write to the bus such that no collision occurs. A time slot can be requested by any core that is implemented as a bus master. Hence, when two masters request for a time slot at the same time, at least one of both is halted temporarily since they cannot use the bus during the same time interval. Halting can be avoided by adding a shared bus for each master and connecting all slaves to multiple shared buses. For example, the Advanced Microcontroller Bus Architecture (AMBA) interconnect [8] includes a multi-layer Advanced High-performance Bus (AHB) [9].

Ultimately, there is a direct connection between each of the processors in the system. However, such topology would implicate very large costs since it requires many (possibly long) wires. By having a connection between one processor and its direct neighbors, only a small number of connections need to be made. Moreover, in such an approach multiple communications can run in parallel leading to a high aggregated bandwidth. Such an interconnection medium, providing a balance between flexibility in connections, total aggregated bandwidth and chip area, is called a Network-on-Chip (NoC). An example MPSoC consisting of different types of cores and IO devices interconnected via a NoC is shown in Figure 2.1. The IO devices are used for off-chip communication.

In such a structured topology, a single processing unit is called a processing tile (see Figure 2.2). It consists of one processor, which is called a Tile Processor (TP), and one interface to the NoC, which is called a Network Interface (NI). The TP has a small Local Memory (LM) at its disposal for storing intermediate results.

Two tiles are called neighbors if their routers are connected via a direct link. For communication between two tiles that are not neighbors, a route needs to be created along which the data is communicated. Therefore, the data is sent by the initiating TP via its NI to its local NoC router, which routes the data to one of its neighbors. The neighboring router can either route the data to its TP or forward it to another

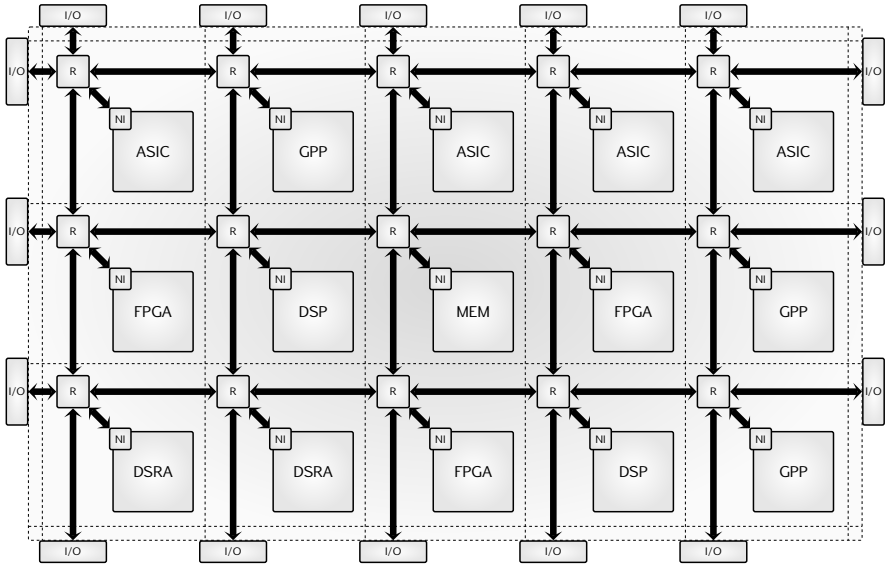


Figure 2.1 – MPSoC example with several different types of tiles

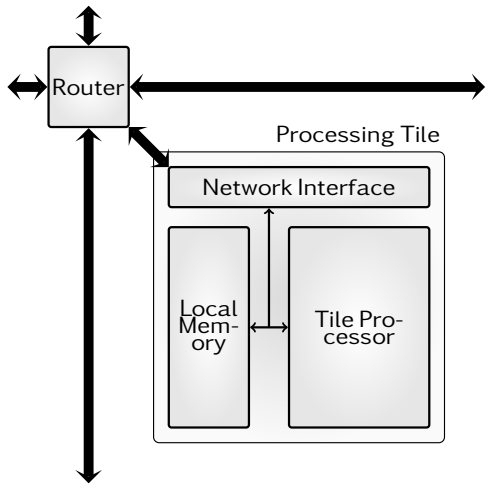


Figure 2.2 – Processing Tile structure showing network interface, tile processor and local memory and its connection to the NoC

neighboring router. The NI enables communication with other cores in the MPSoC. It translates the TP interface protocol to the NoC protocol and vice versa.

2.1.1 Tile Processors

In the example MPSoC shown in Figure 2.1, several types of tile processors can be identified, for example Application Specific Integrated Circuits (ASICs), General Purpose Processors (GPPs), Digital Signal Processors (DSPs), fine-grained reconfigurable architectures like Field Programmable Gate Arrays (FPGAs), coarse-grained Domain Specific Reconfigurable Architectures (DSRAs) and memory tiles (indicated by MEM)[1]. Each tile processor type has a specific instruction set. Furthermore, a tile processor has a certain small local memory that is available for temporary storage.

In our application domain, typical algorithms that are executed by tiled architectures are Digital Signal Processing (DSP) algorithms like Fast Fourier Transform (FFT), Discrete Cosine Transform (DCT) and Finite Impulse Response (FIR) filters. Such applications have to be partitioned in processes that can be executed by tile processors. The input, output and intermediate results for such a process are stored within the local memory of the tile. On an MPSoC level, this can be seen as a distributed memory (with a typical storage size in the range of 10 kB to 100 kB per tile).

2.1.1.1 Memory tile

A memory tile is used by other tile processors for temporal storage of their data. Therefore, it contains a relatively large local memory and a memory controller that connects the memory to the NoC as if it were a processor. With this controller, the underlying memory architecture is hidden.

2.1.1.2 General Purpose Processor

Some applications require a very generic processor architecture, since they contain many different kernels which differ so much that no optimized architecture can be designed for it. For such applications, a GPP is used. It supports a wide range of instructions that can be executed in an arbitrary order. Such flexibility comes at the price of reduced performance or increased silicon size. Typically, a GPP is based on a combined instruction and data memory and a data path that loads instructions and data from the memory, which is also known as a Von Neumann machine [10].

In Multi-Processor Systems-on-Chip, often used GPP architectures are based on a Reduced Instruction Set Computer (RISC) architecture (for example, the Advanced RISC Machine (ARM) family, IBM's PowerPC and Sun's SPARC). An extensive overview of microprocessors is presented in[11].

2.1.1.3 Digital Signal Processor

The DSP is a GPP that has been optimized for DSP applications. It is based on complex instructions that may specify multiple operations in parallel. For example, vector operations may be used to execute the same operation on multiple operands

[1] Although other processor types could be added to this list, we consider these types as the relevant processors types for embedded multi-processor architectures.

simultaneously. Composite operations, like the Multiply Accumulate (MAC), decrease the number of memory operations as intermediate values can be directly stored in local register files. The Harvard architecture [12] was designed to improve the processor performance for DSP applications. In contrast to the Von Neumann architecture, where instructions and data are stored in the same memory, the Harvard architecture uses separate memories for the storage of instructions and data. The advantage of this separation is an increased memory bandwidth, as the instruction fetch can be done simultaneously with the memory read/write operations. Furthermore, since instructions are read from a separate memory, they can be stored in a Read-only Memory (ROM), which can be implemented at relatively low costs and a high performance in terms of access latency and bandwidth.

By adopting the Harvard architecture and increasing the number of data memories, the bandwidth offered by the memory or the IO controller is increased such that the instructions and data can be fetched from the instruction memory simultaneously. DSP applications typically have strict real-time constraints. For example, if the decoding of an audio stream is not executed fast enough, the played audio may contain clicks and noise. Typically, for GPPs and DSPs it is difficult or impossible to give real-time guarantees, as they are usually not able of satisfying (guaranteed) real-time constraints. An overview of typically used DSP architectures is given in [13].

2.1.1.4 Application Specific Integrated Circuit

The most efficient execution of an algorithm can be obtained by performing the entire algorithm with one large hardware accelerator block [14, 15]. In this case, the algorithm is directly synthesized to transistors and etched on silicon. The main advantage of this approach is its efficiency, as it requires a minimum of silicon area and has a very low energy budget. However, this comes at the costs of inflexibility, since later modifications cannot be made anymore.

The manufacturing costs of a single ASIC core are mainly determined by the preparation before manufacturing. Once the design of the hardware accelerator is finished, masks are created for the lithography process which is used for etching silicon. The design of such a mask is very expensive, but once the mask has been made, it can be used for the production of many devices.

The only solution for making a flexible architecture based on ASIC cores, is by combining multiple chips on a Printed Circuit Board (PCB) and having a controller that activates or deactivates individual chips. However, since this requires a complex design consisting of multiple chips, such a design is expensive and inefficient.

2.1.1.5 Fine-grained reconfigurable: FPGA

The FPGA is a bit-level reconfigurable architecture, consisting of a large number of small logic blocks connected via a large number of wires [16]. The logic blocks contain a Lookup Table (LUT) and some memory elements, which can be connected to other logic blocks via the on-chip interconnect. This interconnect consists of wires of different lengths and small router elements that are used to connect these wires. The functionality can be altered such that an FPGA is capable of running many different

applications [17]. However, this comes at the cost of configuration, as each logic block and interconnect router needs to be updated. Typically, this requires configuration files of several (up to tens) of megabytes. Such large configuration streams cannot be put in the FPGA instantly; a full reconfiguration may take up to several seconds. For time-critical applications, this may be too slow. Another disadvantage of the large configuration space is the relatively large physical overhead required to configure each logic block. Therefore, an FPGA device is large and consequently its energy consumption is considerable.

The default programming model for an FPGA is a very low level, as the developer has to describe all individual combinatorial circuits and memory elements. Example programming models include VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL) [18], Verilog [19] and SystemC [20]. This makes the design very complex and fault sensitive [21]. Furthermore, fully testing an FPGA application can be very difficult [22, 23].

2.1.1.6 Coarse-grained reconfigurable: DSRA

As a trade-off between energy-efficiency and flexibility, coarse-grained reconfigurable architectures turn out to be good alternatives [24]. Coarse-grained reconfigurable architectures provide the flexibility needed for a lot of DSP algorithms, while the energy consumption is relatively small compared to the other architectures mentioned (except for the ASIC which has a low energy budget, but is limited to a fixed functionality). Examples of DSRA are the Montium TP [25], the PACT Extreme Processing Platform (XPP) [26], the Silicon Hive AVISPA reconfigurable accelerator [27] and the Pleiades architecture proposed by the University of Berkeley [28]. For a detailed overview of coarse grained reconfigurable architectures, we refer to [29].

In this section a short introduction is given to the Montium TP, as this processor is used throughout this thesis in examples and case studies.

Example DSRA: Montium TP The Montium TP is a coarse-grained reconfigurable tile processor that was developed in the Chameleon project [25, 29, 30]. The hardware architecture and support tooling are now further developed by Recore Systems [31]. Within the core, three main regions can be identified: the Processing Part Array (PPA), a control part consisting of a sequencer and a configurable sub-system consisting of several configurable decoders and instruction registers. Figure 2.3 shows a Montium tile, consisting of the Montium TP (shown in the upper part) and a NI that connects it to the NoC (shown in the lower part).

To enable energy-efficient processing, the Montium TP was designed such that the program execution overhead is as small as possible. Such efficiency can be obtained by reducing the signal activity. Therefore, the datapath is configured such that during several clock cycles only a limited number of control signals changes polarity, by switching from a logical value 0 to a logical value 1 or vice versa. Hence, the energy consumption is mainly caused by data transport and Arithmetic Logic Unit (ALU) activity.

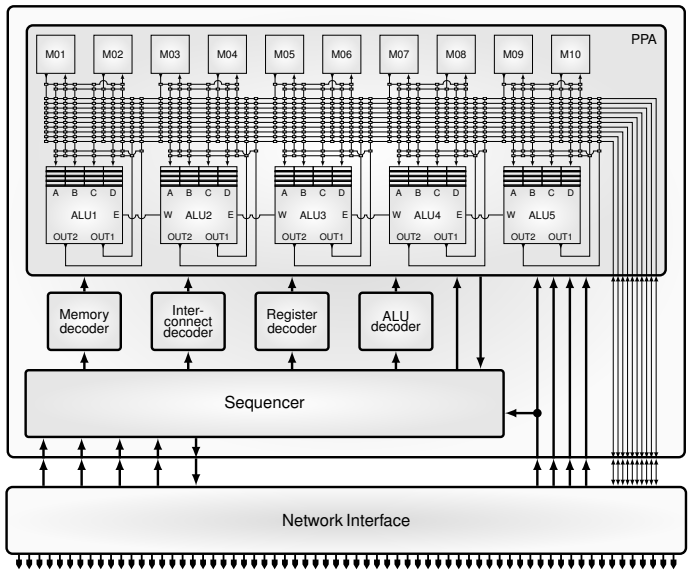


Figure 2.3 – Montium TP (modified from [153])

Processing Part Array The processing core of the Montium TP consists of an array of Processing Parts (PPs), each containing an ALU, a local interconnect and two memory units. A memory unit contains a 1024 × 16-bit Static Random Access Memory (SRAM) and an Address Generation Unit (AGU) that can be configured to generate address patterns for its SRAM. This reduces the load on the ALUs, since they are not bothered with the load caused by address calculation. However, for irregular memory access patterns, the ALUs can be employed to calculate addresses. These calculated addresses can then be loaded into the AGU, which will execute the memory access operation. Only 10 bits out of a 16-bit word are used to address one of the 1024 memory positions. The calculated address can be used in two ways: the *integer lookup* uses the lowest 10 bits, while the *fixed-point lookup* uses the highest 10 bits of the 16-bit word. The SRAM memories are single ported, so either one write operation or one read operation can be done at a time. The ALUs are connected to these memories via ten Global Buses (GBs) which can also be accessed by the NI. In total, each ALU contains 4 register banks (labeled A to D) which can be read simultaneously, and each ALU can receive an intermediate value from its right neighbor ALU via the east-west connections. Using these 5 inputs, multiple operations can be executed simultaneously and from each ALU at most 3 results can be sent to the west output and both outputs connected to the interconnect respectively. Figure 2.4 shows the internal structure of one ALU.

The upper part, level 1, is used for applying bitwise and logic operations like *and*, *or* and *shift*. Additionally, in this level simple arithmetic operations can be done like *add*, *sub* and *neg* and saturated equivalents of these operations, which are useful for DSP applications. Four function units are used for executing the operations

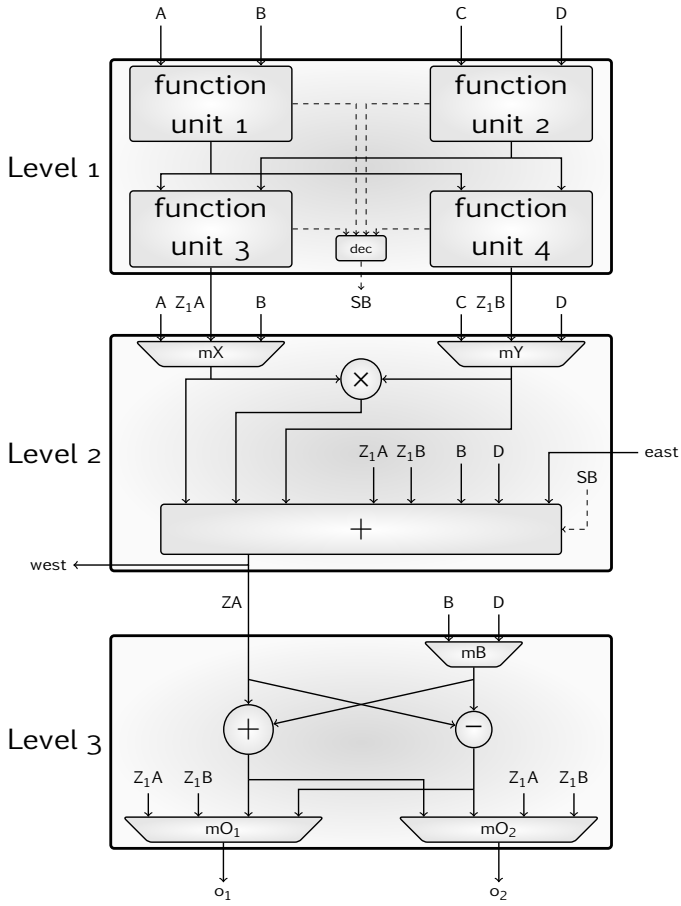


Figure 2.4 – Structure of one Montium ALU

mentioned. Each function unit generates *status flags* to indicate the occurrence of overflow, a negative result or whether its result equals zero. The function units can be used to execute up to four operations in parallel in level 1 of the ALU. Obviously, for this large number of operations a lot of operands need to be available. Four register banks (A to D) can be used as inputs for the ALU.

In the second level a MAC operation can be executed. For the multiplication, the input operands are selected using multiplexers mX and mY. These multiplexers can access either the outputs of the first level, named Z_1A and Z_1B, or the register files A to D. Next, the addition is performed on either the result of the multiplication, mX or mY, and the register files B and D, level 1 outputs and the east input. The east input is connected to the right neighboring ALU to allow a chain of operations over multiple ALUs. For the selection of the right operand, the status flags generated by the four function units can be used. A small encoder takes the four status flags and

creates a *status bit* (SB), which can be used to dynamically select the right operand for the adder. This operand is selected from inputs B and D or from Z_1A and Z_1B. Since the operand selection is done within the same clock cycle as the rest of the ALU operations, it enables an efficient single cycle conditional operation. The result of the addition is made available for the left neighboring ALU via the west signal and can be used in the third level in the ALU via the ZA signal.

In the third level of the Montium TP's ALU, a butterfly operation can be done. This operation is typically used in FFTs to enable an efficient implementation by using symmetry in the operations. More detail on this is given in section 4.1.1. Finally, up to two results of the ALU operation can be selected via the output multiplexers mO_1 and mO_2. Since the ALU is not pipelined, the entire operation from inputs A, B, C and D to outputs o_1 and o_2 is done within one clock cycle. Moreover, because it supports only single-cycle instructions, the program flow is fully deterministic. Almost all arithmetic operations in the ALU can be executed in either *integer* modus (operating on the 16 rightmost bits) or in 1.15 *fixed point* modus (the leftmost bit is used as sign bit whereas the other bits contain the fixed point fraction). In order to avoid overflow, the intermediate values can be saturated.

Control The control part consists of a sequencer, which contains an instruction memory in which the program is configured. Therefore, the instructions do not need to be fetched from the main memory as they are already present within the sequencer. The sequencer could be considered a state machine that defines the current and next system state by generating output signals, which are used for controlling the configuration part.

Configuration The configuration part consists of a set of decoders, in which parts of the instructions are stored. Figure 2.3 shows the 4 decoders: a memory decoder, which contains the instructions required to control the memory units, an interconnect decoder that is used for controlling the Global Buses between memories and ALUs, a register decoder that is used for controlling the local registers and finally, an ALU decoder which contains the control signals for the ALUs.

By using 3 stages of instruction decoding, the instruction size and therefore its memory footprint, is minimized. Figure 2.5 shows an overview of the compression mechanism. First, the sequencer selects the current instruction using the Program Counter (PC). The instruction consists of several fields, each of which is used for addressing one of the four decoders: ALU decoder, memory decoder, register decoder and interconnect decoder. From the selected instruction, the ALU decoder instruction is selected (`dec[4]` in the picture) and used for indexing the ALU decoder (at position 4, in the figure). Similarly, the decoder contains several instruction fields, one for each ALU, that are used to address the *configuration registers* for each of the ALUs. In the example, the decoder addresses the instruction for ALU3 that is stored in `cr[1]`, which is the configuration register instruction 1. This instruction contains the control signals that are used for controlling the data path elements. For example, it selects which register file inputs are used for the ALU (`regA[1]` and `regC[2]`), it selects the ALU operation (*add*) and it selects to which ALU output the result is written (`out2`).

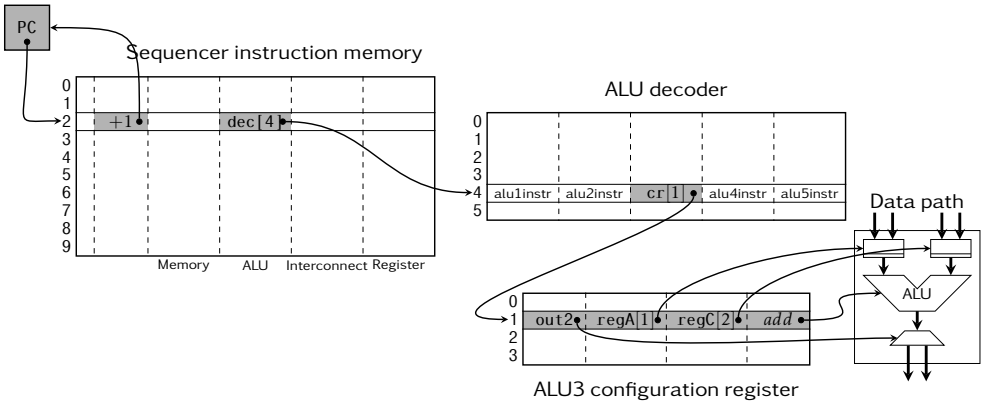


Figure 2.5 – 3-stage encoded Montium instruction showing the instruction decoding for ALU 3

Table 2.1 – Characteristics of the Montium TP

Word size	16 bits
Area	1.8 mm^2
Memory size	10 × 2 kB
Clock frequency	100 MHz
CMOS Technology	0.13 µm TSMC
Voltage	1.2 V
Power	0.25 – 0.6 mW/MHz

By modifying either the contents of the sequencer instruction memory, the decoder memory or the configuration registers, the Montium TP can be reconfigured. Moreover, since the 3-stage encoded instructions can be stored in small memories, reconfiguration can be done quickly. The Montium TP's total configuration address space consists of about 2.8 kB [32].

Table 2.1 summarizes the characteristics of the Montium TP. It has a small area footprint (1.8 mm^2) and a relatively low clock frequency (100 MHz), such that it has low power demands for executing the program.

2.1.2 Network-on-Chip

As discussed before, the cores in an MPSoC are interconnected by a NoC. Many different on-chip networks have been proposed [33–41]. Usually they are based on the same principles but different design choices lead to small differences. The next sections describe typical basic properties of Networks-on-Chip: the topology, communication protocol, routing method and types of communication. These are needed to understand the techniques presented in section 2.2. They stem from the NoC used in the Annabelle chip (see section 2.2.3). An extensive overview of NoC

related techniques is presented in [42].

2.1.2.1 Topology

The NoC consists of a set of routers and a set of links connecting the routers. The way in which routers are connected is determined by the NoC's topology. Examples of topologies are presented in section 2.2 of [41]: for example, the mesh structure (the routers are positioned in a rectangular grid, with a connection between each two neighboring routers on a row or a column) or a torus (comparable to the mesh, but with the leftmost router of each row connected to the rightmost router of that row and the upper router in each column connected to the lower router in that column). For this thesis a regular mesh structure is assumed.

2.1.2.2 Network protocol

The routers in the NoC are connected via *links*, organized in a regular structure as explained in the previous section. A link between two routers consists of one or multiple unidirectional physical channels (called *lanes*), for example as shown in Figure 2.6.

Definition 1. A link is a physical connection between two NoC routers. It consists of one or multiple lanes via which data can be transmitted.

Multiple lanes can be used simultaneously. The bandwidth provided by a single lane is determined by the clock frequency of the routers, the number of parallel wires and the length of the wires. For a more fine-grained bandwidth control, the lane can be shared in time by using one or few Virtual Channels (VCs), such that a single physical channel can be used for multiple logical channels simultaneously. Using an arbitration scheme (for example, Time Divison Multiple Access (TDMA) or Round Robin), the lane is reserved for one VC at a time such that there will be no contentions. Thus, the VC has a guaranteed minimum bandwidth and a guaranteed maximum latency, which is independent of traffic via other VCs.

Definition 2. A lane is a part of a link, which can be used independently from other lanes. It provides flow control, by acknowledging data transmissions. Its bandwidth is shared in time over one or multiple VCs, which form logical channels.

A connection between any source and any destination in the MPSoC can be made by mapping a logical channel on a sequence of VCs via one or multiple routers. The source can write into the channel without any knowledge about the mapping on VCs and routers, but with the guarantee for a certain throughput and latency. Data written into the channel is transported in-order, such that it can be read in the same order by the destination.

The minimum size of a data sample written into a channel is called a *flit* [38]. Figure 2.7 depicts the structure of one flit, as used in our Networks-on-Chip [154]. It consists of a 2-bit type field (FT), which is used to provide control information, along with a 16-bit data field. The four flit types and their encoding are shown in Table 2.2.

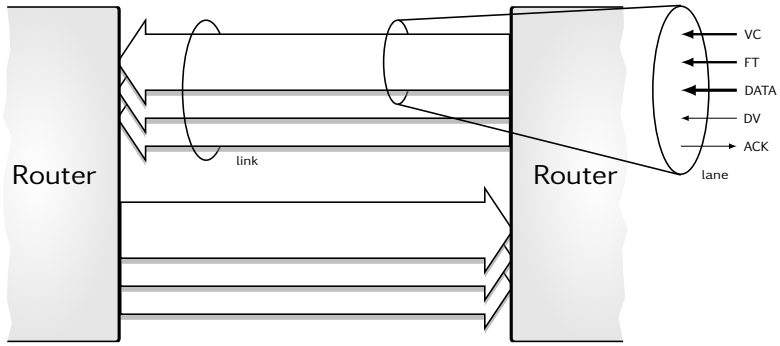


Figure 2.6 – NoC link structure (modified from [41])

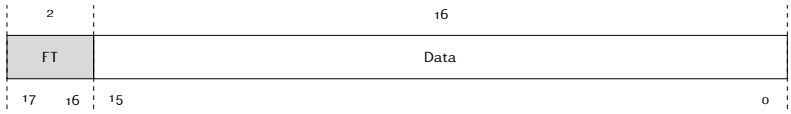


Figure 2.7 – Generic flit structure

Table 2.2 – Flit type encoding

Encoding	Flit type	Symbol
00	Data	*D*
01	Header	*H*
10	Tail	*T*
11	Command	*C*

A sequence of flits forms a *packet*. Typically, a packet is started with a *H* flit and is terminated by a *T* flit.

As depicted in Figure 2.6, a lane consists of the signals for transmitting the flit data (using the FT and Data signals) via a specified VC, and uses 2 signals for *flow control*. Flow control between routers is used to prevent buffer overflow. The transmitting router asserts a *data valid* signal (DV) to indicate that the FT, Data and VC signals are valid. The receiving router replies with an *acknowledgement* signal (ACK) to indicate that it is ready to accept data (see Figure 2.6).

2.1.2.3 Switching techniques

There are two main approaches for routing data through a NoC [42]: by distributing the routing information to each of the routers via a separate circuit (referred as circuit-switched) or by embedding the routing information within the data streams (referred as packet-switched). The two approaches lead to circuits with different characteristics.

Circuit-switching Generally speaking, circuit-switched networks consume a relatively small amount of energy since the routers mainly consist of switches and do not have to inspect the received packets [41, 43, 44]. Therefore, data arriving at a certain input channel can always directly be sent to a pre-determined output channel. When the routing has to be changed, the NoC has to be (partially) reconfigured. For this reason, a circuit-switched NoC is more efficient for high-throughput streams that are fixed for a relatively long time.

Setting up a connection between a source and a destination in a circuit-switched network is done as follows. First a route from the source to the destination via one or multiple routers is calculated. For each router in this route, the appropriate VC of the corresponding input port and VC of the corresponding output port are connected by writing a simple routing table inside the router. This routing table can only be accessed via a dedicated configuration interface, to keep the routing control and data channels independent. If all routers have been configured, the source can safely write data into the channel and the connection will remain open until the routing tables are reconfigured again.

Packet-switching In packet-switched networks, the routers extract the routing information from the packets at runtime. This header extraction process introduces a higher energy consumption for the routers. However, they do not need to be reconfigured when the communication pattern changes. Therefore, they are more flexible compared to a circuit-switched NoC. In the packet-switched NoC presented in [45], the *H* flits are used to create a connection for all following *C* and *D* flits, while a *T* flit tears down the connection (see Table 2.2).

Since the header of a packet defines how a router should forward the remainder of the packet from a VC of an input port to a VC of the output port, the connection can only be made as soon as the header has been received. A tail flit, indicating the end of a packet, also indicates that the routing information inside the router can be erased and the connection is destroyed. This flit is then forwarded to the next router such that the rest of the channel can be closed.

2.1.2.4 Traffic classes

The application domain to be supported by the NoC is a key issue when designing the NoC. There can be large differences between requirements of the various applications within the application domain. The large differences are supported by introducing different traffic classes. For example, an application with strict execution deadlines requires deterministic behavior of the NoC, such that guarantees can be given for the latency and throughput of the communication streams. This behavior is provided by the Guaranteed Throughput (GT) traffic class. Using GT traffic, each of the communication streams is given a fixed bandwidth. The given bandwidth guarantees the throughput of the stream, while the schedule provides a guarantee on the latency of the stream. Since both are deterministic, their combined behavior can be analyzed deterministically.

Another type of traffic is the Best Effort (BE) traffic class. This traffic class typically services applications with less strictly defined behavior (for example, where

bandwidth and latency vary) or have less predictable behavior (for example, control messages sent at irregular time intervals to reconfigure parts of the MPSoC).

The circuit-switched NoC presented in [43] supports GT traffic only, while the packet-switched network presented in [45] supports GT as well as BE traffic. However, the packet-switched network is more suitable for applications with dynamically changing communication patterns [41].

2.1.3 Network Interface

The NI is the medium between the NoC and the tile processor. The most important design decisions depend on the requirements enforced by both the NoC and the tile processor. Typical tasks that are performed by the NI are clock synchronization, communication protocol conversion, buffering and tile processor control.

A vast amount of research has been done on NIs for off-chip communications [46, 47] (for example in Local Area Network (LAN) connections). For on-chip communication, much effort has been spent on bus interfaces and bridges. However, for NoC interfaces, there has been done limited research. The Æthereal NoC is based on time slot allocation at design-time, such that the network routers can be reduced to small switches with minimal buffering and guarantees can be given for data transmitted by a processor via the NoC to another processor. Allocated time slots are assigned to the NI, which consists of a buffer for each possible connection to another processor [48]. Hence, the NI is responsible for transmitting data during the assigned time slots. As a result, communication between two processors is done via two deterministic NIs and a deterministic NoC. Dally and Towles [34] describe a NI that is integrated in the processor. Only the physical interface protocol is implemented in hardware; higher level protocols have to be implemented in software. Liang [33] presented a combined NI and router. A combined synchronous/asynchronous NI for both on-chip and off-chip communication is presented in [49].

A common property of most network interfaces is their NoC abstraction layer that enables global addressing from the perspective of the local tile. Having a notion of a global address space means that each tile processor has to know the entire system state to ensure that a certain address can be read or written without destroying another tile processor's state. Advanced memory arbitration and cache coherency techniques may have to be used to avoid such problems. Hence, the concept of a global address space does not scale well with large numbers of processing tiles. We propose to use a stream based programming model (presented in chapter 3) where addressing is implicit within the data streams. Therefore, our NI does not provide an address conversion but is only accessed via streams. At first sight this sounds like *message passing* [50, 51], but our model is not based on explicit read and write operations that are typically used in message passing. However, on top of our stream model a global shared address space could be used, but we assume that memory coherency is done in software.

2.2 Hydra Network Interface

In the previous section we introduced a minimal set of basic building blocks required to create an MPSoC. As mentioned, not much effort is spent on research on NIs as a separate basic building block, due to the common integration of the NI in either the tile processor or the network router. In this section we present the Hydra NI in three steps. First, an overview is given of the requirements of the NI based on its environment (tile processor and router). Second, the requirements are used to describe the design structure. Finally, the realization and performance results of the design is presented.

2.2.1 Requirements

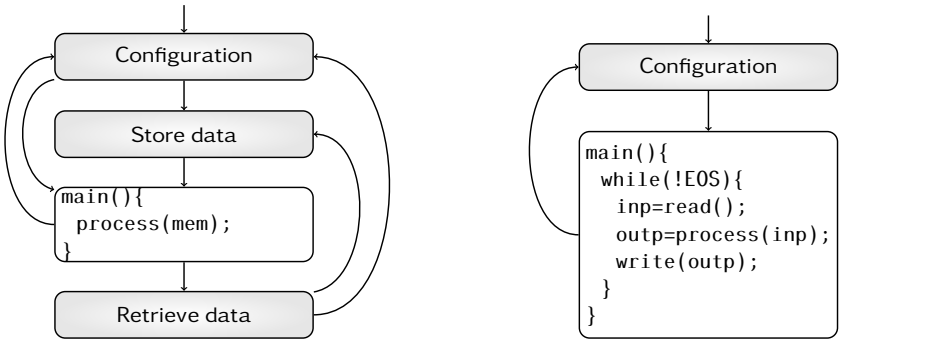
Due to the design choices made in the tile processor and the NoC, the requirements for the NI are well defined. In this section, the design considerations are introduced to support the realization presented in section 2.2.2, where each of these requirements will be evaluated in section 2.2.3.

2.2.1.1 Operation mode

A typical application execution by the tile processor may consist of the following subsequent steps. First, the instruction code is configured into the processor's configuration memory. Next, input data for the operation is transferred to the processor. Then, the processor executes the configured program and the result is transferred, after which the processor can be reconfigured or new input data can be transferred. We have two mechanisms for transferring data between the NoC and the tile processor: block-mode operation and streaming-mode operation, illustrated in figures 2.8(a) and 2.8(b). Two main differences can be identified between both modes: the initiator of a data transfer differs per mode and the processing efficiency due to concurrency in computation and communication varies.

Some processes require all the input data to be present in the local memories before the execution can be started. This operation mode is called *block-mode*. Typically, a block-mode operation is done in three stages, as shown in Figure 2.8(a): (a) the input data is loaded into the local memories via a Direct Memory Access (DMA) transaction, (b) the process is executed and (c) the output data is fetched from the local memories with another DMA transaction. In this operation mode the network interface acts as a master for its tile processor. A second, external tile processor operates as initiator of the input and output DMA transactions. For each of the stages, the initiator configures the NI such that the NI can access the local memory within its processing tile. To ensure that the transactions are completed successfully, the initiator and the NI need to synchronize such that a transition between NI states can be made without loss of data. To make sure the processing does not start before the input data has been fully loaded, the NI halts the tile processor during the data transfers.

In block-mode operation tiles are event driven, which means that the process execution is started when an external source has prepared the input samples and



(a) Block-mode operation

(b) Streaming-mode operation. EOS denotes End of Stream.

Figure 2.8 – State transition diagrams for block-mode and streaming-mode operation

gives a start command. As long as no command is received, the process is halted. Using this operation mode, communication and computation are fully separated. This may be useful in case the input data is provided on an irregular basis, for example if the data was transmitted in a different order than expected by the processing tile or if the input data size varies per packet.

In contrast with the block-mode operated tile processor, some tile processors support reading input data and writing output data during program execution. This operation mode is called *streaming-mode* and its state transition diagram is shown in Figure 2.8(b). Since the tile processor is in control of input and output communication, the NI acts as a slave of the tile processor. Typically, during the execution of a streaming-mode process, connections for the input data and output data remain open. Only at the time a tile processor is reconfigured, its outstanding connections are closed. Hence, after opening the connection (see section 2.1.2.3) the raw data stream can be written to the output channels without the need for packetization. Similarly, the receiving tile processor reading that channel does not need to de-packetize the stream and can use it immediately. This is an advantage for both the sender as well as for the receiver, because it saves the packetization costs for both. Since packetization is not done in streaming-mode, the NI can immediately forward the data stream without introducing a packetization latency while in block-mode the stream is slightly delayed due to the packetization.

In streaming-mode operation the system is data driven, which means that a process on a tile is started as soon as enough samples are available. This may be before the last sample has been received, providing a parallel execution of communication and computation. This means that in general for streaming-mode communication less buffer space is required. However, for streaming-mode processes the usage and order of input data has to be known in advance. This implies that such a process needs to know what to do with the next input data: a possible reordering may have to be done before the data can be used. Clearly, this also holds for the output data.

Whether block-mode or streaming-mode is used, is determined by the application

programmer and strongly depends on the characteristics of the application process. When the application operates in block-mode, no computation and communication occurs at the same time. This eases the implementation of individual processes, but gives some overhead when composing an application out of individual processes. For streaming-mode applications the programmer has to carefully plan how and when the communication takes place. This can be hard, especially when the ordering of samples within a data stream differs from the expected order such that reordering is required, or when the ordering depends on one or more parameters.

2.2.1.2 Throughput and latency

Streaming applications with explicit *throughput* and *latency* requirements form the application domain for our intended MPSoC platform. All involved components within the MPSoC are responsible for the overall performance of the application at run-time. Since the NI is used as a clock synchronization and communication protocol conversion, its throughput and latency may be critical for the performance of a communication stream. Therefore, the throughput of the NI should be optimized such that either the throughput of the tile processor or the throughput of the NoC is the limiting factor. In other words, the NI is not the critical factor.

When assuming the NoC supports guaranteed throughput traffic, the NoC is typically not a limitation for the real-time guarantees that are required for the application. The latency caused by communication via the NI needs to have a strict upper bound and should be relatively small compared to the communication time in order to satisfy the real-time guarantees of the overall system. Data received from a tile processor should be packaged and transmitted as fast as possible by the NI. Therefore, only limited buffering can be allowed inside the NI.

2.2.1.3 Clocking regime

A tiled architecture provides the possibility to operate individual processors at different clock speeds. Such architecture is said to have a Globally Asynchronous Locally Synchronous (GALS) clocking strategy, because the processors are locally synchronous but may operate in an asynchronous fashion at global level [52].

A signal that crosses the boundary of two clock domains should be synchronized before it is read. Otherwise, reading the signal may result in corrupted data. In [53], the following definitions are given:

Definition 3 (Asynchronous)**.** A signal is said to be *asynchronous* when it is not related to any local clock.

Definition 4 (Synchronous)**.** A signal is called *synchronous* if it is updated with the same frequency as the local clock and it is in phase with the clock.

The MPSoC presented in [154] uses a dedicated clock domain for each tile processor, derived from one master clock. Each of the derived clocks is exactly in phase with the master clock, but has a lower clock frequency. The master clock's frequency f_{NoC}

is 100 MHz and the frequency of a derived clock f_{TP} is derived from this frequency using a clock divider that uses a parameter n:

$$f_{\mathrm{TP}}(n) = \frac{f_{\mathrm{NoC}}}{2^n}, \quad \text{where} \quad n \in \{0,1,2,3,4\} \tag{2.1}$$

2.2.1.4 Energy-efficiency

Low-power design is required for efficient architectures [54–56]. A processor's total power consumption can be decomposed in static power and dynamic power, shown in Equation 2.2.

$$P = P_s + P_d \tag{2.2}$$

The static part P_s of the power consumption is mainly caused by leakage of transistors. With each decrease in technology feature size, the thickness of the insulation layer between source and drain of a transistor is decreased. Hence, more electrons are able to leak through this layer, causing loss of energy. The dynamic power component P_d is caused by charging and decharging the capacitances in a circuit and by short circuit currents, which happens if the logic values change polarity.

Equation 2.3 gives an approximation of the dynamic power consumption,

$$P_d = \alpha \cdot C \cdot V_{dd}^2 \cdot f \tag{2.3}$$

where α is the switching activity, C is the capacitance, V_{dd} is the supply voltage and f is the frequency.

In order to lower the power consumption, some of the parameters mentioned can be fine-tuned. The capacitance C in Equation 2.3 is technology dependent and thus cannot be modified dynamically. The clock net has the highest α of all parts of a synchronous digital design and the C is large since all synchronous components in a VLSI design are driven by the clock. Hence, the clock distribution net consumes a considerable part of the power. When running a processor at a lower clock frequency, it can also be run at a lower core voltage. Combined, this can give a significant reduction in the power consumption. Therefore, it is useful to slow down or shut down the tile clock whenever possible. This can be done by either using Dynamic Frequency Scaling (DFS), Dynamic Voltage Scaling (DVS) or by applying *clock-gating* or *power-gating*. These techniques are explained in the next paragraphs.

DFS can be used to change the clock generator's frequency dynamically. Depending on the required performance, the circuit can be operated at different speeds. The upper frequency is limited by the circuit's maximum delay. Therefore, the clock generator should generate a clock signal with a minimum period length equal to the maximum delay in order to guarantee correct signal behavior.

Usually, clock generators are based on Phase Locked Loops (PLLs) [57]. Such a clock generator consists of a phase detector that drives a Voltage-controlled oscillator (VCO). The output of the VCO is fed back to the phase detector, which compares the phase of a reference clock with the phase of the VCO output. In this way, the output frequency of the PLL locks to the reference frequency. Because of the limited loop bandwidth, a stepwise change of the reference frequency will not immediately yield

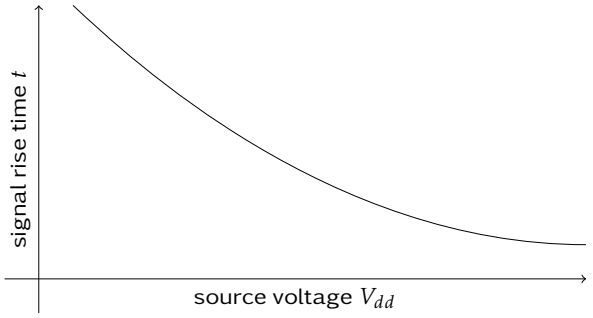


Figure 2.9 – Signal rise time t versus source voltage V_{dd}

the desired new clock signal. Therefore, frequent switches in the clock frequency should be avoided.

By increasing the source voltage, the rise and fall times of signals within a circuit become shorter, resulting in a faster circuit response. The delay of an inverter using a varying operating voltage can be described by an α power model, as given in Equation 2.4 [56, 58].

$$t = \frac{K V_{dd}}{\left(V_g - V_t\right)^h} \tag{2.4}$$

where t is the signal propagation delay, K is a proportionality constant, V_{dd} is the source voltage, V_g is the gate voltage, V_t is the threshold voltage and h a technology dependent parameter, which is typically in between $h = 1$ and $h = 2$ [58].

A reduction of the operating voltage V_{dd} gives a considerable improvement on the dynamic power consumption (see Equation 2.3), but increases the circuit delays at the same time. The relation between V_{dd} and t, as presented in Equation 2.4, is shown in Figure 2.9. To guarantee stable behavior of all individual signal updates in the IC, the total delay of all subsequent transistors between two registers should be smaller than one clock period. In case only parts of the IC are actually used, DVS could be used to lower the voltage such that those parts of the IC are still stable. Similarly, when the clock frequency goes down, the voltage may also be scaled down.

It is possible that, at a certain moment in time, the tile processor has finished its computation before new data samples have arrived. To save energy, the tile processor should be halted until these new samples arrive.

Clock-gating is a technique used to disable parts of the clock tree [59]. When all synchronous components driven by this clock tree are not going to be updated for a certain period, their clock can be disabled during that period. Since clock-gating can reduce the switching activity of parts of the clock tree, this may contribute significantly in reducing the dynamic energy consumption. However, the gating elements used for disabling the clock tree have to be added to the design before the IC is built. Hence, clock-gating can not be altered dynamically for any arbitrary

synchronous component. Though, the gating elements in the clock tree can be controlled at run-time.

A similar technique that is even more drastic is power-gating [54, 60]. With this technique, entire chip regions connected to a common power source (called a *power domain*) are powered down. Therefore, both static and dynamic power consumption can be reduced considerably. Within a power domain, all stateless components can be powered down without problems. Stateful components (for example, memory cells), however, will be reset upon power down, possibly resulting in a corrupted state. Internal circuits may be able to restore the state as soon as the power is back up, but during that period circuits externally to the powered down components may receive wrong input values. To avoid this, isolation cells are added that store the last output value of the circuit before power down, such that the external circuits remain operational. Furthermore, the state has to be retained by either copying it to a memory cell in a power domain that is not powered down, or by adding special retention memory elements that are capable of saving the internal state when they are powered down.

2.2.1.5 Communication to Computation ratio

As mentioned in section 2.2.1.1, depending on the operation mode, computation might overlap with communication. We introduce the following two definitions:

Definition 5 (Computation time)**.** The computation time T_{comp} is the time required to execute the process P.

Definition 6 (Communication time)**.** The communication time T_{comm} is the time needed for both the input data transfer C_{in} and the output data transfer C_{out}, while the processor is not processing.

Note that the computation time T_{comp} is related to the operating frequency of the tile processor: computation time $T_{\text{comp}} = cc_{\text{comp}} / f_{\text{TP}}$, where cc_{comp} denotes the number of clock cycles required for computation. The same holds for the communication time: $T_{\text{comm}} = cc_{\text{comm}} / f_{\text{TP}}$ where cc_{comm} denotes the number of clock cycles spent during communication.

The communication time can also be considered as the overhead due to communication. The ratio of the communication and computation times is called the *Communication to Computation ratio* (C/C):

$$C/C = \frac{T_{\text{comm}}}{T_{\text{comp}}} \tag{2.5}$$

Block-mode processes can only be started when all input data is received. This means there is no overlap in time between the communication and computation and, therefore, a change in the communication performance does not influence T_{comp}. However, delays caused during the communication (for example due to blocking in the network) do influence T_{comm}.

Streaming-mode processes behave slightly different. Since communication and computation are done in parallel as much as possible, T_{comm} is reduced significantly.

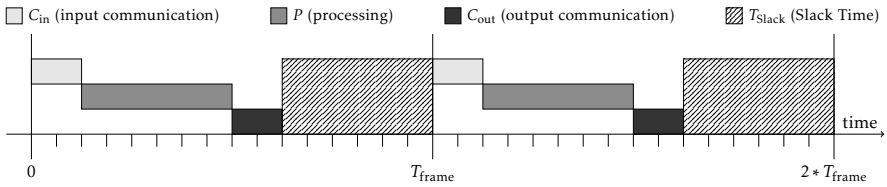


Figure 2.10 – Execution of a process, indicating computation, communication and slack time

Note that this also makes the total computation time T_{comp} dependent on delays caused by the communication. While T_{comp} is comparable for both modes, T_{comm} for block-mode processes is a lot bigger than for the streaming-mode processes. Therefore, the C/C ratio for a block-mode process is considerably higher than the C/C ratio for a streaming-mode version of this process. For example, consider a filter that operates on a sequence of samples. A block-mode implementation of that filter would require all input data to be available before the execution can be started. Obviously, such an implementation would have a C/C ratio that is much higher than a streaming-mode implementation of the same filter, which can start with the processing immediately upon reception of the first sample.

Assuming that, for the streaming application domain, T_{comm} and T_{comp} remain constant for a reasonable time and that no blocking occurs during the input and output data transfers, some estimates can be given about the required clock frequency f_{TP} of the tile processor. Assume a process has a periodic execution, where each execution has to be done within one time slot T_{frame} and the tile processor only runs process P. During the period within one time slot when the execution has already finished, the tile processor is idle. This time period is called *slack-time* and is defined as follows:

$$T_{slack} = T_{frame} - \left(T_{comp} + T_{comm}\right) \tag{2.6}$$

An example of two process executions is shown in Figure 2.10. More detail on the influence of both operating modes on the execution of a process is given in section 3.2.8.

Furthermore, we assume the clock is generated with a clock divider as specified in Equation 2.1. If $T_{slack} > \frac{T_{frame}}{2}$, the clock divider setting n can be increased by 1 without consequences. In fact, the clock divider parameter n used for the derivation of the tile processor clock f_{TP} can be safely chosen as follows:

$$n = \min\left(\left\lfloor \log_2 \frac{T_{frame}}{T_{comm} + T_{comp}} \right\rfloor, 4\right) \tag{2.7}$$

where the range $0 \leq n \leq 4$ is determined by the clock divider settings supported by the architecture as specified in Equation 2.1.

From this estimation we can conclude that a decrease of the C/C ratio enables a lower tile processor clock frequency, which contributes to a decrease of the energy consumption. However, this choice for n also influences T_{comm}. Obviously, the delay (in NoC clock cycles) increases when the tile processor clock is slowed down. This

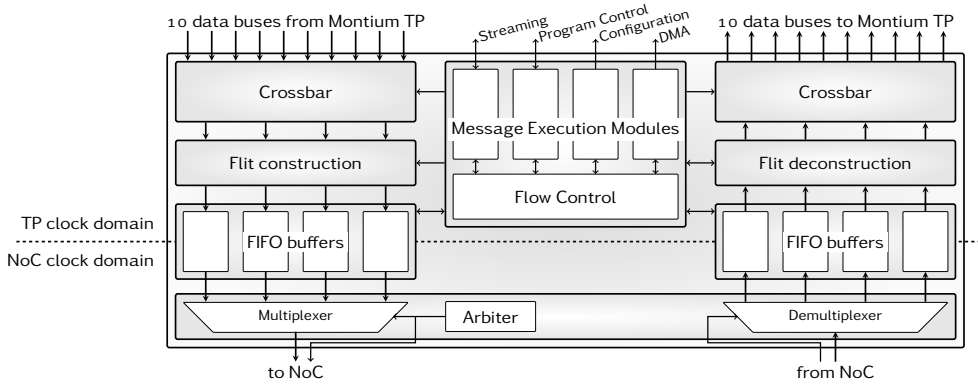


Figure 2.11 – Hydra network interface

may give communication problems for the tile processor that has to receive the output data. Therefore, n has to be chosen carefully and requires an application-level analysis. In chapter 4, the C/C ratio will be evaluated for different kernels.

2.2.2 Design

The NI used for connecting the Montium TP to the Networks-on-Chip is implemented by the Hydra [151][2]. Figure 2.11 shows the structural organization.

At the left side, the outgoing data path from the Montium TP to the NoC is found. The right hand side shows the incoming data path, from the NoC to the Montium TP. In between, the control unit is depicted. It controls both the incoming and outgoing data paths as well as the control signals to the Montium TP. Furthermore, Figure 2.11 shows the two clock domains in which the NI operates. The lower parts (channel multiplexing and a part of the buffering) are operated using the NoC clock frequency. The components located in the rest of the NI (flit (de-)construction, crossbars and control unit) operate at the Montium TP's clock frequency.

The Hydra was primarily designed for the circuit-switched NoC presented in [43]. This is a NoC operating at 500 MHz, consisting of 4 lanes with 4 VCs per lane. Such a NoC provides a huge bandwidth, which is not required for typical energy efficient applications. A scaled down version of the NoC router was also used in the Annabelle architecture [154], where it is operated at 100 MHz. As this equals the upper operation frequency of the Montium TP, a maximum of 4 channels is expected to be large enough to provide the bandwidth required by a typical Montium TP application. The 4 channels are implemented by 4 lanes, each providing one VC. The choice for a single VC was made because this decreases the NoC router complexity

[2] In Greek mythology, the Hydra was a mythological water creature with many heads. It guarded the lake of Lerna, which covered an entrance to the underworld. The heads symbolize the data parallelism of our network interface and the covered entrance can be seen as the entrance to the NoC, which is abstracted by the network interface

while the additional costs (larger area due to increased buffering and wires) are still acceptable.

The NI could also be used in combination with the packet-switched NoC presented in [45]. That NoC architecture uses a single lane per link, with 4 VCs per lane. In the next sections we will elaborate on the generic design of the Hydra NI, where the NoC instance of the Annabelle architecture presented in [154] is considered as reference architecture.

2.2.2.1 Data path

To ensure a high throughput, the Hydra consists of multiple parallel data paths. For each VC in the NoC, it has a separate FIFO buffer and crossbar connection such that it offers the full bandwidth as provided by the NoC. Furthermore, since input and output communication can be done in parallel by the Montium TP, the Hydra's data path consists of separate input and output parts that run in parallel.

Channel (de-)multiplexing Packets arriving from a NoC lane are split into separate VCs by the demultiplexer. The VC packets are forwarded to the FIFO buffers for storage and synchronization. This approach enables a scalable design, as all physical channels in the NoC connection are mapped on VCs such that the NI only has to scale with the number of VCs. Therefore, the internal behavior of the NI is independent of the NoC architecture chosen.

Buffering Per VC a dedicated buffer is used to store received packets. Such a buffer is implemented by First In, First Out (FIFO) [61, 62]. The buffers are located in two clock domains: the NoC side operates at the NoC frequency where the Montium TP side operates at the tile frequency.

Figure 2.12 shows the internal structure of a FIFO. It consists of a memory which can be simultaneously written by a source and read by a sink. The source uses a write clock clk_{w} and accompanying *write enable* signal *we* to store its value d_{in}. Similarly, the sink uses a read clock clk_{r} and a *read enable* signal *re* to fetch a value d_{out} from the memory. Internally, addresses are generated for the memory using read and write pointers. Furthermore, Figure 2.12 shows the *full* and *empty* signals, which are status signals that give feedback to the source and sink.

A write action to the FIFO can be done as long as it is not full and a read action from the FIFO can be done as long as the FIFO is not empty. The FIFO buffer is implemented by a dual-port memory with separate read and write pointers, behaving as a cyclic buffer. Equation 2.8 defines the functionality of the pointer logic blocks.

$$
\begin{aligned}
rp[t+1] &= \begin{cases} rp[t]+1 \mod \texttt{FIFO\_DEPTH} & \text{if } (re[t] \;\&\; \neg empty[t]) \\ rp[t] & \text{otherwise} \end{cases} \\
wp[t+1] &= \begin{cases} wp[t]+1 \mod \texttt{FIFO\_DEPTH} & \text{if } (we[t] \;\&\; \neg full[t]) \\ wp[t] & \text{otherwise} \end{cases}
\end{aligned} \tag{2.8}
$$

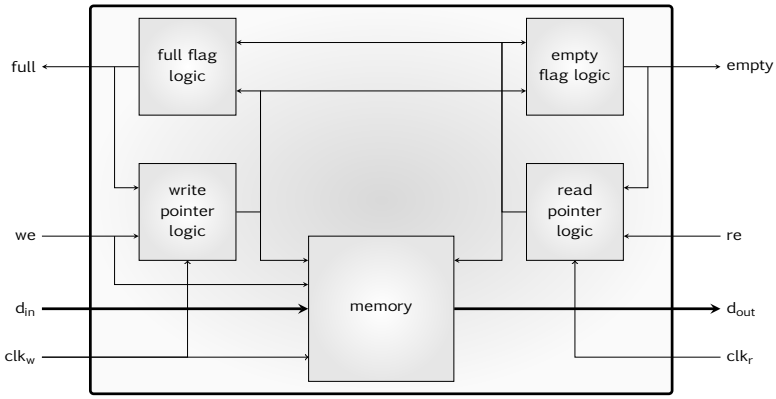


Figure 2.12 – Internal FIFO structure

As shown in Figure 2.12, the *full* and *empty* flags are derived from the pointers. A straight-forward implementation is:

$$
\begin{aligned}
empty[t] &= \begin{cases} \texttt{false} & \text{if } (rp[t] < wp[t]) \\ \texttt{true} & \text{otherwise} \end{cases} \\
full[t] &= \begin{cases} \texttt{false} & \text{if } (wp[t] < rp[t]) \\ \texttt{true} & \text{otherwise} \end{cases}
\end{aligned} \tag{2.9}
$$

However, the definitions in Equation 2.9 are ambiguous for the case where the pointers are equal. In that case, the flags would indicate both that the buffers are *full* and *empty* at the same time, which is impossible. Hence, the pointer definitions are extended with a phase ϕ. The internal pointers are incremented in size such that they count twice the buffer length:

$$
\begin{aligned}
rp'[t+1] &= \begin{cases} rp'[t] + 1 \bmod 2 * \texttt{FIFO\_DEPTH} & \text{if } (re[t] \ \& \ \neg empty[t]) \\ rp'[t] & \text{otherwise} \end{cases} \\
wp'[t+1] &= \begin{cases} wp'[t] + 1 \bmod 2 * \texttt{FIFO\_DEPTH} & \text{if } (we[t] \ \& \ \neg full[t]) \\ wp'[t] & \text{otherwise} \end{cases}
\end{aligned} \tag{2.10}
$$

The pointers that are actually used to address the memory element are derived from these internal pointers:

$$
\begin{aligned}
rp[t] &= rp'[t] \bmod \texttt{FIFO\_DEPTH} \\
wp[t] &= wp'[t] \bmod \texttt{FIFO\_DEPTH}
\end{aligned} \tag{2.11}
$$

Note that only half of the internal pointer range is used for the memory pointers. The other half is used for the generation of the *full* and *empty* flags. This half defines

the phase in which the read and write pointers are:

$$\phi_{rp'}[t] = \begin{cases} \texttt{false} & \text{if } (rp'[t] < \texttt{FIFO\_DEPTH}) \\ \texttt{true} & \text{otherwise} \end{cases}$$
$$\phi_{wp'}[t] = \begin{cases} \texttt{false} & \text{if } (wp'[t] < \texttt{FIFO\_DEPTH}) \\ \texttt{true} & \text{otherwise} \end{cases} \tag{2.12}$$

Now, the *full* and *empty* flags can be generated from the memory pointers and the phase information:

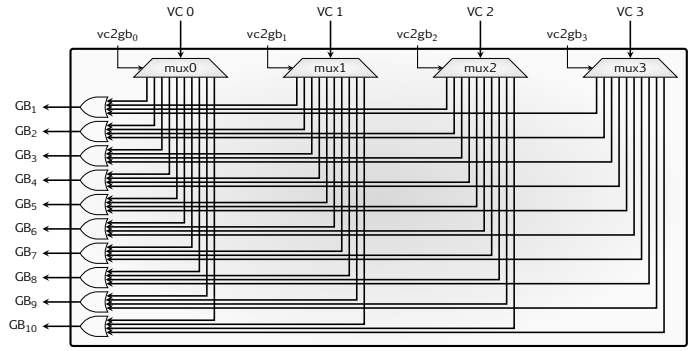
$$empty[t] = \begin{cases} \texttt{true} & \text{if } \left(wp[t] = rp[t] \ \&\ \phi_{rp'}[t] = \phi_{wp'}[t]\right) \\ \texttt{false} & \text{if } \left(wp[t] > rp[t] \ \&\ \phi_{rp'}[t] = \phi_{wp'}[t]\right) \\ \texttt{undefined} & \text{otherwise} \end{cases}$$
$$full[t] = \begin{cases} \texttt{true} & \text{if } \left(wp[t] = rp[t] \ \&\ \phi_{rp'}[t] = \neg\phi_{wp'}[t]\right) \\ \texttt{false} & \text{if } \left(wp[t] < rp[t] \ \&\ \phi_{rp'}[t] = \neg\phi_{wp'}[t]\right) \\ \texttt{undefined} & \text{otherwise} \end{cases} \tag{2.13}$$

Note that the case where the phases are equal and the write pointer is behind the read pointer and the case where the phases differ and the write pointer is ahead of the read pointer are not valid cases. In order to get in such a situation, the reader must have read from the buffer while is was empty. Such a situation can never occur.

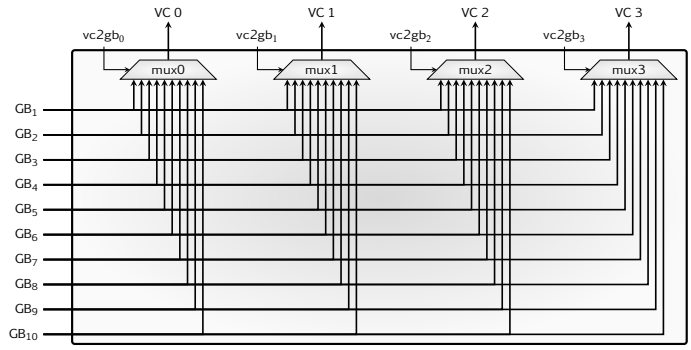
Since the FIFO is used as a synchronization device between two clock domains, it has separate clock inputs for the read side and the write side (shown in Figure 2.12). The pointer update takes place immediately after the transfer. However, due to setup and hold times the updated pointer value is only valid after a certain period τ. During that period, glitches may occur in the individual bits of the counter. Hence, if the full and empty signals are implemented with asynchronous logic, they are sensitive to these glitches. If the rising edges of the read clock clk_r and the write clock clk_w of the FIFO are close together ($|clk_{r,\uparrow} - clk_{w,\uparrow}| \ll \tau$), the full and empty signals may be read incorrectly due to these glitches [63]. This can be avoided by adding synchronizers before the full and empty flag logic blocks. For example, the read pointer (updated at the rising edge of the read clock) is fed into a flip-flop that is clocked at the rising edge of the write clock and its output is fed into the full flag logic block.

In our case, the clocks are generated with different frequencies but the rising edges are tightly aligned. This is because the generated clocks satisfy the relation defined in Equation 2.1. As a result, glitches on the full and empty signals only occur immediately after the rising edges of both clocks, hence the read or write transaction is done correctly. Therefore, additional synchronization means are not required.

Crossbar The NI contains two fully connected 4×10 crossbars. These crossbars consist of large multiplexers to allow any connection between input and output. Figure 2.13(a) shows the internal structure of the input crossbar, which is the crossbar on the right side in Figure 2.11. The outgoing crossbar is depicted in Figure 2.13(b). It is possible to connect multiple VCs to one Global Bus (GB) at the same time, as



(a) Crossbar connected to input channels



(b) Crossbar connected to output channels

Figure 2.13 – Internal structure of the crossbars

can be seen in Figure 2.13(a). Since each GB is connected to the input VCs via a wired-or, there will not be a risk for short circuit current but in the case multiple VCs are connected to the same GB, data corruption will occur. However, since the control signals for each of the VC multiplexers are generated synchronously, this situation can easily be avoided.

Flit construction & deconstruction The network protocol conversion is done by the flit construction and deconstruction blocks. For outgoing NoC communication, packets are created and transmitted to the NoC. The data received from the Montium TP via the crossbar is annotated with a flit type and stored in the FIFO buffers. The flit construction is done asynchronously, so the latency from the crossbar output to the buffer inputs is less than one clock cycle. Alternatively, it is also possible to transmit a few flits out of a small configurable storage ROM[3]. For each VC, one ROM

[3]Technically, a storage ROM is implemented as a register bank that can only written at run-time via the configuration interface. Functionally, however, a storage ROM can only be used as a ROM by the Montium TP.

is available in which up to 4 flits can be configured. During execution, the Montium TP can access these Read-only Memories, select a flit and send it to the NoC (see also section 2.2.2.2).

The flit deconstruction block separates the flit into type and data. The data is transmitted to the crossbar and the flit type is passed to the control part. Again, this operation can be done asynchronously, hence the latency of the data stream is not increased. How an incoming flit is deconstructed and which part of the data is passed to the crossbar, is determined by the control part of the Hydra.

2.2.2.2 Control part

The middle part of Figure 2.11 shows the control part, where incoming packets are analyzed and the internal data path control signals and Montium TP interface signals are generated. These signals are all generated at the TP clock frequency by *Message Execution Modules*, where each module has a dedicated interface to the Montium TP. Figure 2.11 shows the *Streaming interface*, which is used by the *Program control module*, the *Program Control interface* which is used by the *Program control module* and *Flow control module*, the *Configuration interface* that is used by the *Configuration module* and the *DMA interface* that is used by the *DMA module*.

Flow control The packets received from the NoC are inspected by the flow control block. It detects the start of the package, indicated by a command flit. After decoding the command, it enables the related message execution module for further packet handling. The rest of the packet is not modified by the flow control block, except for the tail flit. After receiving a tail flit, the flow control block disables the message execution module that was handling the packet. Furthermore, it handles IO requests by the message execution modules for controlling the flit formatting and FIFO buffers.

Message Execution Modules The message execution modules are used for handling commands received from the NoC. The packets containing these commands are typically formatted using the following flit sequence:

Definition 7 (Hydra packet structure)**.** $C\left[H\,[D]^{+}\right]^{+}T$

Here the notation $[x]^{+}$ defines one or multiple occurrences of x. By analyzing the first flit of a packet (the C flit), the encoded command is obtained. It defines the structure of the rest of the packet. The command is stored in the 3 Most Significant Bits (MSBs) of the flit (bits 15..13 in the data part of the flit, as depicted in Figure 2.14). Depending on the command encoded in the 3 MSBs, the other 13 Least Significant Bits (LSBs) of the flit (bits 12..0) may contain additional information used by the message execution module that handles the command. An overview of the encoding of all possible commands and the usage of additional information is given in Table 2.3.

An overview of the different packets that can be transmitted using these commands is given in Table 2.4. The message handling is performed by several message

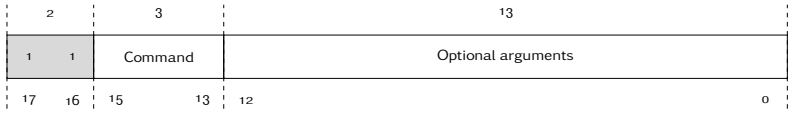


Figure 2.14 – Structure of a command flit

Table 2.3 – Encoding of the command flit in a packet

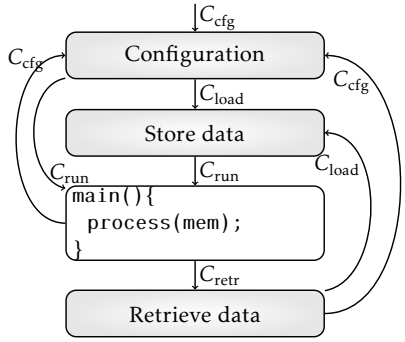
Symbol	Flit encoding (bit range)	
	15..13	12..0
C_{cfg}	000	-
C_{load}	001	input FIFO mask (3..0)
C_{retr}	010	input FIFO mask (3..0)
C_{status}	011	gpo mask (5..0)
C_{run}	100	gpi mask (5..0)
C_{wait}	101	-
C_{rst}	110	input/output FIFO mask (7..0)
`reserved`	111	-

Table 2.4 – Overview of the structure of messages in the Hydra message protocol

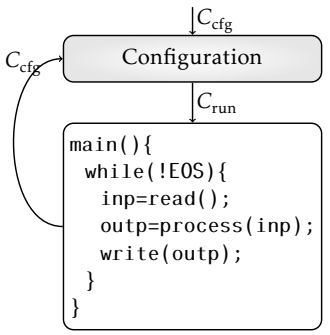
Message Handler	Message type	Packet format
Configuration module	Configuration	$C_{cfg}\left[H\,[D]^{+}\right]^{+}T$
DMA control	DMA load	$C_{load}\left[H\,[D]^{+}\right]^{+}T$
	DMA retrieve	$C_{retr}\,[HD]^{+}\,T$
Program control	Get status	C_{status}
	Run	C_{run}
	Wait	C_{wait}
Flow Control	Reset	C_{rst}

execution modules, where each module is responsible for one or a few packet types as displayed in Table 2.4.

There is a slight difference in the usage of control messages in both operating modes (streaming-mode and block-mode) as shown in Figure 2.8. Figure 2.15 shows the state transition diagrams of Figure 2.8, where the state transitions are now annotated with the accompanying control messages that cause a state change. This example shows the difference between the usage of control messages in both operation modes. As can be seen, the main difference is in the DMA load and retrieve transactions, which are not required for the streaming-mode. In the next sections the operation of the various message handlers is explained.



(a) Control messages used for block-mode operation



(b) Control messages used for streaming-mode operation

Figure 2.15 – Control messages used to make state transitions in block-mode and streaming-mode operation. The control overhead for streaming-mode is considerably smaller than the control overhead for block-mode.

Configuration module The *configuration module* is used to reconfigure the functionality in the Montium TP using the NI's configuration interface (shown in Figure 2.11). This interface consists of a dedicated address/data based configuration bus, that is connected to all configurable entities within the Montium TP. The configuration bus protocol is very basic, as it is a write-only protocol using a 16-bit data signal (`c_data`), a 16-bit address line (`c_addr`) and a 1-bit `c_dv` signal that indicates the transaction is valid. There is no acknowledgement signal to confirm a configuration, hence these 3 signals together form the entire configuration interface. A reconfiguration of (a part of) the Montium TP is started by enabling the `c_dv` signal and simultaneously writing address and data information to the `c_addr` and `c_data` buses. By disabling the `c_dv` signal the configuration transaction is ended.

This configuration bus protocol is implemented in the configuration module and can be enabled by sending a configuration packet to the NI. For example, Figure 2.16 shows such a packet. A timing diagram of the transmission of this packet and the start of the execution by the NI is shown in Figure A.1, where the signals in the upper part show the behavior at the NoC side according to section 2.1.2.2. The input flits are stored in the FIFO buffers which are operated at the NoC clock `clk`. The lower part shows the Montium TP configuration interface, operated at the tile clock `clk_t`.

For a typical configuration, several ranges within the configuration address space are addressed. Hence, the addresses written to the address bus are often incremented by 1. The configuration interface automatically increments the previous address unless a new address is received, as shown in Figure 2.16 and Figure A.1. Hiding incrementing addresses in the input stream is called *address compression*. This addressing approach enables arbitrary-length transactions with minimum addressing overhead. Hence, packets will consist of less flits (decreasing bandwidth requirements) and data flits can be transmitted without any redundant information in between (decreasing packet latency).

VC_0	
C 0000	Configuration command
H 0000	Set offset address 0x0000
D 0000	Write 0x0000 to address 0x0000
D 0000	Write 0x0000 to address 0x0001
D 0011	Write 0x0011 to address 0x0002
D 0000	Write 0x0000 to address 0x0003
D 8000	Write 0x8000 to address 0x0004
D 8000	Write 0x8000 to address 0x0005
H 0100	Set offset address 0x0100
D 0006	Write 0x0006 to address 0x0100
D 8424	Write 0x8424 to address 0x0101
D 0447	Write 0x0447 to address 0x0102
D 0070	Write 0x0070 to address 0x0103
⋮	
T 0000	End of configuration

Figure 2.16 – Example configuration packet to be transmitted in Figure A.1

DMA controller Data can be written to and read from the Montium TP's memories and register files using DMA controller inside the Hydra NI. A block of data can be written directly into a memory of the Montium TP using a burst write, which is called a *DMA load* operation. Because, in normal operation, the Montium TP's internal memories and register files are not available to external processors, an external processor has to write a block of data to the Hydra NI which has access to the internal memories. Similarly, reading a block of data directly from a memory of the Montium TP can also be done by sending a request to the Hydra NI, which then returns the requested data. These operations can be done by using the *DMA load* and *DMA retrieve* commands, which are handled by the DMA controller inside the Hydra NI. The DMA controller uses the Montium TP's DMA interface to access the memories and register files. This interface consists of a `dma_sel` signal that indicates the selection of the DMA interface. It freezes the Montium TP such that the program is halted. The `dma_rw` signal indicates whether the DMA operation is a read (`dma_rw`= 0) or a write (`dma_rw`= 1) transaction. For making the selection between reading the memories or the registers, the `dma_mr` is made low to select the memories or high to select the register files. In case of a register file transaction, the `dma_rs` signal is used to choose between accessing the register files A and C (`dma_rs`= 0) or B and D (`dma_rs`= 1). The offset address within the selected memory or register file is indicated by the 10-bit `dma_addr` signal. By writing the 10-bit `bus_en` signal, the Montium Global Buses are directly connected to the data buses of the Static Random Access Memories and to the selected register files (selected via `dma_rs`) such that $SRAM_x$ can be accessed via GB_x. This enables the selection of individual Global Buses for the DMA transaction.

The command flit indicating the start of a DMA load packet enables the DMA interface. A header flit in the DMA load or retrieve packet is used to select an offset address within the Montium memory range. For more detailed information on the memory mapping, refer to [155]. The next data flit is then written to the decoded memory location. Similar to the configuration packet, for subsequently received data flits the memory address is incremented automatically. An example packet

VC_0	
C 2001	DMA load command
H F300	Set offset address 0xF300 (PP1 Register file C)
D FEF3	Write 0xFEF3 to address 0xF300
H F500	Set offset address 0xF500 (PP2 Register file C)
D 16F2	Write 0x16F2 to address 0xF500
H F700	Set offset address 0xF700 (PP3 Register file C)
D 5436	Write 0x5436 to address 0xF700
H F900	Set offset address 0xF900 (PP4 Register file C)
D 16F2	Write 0x16F2 to address 0xF900
H FB00	Set offset address 0xFB00 (PP5 Register file C)
D FEF3	Write FEF3 to address 0xFB00
⋮	
T 0000	End of DMA load

Figure 2.17 – Example DMA load packet for writing data in the register files, as transmitted in Figure A.3

that illustrates a DMA load operation into the Montium TP register files is shown in Figure 2.17. It shows how a set of coefficients for a FIR filter is stored in the register files, where one coefficient is stored per PP at a time. The timing diagram of this transaction can be found in Figure A.3.

Multiple Global Buses can be accessed in parallel, so it is also possible to perform multiple DMA transactions in parallel. The DMA interface is shared, so these DMA transactions can only be performed simultaneously if they are similar (for example, two write transactions to two memories using the same offset address). This is useful in situations where multiple DMA transactions are related, for example when one transaction stores the real part of a stream containing complex numbers to memory 1 and a second DMA transaction stores the imaginary part of that stream to memory 3. Such a transaction can be initiated by sending a packet via multiple channels, as shown in Figure 2.18. This example shows how a set of complex numbers are stored in memories 9 and 10 via GB_9 and GB_{10}. The command flit `C 2003` is decoded to a DMA load command with argument 0x3. This argument masks the input VCs that are used for the current DMA transaction. The next expected flit in the DMA load packet is a header flit. For each masked VCs such a flit is expected. In this case, at VC_0 a header flit `H 9000` is received, which indicates that the destination for the data received from that channel is memory 9, using offset address 0. The header flit `H A000` received from VC_1 indicates that the destination for that channel is memory 10[4]. After decoding the header flits, the DMA transaction is started by writing the data from all masked channels to the selected memories. This transaction is also shown in Figure A.2.

To read the contents of a memory or register file, the DMA retrieve command can be used, which is also handled by the DMA controller. A valid DMA retrieve packet consists of a command flit containing a 4-bit mask indicating via which output VCs the retrieved data should be written (similar to the DMA load VC masking as described above), for each masked VC a header flit is expected that contains an offset

[4] Since the DMA interface uses a single address bus, the offset address for DMA transactions via any VC other than VC_0 uses the address encoded in the header flit received via VC_0.

VC_0	VC_1	
C 2003		DMA load command (via VC_0 and VC_1)
H 9000	H A000	Set offset address 0x0000, $SRAM_9 \Leftarrow VC_0$ and $SRAM_{10} \Leftarrow VC_1$
D 7FFF	D 0000	Write 0x7FFF to mem9 / 0x0000 to mem10 (address 0x0000)
D 7642	D CF04	Write 0x7642 to mem9 / 0xCF04 to mem10 (address 0x0001)
D 5A82	D A57E	Write 0x5A82 to mem9 / 0xA57E to mem10 (address 0x0002)
D 30FC	D 89BE	Write 0x30FC to mem9 / 0x89BE to mem10 (address 0x0003)
D 0000	D 8000	Write 0x0000 to mem9 / 0x8000 to mem10 (address 0x0004)
D CF04	D 89BE	Write 0xCF04 to mem9 / 0x89BE to mem10 (address 0x0005)
D A57E	D A57E	Write 0xA57E to mem9 / 0xA57E to mem10 (address 0x0006)
D 89BE	D CF04	Write 0x89BE to mem9 / 0xCF04 to mem10 (address 0x0007)
T 0000	T 0000	End of DMA load

Figure 2.18 – Example acrshortDMA load packet for writing data in the memories via multiple channels in parallel, as transmitted in Figure A.2

VC_0	
C 4001	DMA retrieve command
H 2000	Set offset address 0x2000 (Memory 2)
D 03FF	Set block length to 512 samples
⋮	Wait until all 512 samples have been received
T 0000	End of DMA retrieve

Figure 2.19 – Example DMA retrieve packet for reading data from memory 2, as transmitted in Figure A.4

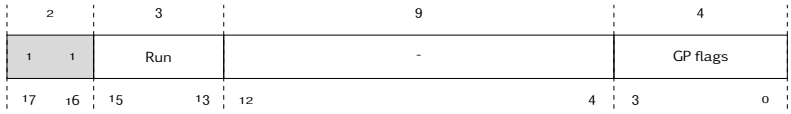


Figure 2.20 – Flit encoding for the run command

address indicating which memory or register file to be read, then a data flit that indicates the block size starting at the offset address and the packet is closed with a tail flit. The block size transmitted in the data flit should contain the memory block size to be returned, decremented by one (for example, to return a full Montium TP memory which consists of 1024 addresses, the value 1023 should be transmitted in the data flit). As soon as the tail flit is received, the retrieve action is stopped. Hence, for a valid completion of a DMA retrieve operation, the tail flit should only be sent as soon as the entire requested block has been transmitted to the NoC. Figure 2.19 shows an example packet that is used to retrieve a block of 512 samples from offset address 0 in memory 2 of the Montium TP.

Program control Once the configuration and (optional) memory initialization have taken place, the Montium TP is enabled and will start its execution. The start of execution is triggered by the C_{run} command. Along with sending this command, it is possible to include several general purpose flags that are stored into the General Purpose (GP) register. Figure 2.20 shows the structure of a C_{run} command including these 4 GP flags (shown in Table 2.5).

Table 2.5 – C_{run} command argument GP flags

Flag #	Function	Description
0	Handshake 0	Initiate handshake procedure with TP
1	Handshake 1	Initiate handshake procedure with TP
2	Pulse 0	Send pulse to TP
3	Pulse 1	Send pulse to TP

Table 2.6 – GP flag register usage

Bit #	NI → TP	TP → NI
0	Initiate handshake 0	Acknowledge handshake 0
1	Initiate handshake 1	Acknowledge handshake 1
2	Initiate pulse 0	-
3	Initiate pulse 1	-
4	Header [1]	-
5	Tail [1]	Interrupt

[1] Automatically generated by NI to indicate flit type of next flit to be read from the input FIFO

Upon reception of a C_{run} command, the GP flags are stored in the GP register. Additionally, the GP register contains 2 flags which are automatically asserted when the flit type of flits received from any input channel equals a header flit (GP_4) or a tail flit (GP_5). This is useful since the flit deconstruction block removes the flit types. The outputs of this GP register are connected to the Montium TP. Similarly, the Montium TP has 3 output GP pins, which are connected to a second GP register inside the program control block in the NI. An overview of the contents of these GP registers is given in Table 2.6.

The handshake and pulse flags can be used to synchronize between the Montium TP program currently executed and some external input. For example, the Montium TP can assert a pulse flag when it reaches a certain point in the current program, which may be useful for debugging. The handshake and pulse flags can only be set to 1 using the C_{run} command. They will be automatically cleared upon completion of the transaction. For the handshake flag, the transaction can only be completed when the TP acknowledges the handshake by asserting a handshake signal. Then, the handshake flag will be reset and can be used again in the next C_{run} command. If the pulse flag is enabled, a single clock cycle lasting pulse is generated. At the next rising edge of the clock, the pulse flag is automatically reset.

Stream controller After receiving a C_{run} command, the Montium TP is enabled and starts the execution of the program currently configured. During execution, the Montium TP can use its streaming interface of the Hydra NI to read data from a network stream and write to a network stream. Next to control signals for all

Table 2.7 – Streaming IO configuration register instruction format

Bit #	Signal	Size	Description
37..36	ft	2	Flit type for $\texttt{gb2vc}_x$ data
35..29	$\texttt{vc2gb}_0$	4	GB destination for VC_0
31..28	$\texttt{vc2gb}_1$	4	GB destination for VC_1
27..24	$\texttt{vc2gb}_2$	4	GB destination for VC_2
23..20	$\texttt{vc2gb}_3$	4	GB destination for VC_3
19..15	$\texttt{gb2vc}_0$	5	GB/storage ROM_0 source for VC_0
14..10	$\texttt{gb2vc}_1$	5	GB/storage ROM_1 source for VC_1
9..5	$\texttt{gb2vc}_2$	5	GB/storage ROM_2 source for VC_2
4..0	$\texttt{gb2vc}_3$	5	GB/storage ROM_3 source for VC_3

decoders, a Montium sequencer instruction also contains a Streaming IO (SIO) control signal, which is made available to the Hydra NI via the streaming interface. The 3-bit control signal is used to select an instruction in the SIO configuration register (which contains up to 8 different SIO instructions). The SIO configuration register is located inside the Hydra NI's stream controller, such that a Montium sequencer instruction directly controls the Hydra NI's behavior while operating in streaming-mode.

The SIO configuration register in the NI is formatted as shown in Table 2.7. Up to 8 SIO instructions can be programmed within this register. Each instruction defines the streaming from input VCs and output VCs simultaneously. The $\texttt{vc2gb}_x$ field encodes to which GB the data from VC_x is written and for making the connection from a GB to VC_x the $\texttt{gb2vc}_x$ field is used. Together, the output of these fields is directly used as control signals for both crossbars. The `flit type` field is used to instruct the Flit Construction block which flit type should be added to outgoing data to the NoC.

As mentioned before, the Flit Construction block contains a storage ROM for each VC. Per VC a maximum of 4 flits can be configured within the ROM. These Read-only Memories can be accessed by the SIO instruction. The full encoding of the $\texttt{vc2gb}_x$ and $\texttt{gb2vc}_x$ fields is shown in Table 2.8.

2.2.3 Realization

In the Smart Chips for Smart Surroundings (4S) project [2], the *Annabelle* MPSoC was designed as a proof of concept of a reconfigurable tiled architecture. It consists of 2 circuit-switched NoC routers as described in section 2.1.2, 4 Montium TPs connected to the routers via 4 Hydra NIs, and an ARM-926 subsystem based on a AHB bus connecting the ARM-926 processor to a local memory and several IO interfaces. The goal of defining such an MPSoC was to show the feasibility of a reconfigurable subsystem as an accelerator for multi-media applications. The feasibility study targeted the gate-level design, synthesis, functional testing, and performance analysis of the run-time system using measurements on system-wide energy consumption

Table 2.8 – $vc2gb_x$ and $gb2vc_x$ encoding

$vc2gb_x$	Connection	$gb2vc_x$	Connection
0000	Disconnected	00000	Disconnected
0001	$GB_1 \Leftarrow VC_x$	00001	$VC_x \Leftarrow GB_1$
0010	$GB_2 \Leftarrow VC_x$	00010	$VC_x \Leftarrow GB_2$
0011	$GB_3 \Leftarrow VC_x$	00011	$VC_x \Leftarrow GB_3$
0100	$GB_4 \Leftarrow VC_x$	00100	$VC_x \Leftarrow GB_4$
0101	$GB_5 \Leftarrow VC_x$	00101	$VC_x \Leftarrow GB_5$
0110	$GB_6 \Leftarrow VC_x$	00110	$VC_x \Leftarrow GB_6$
0111	$GB_7 \Leftarrow VC_x$	00111	$VC_x \Leftarrow GB_7$
1000	$GB_8 \Leftarrow VC_x$	01000	$VC_x \Leftarrow GB_8$
1001	$GB_9 \Leftarrow VC_x$	01001	$VC_x \Leftarrow GB_9$
1010	$GB_{10} \Leftarrow VC_x$	01010	$VC_x \Leftarrow GB_{10}$
1011	Reserved	01011	Reserved
1100	Reserved	01100	Reserved
1101	Reserved	01101	Reserved
1110	Reserved	01110	Reserved
1111	Reserved	01111	Reserved
		1--AA	$VC_x \Leftarrow ROM_x(AA)$

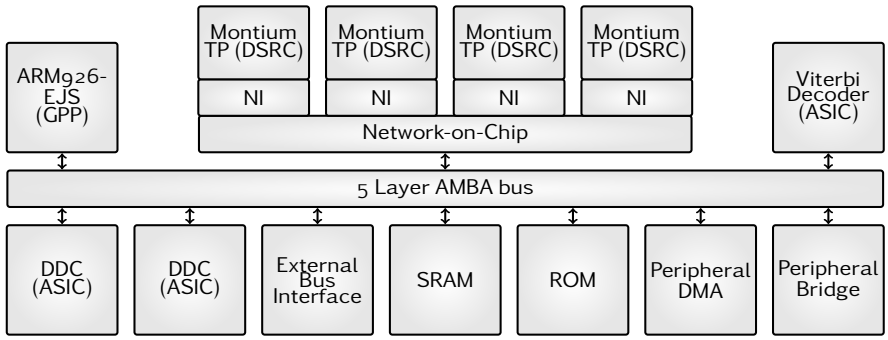


Figure 2.21 – Annabelle MPSoC schematic

and overall processing performance. For the performance analysis, the application presented in section 4.2 was used.

2.2.3.1 Annabelle MPSoC

The VHDL model of the Hydra NI was synthesized in 0.13 µm Atmel technology, while the clock frequency was constrained to 200 MHz. With this constraint the area was 19k gates (0.106 mm^2), which is about 5% of the area of the Montium TP. The area distribution of the several components is given in Table 2.9.

A large part of the total area (41.45%) is needed for the input and output buffering.

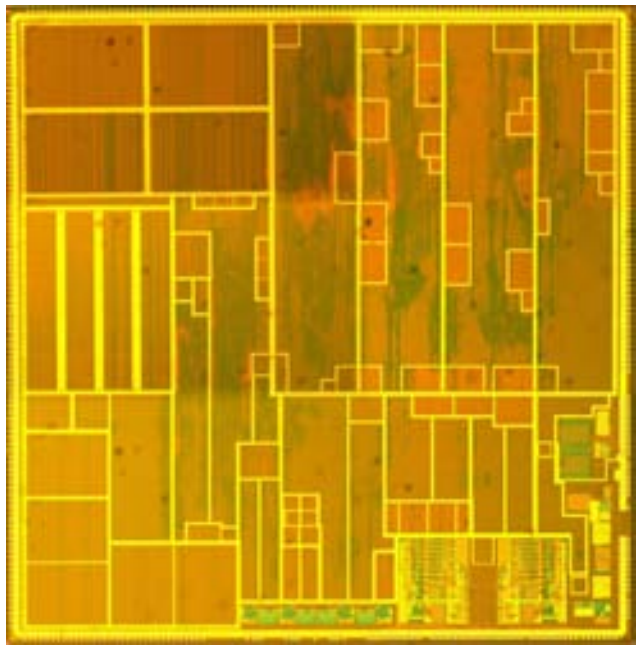


Figure 2.22 – Annabelle MPSoC die photo. The reconfigurable part containing four Montium TPs connected to two NoC routers via four Hydra NIs is shown in the upper right part and the ARM GPP can be found in the middle at the left side.

Table 2.9 – Hydra NI area distribution

Component	#gates	Area (mm^2)	Area (%)
Crossbar	1810	0.010	9.47
Flit formatting	3695	0.021	19.34
Flow control	3893	0.022	20.38
Buffering	7920	0.044	41.45
Message execution	1788	0.010	9.36
Total	19106	0.106	

The flow control and flit formatting each contribute 20% of the total area, which is caused by the storage Read-only Memories and the instruction decoder. For the other components (the crossbar and the message execution modules) the area is related to the multiplexers in the data path and the control logic around it.

2.2.3.2 Block-mode vs. streaming-mode

As was shown in Figure 2.15, the block-mode and streaming-mode implementations of an algorithm require different control messages. The larger the number of control messages for the execution of an algorithm becomes, the more synchronization

is required between the controlling tile and the Montium TP tile. While a DMA operation is handled by the NI, the Montium TP is temporarily halted such that its memories can be read. This reduces the total time available for processing, which means the utilization of the Montium TP is decreased.

Using block-mode operation does have some advantages. For example, it is possible to read different parts of the Montium TP memory after each execution of the algorithm. This can be useful for debugging or for algorithms where the address patterns are data dependent such that only the controlling tile can decide which part of the memory contains the useful information.

The streaming interface used by the Montium TP in a streaming-mode situation can be used very effectively. It enables communication between tile processors without control messages, such that the controlling tile processor does not need to continuously monitor the Montium TP state and transmit time-critical control messages. As a result, the total MPSoC performance increases when using streaming-mode communication.

All functionality for the block-mode and streaming-mode operation is implemented in the DMA controller and the SIO configuration register. Together, these are the largest components within the Message Execution Modules (in terms of silicon area). As can be seen in Table 2.9, the message execution is less than 10% of the Hydra NI's total area while the main costs are in the buffering. Hence the overhead in hardware costs for both modes are minimal.

2.2.3.3 Throughput and latency

The crossbars were designed such that they scale with the number of input VCs and the number of Global Buses. A dedicated FIFO buffer for each input VC and each output VC enables high parallelism in communication.

The message protocol introduces a small input latency at the beginning of each packet, but provides a throughput that is near optimal. This is possible due to the use of address compression, which reduces the interconnect bandwidth required for sending a certain data transfer. Using a typical bus interface with dedicated address and data lines, sending a configuration for the FFT-1024 to the Montium TP would require 597 16-bit addresses and 597 16-bit words (see Table 2.10). Address compression reduces the amount of 16-bit addresses to only 93, which is a compression of 84.5% of the bandwidth required for transmitting addresses, or 43.2% of a full configuration packet. For DMA transfers, this compression can become even larger as the data is typically written in large blocks to the memory, hence only few offset addresses are needed in the input data. The overhead of the C, H and T flits compared to the D flits is relatively high for small packets (for example, DMA transfers for loading a few coefficients) and for configuration packets, because those packets contain a large amount of addressing information. There is no significant difference in the protocol overhead and address compression between the both operating modes. Configuration sizes are about equal in size and address compression can be done equally efficient for the both modes.

Since the packet (de-) construction and crossbars are combinatorial circuits, they do not add any latency to the communication. Hence the only latency is in the

Table 2.10 – Flit type distribution for different packet types

Algorithm	Packet type	C	H	D	T	Overhead
Block mode FFT-1024	Configuration	1	93	597	1	13.7%
	DMA load (twiddle factors)	1	2	1024	1	0.4%
	DMA load (input data)	1	2	1024	1	0.4%
	DMA retrieve (output data)	1	2	1024	1	0.4%
Block mode FIR-5	Configuration	1	90	148	1	38.3%
	DMA load (coefficients)	1	5	5	1	58.3%
	DMA load (input data)	1	1	1024	1	0.3%
	DMA retrieve (output data)	1	1	1024	1	0.3%
Streaming mode FFT-1024	Configuration	1	91	595	1	13.5%
	DMA load (twiddle factors)	1	2	1024	1	0.4%
	Run + stream input / output	1	0	2048	1	0.1%
Streaming mode FIR-5	Configuration	1	87	135	1	39.7%
	DMA load (coefficients)	1	5	5	1	58.3%
	Run + stream input / output	1	0	2048	1	0.1%

FIFO buffers and in the message handling modules. The message protocol has been designed such that the message handling modules can be operated at the full tile clock frequency, processing one flit in the packet per clock cycle. Therefore, they do not decrease the throughput bandwidth and the latency is only one clock cycle per flit.

2.2.3.4 Clocking regime

The Hydra NI operates almost fully in the tile clock domain, except for the buffering parts. Due to the parallelism in the NI's datapath, the data bandwidth to the Montium TP is not restricted by the NI. The FIFO buffers were designed for synchronization between clock domains with in-phase rising clock edges. Using a clock divider that divides a global system clock by a factor of 2^n (with $0 \leq n \leq 4$) the clock edge synchronization can be done more efficiently. Therefore, the read and write pointers within the FIFO buffers can be implemented without complex address generators and multiple synchronization registers that suppress the risk for metastability.

2.2.3.5 Energy-efficiency

The energy-efficiency of the Hydra NI was estimated based on the post-layout netlist of the Annabelle MPSoC. Since the operation of the Hydra NI is tightly connected to the operation of the Montium TP, the estimation was done for an entire processing tile and, where possible, specified per component. The estimations include a subdivision of static power and dynamic power. Algorithms used to perform the estimations are a radix-2 FFT (for more detail, see section 4.1.2), a non-power-of-two FFT (see section 4.1.3) and a FIR filter (see section 4.1.4).

Table 2.11 – Static and dynamic power distribution over a Montium processing tile for different streaming algorithms. Dynamic power is normalized per MHz.

Montium TP component	Static power (µW)	Dynamic power (µW/MHz) FIR-5	FFT-512	FFT-288
Datapath processing	94.8	187.63	237.81	151.27
Memories	68.0	3.60	276.99	212.57
Sequencer processing	0.6	2.81	4.02	3.45
Sequencer memory	13.2	20.27	63.42	50.11
Decoders	26.5	0.01	1.08	3.06
Hydra	7.1	20.40	19.27	15.26
Total	210.2	234.72	602.59	435.72

These numbers show that the static power consumption of the Hydra NI is about 3% of the Montium TP's static power consumption and the dynamic power consumption is in the range of 3% to 10% depending on the type of algorithm. On average, this means that the Hydra NI contributes to less than 10% of the total power consumption. The ratio in area of the Hydra NI with respect to the Montium TP is comparable: 0.106 mm^2 compared to 1.8 mm^2 for the Montium TP is about 6%.

2.3 Conclusion

In this chapter we presented the design and implementation of the Hydra NI, an energy-efficient and reconfigurable network interface for the Montium TP and an NoC. It supports both block-mode and streaming-mode data transfers and is controlled by a lightweight message protocol. The bandwidth provided by the Hydra is only limited by the chosen NoC and the Montium TP, as its data path provides a maximum connectivity between all input channels from the NoC to the Montium TP and vice versa. If constrained to operate at a maximum clock frequency of 200 MHz, the Hydra NI has a total area of 19k gates or about 0.106 mm^2 in 0.13 µm ASIC technology. Looking at the power distribution of one Montium TP processing tile for typical streaming DSP applications, the Hydra NI contributes for about 3% to 10% of the total power budget of one processing tile.

In conventional bus-based architectures, the processors connect to the interconnect directly. A bridge is then responsible for the physical level protocol of the interconnect, dealing with arbitration, timing and acknowledgements. Since routing and arbitration are determined on beforehand in our NoC architecture, such bridging functionality is not required and the network interface can be optimized for throughput and latency.

Moreover, when the communication time is decreased, the tile clock can be slowed down while the tile processor is still capable of performing enough computations to finish the process within its required period. This can contribute to a significant reduction in energy consumption.

The design of a lightweight message protocol developed to support the functions that are required by the processor contributes to a straight-forward implementation of the network interface. The packet overhead introduced for adding header, control and tail information to packets is low for large packets. To avoid transmitting redundant information within one packet, large memory blocks can be transported efficiently by applying address compression.

Although the Hydra NI is presented as a Montium TP specific network interface, the basic mechanisms are reusable for the design of a network interface for any stream processor architecture where parallel IO transactions are useful to increase the processing performance.

Chapter 3

Design flow for Streaming DSP Applications

Abstract

Application design for multi-processor architectures has been a topic of research for many years. Much effort has been spent on porting existing applications developed for single-processor architectures to multi-processor systems, for example by using parallelizing compilers and by using a shared memory model and cache coherency protocols. The reconfigurable tiled architectures presented in the previous chapter are optimized for streaming applications and have a different memory abstraction that does not require shared memory and cache coherency protocols. Moreover, the on-chip network allows for multiple data streams to exist simultaneously, while for each stream guarantees can be given on latency and throughput. This chapter presents new ideas for a design flow, where applications are written using a mathematical programming language. The mathematical programming language gives the flexibility to manipulate (parts of) applications, by applying transformations to the expressions forming such an application. Using a dataflow simulation framework, the execution of applications described in this programming language on multi-processor systems is simulated. The simulation framework includes a graphical user interface that visualizes the effects of application transformations.

Applications that are operating on long sequences of data are called *streaming applications*. Such applications are mainly data-driven, meaning that without input data no processing has to be performed. Therefore, the required throughput determines the time frame available for processing. A typical characteristic of streaming applications is the relative simplicity of calculations on long streams of data. Due to regularity in the calculations, *data parallelism* can be exploited by applying one instruction to multiple data words simultaneously using Single Instruction Multiple Data (SIMD) instructions. Next to data parallelism, other forms of parallelism can be found within

Parts of this chapter have been presented at the 18th, 19th and 20th Annual Workshop on Circuits, Systems and Signal Processing [156–158], and at the Scientific ICT Research Event Netherlands (SIREN 2009) [159].

applications. For example, a streaming application can be modeled as a set of small independent processing blocks called *kernels*. Data dependencies between these kernels are made explicit in the form of *communication channels*. This separation of kernels allows for concurrency at kernel level (called *task-level parallelism*). Separate kernels can be mapped on separate processing elements, such that the result from one kernel is sent to the next kernel via their common communication channel. The next kernel can then be executed using the data received via the communication channel, while the first kernel can be executed on the next part of the input data, such that a pipeline of kernels is formed.

A kernel is said to be *stateless* if its output only depends on the current input. If its output also depends on previous inputs, its state can be made explicit by providing the previous state as an input to the function (in addition to the current input) and the new state is provided as an additional output (next to the output data streams).

3.1 State of the Art

DSP application design for single-processor architectures has been studied for many years. When the processing capacity of individual processors became a limitation, architectures were composed from multiple single-core chips to increase the total computational performance. The requirements, however, for partitioning and allocating an application over multiple single-core chips differs considerably from partitioning and allocating over a single-chip multi-core architecture.

One of the reasons is that the energy consumption per communicated bit is much higher when communicating via an off-chip link. In general, we can observe that energy-wise processing is becoming cheaper and communication is becoming relatively expensive. The Multi-Processor Systems-on-Chip considered in this thesis provide a large aggregated computational capacity with a considerable communication bandwidth between the cores.

3.1.1 Design flow

Many approaches for DSP application design for multi-core architectures have been presented before. Several approaches try to realize a fully automatic tool chain, such that any DSP application can be ported to the targeted architecture. Others advocate a fully manual approach, where the entire application and hardware architecture are fine tuned manually. Our proposed method is a hybrid approach that is partly manual and partly automatic.

3.1.1.1 Automatic approach

Several research projects focus on automatic parallelization of sequential code, typically based on a legacy C or Matlab program. They have in common, that they all use a Kahn Process Network (KPN) (see section 3.1.2) to model the communication and synchronization of the partitioned application. Within the Artemis project [64, 65], several tools were developed. The Simulation of Embedded Systems Architectures for Multi-level Exploration (SESAME) framework targets modeling and

simulation methods and tools for efficient design space exploration of heterogeneous embedded multimedia systems [66]. Recently, it has been merged with the Daedalus framework [67], which aims at bridging the gap between system-level models and implementation. Another tool developed in the Artemis project is Compaan, which is a tool for automatic parallelization of sequential code to KPNs [68]. It can be used to transform affine nested loop programs to parallel programs, either for simulation on a desktop processor or for compilation to an embedded architecture.

The approach presented by Bijlsma et al. [69] derives Cyclo-static Data Flow (CSDF) graphs (see section 3.1.2) from sequential code. Code blocks use explicit read and write operations to synchronize with each other. Each code block is based on nested loops, typically used for indexing arrays and matrices. Synchronization between these code blocks is done via circular buffers where a *read window* denotes that part of the buffer that can be safely read by the reading code block and the *write window* denotes that part of the buffer where the writing code block can safely write its outputs. The windows are updated by both code blocks by using an *acquire* (for increasing the window size such that buffer space can be accessed) and a *release* instruction (for releasing that part of the window that is not required anymore, such that the buffer space can be reused by the other code block). By analyzing the buffer access patterns, a minimum buffer size can be calculated such that a required throughput or latency can be guaranteed [69].

Hansson proposed an integrated approach for the definition of an architecture and an application [70]. This approach is based on *use-cases*, which identify a combination of applications that can be executed simultaneously. From the total set of defined use-cases, an architecture is composed that consists of enough resources (both in terms of processor capacity, memory and interconnect bandwidth) such that a guarantee can be given on the real-time performance of any application contained in the use-cases.

3.1.1.2 Manual approach

Another design approach is the manual approach, where no automatic tools are used for the optimization of the design. In such approaches, typically simulations are based on a combination of Matlab and Simulink [71] for the functional testing. Although the composition of blocks in a Simulink environment can be done quickly, it is quite hard to apply transformations to the application or the targeted architecture. Therefore, even a small modification in the structure of a Simulink application requires a lot of manual work. The automatic identification of parallelism in the code is hard and thus, parallelization still needs to be done by hand [72]. This increases the risk of human errors and moreover, it is very time consuming to create and maintain. Graphical programming enables a very structural approach. However, it also limits the design flexibility, because the interface of a block (which consists of all input and output ports) needs to be defined before the block is implemented. A small modification of the interface of a block may require all ports of that block to be reconnected to other blocks.

The Ptolemy project studies design, modeling and simulation of concurrent real-time embedded systems [73]. It provides a framework for system simulation and

focuses on experimenting with various Models of Computation (MoCs) and architectural designs. The modeling is based on graphical programming, with Java as the underlying language for implementation of blocks. Ptolemy focuses on diagrams showing composition and interaction between blocks, similar to Matlab/Simulink. The provided Models of Computation make Ptolemy more useful as a tool for architectural design than Matlab/Simulink.

The Y-chart approach, presented by Kienhuis [74], is a methodology to provide designers with quantitative data obtained by analyzing the performance of architectures for a given set of applications. The approach includes tools that indicate which parameters (for example, the number of function units in a processor or the instruction set) should be fine-tuned to improve the application performance. Applying these improvements, however, has to be done manually. In multiple steps, the hardware design is then optimized such that a better utilization can be obtained when executing applications from the given set. The approach is used for designing a streaming DSP architecture using data flow principles. A programming framework implementation of this approach is the Y-chart Application Programmer's Interface (YAPI) [75]. YAPI can be used to implement data flow models, describing structural properties of the application. Such models can be generated with SDF for Free (SDF3), a tool for generating random Synchronous Data Flow (SDF) graphs that have a structure similar to typical DSP applications [76]. The tool does not include functional simulation; however, its SDF graphs can be exported to a YAPI compatible format such that YAPI can be used for further simulation.

Another design flow was proposed within the 4S project [4]. It is based on a separate design-time part and a run-time part. A graphical representation of the flow is given in Figure 3.1. The targeted hardware architecture was an MPSoC consisting of existing cores (for example, GPPs, DSPs, coarse-grain and fine-grain reconfigurable processors and hardwired accelerators), for which existing tooling is available (shown at the third layer of blocks in the figure). An existing application, written in C, is manually partitioned in smaller processes and compiled to the different processor types (visualized in the second layer). For each of these processor types, the compiled code is simulated and analyzed using performance estimation tools and power estimation tools (see the fourth layer of blocks). Based on the results of these tools, for example execution times and energy consumption of the compiled code when executed on a specific processor type, the application designer may want to start a new iteration of partitioning, compilation and analysis. Once the results of the application partitioning are accepted by the designer, the processes are prepared for integration in a Real-Time Operating System (RTOS) called Operating System for Real-Time Embedded Systems (OSYRES) [77]. In the run-time part of the design flow (the lower part of Figure 3.1), the RTOS allocates processes to processors by supporting run-time mapping (see section 3.1.3), based on user demands, power constraints and Quality of Service (QoS) requirements posed by the application.

3.1.2 Data flow modeling techniques

Due to their regular behavior, streaming applications can be modeled well using data flow models. The input, output and intermediate streams consist of a long sequence

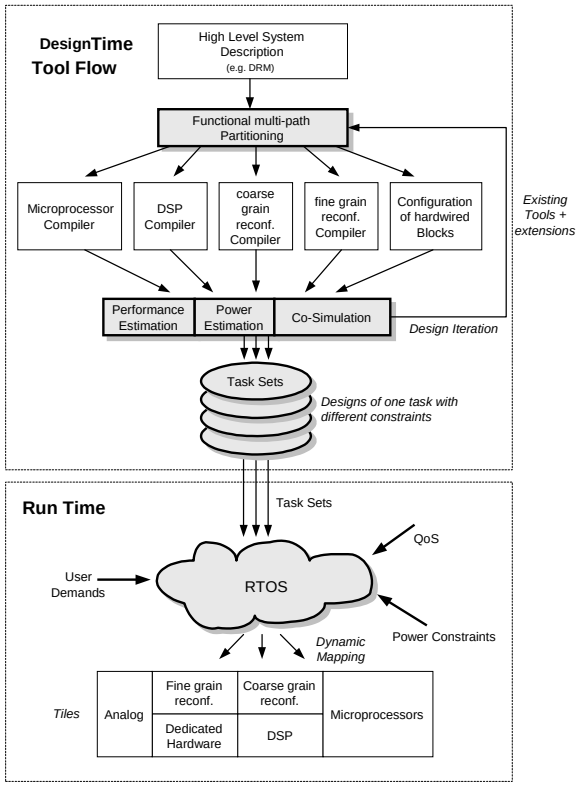


Figure 3.1 – 4S project mapping flow from application to hardware

of data on which operations are performed. By modeling the application as a graph consisting of nodes (the kernels) and edges (data dependencies between the kernels), the operations performed on the streams and the communication between kernels are made explicit. By looking at the data dependencies between the nodes, functional parallelism in the application can be identified [78]. From these data dependencies, a schedule can be calculated for the execution of each of the kernels, such that required input data is available when a kernel is executed. It is also possible to make a schedule for executing nodes on multiple processors, exploiting the processing capacity provided by these processors. A formal definition used to calculate such multi-processor schedules was proposed by Kahn [79].

3.1.2.1 Kahn Process Network

In a so-called Kahn Process Network (KPN), kernels are translated to *processes* and data dependencies between kernels are made explicit as communication channels between these processes. A process is executed by a *computing station* and communication between processes is done via *communication lines* between the computing

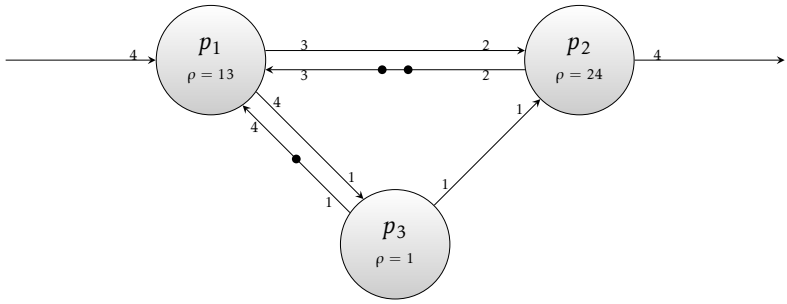


Figure 3.2 – SDF model of an application.

station on which they are mapped. For the execution of the process, the computing station is not limited to a finite amount of memory. At any given time, a computing station is either executing its process or halted due to a blocking read operation on one of its input channels. Data produced on a channel can have an arbitrary, but finite, delay before it can be read by the next process. Within the channel, data are transported in FIFO order. KPN channels can contain an arbitrarily large (possibly infinite) number of data samples.

If multiple processes depend on each other's output and are currently blocked due to a blocking read operation, the application is deadlocked. To overcome the risk for deadlock, a more restrictive data flow model was introduced for which deadlock analysis can be done. This model is presented in the next sections.

3.1.2.2 Synchronous Data Flow

SDF is a more restrictive model than KPN, as it introduces *firing rules* that define the conditions for the execution of a process [80]. A firing rule for a process defines per input channel how many *tokens* the process consumes from that channel, where a token can either store data or events produced by another process to that channel. While KPN allows read operations on input channels and write operations to all output channels at any time during the execution of a process, an SDF process instantaneously consumes its tokens from all input channels before its execution starts and instantaneously produces its tokens to all output channels upon finishing the execution. The interval between the start of an execution and the end of the execution is called the *execution time* ρ. Furthermore, in the SDF model, channels introduce zero latency on tokens such that a token produced on a channel can be consumed by the next process immediately. An example SDF application is given in Figure 3.2.

A finite FIFO buffer between a producing process p_1 and a consuming process p_2 can be modeled by a channel from p_1 to p_2 (denoting the flow of data from p_1 to p_2) and a channel in the opposite direction (modeling the free buffer positions). When analyzing the number of tokens in these channels, together with the consumption and production rates of both processes, an execution schedule of these two processes can be created [81, 82]. Using the analysis techniques presented in [83–85], the

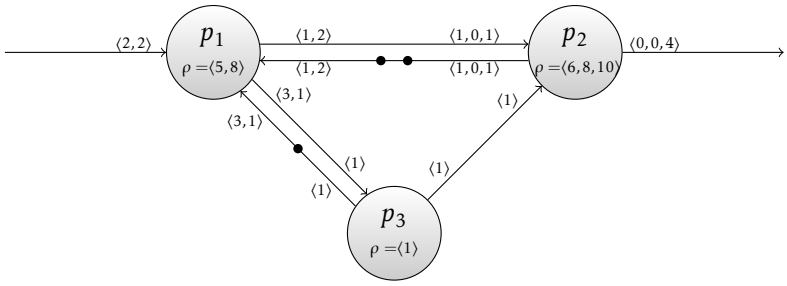


Figure 3.3 – CSDF model of an application. Different stages in the execution of a process are indicated in angle brackets, where the first number indicates the first stage (for token consumption, production and execution time).

minimum buffer capacity required to store the tokens communicated between these processes can be found. If the buffer size is chosen too small, deadlock can occur in the application.

3.1.2.3 Cyclo-static Data Flow

Since the SDF model requires all input tokens to be available before the execution can be started, it requires considerable buffer space for storage of the tokens as long as they are not consumed. Upon firing, all tokens are consumed at once and produced at the very end of the execution. Hence, the number of tokens in the input buffer grows until the process is fired. CSDF is a variant of SDF that enables more intuitive modeling of applications by allowing a process execution to consist of multiple sequential stages [86, 88]. A CSDF model can be translated into a SDF model [87]. For example, the CSDF application given in Figure 3.3 is equivalent to the SDF application given in Figure 3.2. Each stage behaves as an SDF process, as it has a fixed token consumption, a fixed token production and an execution time. The stages are executed in a round robin schedule, such that each stage is executed once per execution cycle. Using a CSDF model, token consumption and production can be divided over multiple stages, such that the maximum number of tokens stored in a buffer at any time during the execution is smaller than in the SDF version of the same application. Then, the minimum buffer sizes can be reduced while maintaining a performance compared to the SDF model. Next to estimation of buffer sizes, for both models it is possible to identify critical parts in the application that limit the performance in terms of latency or throughput [89].

3.1.2.4 Other data flow models

The data flow models presented here have a fixed schedule as they allow only a single firing rule per process. Therefore, the number of tokens consumed and produced per process per execution does not depend on the values stored inside the received tokens. As a result, a large class of algorithms can not be modeled using these techniques. For example, a Variable Bit Rate (VBR) decoder may have to read an arbitrary number

of bits before these bits can be successfully decoded. Wiggers [89] proposes a more flexible model that is capable of including data dependent token consumption and production rates. The Variable Phase Data Flow (VPDF) model is comparable with CSDF, as it supports multiple stages which are executed after each other. However, for each stage the token consumption and production rates may be determined by another process. Although this model is more expressive, we consider CSDF to be expressive enough in the context of this thesis.

3.1.3 Design-time vs. run-time mapping

The assignment of processes to processors and communication channels to the interconnect of the MPSoC can be done in many ways. As a matter of fact, obtaining the optimal mapping is an NP-complete problem [90]. Assuming all resources (processors, memory, and interconnect) of the MPSoC are available at run-time, the best possible mapping of an application to the MPSoC can be calculated at design-time. However, if a part of the MPSoC's processing or communication capacity is not known at design-time (for example, due to other resource sharing with running applications or due to malfunctioning hardware), a pre-calculated mapping may not be feasible, or it may result in a decreased application performance. Run-time mapping uses information about the MPSoC resource allocation as soon as the application needs to be mapped. If multiple applications are executed on one MPSoC simultaneously, different orders of startup of both applications may result in different mappings. In a design-time mapping based architecture, individual applications or fixed combinations of applications (called *use cases* [70]) are assigned to specific processor elements such that the real-time execution is always guaranteed. This, however, may result in a very specific MPSoC architecture for the selected set of applications, since the hardware usage is known on beforehand. For next generation Multi-Processor Systems-on-Chip with hundreds of processing elements in advanced CMOS technology, we expect that the probability of manufacturing faults will increase considerably. Therefore, the mapping calculated at design-time may become useless since the allocation may require specific processors which turn out to be faulty [152].

By using heuristics to calculate a sufficient feasible mapping at run-time, the restriction of fixed applications or fixed combinations of applications can be relaxed. Heuristics cannot give an optimal solution for an NP-complete problem, but lead to a solution that is close to the optimal solution. An efficient mapping can be made by choosing a processor type for each process to be executed (called *binding*), where that processor type can execute the process most efficiently [91]. Then, for each process, a processor of the chosen processor type is selected. For this selection, communication between the process and its source and destination processes is considered, such that the selected processor is close the processors that execute those processes. As a result, communication channels between processes are short, which results in low latency between processes and low energy costs for communication.

3.2 Mathematical programming based tool-flow

The design methodologies discussed in section 3.1 all assume imperative sequential code (for example an application written in C) or a fixed manual partitioning and synchronization of the application (for example into a set of parallel threads). For each of the proposed flows, this is mainly caused by the requirement that legacy code should be reusable. The imperative sequential code is analyzed to identify possibilities for parallel execution such that the application can be mapped on a multiprocessor architecture. In general, this is a very hard problem that has been studied extensively for many years [92, 93]. A typical DSP application, however, consists of mathematically defined kernels that are composed using block diagrams which are inherently parallel when pipelining is applied (called *task level parallelism*). Therefore, traditionally, first the mathematical definition is translated to a sequential language manually. Second, that implementation is analyzed by parallelizing compilers to find data dependencies such that independent instructions can be executed in parallel.

Figure 3.4 gives an overview of the design flow presented in this section, showing the relation between the different models, the simulation environment and the MPSoC platform. A mathematical specification of a streaming application is implemented in a custom language. Using transformation rules, the implemented application is partitioned into a set of parallel kernel operations. After analyzing the execution times of all kernels, communication between kernels is made explicit by modeling the application as an SDF application. This application can then be simulated by the simulation framework presented in this chapter or it can be mapped to an MPSoC.

3.2.1 Language construction

In particular streaming applications are often specified by mathematical equations. Mathematical equations define *relations* between operands. Since all relations exist simultaneously, a mathematical definition is parallel by nature [94]. Hence, it seems to be counter-intuitive to remove this parallelism by rewriting the mathematical equations to a set of sequential assignments and then trying to reconstruct the original relations such that parts of the algorithm can be executed in parallel. Therefore, we propose a mathematical programming based approach implemented in a functional language. In this thesis we use the functional language Haskell as a host language, which is used to construct a so-called Embedded Domain Specific Language (EDSL) [95]. A short overview of Haskell examples, useful for understanding the code listings in this section, can be found in section B.1. The grammar rules of the language are implemented using an Algebraic Data Type (ADT). A fragment of the language is shown in Listing 3.1. Here, an expression of type `Expr a` can consist of an `Add` of two expressions of type `Expr a`, or a `Sub` of two expressions of type `Expr a`, or can be a terminator of type `Const a`. Other expressions can be added to the grammar by adding additional lines. With these operators a very simple language can be defined as shown in Listing 3.1.

There are two ways to look at the EDSL. An application programmer uses the EDSL as a programming language, where the typing system prevents making errors.

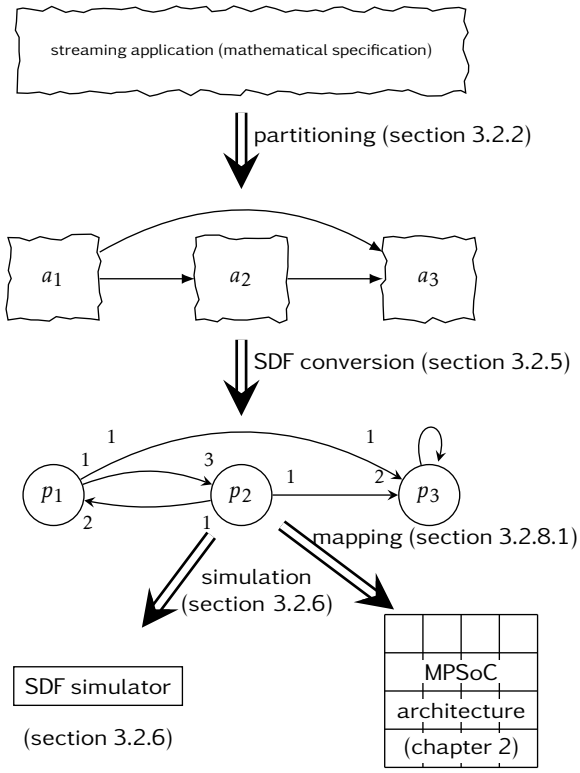


Figure 3.4 – Mathematic programming based tool flow

```
1  data Expr a
2          = Add (Expr a) (Expr a)
3          | Mul (Expr a) (Expr a)
4          | Sub (Expr a) (Expr a)
5          | Div (Expr a) (Expr a)
6          ...
7          | Const a
8          | Var   String
```

Listing 3.1 – Simple EDSL for DSP operations

Transformation tools and compilers use the EDSL as a data type (`Expr a`), which can be used to generate a modified version of the application or to translate the EDSL to instruction code. Any expression defined in this language is also an equation and can therefore be used for calculations. For example, a function of type `f :: Expr a -> Expr a` can be defined to transform the equation in the input expression to another equation. By using the constructors like `Add`, `Sub`, `Const`, such a transformation function can be applied to specific parts of the language using pattern matching.

To show the relation between the mathematical definition and our language

implementation, consider the following equation:

$$y = 2 * x + 15 \tag{3.1}$$

A straightforward implementation of a kernel with one output (indicated by `y`) and one input (indicated by `Var "x"`) is:

```
y = Add ( (Mul (Const 2) (Var "x")) (Const 15) )
```

3.2.2 Partitioning

Before this expression is evaluated, transformations can be applied to it. Since the expression is of type `Expr a`, any function of type `Expr a -> Expr a` can be applied to this input expression. Consider the `Add` operation, which is associative. It can be rewritten from $(a + b) + c$ to $a + (b + c)$ which is equivalent. This rewrite rule is implemented in our EDSL as follows:

```
1  assoc :: Expr a -> Expr a
2  assoc (Add (Add a b) c) = Add a (Add b c)
3  assoc (Mul (Mul a b) c) = Mul a (Mul b c)
4  assoc x                 = x
5
6  commut :: Expr a -> Expr a
7  commut (Add a b) = Add b a
8  commut (Mul a b) = Mul b a
9  commut x         = x
```

Listing 3.2 – Rewrite rules for associativity and commutativity

The `assoc` rewrite rule shown in Listing 3.2 is based on *pattern matching*, a commonly used technique in functional programming languages. Different cases of the function can be implemented via separate patterns, for example as shown in Listing 3.2 where one pattern is used to define the `assoc` transformation function when applied to additions (line 2), another pattern is used to match the transformation for multiplications (line 3) and all other language constructs are matched by the last pattern (line 4). Note that all three cases of the function `assoc` are of the same type (`Expr a -> Expr a`). Additional operators can be added to the basic language defined in Listing 3.1, without breaking the existing rewrite rules. Patterns are matched in the order as they occur; the first matching pattern is chosen. Therefore, if a new operator is added to the language, the rewrite rules can be implemented by adding a pattern for the new operator right before the general `x` pattern that matches all other operators. If the correctness of each of the cases of a rewrite rule is proven, its application to any expression in our language is guaranteed to be correct. The same rule can be applied recursively to the operands of an expression to transform the entire expression as shown in Listing 3.3. In this case, all non-associative operations in the language are skipped and the transformation is applied to their operands (for example in the case of a `Sub a b` operation).

```
1  assocRec :: Expr a -> Expr a
2  assocRec (Add (Add a b) c) = Add (assocRec a) (assocRec (Add b c))
3  assocRec (Mul (Mul a b) c) = Mul (assocRec a) (assocRec (Mul b c))
4  assocRec (Add a b)         = Add (assocRec a) (assocRec b)
5  assocRec (Mul a b)         = Mul (assocRec a) (assocRec b)
6  assocRec (Sub a b)         = Sub (assocRec a) (assocRec b)
7  assocRec (Div a b)         = Div (assocRec a) (assocRec b)
8  ...
9  assocRec (Const a)         = Const a
10 assocRec (Var   s)         = Var   s
11
12 commutRec :: Expr a -> Expr a
13 commutRec (Add a b) = Add (commutRec b) (commutRec a)
14 commutRec (Mul a b) = Mul (commutRec b) (commutRec a)
15 commutRec (Sub a b) = Sub (commutRec a) (commutRec b)
16 commutRec (Div a b) = Div (commutRec a) (commutRec b)
17 ...
18 commutRec x         = x
```

Listing 3.3 – Recursive rewrite rules for associativity and commutativity

By combining transformations, new composite transformations can be defined. An example of such a combined transformation is the distributivity rewrite rule, shown in Listing 3.4.

```
1  distrib :: Expr a -> Expr a
2  distrib (Mul (Add a b) c)         = Add (Mul a c) (Mul b c)
3  distrib (Add (Mul a c) (Mul b c)) = Mul (Add a b) c
4  ...
5  distrib x = x
```

Listing 3.4 – Rewrite rule for distributivity

The examples presented here are based on unary and binary operations on scalar values. Similarly, transformations can be defined for lists of values. A list is defined as a recursive structure, consisting of a list constructor `Cons` that contains two values: a value of type a indicating the head of the list and another list of type `List a` defining the rest of the list. The end of the list is represented by a `Nil` constructor. The definition of a list containing the integer values `[1, 2, 3]` is given at line 4 of Listing 3.5.

```
1  data List a = Nil
2              | Cons a (List a)
3
4  numbers1 = Cons 1 (Cons 2 (Cons 3 Nil))
```

Listing 3.5 – ADT definition of a list containing elements of type a

Usually, the shorthand notation `a:bs` is used instead of the constructor `Cons a bs`. A compact form of the `Nil` constructor is the notation `[]`. Using *higher order functions*, an operation can be executed on a list with a very compact notation [96]. The basic operation that can be used to construct any other function is the *fold*, which

exists in two variants: a left-associative variant (foldl) and a right-associative variant (foldr). Their definitions are given in Listing 3.6. Note the use of the shorthand notation (x:xs) at lines 3 and 7, defining a pattern for matching non-empty lists. Other higher order functions include map (which applies a given function to each element in a list, resulting in a new list of results) and the filter function (which uses a given boolean function to remove elements from the given list).

```
1  foldl :: (a -> b -> a) -> a -> [b] -> a
2  foldl f z []     = z
3  foldl f z (x:xs) = foldl f (f z x) xs
4
5  foldr :: (a -> b -> b) -> b -> [a] -> b
6  foldr f z []     = z
7  foldr f z (x:xs) = f x (foldr f z xs)
8
9  map :: (a -> b) -> [a] -> [b]
10 map f xs = [f x | x <- xs]
11
12 filter :: (a -> Bool) -> [a] -> [a]
13 filter c xs = [x | x <- xs, c x]
```

Listing 3.6 – Higher order list functions

3.2.3 Language usage and evaluation of expressions

An example of how the fold operator is used, is shown in Listing 3.7. It shows the evaluation of a summation of the elements of the list [1..5]. First, the fold is expanded recursively until the last element in the list has been accessed (line 9, where the remaining list equals []), which is then followed by the evaluation of the expression.

```
1  sum xs = foldl (+) 0 xs
2
3  sum [1..5]
4  = foldl (+) 0 [1..5]
5  = foldl (+) ((+) 0 1) [2..5]
6  = foldl (+) ((+) ((+) 0 1) 2) [3..5]
7  = foldl (+) ((+) ((+) ((+) 0 1) 2) 3) [4..5]
8  = foldl (+) ((+) ((+) ((+) ((+) 0 1) 2) 3) 4) [5]
9  = foldl (+) ((+) ((+) ((+) ((+) ((+) 0 1) 2) 3) 4) 5) []
10 = ((+) ((+) ((+) ((+) ((+) 0 1) 2) 3) 4) 5)
11 = ((+) ((+) ((+) ((+) 1 2) 3) 4) 5)
12 = ((+) ((+) ((+) 3 3) 4) 5)
13 = ((+) ((+) 6 4) 5)
14 = ((+) 10 5)
15 = 15
```

Listing 3.7 – Evaluation of a sum function, implemented by a higher order fold function

Lists can grow very long and therefore it may be needed to split them in multiple sub-lists for mapping them on a multi-processor architecture. If the sum operation in Listing 3.7 operated on a list containing values [1..1000000], this could be beneficial. Instead of calculating the sum via a single recursive expansion, the list

can be split in multiple parts that are summed individually, and the result of all sub-lists can be summed afterward (see Listing 3.8). This partitioning is correct because the sum operation only consists of additions, which are both commutative and associative.

```
1  sum [1..1000000]
2  = sum [ sum [1..100000]
3        , sum [100001..200000]
4        , ...
5        , sum [900001..1000000]
6        ]
7  = foldl (+) 0 [ foldl (+) 0 [1..10000]
8                , foldl (+) 0 [10001..20000]
9                , ...
10               , foldl (+) 0 [990001..1000000]
11               ]
12 = ...
```

Listing 3.8 – Partitioned sum calculation

The same partitioning can be adopted within the EDSL. First, the higher order functions for lists are rewritten. For example, the implementation for `foldr` and `foldl` in our language (implemented as `foldrE` and `foldlE`) is given in Listing 3.9. These folding functions operate on a list of values and return a expression tree where the values are added as `Const` leafs in the tree. Any expression built with the language constructs listed in Listing 3.1 can be represented as an abstract syntax tree, where the nodes in a tree define the operations and the leafs of the tree contain the constants or variables. After applying transformations to the syntax tree, a new valid syntax tree is returned. The tree can be evaluated by feeding it to the `evalE` function in Listing 3.9.

```
1  foldrE :: (Expr a -> Expr b -> Expr b) -> Expr b -> [a] -> Expr b
2  foldrE f z []     = z
3  foldrE f z (x:xs) = f (Const x) (foldrE f z xs)
4
5  foldlE :: (Expr a -> Expr b -> Expr a) -> Expr a -> [b] -> Expr a
6  foldlE f z []     = z
7  foldlE f z (x:xs) = foldlE f (f z (Const x)) xs
8
9  evalE :: Expr a -> a
10 evalE (Add a b) = evalE a + evalE b
11 evalE (Sub a b) = evalE a - evalE b
12 evalE (Mul a b) = evalE a * evalE b
13 evalE (Div a b) = evalE a / evalE b
14 ...
15 evalE (Const a) = a
```

Listing 3.9 – Higher order functions implemented for our EDSL

3.2.3.1 Example evaluation

Using the higher order functions, the example summation presented in Listing 3.7 can also be defined (see Listing 3.10). First, the definition of `sumE` is expanded to

a syntax tree. This tree can be evaluated by applying the evalE function, which recursively expands the syntax tree until it can evaluate the Const a values.

```
1  sumE xs = foldlE (Add) (Const 0) xs
2
3  sumE [1..5]
4  = foldlE (Add) (Const 0) [1..5]
5  = foldlE (Add) (Add (Const 0) (Const 1)) [2..5]
6  ...
7  = Add (Add (Add (Add (Add
8         (Const 0) (Const 1)) (Const 2)) (Const 3)) (Const 4)) (Const 5)
9
10 evalE (sumE [1..5])
11 = evalE (Add (Add (Add (Add (Add
12        (Const 0) (Const 1)) (Const 2)) (Const 3)) (Const 4)) (Const 5))
13 = (evalE (Add (Add (Add (Add (Const 0)
14        (Const 1)) (Const 2)) (Const 3)) (Const 4))) + evalE (Const 5)
15 ...
16 = (((((evalE (Const 0) + evalE (Const 1))) + evalE (Const 2)) +
17        evalE (Const 3)) + evalE (Const 4)) + evalE (Const 5)
18 = (((((0 + evalE (Const 1))) + evalE (Const 2)) + evalE (Const 3)) +
19        evalE (Const 4)) + evalE (Const 5)
20 ...
21 = (6 + 4) + evalE (Const 5)
22 = 10 + evalE (Const 5)
23 = 10 + 5
24 = 15
```

Listing 3.10 – Evaluation of a sum function within our EDSL

This example shows how the evaluation of a short list expands to a large expression tree. To avoid an explosion in the expansion of the expression tree resulting from the evaluation of a larger list, the list can be partitioned into smaller sublists that are expanded by different processors [157]. Each processor is then responsible for the evaluation of a part of the list. The partial results are sent to the processor executing the parent node, which performs the addition of the sublists. The partitioning of a list in a number of smaller lists can be done by applying the part function defined in Listing 3.11.

Which value of n is chosen for the part() function depends on the architecture. For example, if a processor can only store 10k of data and the list has length 38k, the input list has to be partitioned over at least 4 processors.

3.2.4 Example application specification

Typically, for streaming DSP applications, a specification is available that very accurately describes the desired operation. The specification usually consists of a block diagram with mathematical definitions for the requirements of the individual blocks. An example block is a FIR filter, which consists of a convolution operation with a set of filter coefficients that define the filter's behavior, for example a 20 dB attenuation low pass filter. The discrete-time FIR filter is mathematically defined as:

$$y[k] = (h * x)[k] = \sum_{i=1}^{N} h[i] \cdot x[k-i] \tag{3.2}$$

```
1  part :: Int -> List a -> List (List a)
2  part _ Nil = Nil
3  part n  xs = Cons a bs
4              where
5                a  = takeE n xs
6                bs = part  n (dropE n xs)
7
8  takeE :: Int -> List a -> List a
9  takeE _         Nil  = Nil
10 takeE o           _  = Nil
11 takeE n (Cons a bs) = Cons a (takeE (n-1) bs)
12
13 dropE :: Int -> List a -> List a
14 dropE _         Nil  = Nil
15 dropE o          bs  = bs
16 dropE n (Cons _ bs) = dropE (n-1) bs
17
18
19 sumE xs = foldlE (Add) (Const 0) xs
20
21 part 4 (sumE [1..1000000])
22 = ...
```

Listing 3.11 – Partitioning of a list within our EDSL

where N defines the filter order, h is the set of coefficients, $x[k]$ denotes the k^{th} input sample and $y[k]$ denotes the k^{th} filter output.

First, the *functional correctness* of individual kernels is tested by generating synthetic test patterns for which the result is known on beforehand. If all kernel tests have been performed successfully, the individual kernels can be combined and the resulting application should be tested again to check for correct interaction between kernels [159].

Implementation costs of a mathematical relation are not explicit and not always obvious. For example, a direct implementation of the filter of Equation 3.2 operating on the input data $x[k-1, k-2, \ldots, k-N]$ to calculate the output value $y[k]$ would require one read operation for each input sample, hence N read operations in total. For the next output value $y[k+1]$, the values $x[k, k-1, \ldots, k-N+1]$ are read. When analyzing the indices of the values read, it can be seen that there is an overlap in read operations on values $x[k-1, k-2, \ldots, k-N+1]$. Instead of reading the entire sequence of N elements for each calculation, $N-1$ elements can be stored in local memory and only the new value $x[k]$ needs to be read. Hence, the number of read operations is reduced by a factor N. However, by storing the elements in local memory, the stateless implementation of the FIR algorithm has been converted to an implementation with state. The introduction of state significantly reduces the necessary communication bandwidth due to *locality of reference* [97].

Consider the FIR filter from Equation 3.2. When introducing state, this equation

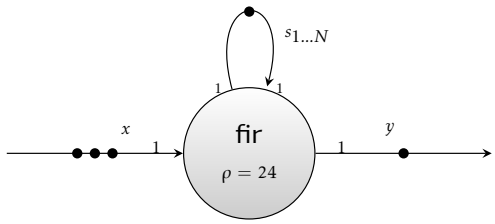


Figure 3.5 – SDF model of a FIR filter

can be rewritten as follows:

$$\begin{aligned} s_1[k] &= x[k] \\ s_i[k] &= s_{i-1}[k-1] \qquad 1 < i \leq N \\ y[k] &= \sum_{i=1}^{N} h_i \cdot s_i[k] \end{aligned} \tag{3.3}$$

where the vector s indicates the state, which consists of N intermediate results (s_i) that are used for the calculation of the output y. The recurrent relation between s_i and s_{i-1} (shown at the second line in Equation 3.3) can be implemented very efficiently by a shift register. Although the manual state derivation seems trivial in this example, in general an automated approach is much harder. This requires an advanced dependency analysis in order to obtain an efficient solution. If the state is made explicit, the FIR filter can be described with an SDF model as given in Figure 3.5.

When state has been introduced, the communication bandwidth is reduced. However, the computational costs are not reduced, so executing the algorithm in real-time on a single tile processor may still be infeasible. Therefore, the execution of the mathematical operations must be separated in smaller kernels which can be mapped on multiple tile processors, such that the throughput of the architecture becomes high enough to guarantee real-time execution of the application. If the application is mapped on multiple processors, the communication between these processors becomes visible. To illustrate the increase of communication costs when partitioning, Equation 3.3 can be rewritten as follows:

$$\begin{aligned} y[k] &= \sum_{i=1}^{N} h_i \cdot s_i[k] \\ &= \sum_{i=1}^{\frac{N}{2}} h_i \cdot s_i[k] + \sum_{i=\frac{N}{2}+1}^{N} h_i \cdot s_i[k] \end{aligned} \tag{3.4}$$

Note that the rightmost sum shown in Equation 3.4 requires the state value $s_{N/2+1}[k]$. Using Equation 3.3, we find $s_{N/2+1}[k] = s_{N/2}[k-1]$, which is only present in the

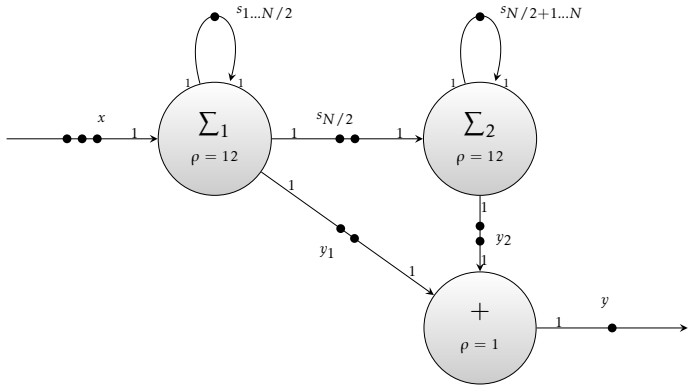


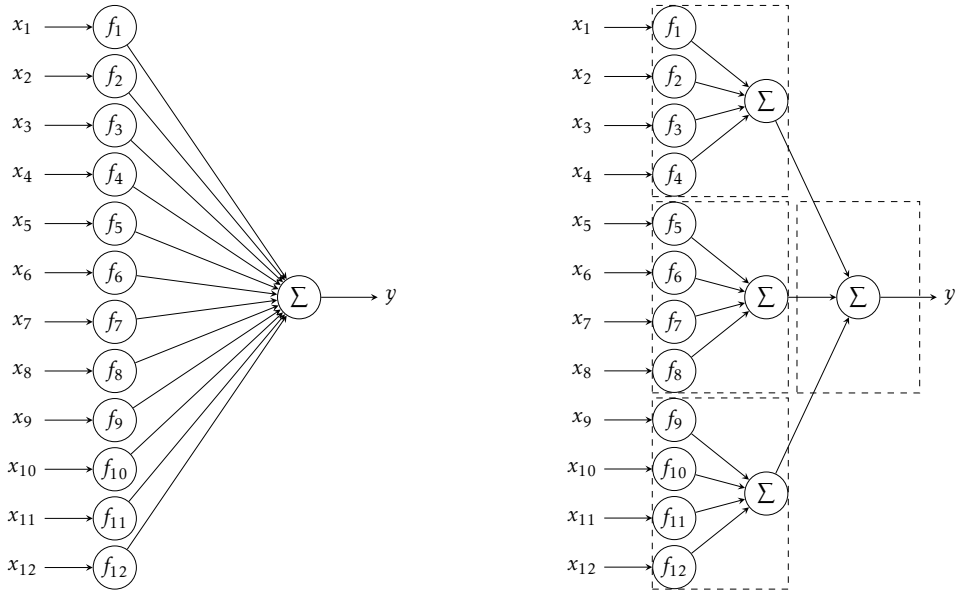
Figure 3.6 – SDF model of a partitioned FIR filter, where the execution times of the partial summations is reduced.

leftmost sum shown in Equation 3.4. Hence, if both sums are mapped on different tile processors, this intermediate state value has to be communicated between the tile processors. Also note that the rightmost sum in the equation is the same as the original FIR filter equation. The result of this partitioning is shown in Figure 3.6.

The FIR example shows a stepwise approach for partitioning one process into multiple processes. This approach is done semi-automatically, as manual input is still required to decide which rewrite rule should be applied next. For example, one still has to specify the number of tile processors to be employed or the amount of operations that can be mapped onto one processor simultaneously. With a predictable processor and a predictable NoC, partitioning can be done automatically.

Figure 3.7(a) shows the calculation of $y = \sum_{i=1}^{12} f_i(x_i)$. In this example, 12 inputs are read and a certain function f is applied to each of them. If the computational complexity of these functions is sufficiently high, they have to be calculated by different processors. In this case, the summation has to be done by another processor. However, adding all results simultaneously may be impossible, for example because of the communication bandwidth required. Then, the summation can be partitioned in multiple smaller summations, as shown in Figure 3.7(b).

Such partitioning requires knowledge about the targeted hardware architecture. However, if the required throughput is known, and the amount of clock cycles required for the execution of the function f on the targeted core is known, and if the communication bandwidth to the NoC is known, this type of partitioning can be done automatically. For example, the Montium TP presented in section 2.1.1.6 has a predictable performance for most algorithms and the Networks-on-Chip used in the Annabelle MPSoC has predictable throughput and latency. Therefore, the partitioning can be done automatically for a number of applications.



(a) Operation tree consisting of a single large adder with 12 inputs

(b) Operation tree consisting of multiple small adders with maximally 4 inputs

Figure 3.7 – An example operation tree with a large adder, that is partitioned into smaller adders.

3.2.5 Composition of the dataflow model

After the decomposition of an application into processes, the processes are mapped onto tile processors. Typically, each tile processor is capable of executing only a few processes. In order to have the entire application meet its real-time constraints, all processes and processor schedules need to meet real-time constraints [83, 89]. Therefore, guarantees need to be given to be sure none of the processes can endanger the real-time execution behavior of the application.

The correctness of transformation functions (expressed in terms of functions on our EDSL) can be proven; all correct transformations preserve the functional behavior of an application. However, errors that exist in the original application remain in the partitioned application. Additionally, the temporal behavior of an application will change after partitioning. To determine the correctness of the temporal behavior, an SDF model is compiled from the partitioned application.

As explained before, the amount of communication within an application can be reduced by introducing local state. In the dataflow model, this local state is added by adding a self edge to the process such that the process reads its state token from this self edge, performs the operation and then writes the new state token back to the self edge. Interaction between processes is made explicit by adding communication channels.

We define the following terms:

Definition 8 (Token)**.** A token is a typeless container for any kind of data.

Definition 9 (Stream)**.** A stream is a sequence of tokens.

Definition 10 (Channel)**.** A channel, defined by a record (`chid,from,to,cont`), connects to a port of a source process (indicated with `from`) and to a port of a destination process (indicated by `to`). The channel has an identifier `chid` and contains a stream of tokens `cont`.

Definition 11 (Port)**.** A port of a process models a unidirectional interface from that process to other processes.

Definition 12 (Process)**.** A process v is defined as a record (`pid, f, s, is, os, wcet`), with identifier `pid`, function `f()`, state `s`, token consumption rule `is`, token production rule `os` and the worst-case execution time `wcet`.

Definition 13 (Token consumption rule)**.** The token consumption rule `is` for a process v with m input ports is described as a list of tuples $(i, c_{v,i})$, where $c_{v,i}$ denotes the integer number of tokens consumed from the i^{th} input port of v ($c_{v,i} \geq 0$ for all $0 \leq i < m$) per execution of v.

Definition 14 (Token production rule)**.** The token production rule `os` for a process v with n output ports is described as a list of tuples $\left(j, p_{v,j}\right)$, where $p_{v,j}$ denotes the integer number of tokens produced on the j^{th} output port of v ($p_{v,j} \geq 0$ for all $0 \leq j < n$) per execution of v.

Listing 3.12 shows the definition of a process, as used in the simulator framework.

```
1  data Process
2         = PS { pid   :: ProcessID
3              , f     :: Fun
4              , state :: State
5              , is    :: [(Port,Int)]
6              , os    :: [(Port,Int)]
7              , wcet  :: Int
8              }
```

Listing 3.12 – Process type definition

Since our goal is to simulate any application graph $G = (V, E)$, consisting of a set of processes V and a set of channels E, the simulation framework needs to be flexible and generic to operate on arbitrary large sets. A graph could be constructed using advanced network combinators that connect processes and channels, for example as used in S-Net [98]. However, we prefer using a list representation for these sets, because it allows for simple construction of graphs and enables applying higher order functions like `map` to the set of processes and to the set of channels. In order to be able to execute the processes in V and to transport tokens via the channels in E, the simulator framework requires knowledge about the type of the function `f` in a process and its state. Thus, it needs to know what type of input and output tokens are consumed and produced by the process.

```
1  f :: (a, b) -> c
2  f1 :: (Int, Char) -> String
3  f2 :: (Double, Int) -> (Char,Int)
4
5  f' :: [Stream] -> [Stream]
```

Listing 3.13 – Generalization of function types

In Haskell it is not possible to create a list containing values of different data types, for example a list of functions with different input types. However, if all processes use a common type for their functions and states, they could be joined in one list. As an illustration, assume a function `f` of type `(a,b) -> c` (see Listing 3.13), where a, b and c can be chosen arbitrarily. Such a function is called a *polymorphic function* as its types a, b and c are not bounded. The two functions `f1` and `f2` comply with the type definition of `f` (for `f1`, choose `a=Int`, `b=Char` and `c=String` and for `f2` choose `a=Double`, `b=Int` and `c=(Char,Int)`). As mentioned above, in Haskell, it is impossible to create a list containing `f1` and `f2` because their types cannot be unified to one combined type. However, Haskell has the possibility of using dynamic types to pack an item of any data type in the so-called *Dynamic* type, which internally preserves the value and type of the item stored. Then, it is possible to create a list of dynamics (`[Dynamic]`) because all values in the list are of the same type. In the simulator, this Haskell property is used for storing data in *Token*s and process *State*, such that process functions can be defined as a function with type `[Stream] -> [Stream]`, where a `Stream` denotes a list of tokens received on an input of the function. The explicit conversion functions for packing and unpacking function arguments to and from tokens are depicted in Listing 3.14. The `fromToken()` function is used to open an abstract `Token` and returns the contents of the token, and the `toToken()` function is used to convert any data of type `v` to a token.

```
1  fromToken :: (Typeable v) => Token -> v
2  fromToken t = fromJust (fromDynamic t)
3
4  toToken :: (Typeable v) => v -> Token
5  toToken t = toDyn t
6
7  type Token  = Dynamic
8  type Stream = [Token]
```

Listing 3.14 – Token conversion functions

A channel is connected to one output port of a process and one input port of a process (see Listing 3.15). These connections are defined by a tuple containing the connected process and its port number, where the channel uses such a tuple for both the source connection (`from`) and the destination connection (`to`). A channel can be identified using a `ChannelID`, and tokens currently buffered in the channel are stored in a `Stream` field. Different types of tokens can be stored in the channel, similar to the generalization of process functions. This makes it possible to maintain a list of

all channels in the application.

```
1  data Channel
2         = C { chid :: ChannelID
3             , from :: (ProcessID, Port)
4             , to   :: (ProcessID, Port)
5             , cont :: Stream
6             }
```

Listing 3.15 – Channel type

The resulting data structure, containing the application graph $G = (V, E)$, can now be used as input for the dataflow simulator. In section 3.2.6.1, the application graph definition will be discussed in more detail.

3.2.6 Simulation

To verify that, after the partitioning, the entire application meets its real-time constraints, we have developed a simulator that performs functional simulation of an SDF application. Communication between two processes is done via explicit token production and explicit token consumption operations. A global simulation clock triggers the execution of all processes. First, the firing rule for a process is checked to see whether there are enough tokens available on the input edges, and in case of success the function associated to that process is executed. After an integer number of simulator clock periods (the execution time of the process), the process is finished. At any time during the simulation, the simulator has a state in which the list of processes, the list of channels, and the execution state of all processes is stored (shown in Listing 3.16).

```
1  data SimState
2         = SS { ps     :: [Process]
3              , cs     :: [Channel]
4              , exec   :: [ProcessExec]
5              }
```

Listing 3.16 – Simulator state type definition containing all application information

The process execution state indicates whether a process is currently `Waiting` to be fired, `Running` or has `Finished` its computation (see Listing 3.17). When the process is fired, its execution state is set to `Running` n, where n indicates the execution time of that process. Also, the input tokens are read from the channels and the function associated to the process is evaluated, using the read tokens as arguments. The evaluation of this function is done immediately and the new state and output tokens are stored in a temporal buffer that is part of the `ProcessExec` state used by the simulator (see Listing 3.16 and Listing 3.17). After each simulation cycle, the `Running` counter is decreased, until it reaches the state `Running 0`. Then, the `ProcessExec` buffer is flushed to the channels connected to the output ports and the process state is set to `Finished` and a new firing can be started.

```
1  type ProcessExec = (Exec, Buffer)
2  type Buffer = [Stream]
3
4  data Exec
5          = Waiting
6          | Running Int
7          | Finished
```

Listing 3.17 – Process execution state type definition

The simulation of one process iteration is shown in Listing 3.18. The updateProcess function operates on the tuple (v, pe), where v indicates the process that is updated and pe refers the current ProcessExec state information of v. One update results in a 4-tuple (v, pe', rds, wrs) containing the original process information v, the updated ProcessExec state pe', the consumed tokens rds and the produced tokens wrs.

```
1  -- Process is waiting
2  updateProcess s (pr, (Waiting,_))
3          | enabled   = ( pr', (Running (wcet pr), buf), rds, [] )
4          | otherwise = ( pr,  (Waiting,           []),  [],  [] )
5          where
6
7                  -- Check firing rule
8                  enabled = tokensAvailable s (is pr)
9
10                 -- Read tokens and execute function
11                 rds = readTokens s (is pr)
12                 (fs',buf) = (f pr) (state pr) rds
13
14                 pr' = pr {state = fs'}
15
16 -- Process just executed its last cycle
17 updateProcess s (pr, (Running o, buf)) = updateProcess s (pr, (Finished, buf))
18
19 -- Process is currently executing
20 updateProcess _ (pr, (Running n, buf)) = ( pr, (Running (n-1), buf),[],[])
21
22
23 -- Process just finished
24 updateProcess s (pr, (Finished, buf))
25         = ( pr', pe', rds', wrs ++ wrs' )
26         where
27           wrs = fromBuffer buf
28                 ( pr', pe', rds', wrs') = updateProcess s (pr, (Waiting, [] ))}
```

Listing 3.18 – updateProcess function (in pseudo-Haskell code)

The simulation model for the entire application works as follows. On each global simulation clock, one time step is simulated (see Listing 3.19). For all processes, this involves an update of one time step (line 5). The process update results in a new process state, a list containing the token consumption per input port, and a list containing the token production per output port during that update step (p_i, used at lines 8, 9, 12 and 13). After updating a process, the consumption of tokens from the input channels associated to the input ports of the process is performed (line 16).

Next, the produced tokens are put in the corresponding channels connected to the process output ports (line 17). The simulator state is updated and returned for the next step (line 20). This sequence of steps is repeated n times (see line 2).

```
1  step n ss
2          = step (n - 1) ss'
3          where
4                  -- Execute all processes
5                  p_i = map (procExecute ss) ( zip (ps ss) (exec ss) )
6
7                  -- Extract next state
8                  ps'   = [ p | (p,_,_,_) <- p_i]
9                  exec' = [ e | (_,e,_,_) <- p_i]
10
11                 -- Create lists of read and write operations to be performed
12                 consumed_tokens = concat [ i | (_,_,i,_) <- p_i]
13                 produced_tokens = concat [ o | (_,_,_,o) <- p_i]
14
15                 -- Apply read and write operations to channels
16                 cs_rd = foldl readFromChannel (cs ss) consumed_tokens
17                 cs_wr = foldl writeToChannel   cs_rd  produced_tokens
18
19                 -- Update simulation state
20                 ss' = ss { ps = ps', exec = exec', cs = cs_wr }
```

Listing 3.19 – Evaluation of one simulation clock (in pseudo-Haskell code)

The data structure containing the initial simulator state, is called the application structure and is discussed in more detail in the next section. The implementation of functionality that is executed within one process is discussed in section 3.2.6.2.

3.2.6.1 Application structure

A fragment of an example application graph $G = \{V, E\}$ as used in the simulator is displayed in Listing 3.20. The listing shows the data structure of graph g (line 1) which consists of the list of processes vs and the list of channels es. Each process v in vs (line 2) is defined by a PS record, as introduced in Listing 3.12. The PS record contains a function field f (implemented by the `sum_1` partial summation function shown at line 3 in the example of Listing 3.20), the initial state of that function (line 4, here set to 0), and the token consumption rule `is` and production rule `os` (lines 5 and 6 of Listing 3.20, respectively). For this example, per execution of the function `sum_1` from input port 0 a total of 2 tokens is consumed, and on both output ports 0 and 1 a single token is produced after 4 time steps. A list of the channels between processes in the application graph is denoted by E (displayed as es in Listing 3.20 at line 9). A channel e is defined as a C record (see Listing 3.15) containing the channel's identifier, the output port of the source process that produces tokens into the channel, the input port of the destination process that consumes tokens from the channel and a list of tokens `cont` currently present in channel e (line 12 of Listing 3.20).

A graphical representation of the application structure within the simulator framework is shown in Figure 3.8. Here, the squares indicate processes, the pointy blocks on the boundary of the squares indicate the input and output ports, and the function associated to the process is displayed below the process rectangles. Per

```
1  g = SS { ps = vs, cs = es, exec = [(Waiting,[]),...] }
2  vs = [ PS { pid   = 1
3            , f     = sum_1
4            , state = 0
5            , is    = [(0, 2)]
6            , os    = [(0, 1),(1, 1)]
7            , wcet  = 4 }
8       , PS { ... } ]
9  es = [ C { chid = "c0"
10           , from = (1, 0)
11           , to   = (2, 0)
12           , cont = Stream []
13           }
14      , C { ... } ]
```

Listing 3.20 – Application data structure example as used in the simulator

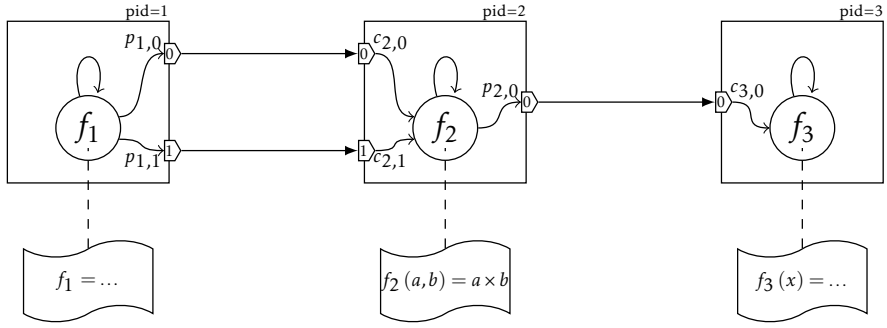


Figure 3.8 – Internal representation of an application within the simulator

port, the token consumption and production rates are displayed (for example, $c_{2,0}$ denoting the token consumption of process 2 from its 0^{th} input port). The self-edge from a function to itself denotes the explicit state.

3.2.6.2 Process implementation

Since the simulator framework takes care of the process firing and the communication between processes via a channel connected to their ports, the process' function implementation only has a local memory map (or stack) in which the input tokens are stored. As soon as there are enough tokens available on the input channels, they are moved to the local memory map for the process and the function can be executed using that local memory map.

An example process implementation in the simulator framework is given in Listing 3.21. The example shows a FIR filter, as presented before in section 3.2.4. At lines 4 and 5 of Listing 3.21, the input values x, and current state (n, h, s) that contains the number of taps, the coefficients and the intermediate summation results, are obtained from the `fromToken` function. These support functions are used to unpack the tokens and copy the contents to the local memory. The FIR filter operation itself is defined by lines 7 and 8. A part of the state contains the delay register

```
1  fir :: (State, [Stream]) -> (State, [Stream])
2  fir (st, [ Stream [ t ] ]) = (st', [ Stream [ t' ] ])
3    where
4      x = (fromToken t) :: Int
5      (n, h, s) = (fromToken st) :: (Int, [Int], [Int])
6
7      s' = [x] ++ take (n-1) s
8      y = sum ( zipWith (*) h s )
9
10     t' = toToken (y :: Int)
11     st' = toToken (n, h, s')
```

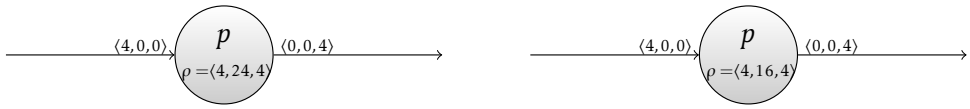
Listing 3.21 – SDF function type definition and example n-taps FIR filter implementation

contents. As explained in Equation 3.3, these delays are implemented using a shift register. The shift functionality is obtained by adding the new input `x` at the front and dropping the last element of the list of delayed elements `s` (line 7). The filter output `y` is obtained by applying an element-wise multiplication (using the `zipWith (*)` function) of the delayed elements `s` and the list of coefficients `h`, followed by an accumulated summation of the resulting list using the predefined function `sum` (line 8). The output `y` of the filter as well as the new state (n,h,s') are stored in a token using the `toToken` function and returned as new state output. Lines 10 and 11 show the type casting and storage of the filter output and the new local state in tokens `t'` and `st'`. Note that for this implementation, the FIR filter was not partitioned to smaller processes. The implementation shown is identical to Equation 3.3.

3.2.7 Testing

Due to the strong typing system used in Haskell, type errors appear very early when designing an application [99, 100]. Therefore, possible bugs manifest themselves during the design of individual processes. When combining these processes to an application, a different type of errors is likely to occur. For example, if tokens transmitted via a stream contain different typed data, the receiver may have problems with the interpretation of the tokens. Moreover, if two communicating processes use different ordering of a data stream, no typing errors are made but the received data is interpreted differently than how it was sent. Another problem may be in the synchronization of multiple processes: cyclic connections between two or more processes may lead to deadlock if the buffer capacities are insufficient. This problem can be avoided by applying buffer capacity analysis techniques, for example as presented by Wiggers [89].

The simulation framework includes a Graphical User Interface (GUI) that is generated from the `SimState` application specification. It shows the structure of the application by drawing the SDF model. The contents of tokens stored in channels can be displayed, such that the functionality of the processes can be verified. Applications may be either defined using parameters (for example, when several copies of one process have to run in parallel) or their structure can be modified afterwards by applying transformations to the application. In both cases, the GUI shows the application structure and token flow.



(a) CSDF variant for block-mode (b) CSDF variant for streaming-mode

Figure 3.9 – CSDF equivalents for both operations modes (assuming maximum consumption/production of 1 token per cycle)

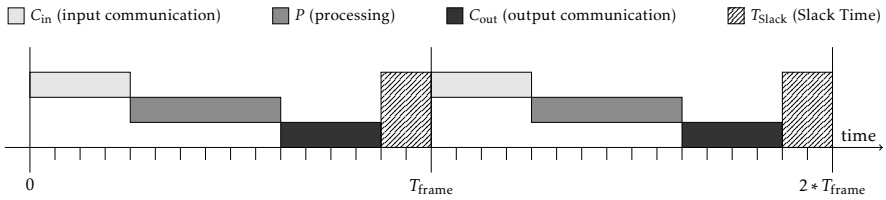
3.2.8 Performance of communication modes

Once the application is partitioned and tested, it is mapped to an MPSoC architecture. Hence, the processes in the application are mapped on processors and the communication channels between processes are mapped on the interconnect between these processors [101]. The SDF and CSDF models used in this chapter assume zero latency in the channels. However, when these channels are mapped on the physical interconnect, a certain latency is introduced due to routing and arbitration. Therefore, the mapping of the application to the hardware platform may cause a different application behavior in terms of time, while it still shows identical behavior in terms of functionality. However, it is possible to model the latency and throughput limitations introduced by the interconnect in the application [89].

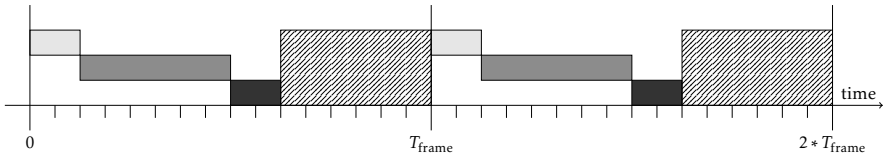
The influence of the operation mode (block-mode or streaming-mode) of a process and the mapping of its input and output streams on the NoC via the NI on the data flow can be made explicit by using a CSDF model. For example, consider the SDF process that is shown in Figure 3.2. For the block-mode operation, all input tokens must be loaded via a DMA load transaction before the execution can be started. Hence, an additional stage before the execution stage is added to model the token consumption. Similarly, the output tokens are produced in an additional stage after the execution stage, which models a DMA retrieve transaction to read the result. The resulting CSDF model of a block-mode is shown in Figure 3.9(a). It consists of three stages, where the first stage has an execution time of 4 cycles, during which 4 input tokens are read. In the second stage, the function associated to the process is executed (in 24 cycles) and its results are stored. The third stage, where 4 result tokens are produced, also has an execution time of 4 cycles. In total, this implementation lasts for 32 cycles.

In case of streaming-mode operation, a part of the execution stage is combined with the token consumption stage. The consumption is done exactly as fast as in the block-mode example of Figure 3.9(a) (4 cycles). After the token consumption stage, the execution is continued in 16 cycles during the second stage. Then, the result tokens are produced in the last stage simultaneously with a part of the execution (again in 4 cycles), such that the total token consumption and production are equal to the block-mode version but the total execution time of all 3 stages is reduced from 32 to 24 cycles. Effectively, this reduces overhead in cycles due to execution being halted during communication, as the streaming-mode exploits concurrency between communication and computation during the communication stages.

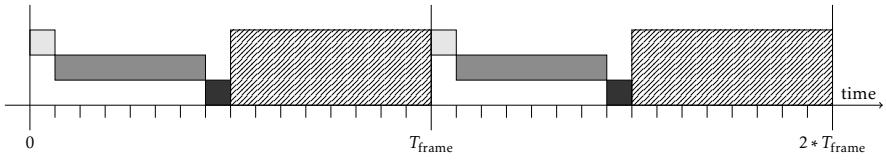
In this example, the token consumption and production rates are assumed to be 1



(a) CSDF schedule when communication is limited to 1 word per clock cycle (single channel DMA transactions)



(b) CSDF schedule when communication is limited to 2 words per clock cycle (dual channel DMA transactions)

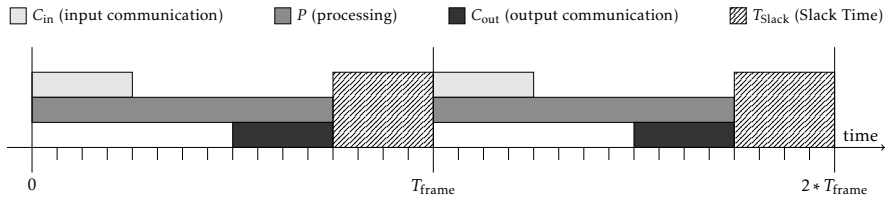


(c) CSDF schedule when communication is limited to 4 words per clock cycle (4-channel DMA transactions)

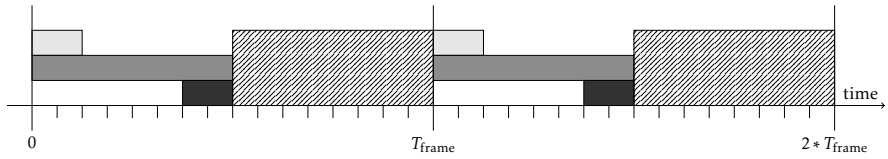
Figure 3.10 – Three possible CSDF schedules for block-mode operation of a process with $\rho = 24$, $C_{in} = 4$ and $C_{out} = 4$. The effect of increasing the number of channels for DMA transactions is clear: lower bandwidths result in longer execution times of the communication stages.

token per cycle, where each token contains one word. However, if multiple words can be stored in a token with an identical token rate (or if multiple tokens can be consumed per cycle), the execution time of the token consumption and production stages may be decreased. An illustration is given in Figure 3.10, which shows how the stages of a CSDF process (operating in block-mode) could be scheduled for different consumption and production rates. The lightest blocks in the upper part of each picture represent input data transfers (indicated by C_{in}), the middle gray blocks represent the processing (indicated by P) and the darker blocks in the lower part show output data transfers (indicated by C_{out}). The striped area indicates the remaining time in a frame, called slack time (T_{slack}). As can be seen, the communication times are reduced with a larger bandwidth for the DMA transactions, resulting in a larger slack time.

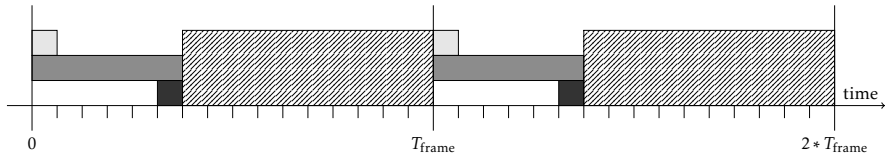
The same figures can be drawn for a streaming-mode implementation of the process. Figure 3.11 shows the CSDF schedule of the same process, operated in streaming-mode operation. Here, the number of channels used to stream input tokens into the process is also increased. The streaming-mode implementations



(a) CSDF schedule when communication is limited to 1 word per clock cycle (single channel streaming communication)



(b) CSDF schedule when communication is limited to 2 word per clock cycle (dual channel streaming communication)



(c) CSDF schedule when communication is limited to 4 word per clock cycle (4-channel streaming communication)

Figure 3.11 – Three possible CSDF schedules for streaming-mode operation of an SDF task with $\rho = 24$, $C_{in} = 4$ and $C_{out} = 4$. The schedules shown represent best-case situations where communication and processing can be done fully in parallel.

enable longer slack times than their block-mode counterparts with comparable communication bandwidth.

3.2.8.1 Run-time Execution

The previous sections dealt with the tool flow's design-time part. For the run-time part, we rely on the work by Hölzenspies et al. [90] and ter Braak et al. [91]. The application definition `SimState` can be converted to an Extensible Markup Language (XML) format that is used by the run-time mapping tool to choose at run-time the most suitable processors for executing the application. The XML format required by the run-time mapping tool resembles the `SimState` format, although much more architectural knowledge is included (for example, the amount of processors on the MPSoC, and the NoC infrastructure).

3.3 Conclusion

Mapping streaming DSP applications to a multi processor architecture is a complex task that has proven to be very hard. Typically, such applications are specified by block schematics that can also be described by mathematical relations. Existing design flows often translate the mathematical relations to a sequence of instructions, using an imperative programming model. The sequential implementation is then analyzed to find suitable locations for partitioning the code into small code segments that can be compiled for individual processors.

In this chapter we introduced an alternative design method that directly translates the mathematical relations in parallel processes, such that transformation to sequential code can be avoided. The design method uses an EDSL implemented in the functional programming language Haskell. Basic transformations like rewriting associative operators and partitioning of long lists of operations into multiple shorter lists of operations can be applied to any application implemented in the EDSL. Therefore, the EDSL is both a data type and a language. After partitioning, the EDSL implementation is converted to an SDF model. This model can be used for functional simulation and for the run-time execution on a MPSoC architecture.

Using the simulation framework, correctness of the application can be tested by executing processes and transporting their results via channels to other processes. Such testing can identify possible typing errors in communication (for example, one process produces a list of integers while the next process expects a list of characters) and allows for evaluation of the EDSL definitions. A GUI is used to present the feedback from simulation by drawing the process graph, token contents on the channels and gives an indication of the throughput of the application.

Finally, we showed how the communication modes supported by the NI as presented in chapter 2 translate into the SDF and CSDF data flow models. These models will be used in the next chapter.

Chapter 4

Case Studies from Mobile Communication Receivers

Abstract

Streaming DSP applications can be found in many embedded systems. Software defined radio receivers are a suitable case for streaming DSP applications, because they require a considerable amount of signal processing with relatively high bandwidth requirements and low latency requirements. Reconfigurable tiled architectures have proven to be useful for applications with such requirements. In this chapter, two different mobile communication receivers are discussed. The first example is a DRM radio receiver for mobile handheld devices with limited battery capacity. Thus, energy efficiency is a primary target for the implementation of the digital receiver. The second example is a DVB-S satellite receiver for in-car infotainment. A receiver uses an electronically steered array antenna that tracks satellites transmitting the signal of interest. Due to the large number of antenna elements used in an array, this communication receiver is a good example of a high performance application with strict constraints.

Analog radio and tv broadcasts have been transmitted for decades using Amplitude Modulation (AM) and Frequency Modulation (FM) broadcasting. For such modulation, receivers could be implemented very efficiently using a few analog components. However, these modulation techniques have a poor spectrum utilization, as the number of bits transmitted per Hz of bandwidth is typically small. Digital modulation schemes have been proposed to improve the spectrum utilization by increasing the number of bits transmitted per Hz of bandwidth. As a result, the channel capacity is increased at the cost of additional digital processing.

Nowadays, broadcast streams are transmitted via different media, each having specific advantages and disadvantages [102]. A popular transmission technique that

Parts of this chapter have been presented at the International Symposium on System-on-Chip [160], at the Scientific ICT Research Event of the Netherlands [161] and at the Euromicro Conference on Digital System Design [162], parts were published in the International Journal of Reconfigurable Computing [163] and parts have been accepted for publication at the IEEE Vehicular Technology Conference [164].

is suitable for short distance communication is wireless transmission via radio base stations, called terrestrial communication. Typically, the signals transmitted by a radio base station can be received within an area with a radius of a few kilometers around the base station. However, wireless transmission of signals is quite expensive as the signal quality depends on many factors like interferers and weather circumstances. The total capacity of terrestrial communication is restricted due to power- and spectrum limitations. Furthermore, to cover a large area, multiple transmitters are required and therefore, there may be some overlap in coverage. In the overlapping area, interference will occur if both transmitters use the same spectrum. Therefore, to avoid interference, the spectrum cannot be used completely. Digital terrestrial broadcasts typically use Orthogonal Frequency Division Multiplexing (OFDM), as it provides high spectral efficiency and allows simple filters in the transmitter and receiver [103]. A disadvantage of OFDM is its sensitivity to interference due to multipath effects (caused by reflections via buildings and large objects) and Doppler shift (caused by movement of the receiver), hence OFDM based communication receivers should include mechanisms for time/frequency synchronization and channel estimation.

Another application area of wireless communication is satellite communication. Many satellites are orbiting around the earth at different altitudes and with different speeds with respect to the earth. Due to their high altitude, each satellite can broadcast a signal to a very large area. However, this altitude also decreases the signal quality, as climatological circumstances have a varying impact on the signal of the earth-based receiver. Keeping the satellite at any arbitrary altitude moving at any arbitrary speed would require continuous corrections, for example by using a large booster engine that can be turned on to reposition the satellite. By using the earth's gravity field, for each altitude, one trajectory can be chosen in which the satellite can stay without requiring an engine. When a satellite traverses a trajectory around the earth in exactly 24 hours, the trajectory is called a *geosynchronous* trajectory as it enables traveling around the earth in exactly one earth rotation [104]. There is one special geosynchronous trajectory, which enables satellites to move with the exact same speed as the earth's surface. This *geostationary* trajectory is a trajectory at 36000 kilometers around the equator and is also known as the *Clarke belt* [105] (see Figure 4.1). From the earth's point of view, a satellite in that trajectory is fixed at its position. A fixed directional antenna can then be used to receive signals transmitted by such a 'fixed' satellite. Typically, dish antennas are used to focus and to amplify the signals received from a particular satellite. The pointing of the dish antenna should be done very carefully to enable optimal signal reception. Hence, this type of antenna is only useful in static scenarios. Moreover, for satellite broadcasting, the transmission channel is continuously changing, for example due to changing weather circumstances. The varying channel quality typically distorts the signal amplitude, hence the signal modulation is based on Quadrature Phase Shift Keying (QPSK) as this enables modulation in phase only.

In this chapter we present two mobile communication receivers that utilize different transmission media. First we introduce Digital Radio Mondiale (DRM) as an example digital radio application which is broadcasted using terrestrial base stations, in section 4.2. The reception of DRM signals using a mobile handheld device requires

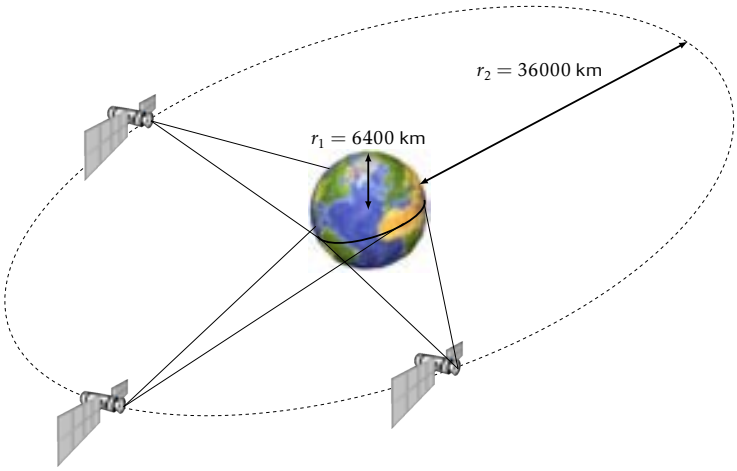


Figure 4.1 – Clarke belt at 36000 km around the equator

an energy-efficient architecture that allows for long battery life when listening to a digital radio broadcast. The second example, a mobile Digital Video Broadcast for Satellite (DVB-S) receiver for in-car infotainment, is described in section 4.3.

The mobile communication receivers are used to show how a generic stream processing platform, as presented in chapter 2, can be used in different circumstances. For both receivers, digital processing is used to allow for high spectrum utilization and for reducing the analog front-end constraints (like filter selectivity, energy per transmitted bit, and tuner flexibility). The digital processing can be modeled using streaming dataflow models. The DRM application has been used as a key application within the 4S project [2] and DVB-S was used as a research vehicle in the CMOS Beamforming Techniques project [3]. The DRM application is an example of a low-power application for handheld devices, where battery capacity is limited. The DVB-S application poses less power constraints, as its application to in-car infotainment enables a reasonable battery capacity. However, due to the large number of antenna elements, the power budget and processing performance of the overall receiver are important. Before discussing these applications, we first discuss some of the frequently used DSP kernels and their implementation on the Montium TP based MPSoC, such that they can be referred to in the next two sections discussing the applications.

4.1 Common DSP kernels

In this section, frequently used DSP algorithms are presented. First several classes of FFTs and their implementations on the Montium TP are discussed in section 4.1.1. Then, a generic implementation of FIR filters on the Montium TP is analyzed.

4.1.1 Fast Fourier Transform

The Discrete Fourier Transform (DFT) transforms a digital signal from the time domain to the frequency domain. It is defined by the following relation between N input samples $x[n]$ and N output samples $X[k]$:

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}, \quad k = 0, 1, \ldots, N-1 \tag{4.1}$$

where $W_N^{nk} = e^{-j2\pi \frac{nk}{N}}$ are primitive roots of the unit circle, also called *twiddle factors*. Each of the N outputs is the sum of N terms, so a direct computation of this formula requires $O\left(N^2\right)$ operations. The inverse operation, the Inverse Discrete Fourier Transform (iDFT), converts the frequency domain samples back to the time domain and is defined as follows:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] W_N^{-nk}, \quad n = 0, 1, \ldots, N-1 \tag{4.2}$$

The FFT efficiently implements the DFT by exploiting symmetry in its twiddle factors [1]. The best known FFT algorithm is the algorithm proposed by Cooley and Tukey [106], who generalized the decomposition into any arbitrarily sized FFT. Their algorithm recursively re-expresses a DFT of length $N = N_1 \cdot N_2$ into smaller DFTs of size N_1 and N_2. For an FFT with a length that is a power of x (called radix-x), the recursion can be done in $\log_x(N)$ stages using an x-input butterfly. For example, Equation 4.3 shows how Equation 4.1 can be rewritten into a radix-2 FFT:

$$\begin{aligned} X[k] &= \sum_{m=0}^{\frac{N}{2}-1} x[2m] W_N^{(2m)k} + \sum_{m=0}^{\frac{N}{2}-1} x[2m+1] W_N^{(2m+1)k} \\ &= \sum_{m=0}^{\frac{N}{2}-1} x[2m] W_{\frac{N}{2}}^{mk} + W_N^k \sum_{m=0}^{\frac{N}{2}-1} x[2m+1] W_{\frac{N}{2}}^{mk} \end{aligned} \tag{4.3}$$

where $k = 0, 1, \ldots, N-1$. Hence, the FFT has been reduced to two smaller FFTs which, on their turn, can be rewritten again according Equation 4.3.

The partitioning presented here is only useful for radix-2 FFTs. For other FFT sizes, different algorithms can be used. A flexible FFT algorithm is the Prime Factor Algorithm (PFA), as presented by Good [107]. For any FFT of length $N = N_1 \cdot N_2$ where N_1 and N_2 are coprime, a partitioning similar to Equation 4.3 can be applied where the intermediate multiplication with a twiddle factor can be skipped. The smaller FFTs of length N_1 and N_2 can be implemented with any FFT algorithm.

Good's mapping optimizes the PFA for the number of calculations to be done, but assumes that input data is ordered in Ruritanian Correspondence (RC) order and

[1] In this thesis, we use the notation DFT-N and FFT-N to indicate an DFT and FFT operating on N time samples and resulting in N frequency components.

output data in Chinese Remainder Theorem (CRT) order or vice versa, as presented by Temperton [108]. Therefore, the input n and output k are reindexed using the following equations:

$$n = \langle n_1 N_2 + n_2 N_1 \rangle_N \tag{4.4}$$

$$k = \left\langle k_1 N_2^{-1} N_2 + k_2 N_1^{-1} N_1 \right\rangle_N \tag{4.5}$$

where $\langle a \rangle_N$ denotes $a \pmod N$. N_1^{-1} denotes the modular multiplicative inverse of $N_1 \pmod{N_2}$, such that $N_1 N_1^{-1} = 1 \pmod{N_2}$.

The re-indexing of n is called RC reordering and the re-indexing of k is called CRT reordering. Using these new indices, the DFT can be rewritten as follows:

$$X\left[k_1 N_2^{-1} N_2 + k_2 N_1^{-1} N_1\right] = \sum_{n_1=0}^{N_1-1} \left(\sum_{n_2=0}^{N_2-1} x\left[n_1 N_2 + n_2 N_1\right] W_{N_2}^{n_2 k_2} \right) W_{N_1}^{n_1 k_1} \tag{4.6}$$

where the inner sum is a DFT of size N_2 and the outer sum is a DFT of size N_1 while no intermediate multiplication is required.

Using Equation 4.4, the input vector $x[n]$ is remapped in RC order as follows:

$$x_{\mathrm{RC}}[n_1, n_2] = x[n] \tag{4.7}$$

For example, with an FFT-6 (where $N_1 = 3$ and $N_2 = 2$), the inputs for the first FFT-3 are $x_{\mathrm{RC}}[0,0] = x[0]$, $x_{\mathrm{RC}}[1,0] = x[2]$, and $x_{\mathrm{RC}}[2,0] = x[4]$, while the second FFT-3 operates on the other inputs ($x_{\mathrm{RC}}[0,1] = x[3]$, $x_{\mathrm{RC}}[1,1] = x[5]$, and $x_{\mathrm{RC}}[2,1] = x[1]$).

Similarly, the CRT reordering of the output $X[k]$ is done using Equation 4.5:

$$X[k] = X_{\mathrm{CRT}}\left[\langle k \rangle_{N_1}, \langle k \rangle_{N_2}\right] \tag{4.8}$$

A graphical description of the steps required for a PFA decomposed FFT using Good's mapping is given in Figure 4.2. First, the input reordering is done, then N_2 times an FFT-N_1 is performed (visualized with the horizontal planes in Figure 4.2) and N_1 times an FFT-N_2 is performed on the results (visualized by the vertical planes in Figure 4.2) and finally the outputs are reordered.

4.1.2 Radix-2 FFT

The radix-2 decomposition, as presented in Equation 4.3, is the most used FFT implementation. However, other efficient implementations do exist. For example, the *split radix* FFT was proposed by Yavne [109] and later reintroduced by Duhamel and Hollman [110]. Their solution is based on a partitioning different from the partitioning used by Cooley and Tukey. In stead of re-expressing an FFT-N in smaller FFTs of size N_1 and N_2, it recursively expresses the FFT-N in one FFT of size $\frac{N}{2}$ and two FFTs of size $\frac{N}{4}$. For a long time, this partitioning has been considered to have the lowest arithmetic operation count (total number of additions and multiplications) for computing power-of-2 FFTs.

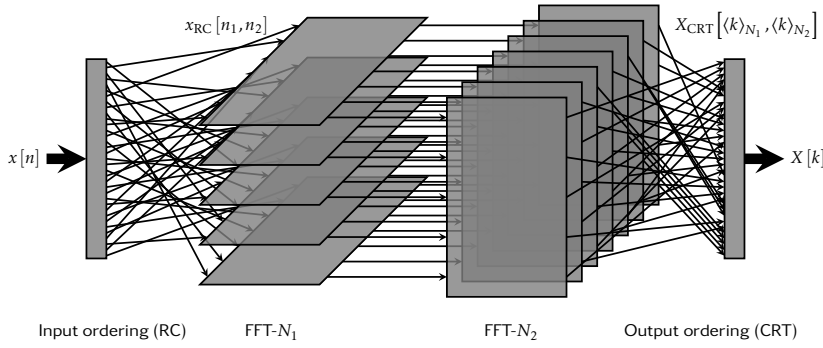


Figure 4.2 – Steps in a PFA decomposed FFT

An efficient implementation of a mixed radix-2/4 FFT processor is presented in [111]. The design is flexible and can be reconfigured at runtime to vary the FFT size between 16-point and 4096-point. However, due to its specialized design, other FFTs are not supported.

4.1.2.1 Implementation

The implementation of a radix-2 butterfly operation can be done very efficiently on the Montium TP. Due to the symmetry in the twiddle factors ($W_{N/2}^{n\cdot(k+N/2)} = W_{N/2}^{nk}$ and $W_N^{n\cdot(k+N/2)} = -W_N^{nk}$), Equation 4.3 can also be expressed as:

$$X[k] = \sum_{m=0}^{\frac{N}{2}-1} x[2m]\, W_{\frac{N}{2}}^{mk} + W_N^k \sum_{m=0}^{\frac{N}{2}-1} x[2m+1]\, W_{\frac{N}{2}}^{mk} \tag{4.9}$$

$$X\left[k+\frac{N}{2}\right] = \sum_{m=0}^{\frac{N}{2}-1} x[2m]\, W_{\frac{N}{2}}^{mk} - W_N^k \sum_{m=0}^{\frac{N}{2}-1} x[2m+1]\, W_{\frac{N}{2}}^{mk} \tag{4.10}$$

Note that $\sum_{m=0}^{\frac{N}{2}-1} x[2m]\, W_{\frac{N}{2}}^{mk}$ and $\sum_{m=0}^{\frac{N}{2}-1} x[2m+1]\, W_{\frac{N}{2}}^{mk}$ both define an FFT-$N/2$. These FFTs can be rewritten to FFT-$N/4$ by reapplying Equation 4.3. Finally, this results in an FFT-2 (where $N = 2$ and $W_{\frac{N}{2}}^{mk} = W_1^{mk} = 1$ for any m and k):

$$X[k] = x[0] + W_2^k x[1] \tag{4.11}$$

$$X[k+1] = x[0] - W_2^k x[1] \tag{4.12}$$

which is also called the *butterfly operation*. This operation can be mapped efficiently on the Montium ALUs (see Figure 4.3). In one clock cycle, the ALUs can execute a complex multiplication and perform the butterfly operation.

With one butterfly operation, two results out of N are calculated. Therefore, $N/2$ butterfly operations are required to calculate all N results. Since the FFT-N

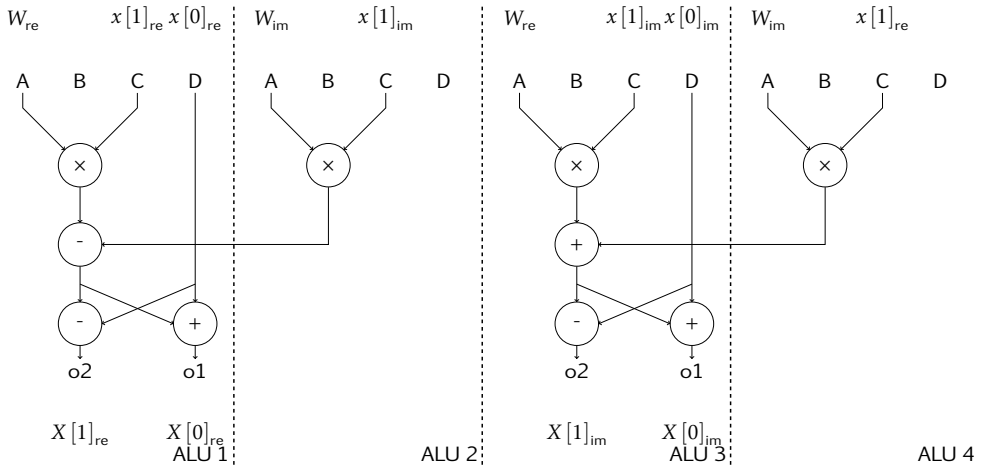
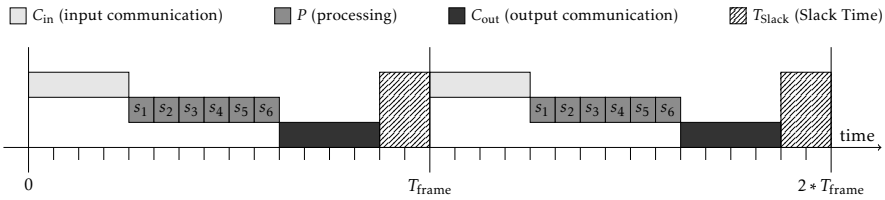


Figure 4.3 – FFT butterfly ALU mapping

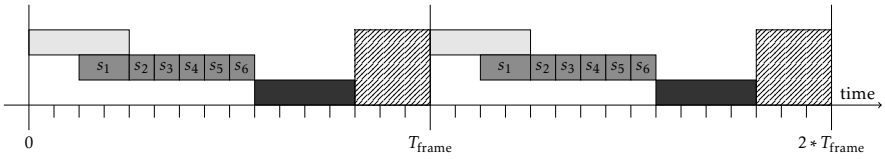
can be calculated by dividing it into two FFT-$N/2$ and combining the intermediate results with $N/2$ butterfly operations, the calculation of one FFT-N requires $N/2$ butterfly calculations, followed by twice an FFT-$N/2$. In this way, an FFT-N can be decomposed into $\log_2 N$ stages consisting of butterfly operations. These operations can be fully pipelined (which costs 2 additional cycles for the start and end of the pipeline), hence $N/2+2$ cycles are required for one stage. In total, the FFT-N can be executed in $\left(\frac{N}{2}+2\right)\log_2 N$ clock cycles [112]. Due to the partitioning, the output data of the last stage has to be reshuffled. The FFT results are produced in a bit-reversed address order. Such reordering is supported in hardware within the AGUs of the Montium's memories. Hence, reversing the address bits costs no clock cycles as it is done simultaneously with the calculation of the next memory address.

4.1.2.2 Block-mode versus streaming-mode

The block-mode version of the radix-2 FFT mentioned above operates on an input vector x (consisting of N time samples) stored in memory and results in an output vector X (consisting of N frequency components) that is stored in another part of the Montium TP's memory. Before the execution of the FFT butterflies, the input vector is first received and stored via a DMA transfer. Then, the execution is started and upon completion, the results are retrieved using another DMA transfer. For the operation of a radix-2 FFT in streaming-mode, a part of the input can be loaded simultaneously with the first butterfly calculations and the results can be streamed to the output as soon as a part of the results has been reordered. For both modes, the communication bandwidth determines the communication overhead of the FFT. An example of the execution of an FFT-64 in both modes is shown in Figure 4.4, where



(a) Schedule of a radix-2 FFT operated in block-mode



(b) Schedule of a radix-2 FFT operated in streaming-mode

Figure 4.4 – Example schedules of the block-mode and streaming-mode implementations of an FFT

the notation s_x shows the execution of the x^{th} stage of the FFT[2].

In the example of Figure 4.4, an FFT-64 is executed. For the block-mode version of the FFT, the total number of tile clock cycles required to process one FFT is:

$$\begin{aligned} T_{\text{block}} &= T_{\text{comm}} + T_{\text{comp}} \\ &= 2 \cdot \frac{2N}{L} + \left(\frac{N}{2} + 2\right)\log_2(N) \\ &= \frac{4N}{L} + \left(\frac{N}{2} + 2\right)\log_2(N) \end{aligned} \tag{4.13}$$

where L indicates the number of samples that is loaded into the Montium TP simultaneously (see also the notes on parallel DMA transactions supported by the DMA controller discussed in section 2.2.2.2). The input communication time for N complex numbers (so $2N$ words) is equal to the output communication time, hence $T_{\text{comm}} = 2 \cdot \frac{2N}{L}$. In the example used in Figure 4.4, $N = 64$ and $L = 1$. As a result, the worst-case C/C ratio for the block-mode operated FFT-64 is $\frac{T_{\text{comm}}}{T_{\text{comp}}} = \frac{256/L}{204} = 1.25/L$. Table 4.1 shows an overview of the computation and communication times for all radix-2 FFTs between FFT-16 and FFT-1024.

The streaming-mode implementation is capable of reading the input data during the first stage of the FFT and writing the results during the last stage of the FFT. In the first stage, each butterfly operation is performed on the samples $x[i]$ and $x[i + N/2]$. Hence, if the samples are read in linear order (sample $x[i]$ is followed by $x[i + 1]$), the first $N/2 + 1$ samples need to be read before the first butterfly operation can be performed. Figure 4.4(b) shows how a part of the inputs is read, and then the processing of stage 1 is started during the input reading. As mentioned

[2] The schedule does not include the possible delays caused by the NI message protocol that is used to start the DMA transactions and algorithm execution.

Table 4.1 – Communication to computation ratio of different radix-2 FFTs.

(a) Block-mode

N	T_{comp}	T_{comm}	C/C ratio
16	40	$\frac{64}{L}$	$\frac{1.60}{L}$
32	90	$\frac{128}{L}$	$\frac{1.43}{L}$
64	204	$\frac{256}{L}$	$\frac{1.25}{L}$
128	462	$\frac{512}{L}$	$\frac{1.11}{L}$
256	1040	$\frac{1024}{L}$	$\frac{0.98}{L}$
512	2322	$\frac{2048}{L}$	$\frac{0.88}{L}$
1024	5140	$\frac{4096}{L}$	$\frac{0.80}{L}$

(b) Streaming-mode

N	T_{comp}	T_{comm}	C/C ratio
16	40	$\frac{64}{L}$-10	$\frac{1.60}{L}$-0.25
32	90	$\frac{128}{L}$-18	$\frac{1.43}{L}$-0.20
64	204	$\frac{256}{L}$-34	$\frac{1.25}{L}$-0.17
128	462	$\frac{512}{L}$-66	$\frac{1.11}{L}$-0.14
256	1040	$\frac{1024}{L}$-130	$\frac{0.98}{L}$-0.13
512	2322	$\frac{2048}{L}$-258	$\frac{0.88}{L}$-0.11
1024	5140	$\frac{4096}{L}$-514	$\frac{0.80}{L}$-0.10

in section 2.2.1.1, communication and computation can be done simultaneously. However, if the input stream is delayed, the processing is also delayed. Since T_{comm} is defined as the overhead caused by communication, for the streaming-mode FFT we define $T_{comm} = T_{comm,input} - T_{comp,stage1} + T_{comm,output} = \frac{4N}{L} - \left(\frac{N}{2} + 2\right)$.

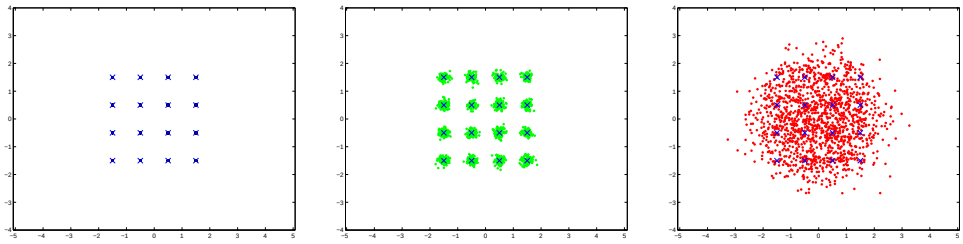
$$\begin{aligned} T_{streaming} &= T_{comm} + T_{comp} \\ &= \frac{4N}{L} + \left(\frac{N}{2} + 2\right)(\log_2(N) - 1) \end{aligned} \tag{4.14}$$

Due to the bit-reversed order in which results are produced in the last stage [112], the data can only be streamed out in linear order after finishing that stage[3]. Hence, computation and communication cannot be combined during the last stage. In total, the communication overhead includes half of the input streaming and all output streaming. For the streaming-mode example depicted in Figure 4.4 (where $N = 64$ and $L = 1$), the C/C ratio then becomes $\frac{1.25}{1} - 0.17 = 1.08$, which is about 14% smaller than the block-mode implementation.

4.1.3 Non-power-of-two FFT

The required DFTs for DRM are those operating on 1920, 576, 512, 352, 288, 256, 224, 176 and 112 samples [113, 114]. For the iDFTs, the required sizes are 1920, 288, 256, 176, 128, 112 and 32 points. The restriction of the radix FFT is that it can only handle FFTs that have a length that is a power of the radix value (for example two for radix-2). Hence, of the DFTs mentioned above only the 1024, 512, 256 and 32-point can be implemented using the radix-2 solution. A frequently used method to implement non-radix-2 FFTs is by applying *zero padding*, which appends zeros

[3] Output ordering is required if the next process does not support bit-reversed indexing of the stream. Therefore, the output communication overhead could be reduced considerably. However, this optimization is not useful for the general case and therefore, we consider in-order communication as a requirement.



(a) Transmitted data (b) Received data (c) Received data using zero padding

Figure 4.5 – 16-QAM bit errors occurring due to transmission and decoding

to the input vector and increases its length to a power-of-two, such that a regular radix-2 FFT can be applied. However, this changes the filter response of the FFT and therefore the data stream will lose its orthogonal characteristics.

To illustrate the effects of zero padding on a stream of OFDM symbols, we simulated an OFDM system with 16-Quadrature Amplitude Modulation (QAM) modulation transmitting 1500 symbols. Figure 4.5(a) shows the input bits after modulation by the transmitter. After transforming the input samples with an Inverse Fast Fourier Transform (iFFT), it was sent via an Additive White Gaussian Noise (AWGN) channel. Two different receivers were used to transform the samples back to the frequency domain: one of them used an FFT with a length equal to the length of the sender's iFFT (see Figure 4.5(b)), while the other receiver used a larger FFT with zero padding (see Figure 4.5(c)). The effect of the white noise added by the channel is clear: small errors occur in the received samples. However, for the zero padding based receiver, the input samples are not recognizable at all. Usually, in most applications the error introduced by zero padding is acceptable as the gain in performance is more important. However, OFDM-based applications use the orthogonal characteristics of FFTs to improve the spectral efficiency and, therefore, the requirements for the FFT are more stringent. In order to obtain an acceptable performance, efficient non-power-of-two FFT implementations are required.

An example of a suitable FFT algorithm for implementing the non-power-of-two FFT lengths is the mixed-radix algorithm [115]. An implementation of the PFA based FFT was presented in [116]. Bi and Chen present a fully optimized FFT algorithm for any FFT of length $q*2^N$ [117]. Their solution is based on a split-radix algorithm which is manually optimized. A more flexible solution using a split-radix FFT is presented in [118], which is optimized to reduce the complexity of address pattern generation and memory lookup with a butterfly efficiency comparable to the Cooley-Tukey. Another efficient FFT algorithm is the Winograd FFT [119]. Several implementations of the Winograd FFT algorithm were proposed [120–122].

A manual partitioning of an FFT-1920, based on the PFA, was published by Rivaton et al [113]. By partitioning the FFT in a radix-2 part and a non-radix-2 part, an efficient implementation can be made that consists of a regular structure. The partitioning was generalized in [160, 163], such that any FFT required for DRM can

Table 4.2 – A selection of the FFTs that can be generated with the PFA mapping, where $N = (2p+1) \cdot 2^q$. FFTs used in DRM are underlined.

p	q			
	4	5	6	7
2	80	160	320	640
3	<u>112</u>	<u>224</u>	448	896
4	144	<u>288</u>	<u>576</u>	1152
5	<u>176</u>	<u>352</u>	704	1408
6	208	416	832	1664
7	240	480	960	<u>1920</u>

be generated from one template. Table 4.2 presents the FFTs that can be generated from the template. The implementation of this generalized partitioning is explained in the next section.

4.1.3.1 Implementation

Using the PFA decomposition described in Equation 4.6, any FFT required for DRM mentioned in Table 4.2 can be implemented. These FFTs can be mapped on the Montium architecture according the same mapping scheme. This makes it possible to generate a large number of configurations based on the same decomposition and mapping structure. As an example of these FFTs, we use the FFT-1920 to describe the implementation of any non-power-of-two FFT.

The FFT-1920 is partitioned using the parameters $N_1 = 2 \cdot 7 + 1 = 15$ and $N_2 = 2^7 = 128$. According to the PFA approach, the FFT is decomposed into 128 times FFT-15 followed by 15 times FFT-128. The order of decomposition can be chosen arbitrarily, but the proposed decomposition leads to a better fit on the Montium architecture.

Figure 4.2 shows the steps that need to be taken in order to compute the FFT-1920. Firstly, the input data is distributed over the input memories. Secondly, the FFT-15 is then performed on 128 blocks of 15 input samples and the results are distributed over the memory such that the FFT-128 can be operated. Thirdly, each FFT-128 then processes a block of data and writes its results in the memory. Finally, the output data from the FFT-128 are reordered. Each of the steps is explained in more detail in the next paragraphs.

Non-radix-2 part Generally, the $N_1 = 2p + 1$ FFTs can be simplified by exploiting the symmetry in the twiddle factors. Because N_1 is odd, Equation 4.1 can be rewritten

to Equation 4.15:

$$X[0] = \sum_{n=0}^{N_1-1} x[n]$$
$$X[k] = \begin{cases} T_{\Re\mathfrak{e}}[k] + T_{\mathfrak{Im}}[k], & 1 \le k \le \frac{N_1-1}{2} \\ T_{\Re\mathfrak{e}}[N_1-k] - T_{\mathfrak{Im}}[N_1-k], & \frac{N_1+1}{2} \le k < N_1 \end{cases} \tag{4.15}$$

where

$$T_{\Re\mathfrak{e}}[k] = x[0] + \sum_{n=1}^{\frac{N_1-1}{2}} (x[n] + x[N_1-n]) \cdot \Re\mathfrak{e}\left(W_{N_1}^{nk}\right)$$
$$T_{\mathfrak{Im}}[k] = \sum_{n=1}^{\frac{N_1-1}{2}} (x[n] - x[N_1-n]) \cdot \mathfrak{Im}\left(W_{N_1}^{nk}\right)$$

For the summations in $T_{\Re\mathfrak{e}}$ and $T_{\mathfrak{Im}}$, the operands are multiplied with a twiddle factor. Two inputs are added before a multiplication and the butterfly structure of the Montium TP is used to calculate $X[k] = T_{\Re\mathfrak{e}}[k] + T_{\mathfrak{Im}}[k]$ and $X[N_1-k] = T_{\Re\mathfrak{e}}[k] - T_{\mathfrak{Im}}[k]$ concurrently. Four ALUs are occupied to compute both outputs in parallel, while the fifth ALU is used to compute the $X[0]$ component simultaneously with the computation of $X[1]$ and $X[14]$. For the calculation of $X[k]$, the values of $x[n]$ and $x[N_1-n]$ are required simultaneously. Therefore, $x\left[0 \ldots \frac{N_1-1}{2}\right]$ and $x\left[\frac{N_1+1}{2} \ldots N_1-1\right]$ are stored in different memories such that these values can be accessed simultaneously. These simultaneous calculations result in a reduction of the number of multiplications by a factor of 4, compared with the calculation of a normal DFT.

In general, the number of clock cycles required to compute an odd-size FFT-N_1 on the Montium TP, using the partitioning presented above, equals $\left(\frac{N_1-1}{2}\right)^2 + \frac{N_1-1}{2} = \frac{1}{4}\left(N_1^2-1\right)$. So, this approach is only viable for small N_1 as for larger odd values the complexity grows exponentially with the size of N_1. The execution of an FFT-15, based on this optimized implementation for the Montium TP, requires 56 clock cycles. Theoretically, the FFT-15 could have been computed more efficiently by again applying the PFA with FFT-3 and FFT-5. Using the PFA, the FFT-3 has to be performed 5 times ($5 \times 2 = 10$ clock cycles), followed by 3 times the FFT-5 ($3 \times 6 = 18$ clock cycles), resulting in 28 clock cycles. However, this requires reordering of the intermediate results which cannot be implemented efficiently on the Montium TP. The reordering costs for an FFT-15 would at least require another two times 15 cycles (for both input and output reordering), hence adding 30 cycles to the costs. Therefore, the presented mapping is the better alternative.

Radix-2 part The FFT-128 is implemented using a standard radix-2 approach. Radix-2 algorithms can be calculated efficiently on the Montium TP since one FFT-2

can be executed in a single clock cycle. A detailed explanation of the mapping is presented in section 4.1.2 and more detailed in [29, 112]. The computation of such an FFT-N_2 requires $\left(\frac{N_2}{2}+2\right)\cdot\log_2(N_2)$ clock cycles. Hence, the execution of an FFT-128 requires 462 clock cycles.

Input and output ordering In traditional radix-2 FFT implementations, the most difficult part is the bit-reversed addressing scheme of either the input or output values. In most DSP architectures, and Montium TP as well, special hardware in the AGU overcomes this problem. However, in the PFA both input and output have to be reordered according the RC or CRT mapping. Because the input reordering is done in the Montium TP, the user of the algorithm has the possibility to stream in the data into the Montium TP in-order.

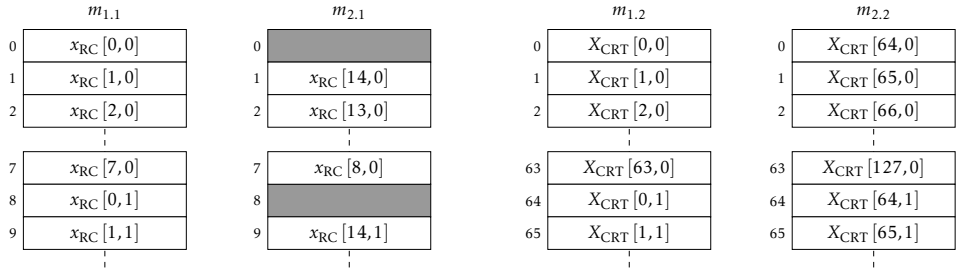
The address patterns for RC ordering cannot be generated efficiently with the AGU. Moreover, since the input values for the FFT-15 are stored in two memories, address patterns become even less regular. A straight-forward solution for the ordering would be to use a LUT containing the reordered addresses for all input values. The first 256 positions in the coefficient memories (the two memories connected to ALU5) are occupied for the twiddle factors used by FFT partitions (the non-radix-2 part and radix-2 part presented in the previous section). Therefore, for a FFT-N with $N > 768$, the Montium TP memories cannot be used as a LUT. For the FFTs used in DRM, this only holds for the FFT-1920. For smaller FFTs this is the preferred approach. This ordering approach can be used for the real and imaginary part of two samples simultaneously, such that all input samples can be reordered in $\lceil\frac{N_1}{2}\rceil * N_2 + 2$ clock cycles.

For the input reordering for the FFT-1920 we use the following steps:

1. The complex input vector is written in-order into 2 local memories $m_{1.1}$ and $m_{2.1}$
2. An indirection read address is calculated using Equations 4.4 and 4.7
3. Using the indirection address, an input value is selected from the local memories $m_{1.1}$ and $m_{2.1}$
4. The value is stored in the other local memories $m_{1.2}$ and $m_{2.2}$ using the current write address (initially set to 0)
5. The write address in memories $m_{1.2}$ and $m_{2.2}$ are incremented by one

Steps 2 to 5 are repeated 1920 times, until all values x_{RC} have been reordered. The calculation of the indirection address and storage of the value in the local memories can be pipelined, such that each clock cycle one value can be stored. In total, the pipelining overhead includes 4 clock cycles, so the full input reordering is done in $N+4$ clock cycles. Figure 4.6(a) shows the FFT-15 input data after input reordering[4]. Figure 4.6(b) shows how the results of the FFT-128 are stored in the memory. For the

[4]Note that only the real part of x_{RC} is shown; the imaginary part is stored in the memories $m_{3.1}$ and $m_{4.1}$ in an identical order and the 5 steps are taken simultaneously for the imaginary part.



(a) Memory organization before FFT-15 (b) Memory organization after FFT-128

Figure 4.6 – Intermediate memory organization for FFT-1920

Table 4.3 – Implementation costs of FFTs used in DRM. Reordering costs are shown for input ordering; output ordering costs are equal.

FFT	N_1	N_2	execution	reordering
112	7	16	472	66
176	11	16	960	98
224	7	32	1014	130
288	9	32	1450	162
352	11	32	1950	194
576	9	64	3116	322
1920	15	128	14098	1924

output ordering we use the same principle, but now the selected complex values are streamed to the NoC via the NI.

The most complex step in the ordering process of the outputs is the calculation of the indirection address. This address has to be calculated using modulo operations. In Appendix C we explain the output ordering for streaming out the complex sample $X[k]$ in linear order.

Computational complexity The total number of clock cycles required to calculate a non-power-of-two FFT of length $N = N_1 \cdot N_2$ is $N_1 \cdot$ cycles(FFT-N_2) $+N_2 \cdot$ cycles(FFT-N_1):

$$T_{\text{comp}} = N_1 \cdot \left(\frac{N_2}{2} + 2\right) \cdot \log_2(N_2) + N_2 \cdot \frac{1}{4}\left(N_1^2 - 1\right) \tag{4.16}$$

In Table 4.3, the implementation costs of the FFTs used in DRM are listed.

4.1.3.2 Scaling

A fixed-point implementation of a digital signal processing algorithm is liable to overflow after an addition. To prevent overflow, the amplitude of the input signal can be limited or the intermediate values can be scaled down. Scaling the intermediate

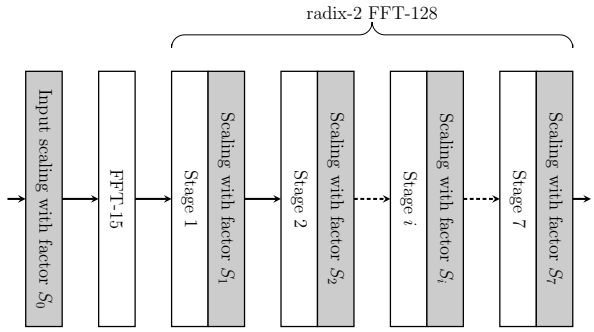


Figure 4.7 – Positions in the FFT-1920 algorithm where scaling can be applied

fixed-point numbers results in a shift of the decimal point. For example, scaling a number in $(1,15)$-fixed-point notation by 64 results in a number in $(7,9)$-fixed-point notation.

Due to the additions in each of the stages, the worst-case increment of intermediate results of an FFT equals $\sqrt{2}$ after each stage [123], but in normal operation with a signal that contains several frequency components, the scaling factor can be smaller. This results in a more accurate output signal. Therefore, we implemented a flexible solution, that supports scaling the signal at predefined positions. After every stage of an FFT, scaling can be applied, as depicted in Figure 4.7. Note that scaling does not necessarily have to be applied at once. Hence, multiple scaling positions can be used to obtain the total scaling by dividing S in smaller fractions S_i. S_0 denotes the input scaling factor and S_i denotes the scaling factor during stage i of the radix-2 FFT. It is up to the user of the algorithm to choose these values. Suppose we have a required total scaling factor of 128. The scaling can be positioned in the beginning ($S_0 = 128$), which results in a less accurate result and low risk of overflow. Moving scaling to the end of the algorithm will improve the accuracy but increases the risk of overflow. In any of the combinations the designer has to adjust the correct fixed-point notation of the output result.

Scaling accuracy To demonstrate the accuracy of the algorithm, the FFT-1920 was executed with several combinations of scaling factors (see Table 4.4). The overall scaling factor S was 128 in all cases. The difference in the cases is the amount of scaling during the algorithm. For the lower case numbers the scaling is put toward the end of the algorithm, which gives a higher accuracy. For the higher case numbers the risk of overflow is lower.

The input used for the test cases was a typical complex DRM sample stream consisting of 9600 samples. The sample stream was cut in 5 segments of 1920 samples and on each segment an FFT-1920 was applied. The maximum amplitude of the stream was scaled to three levels (31%, 63% and 100% of the fixed-point scale) to analyze the effects of the input scaling and intermediate scaling. The results of the FFT computed by the Montium TP are compared with a floating-point FFT calculated

Table 4.4 – 11 cases to demonstrate the accuracy of the FFT-1920

Case	Scaling factors							
	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
1	1	2	2	2	2	2	2	2
2	2	1	2	2	2	2	2	2
3	2	2	2	2	2	2	2	1
4	4	1	1	2	2	2	2	2
5	4	2	2	1	2	1	2	2
6	4	2	2	2	2	2	1	1
7	8	1	1	1	2	2	2	2
8	8	2	1	2	1	2	1	2
9	8	2	2	2	2	1	1	1
10	16	1	1	1	1	2	2	2
11	16	2	2	2	1	1	1	1

by Matlab. For both, a total scaling factor of 128 was used.

Figure 4.8 depicts the maximum and average errors that occur in each of the cases and, per case, for 3 scaling levels of the input signal. The errors are calculated based on the error of all 1920 frequency bins, averaged over the 5 segments. Errors are represented in terms of LSBs of a 16-bit fixed-point represented number. Since parts of the internal datapath in the ALUs of the Montium TP are 18-bit wide, in an ideal situation the arithmetic operations can be performed more accurately than 16-bit. As a result, the error (in bits) can be negative, as shown in several of the cases in Figure 4.8. In the figure, the horizontal lines indicate the error of the ARM9 implementation. Note that the ARM9 implementation is 32-bit, while the Montium TP only operates in 16-bit mode.

From this figure it is clear that, for an input signal with 31% of the range, the low numbered cases have a higher accuracy. However, applying such input scaling decreases the dynamic range of the algorithm considerably, resulting in a less accurate Fourier transform. Therefore, input scaling should be avoided as much as possible. From Figure 4.8 it can be concluded that cases 5, 6 and 8 have the best performance, independent of the input scaling. On the other hand, if input scaling is not applied and the input signal is too strong, the risk of overflow is higher[5].

These results show the benefit for partial reconfiguration, where the system can quickly adjust the scaling factors depending on the input signal level. It can make a trade-off between accuracy and the risk of an overflow.

4.1.3.3 Block-mode versus streaming-mode

The FFTs have been implemented for both the block-mode and streaming-mode operation types. In the block mode, the input samples need to be ordered (in RC order as explained before) before they can be transferred to the memories of the

[5]Overflow is noticed if the maximum error is above the 4.5 bits

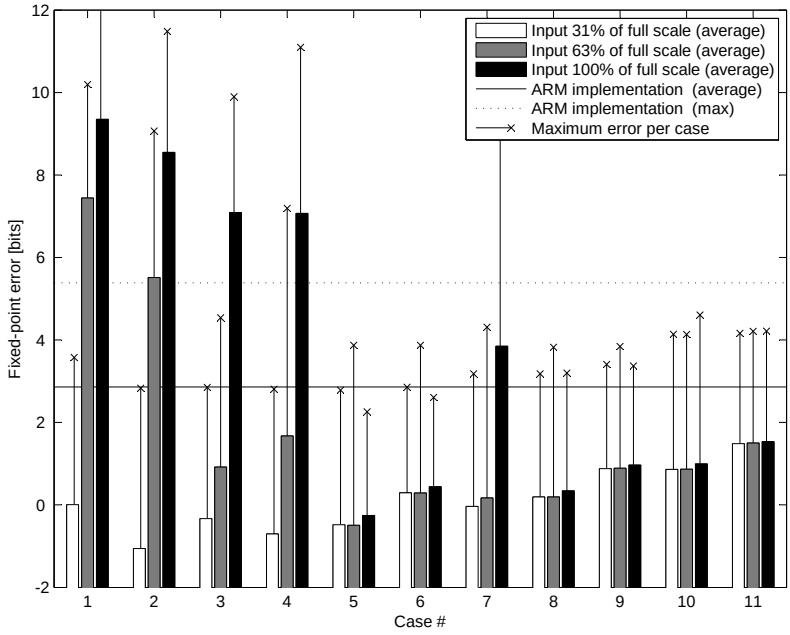


Figure 4.8 – Rounding errors for various scaling combinations

Montium TP. Loading the ordered input samples for an FFT-N takes $2N/L$ clock cycles. When the execution is started, the Montium TP applies input scaling by multiplying the input stream with a factor $\frac{1}{S_0}$. This can be done in $N/2+2$ clock cycles, because 2 complex numbers can be scaled simultaneously and the pipelining effects introduce 2 additional clock cycles. After applying input scaling, the execution is started. Table 4.3 shows the execution times for each of the FFTs. When the FFT has finished, the results can be transferred from the memories and then need to be reordered. Retrieving the output samples for an FFT-N takes another $2N/L$ clock cycles. Both the input and output ordering have to be done outside the Montium TP. Therefore, the total number of clock cycles required for the execution of an FFT-N is $(2N/L)+(N/2+2)+N_1\cdot\left(\frac{N_2}{2}+2\right)\cdot\log_2(N_2)+N_2\cdot\frac{1}{4}\left(N_1^2-1\right)+(2N/L)$.

The streaming mode version requires no external processing. First, data is streamed into the Montium TP memories in the received order. Then, using the indexing as described in section 4.1.3.1 and Appendix C, it is reordered to the correct format. Simultaneously with the input ordering, the input scaling is applied to prevent overflows during the FFT. When the FFT computation is finished, the results are read in CRT order from the memories (as explained in section 4.1.3.1) and written to the network via the NI. Due to the complex address calculation required for input and output ordering, the streaming-mode FFT is only implemented for $L=2$, where the real and imaginary part of each sample are stored simultaneously at identical offset addresses of different Montium TP memories. Therefore, streaming the input samples for an FFT-N is done in N cycles. The execution of a

Table 4.5 – Communication to Computation ratio of FFTs used in DRM

(a) Block-mode

N	T_{comp}	T_{comm}	C/C ratio
112	604	$448/L$	$0.74/L$
176	1156	$704/L$	$0.61/L$
224	1274	$896/L$	$0.70/L$
288	1774	$1152/L$	$0.65/L$
352	2388	$1408/L$	$0.59/L$
576	3760	$2304/L$	$0.60/L$
1920	17946	$7680/L$	$0.43/L$

(b) Streaming-mode

N	T_{comp}	T_{comm}	C/C ratio
112	604	224	0.48
176	1156	352	0.37
224	1274	448	0.44
288	1774	576	0.40
352	2388	704	0.36
576	3760	1142	0.37
1920	17946	3840	0.27

streaming-mode FFT is done similarly to the execution of a block-mode FFT. Reordering and streaming the output samples is also done in N cycles. The total number of clock cycles required for the execution of an FFT-N in streaming-mode operation is $N + N_1 \cdot \left(\frac{N_2}{2} + 2\right) \cdot \log_2(N_2) + N_2 \cdot \frac{1}{4}\left(N_1^2 - 1\right) + N$.

Table 4.5 shows the C/C ratio for all FFTs used for the DRM receiver, for both block-mode and streaming-mode operation[6].

4.1.3.4 Conclusion

The Montium TP is very well suited for executing algorithms with a regular kernel operation. Due to the parallelism in the data path, it can perform up to 5 operations in parallel, while each operation can use up to 4 inputs. The memory bandwidth that is delivered by the 10 local memories is tremendous. For algorithms like the FFT, it is clear that the kernel operation (a butterfly) is done repeatedly. The memory bandwidth required for executing a full butterfly operation in one clock cycle can be provided by the Montium TP, while the address patterns that are used for accessing the memories are generated quite easily. The implementation of a wide range of non-power-of-two FFTs and iFFTs on the Montium TP architecture have been discussed in detail. This range of FFTs showed to be an ideal test-case to explore and validate the flexibility of the coarse-grained architecture.

The class of non-power-of-two FFTs is less regular than the class of radix-2 FFTs. By optimizing the algorithm for regularity and not for the number of multiplications, we managed to map a non-power-of-two FFT on the Montium TP. Using the Prime Factor decomposition, the class of non-power-of-two FFTs could be partitioned such that a radix-2 component was recognized (which can be mapped and executed very efficiently on the Montium TP) together with a small odd DFT. The 10 memories available in the Montium TP enable parallel addressing of multiple inputs for a DFT, such that the DFT can be operated slightly more efficiently. Hence, the DFT's complexity was reduced from N^2 to $\frac{1}{4}\left(N^2 - 1\right)$. We showed that a further decomposition

[6] Due to external input and output ordering for the block-mode implementations, the actual C/C ratio depends on the overhead caused by the processor applying input and output ordering.

of the DFT is not desirable as the regularity of the decomposed algorithm decreases and performance will not increase. Hence, the solution described in section 4.1.3 is considered to be the most efficient FFT mapping for the current Montium TP architecture.

The possibility to use the data path for the generation of addresses makes it possible to map almost any algorithm with less regular addressing patterns to the Montium TP. Although this type of address pattern calculation is difficult, there is still enough regularity left in the non-power-of-two FFT to map the address calculation efficiently. Generic modulo operations are difficult to implement in hardware; however, the (pseudo-) modulo operations required for address calculations can be implemented efficiently using the Compare/Select unit available in each ALU in the Montium TP. Following generations of the Montium TP may be more efficient in such address calculations if a (pseudo-) modulo operations would be implemented in the AGU.

4.1.4 Finite Impulse Response filter

The FIR algorithm is defined as a convolution of an input vector x (consisting of M time samples) with a coefficient vector c (consisting of M coefficients). It is used to apply spectral filtering, by applying frequency dependent gain and phase correction. After the convolution, the result sample stream y is a filtered variant of the input stream x.

$$y[t] = \sum_{k=0}^{M} x[t+k]\, c[N-k] \tag{4.17}$$

where y, x and c are vectors containing real numbers and M equals the number of *filter tap*s, which can be used to design a $M+1^{\text{th}}$ order filter.

An FIR filter can also be implemented as a complex filter (where y, x and c are complex numbers). Both versions are described in the next sections.

4.1.4.1 Real FIR filter

Similar to the FFT operation, we mapped the FIR algorithm to the Montium architecture for both operation modes: block-mode and streaming-mode. Since an FIR filter consists of real multiplications and additions only, the 5 Montium ALUs can be used to perform 5 filter taps simultaneously in one clock cycle. The result s_i of ALU i is sent to its left neighbor ALU $i-1$, that adds the result during the next clock cycle to the next coefficient multiplication (see Figure 4.9). Therefore, for an M-taps FIR filter one output sample y can be calculated in in $\left\lceil \frac{M}{5} \right\rceil + 1$ clock cycles (pipelining in the filter introduces one additional clock cycle).

The block-mode implementation of an FIR filter applies filtering to a block of data with length N. This block of data is stored in one memory and the filtered result is stored in another memory. For each sample in the input block, one convolution is applied. Therefore, the total number of clock cycles needed by the block-mode filter

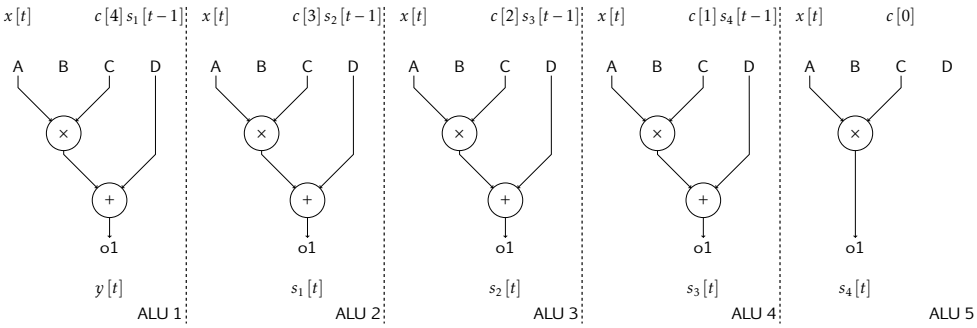


Figure 4.9 – Mapping of a 5-taps FIR filter on the Montium ALUs

is:

$$\begin{aligned} T_{\text{block}} &= T_{\text{comm}} + T_{\text{comp}} \\ &= 2 \cdot \frac{N}{L} + N \cdot \left\lceil \frac{M}{5} \right\rceil + 1 \end{aligned} \tag{4.18}$$

The streaming-mode FIR filter operates on individual samples in the input stream, which are received simultaneously with the processing during the first clock cycle. As a result, the communication overhead is reduced to 0. For a fair comparison between the block-mode and streaming-mode implementations, we consider the costs for processing N input samples with the streaming-mode implementation:

$$\begin{aligned} T_{\text{streaming}} &= T_{\text{comm}} + T_{\text{comp}} \\ &= N \cdot \left\lceil \frac{M}{5} \right\rceil + 1 \end{aligned} \tag{4.19}$$

Hence, for a 5-tap FIR filter and an input data size of 1024 samples, the worst-case C/C ratio for the block-mode version is $\frac{2048}{1025} \approx 2.00$ (again, $L = 1$), whereas the streaming-mode version has a C/C ratio of $\frac{1}{1025} \approx 0.00$.

4.1.4.2 Complex FIR filter

A complex FIR filter is functionally comparable with the real variant. However, since all multiplications are complex multiplications, 4 ALUs are required per filter tap. Therefore, the number of clock cycles for a complex FIR differ from the number of clock cycles for calculating a real FIR filter. The computation of the block-mode implementation can be done in $N \cdot M + 1$ clock cycles. Since complex samples are loaded, the communication time also doubles.

$$\begin{aligned} T_{\text{block}} &= T_{\text{comm}} + T_{\text{comp}} \\ &= 4 \cdot \frac{N}{L} + N \cdot M + 1 \end{aligned} \tag{4.20}$$

such that the C/C ratio becomes $\frac{4*N/L}{N*M+1} = 4/M*L$.

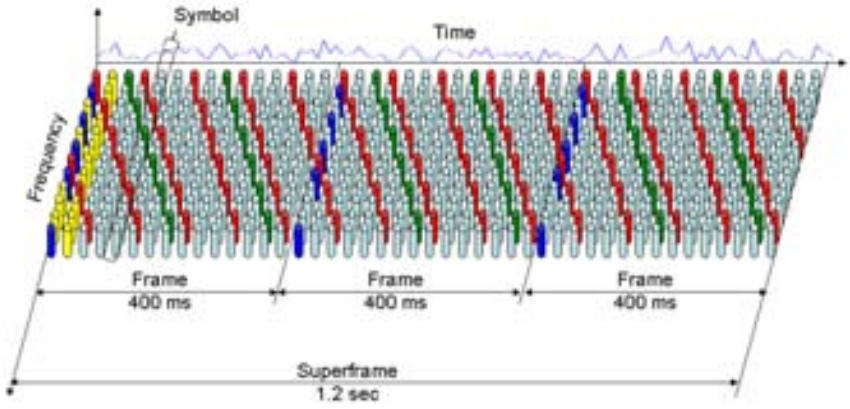


Figure 4.10 – DRM super frame structure showing three frames containing time pilot cells (symbols with mainly dark colored cells), frequency pilot cells (inserted diagonally) and SDC cells (included within the first few symbols at the start of the super frame) (picture modified from [125])

For the streaming-mode filter, loading one complex sample is done simultaneously with computation (thus, requiring $L = 2$ to read one complex sample at once) and therefore only writing the last complex value of the N filtered results is done in one additional cycle:

$$\begin{aligned} T_{\text{streaming}} &= T_{\text{comm}} + T_{\text{comp}} \\ &= 1 + N \cdot M + 1 \end{aligned} \tag{4.21}$$

Hence, the streaming-mode C/C ratio of a complex FIR filter equals $\frac{1}{N \cdot M + 1} \approx \frac{1}{N \cdot M}$.

4.2 DRM receiver

The DRM standard [124] specifies digital radio broadcasting in frequency bands below 30 MHz. DRM is a possible successor of AM radio used for point-to-multipoint broadcast and it is based on OFDM modulation and MPEG-4 audio source coding. Conventional FM and AM radio systems use a separate frequency band for each channel. A DRM stream consists of a multiplex of three information channels that are combined into a DRM *super frame* (shown in Figure 4.10). The first of these three channels, the Fast Access Channel (FAC), describes the encoding used by the physical layer, such that the bit stream can be decoded. The second channel, the Service Description Channel (SDC), describes the configuration of the channel multiplex itself and the demodulation settings that were used to transmit the actual data. The Main Service Channel (MSC) contains the actual data stream that is sent to the MPEG-4 decoder.

Figure 4.11 shows an overview of the baseband processing for DRM. The electromagnetic wave is received by an antenna that is connected to a Radio Frequency

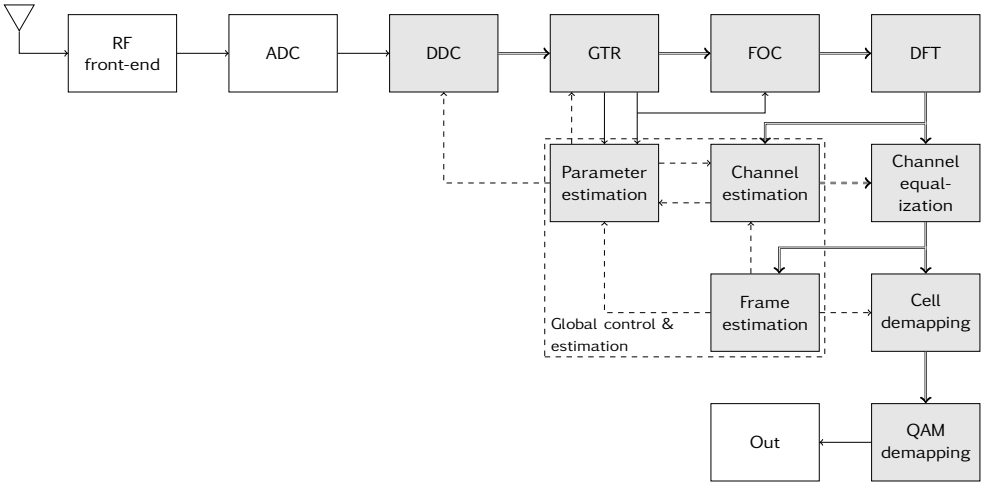


Figure 4.11 – DRM receiver, modified from [126]

(RF) front-end. After the Analog-to-Digital Converter (ADC), a subband is selected digitally by the Digital Down Converter (DDC). To avoid interference between OFDM symbols, the transmitter adds a small guard period to each symbol. This guard period is removed from the received data stream by the Guard Time Removal (GTR) block. Small deviations between the mixing frequency of the transmitter and the receiver (which can occur in the RF front-end as well as in the DDC) result in a frequency shift of the received data stream. The Frequency Offset Correction (FOC) block in the receiver compensates for this effect and results in a corrected data stream, which is converted to an OFDM symbol by the DFT. The symbol consists of a stream of *cells* that contain the samples for the three information channels (as explained before), together with a number of *pilot* cells which can be used for synchronization and channel estimation purposes. Since the transmitter adds known pilot cells to the data stream, the receiver can use the pilot cells to estimate the quality of the channel and compensate all cells in the super frame using the *channel equalization* block. Then, the corrected cells are demapped over the three information channels such that they can be demodulated and decoded. The processing blocks are controlled by the blocks within the *global control & estimation* block, which synchronize the processing blocks and set the parameters for the filters. They provide feedback obtained from the demodulated OFDM symbols to the time domain processing blocks (see section 4.2.1). For brevity reasons, the functionality provided by the global control and estimation blocks is discussed in the next sections together with the functionality of the processing blocks.

The baseband processing of a DRM receiver includes several OFDM demodulation modes (see Table 4.6) which are used for different channel conditions. Mode A is used for short distance broadcasting with very little multipath propagation and Doppler effect. This mode enables the highest data rate of all modes, however, with the lowest robustness. In case multipath propagation does occur, mode B is chosen. With a

Table 4.6 – DRM demodulation modes

	Unit	Symbol	Mode A	B	C	D
Useful Time	[ms]	T_u	24	21.33	14.66	9.33
Guard Time	[ms]	T_g	2.66	5.33	5.33	7.33
Symbol duration	[ms]	T_s	26.66	26.66	20	16.66
Symbol rate	[Hz]	R_s	37.5	37.5	50	60
Samples per symbol (before GTR)			320	320	240	200
Symbols per frame		N_s	15	15	20	24
DFT size		N_u	288	256	176	112
#Modulated carriers		N_m	225	205	137	87
Max. datarate	[kbps]		72.0	56.1	45.5	30.6
Min. datarate	[kbps]		6.3	4.8	9.2	6.1

medium range transmission, this is the most used modulation scheme. For longer range transmission or in case of Doppler effects, mode C is better suited. The longest transmission ranges can be reached with mode D, which is similar to mode B but has a much higher resistance to multipath and Doppler effects.

Each of the modulation schemes leads to different processing requirements. The main difference is in the OFDM demodulation, which is implemented by a DFT. In the next sections the DSP blocks of Figure 4.11 and their implementation on the Montium TP are explained in more detail. An overview of their implementations is given in section 4.2.4.

4.2.1 Time domain processing

With the four modulation schemes, DRM is robust to changes in the environment as both temporal and frequency dependent errors can be detected. First, the received samples (temporal domain) are corrected and then converted to the frequency domain for further postprocessing.

4.2.1.1 Digital Down Converter

DRM uses the frequency band below 30 MHz. Within this band, many subbands can be identified on which different channels are mapped. Since the total DRM band can be sampled directly, a relatively simple front-end can be used together with digital channel selection to create a very flexible receiver. Such digital channel selection is done with a DDC [127]. First, the samples received by the ADC are multiplied by a digital mixer, such that the result contains both the sum and difference frequency as shown in Equation 4.22:

$$v_a(t) * v_b(t) = \frac{A_a A_b}{2} \left[\cos\left(2\pi\left(f_a - f_b\right)t\right) - \cos\left(2\pi\left(f_a + f_b\right)t\right)\right] \tag{4.22}$$

where $v_x(t) = A_x \sin(2\pi f_x t)$ for $x = a, b$. Assuming the channel of interest is located at the carrier frequency f_c and has a bandwidth δ. Then, the transmitted signal v_a at frequency $f_a = f_c$ is multiplied by a digitally generated frequency $f_b = f_{\text{mix}} = f_c - \delta$ (where f_{mix} denotes the mixer frequency of the receiver), the result contains a frequency component $f_a - f_b = \delta$ (which is the selected channel) and a component $f_a + f_b = 2 * f_c - \delta$. The latter component is undesired and therefore, is filtered using a low-pass filter. After the low-pass filter, the information left is only available for frequencies in the range $0 < f < 2\delta$. If $k * \delta = f_{\text{mix}}$ with k an integer value, decimation can be done easily by removing $k - 1$ samples from each sequence of k intermediate results. For other downsampling ratios where $k_1 * \delta = k_2 * f_{\text{mix}}$, the intermediate results are interpolated k_2 times, followed by k_1 times decimation[7]. This can be implemented efficiently with a *decimation filter*.

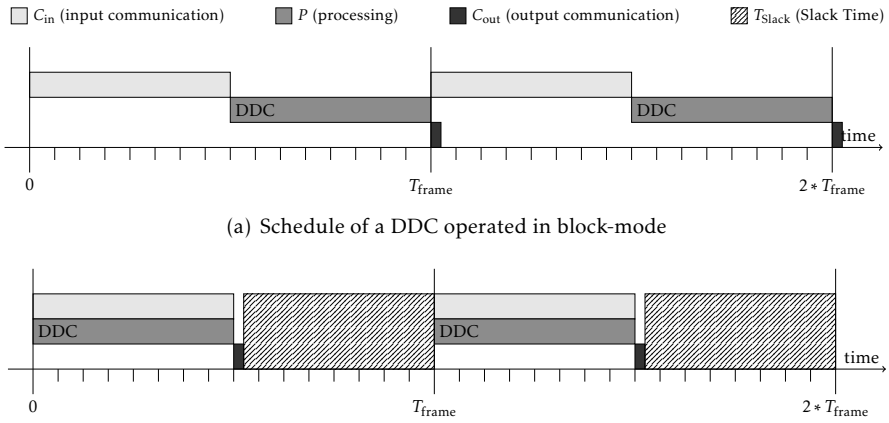
The digital clock is generated by a local Numerically Controlled Oscillator (NCO), that generates a pair of $(\cos(2\pi(f_{\text{mix}} - \delta)t), \sin(2\pi(f_{\text{mix}} - \delta)t))$ signals and multiplies these with the input stream, resulting in an in-phase signal (denoted by I) and a quadrature-phase signal (denoted by Q). The low-pass filter is implemented using two Cascading Integrating Comb (CIC) filters (one for the I and one for the Q part of the input stream) and decimated using a polyphase FIR filter. In this way, for the case discussed by Bijlsma [127], a 64.512 MHz input DRM signal is downconverted to 24 kHz.

The implementation of the DDC on a Montium TP requires 2668 clock cycles [127] to downconvert 2668 real input samples to one complex output sample. If the algorithm is operated in block-mode, the input communication time equals $\frac{2668}{L}$, the processing time equals 2668 cycles, and the output communication time requires $\frac{2}{L}$ cycles. Hence, the block-mode C/C ratio equals $\frac{2668/L+2/L}{2668} \approx 1/L$. In the streaming-mode operation mode, the 2668 input samples are received while processing, such that the communication overhead is only based on the output communication time (1 clock cycle per operation of the entire algorithm), such that the streaming-mode C/C ratio equals $\frac{1}{2668} \approx 0$. Therefore, the clock frequency for streaming-mode operation is only determined by the input sample rate (64.512 MHz), while for the block-mode operation the clock frequency has to be doubled (129.024 MHz) to provide the same output data rate. These effects are also depicted in Figure 4.12, which shows the advantage of the streaming-mode implementation to the block-mode implementation.

4.2.1.2 Guard Time Removal

The transmitted signal may suffer from multi-path effects. For example, caused by reflections of the signal against large objects like buildings. As a result, at the receiver side two or more incoming signals are received simultaneously. This is problematic in case the time difference between symbols is larger than the guard time. This will result in Inter Symbol Interference (ISI). At the transmitter side, the signal to be

[7] For example, a 2.5 MHz data stream can be downconverted to a 1 MHz data stream by first interpolating it such that a 5 MHz sampling rate is obtained, following by a 1/5 decimation resulting in a 1 MHz output stream.



(a) Schedule of a DDC operated in block-mode

(b) Schedule of a DDC operated in streaming-mode

Figure 4.12 – Example schedules of the block-mode and streaming-mode implementations of a DDC. Here, the block-mode implementation borrows a part of the next time frame for output communication, while the streaming-mode implementation introduces a considerable slack time.

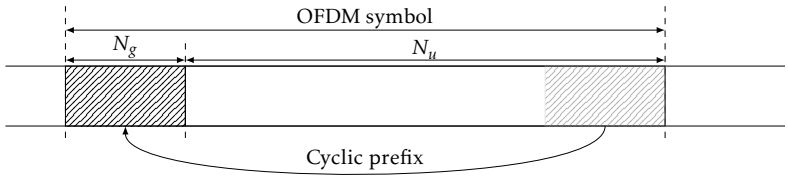


Figure 4.13 – OFDM symbol before Guard Time Removal, showing the useful part and the guard time.

transmitted is extended for a certain period by prepending it with a cyclic copy of a part of the symbol (see Figure 4.13). The extension of the original symbol is called *guard time*. Since the guard time (T_g in Table 4.6) contains redundant information, the GTR block removes that part such that the actual symbol with length $N_u = T_u * f_\delta$ remains, where f_δ denotes the DDC output data rate (see also Table 4.6).

The beginning of a symbol can be found via an auto-correlation of the received signal. The position n_ϵ of the guard time in the input stream x can be estimated using Equation 4.23:

$$\hat{n}_\epsilon = \arg\max_{n_{tr}} \left| \sum_{i=n_{tr}}^{n_{tr}+N_g-1} \frac{x^*[i]}{\left|x^*[i]\right|} \cdot \frac{x[i+N_u]}{\left|x[i+N_u]\right|} \right| \tag{4.23}$$

where x^* denotes the complex conjugate of x, $\hat{n}_\epsilon$ denotes the estimation of n_ϵ, n_{tr} is used to indicate the *trial position* (the candidate frame start position) for the correlation window ($0 \leq n_{tr} < N_u$), $N_g = T_g * f_\delta$ denotes the number of samples in the guard time interval, and arg expresses the index where the maximum correlation is

found.

Before the calculation of the auto-correlation in Equation 4.23, the input stream x first needs to be normalized; for each sample the squared length $x_{\Re\mathfrak{e}}[i]^2 + x_{\mathrm{Im}}[i]^2$ is calculated and fed into a LUT containing the inverse square root function:

$$\mathrm{LUT}(i) = \frac{1}{\sqrt{i}}, \qquad \text{where} \quad i = 0,1,\ldots,1023 \tag{4.24}$$

such that the outcome of the LUT contains $\frac{1}{|x[i]|}$. This calculation is done for each sample in x and the results are multiplied with x to obtain normalized values. Since these operations can be pipelined on the Montium ALUs, the computational costs for normalization are $\left(N_u + N_g\right)+2$ clock cycles per symbol (the 2 additional clock cycles are caused by the filling and flushing of the pipeline) [128]. After the normalization, the multiplication $x^*[i] * x[i+N_u]$ is done once for all values of i, hence it can also be implemented in $\left(N_u + N_g\right)+2$ clock cycles. The correlation sum is calculated for all trial positions in the correlation window. By reusing the intermediate accumulated value for the summation for trial position n_{tr} (see Equation 4.23), the result of the new $x^*[i+1]*x[i+1+N_u]$ is added and the result of the old $x^*[i-N_u]*x[i]$ is subtracted, the summation for trial position $n_{tr}+1$ is calculated. Therefore, the calculation of all trial positions requires $\left(N_u + N_g\right)$ clock cycles. The location of the maximum absolute correlation value within this range of trial positions is searched and used for the further FOC operation. This requires two searches through the correlation results (so $2*\left(N_u + N_g\right)$ clock cycles) to ensure local correlation maxima are not selected (see [128, section 5.4] for a detailed explanation of this problem and the implementation of the search algorithm). In total the GTR can be calculated in about $5*\left(N_u + N_g\right)+4$ clock cycles on the Montium TP.

The GTR operates on $N_u + N_g$ complex input samples[8], which can be loaded in $\frac{2*\left(N_u+N_g\right)}{L}$ clock cycles. After removing the guard time, the result consists of N_u complex samples (which can be retrieved in $\frac{2*N_u}{L}$ clock cycles) and the maximum correlation result (a single complex value, which can be retrieved in $\frac{2}{L}$ clock cycles).

The resulting C/C ratio depends on the values of N_u and N_g, which vary for the different modes of DRM. An overview is given in Table 4.7. Note that this operation can only be done efficiently in block-mode operation, because the correlation requires at least $N_u + N_g$ samples buffered in the local memories.

4.2.1.3 Frequency Offset Correction

The signal is transmitted at a certain carrier frequency f_c within the DRM band. A mixer in the RF front-end shifts the carrier frequency f_c to a low frequency by

[8] This operation requires at least $\left(2 * N_u + N_g\right)$ samples to be available in memory, because of the N_u possible trial positions for which the correlation is done using the next $\left(N_u + N_g\right)$ samples. However, because the last N_u samples are used again for the next symbol, they do not have to be loaded again during the next execution.

Table 4.7 – Communication to computation ratio for the GTR for the 4 DRM modes

Mode	N_u	N_g	T_{comm}	T_{comp}	C/C ratio
A	288	32	$1218/L$	1604	$0.76/L$
B	256	64	$1154/L$	1604	$0.72/L$
C	176	64	$834/L$	1204	$0.69/L$
D	112	88	$626/L$	1004	$0.62/L$

multiplying it by a locally generated frequency f_{mix} close to the carrier frequency, such that an ADC can sample the signal. However, because the actual mixer frequency f_{mix} slightly deviates from the transmitted carrier frequency f_c, the resulting sampled values contain a small *frequency offset*. As a result, the auto-correlation discussed in the previous section contains a small error since the frequency offset for the useful part of the symbol is slightly different from the frequency offset during the guard time period. The FOC block compensates for this error. This means that every sample in the OFDM symbol needs to be multiplied by a correction factor.

The frequency offset Δf introduced by the mixer causes a phase shift of the input signal x_{in}:

$$x[n] = x_{in}[n] \cdot e^{-j\varphi_f[n]+\varphi_0} \tag{4.25}$$

where φ_0 is a constant phase offset that is corrected by the channel equalization block. The other phase error component, φ_f, is defined as follows:

$$\varphi_f[n] = 2\pi \frac{\Delta f}{f_{mix}} n \tag{4.26}$$

The frequency offset can be estimated using the following equation [126]:

$$\Delta\hat{f} = \frac{1}{2\pi}\frac{1}{T_u} \angle \left(\sum_{i=n_\epsilon}^{n_\epsilon+N_g-1} \frac{x^*[i]}{\left|x^*[i]\right|} \cdot \frac{x[i+N_u]}{\left|x[i+N_u]\right|} \right) \tag{4.27}$$

where $\angle x$ equals the phase of the complex value x.

Note that this equation resembles Equation 4.23. By using the phase of the maximum correlation result found by the GTR, the estimated phase error $\hat{\varphi}_f[n]$ can be calculated using Equation 4.26. Then, the sample $x[n]$ is multiplied with a correction factor $e^{j\cdot\hat{\varphi}_f[n]}$ which is calculated using a LUT. Again, these operations can be pipelined such that the correction of all N_u samples can be done in $N_u + 2$ cycles by the Montium TP (due to the pipeline delay of 2 additional clock cycles, similar as mentioned before).

The FOC can be operated in either a block-mode operation or a streaming-mode operation. In the block-mode, the vector containing N_u complex samples and the maximum correlation result from the GTR are read in $\frac{2N_u+2}{L}$ cycles. The result is retrieved in $\frac{2N_u}{L}$ cycles. Hence, the C/C ratio for block-mode operation equals about $\frac{(4N_u+2)/L}{N_u+2} \approx 4/L$ for all DRM modes. For the streaming-mode implementation, the communication adds no overhead to the processing, so the C/C ratio equals 0.

4.2.2 Time domain to frequency domain conversion

The DFT converts a sample stream from the time domain to the frequency domain, where the output of the conversion is called an OFDM *symbol*. The symbol consists of many sub-carriers (for example, a mode B symbol contains 256 sub-carriers) of which a part is used to carry information (for example, 207 out of 256 sub-carriers are used in mode B). Those sub-carriers carrying data are called *cell*s.

In addition to the DFTs presented in Table 4.6, additional DFTs and iDFTs were required for a DRM receiver to improve the channel quality and to suppress possible interferers [129, section 3.4.1]. These transforms can be separated in two classes: the class which consists of DFTs of 2^N points and the class which consists of DFTs of size $2^p * (2q+1)$, where $N = [4 \ldots 11]$, $p = [4 \ldots 7]$ and $q = [3 \ldots 7]$. The implementation of the DFTs of the first class (also known as radix-2 FFT) is presented in section 4.1.2, and for the second class (here referred to as non-power-of-two FFT), an efficient implementation of the conventional DFT is presented in section 4.1.3.

4.2.3 Frequency domain processing

Within each super frame, pilot cells are transmitted that can be used to detect the channel quality. The first few symbols of each super frame contain the Service Description Channel information about the modulation scheme used for the Main Service Channel. For the synchronization of frames, the first symbol of each frame contains *time pilot* cells at fixed sub-carrier positions with a known constant value (called *boost factor*) of which the amplitude has to be normalized to $\sqrt{2}$. The obtained normalization factor for such a sub-carrier position is used for the channel equalization, to correct the same sub-carrier position in all next symbols, until the next time pilot cell is transmitted on the sub-carrier.

4.2.3.1 Channel equalization

During the transmission, the signal is distorted because of noise, Doppler shift and multi-path effects. Some of these may cause frequency dependent distortions, which means that individual cells of the symbol may have to be adapted differently. The *channel processing* block corrects the gain and phase of all cells by multiplying each cell with a complex number obtained from the channel estimation block. In order to detect frequency dependent errors in the signal, a cyclic pattern of *frequency pilot* cells with a known gain and phase is transmitted within each frame. Hence, by analyzing these pilot cells, the channel distortion can be estimated such that the rest of the frame can be corrected.

For the equalization of one symbol consisting of N_m modulated carriers (see Table 4.6), each carrier is multiplied by a correction factor. Hence, the total computational costs for the channel equalization is N_m complex multiplications, which can be done in $N_\mathrm{m} + 2$ clock cycles on the Montium TP (2 clock cycles delay due to the pipelining overhead). During the reception of a DRM superframe, the channel is assumed to be constant. Therefore, the correction factors calculated by the channel estimation block are stored in the local Montium memories once for each superframe.

For the block-mode implementation of the channel equalization, a symbol is stored inside the local memory before processing ($\frac{2*N_m}{L}$ clock cycles because of complex values) and retrieving the result from the local memory is done in an equal number of clock cycles. The total C/C ratio for block-mode is therefore $\frac{4N_m/L}{N_m+2} \approx 4/L$. The streaming-mode implementation combines input and output communication with the computation, such that the C/C ratio becomes $\frac{0}{N_m+2} = 0$.

4.2.3.2 Cell demapping

After equalization, the non-pilot cells in the symbol are split into the three information channels FAC, SDC and MSC. For each of the transmission modes, the order of demultiplexing in a DRM super frame is fixed. Hence, each of the information channels can be created by reading cells from the super frame in that particular fixed order.

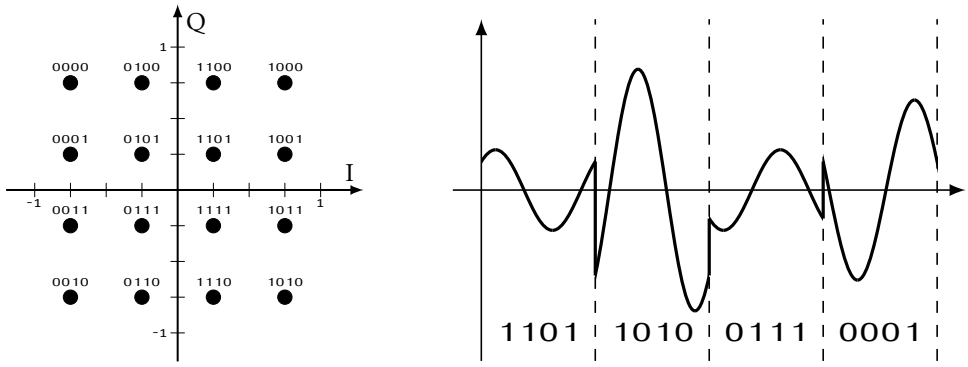
The demultiplexing order of the symbols is defined using a set of equations to calculate the indices [124]. A direct calculation of the equations is relatively expensive, as they cannot easily be defined in terms of differential equations. An effective implementation for the indexing could be the use of a LUT containing indices. In total, a super frame contains at most 10125 cells (for mode A, where 3 frames each consist of 15 symbols containing 225 cells) to 6264 cells (for mode D, where 3 frames each consist of 24 symbols containing 87 cells). Hence, the memory requirements for implementing a LUT containing all indices is considerable. Moreover, cell demapping is not computationally intensive. Therefore, the cell demapping was considered to be executed on a different processor architecture than the Montium TP, for example on an ARM processor [125].

4.2.3.3 QAM demapping

After cell demapping, the information channels are demodulated to a bitstream. This is done via *demapping*, where each cell (a complex (I, Q) signal) is converted to bits. The FAC is always modulated with 4-QAM, whereas the SDC may be either modulated using 4-QAM or 16-QAM and the MSC can be modulated using 4-QAM, 16-QAM or 64-QAM. Here only the 16-QAM modulation is discussed, because its principles are similar to those of the 4 and 64-QAM variants.

Figure 4.14(a) shows an example of the *constellation diagram* for 16-QAM, which contains all possible modulation points (depicted by the black dots) and their associated codes (bit codes close to the dots). Whenever a cell is received, its I and Q component will probably deviate from the constellation points because noise is added at the receiver. The constellation point closest to the point received has the highest probability to be the actual transmitted point, hence that point is selected. The sequence of bits obtained after QAM demapping is further processed, for example with a Viterbi decoder and MPEG-4 decoder.

The mapping of the (I, Q) values onto QAM constellation point associated codes can be varied depending on the encoding technique used. A flexible solution for demapping these varying constellation mappings is the usage of a LUT which is addressed using the I and Q values of a cell and returns the bits associated to that



(a) 16-QAM constellation diagram (b) Example 16-QAM modulated stream

Figure 4.14 – 16-QAM modulation

cell. Since, for the Montium TP, the maximum size of a single LUT is 1024 entries, the I and Q parts of a cell are combined to a 10-bit address. A lookup operation can be done by sending a 16-bit value to the AGU (see also section 2.1.1.6), which can be instructed to select either the 10 MSBs of the value (if the value is a signed fixed-point represented value) or by using the 10 LSBs of the value (if the value is an unsigned integer value). The 10-bit address is composed from the 5 MSBs from the I part of the cell concatenated with the 5 MSBs from the Q part of the cell.

For the integer lookup, first the 5 MSBs of the I part are selected by applying a bit mask and the Q part is shifted over 11 positions to the right (inserting zeroes at the left side):

$$I_{\text{msb}} = I\&\text{“}1111100000000000\text{”} \tag{4.28}$$

$$Q_{\text{shift}} = Q >> 11 \tag{4.29}$$

Then, the address a_{int} is generated by combining the shifted I_{msb} field (with zeroes inserted at the left) and the Q_{shift} field using a bitwise or:

$$a_{\text{int}} = I_{\text{msb}} >> 6 | Q_{\text{shift}} \tag{4.30}$$

The calculation of I_{msb} and Q_{msb} (Equation 4.28 and 4.29) can be done simultaneously in one clock cycle and the calculation of a_{int} (Equation 4.30) can be done in a second clock cycle.

The fixed point lookup can be implemented slightly more efficiently:

$$I_{\text{msb}} = I\&\text{“}1111100000000000\text{”} \tag{4.31}$$

$$Q_{\text{shift}} = Q >> 5 \tag{4.32}$$

where the right shift of the Q part is a logical shift, such that zeroes are inserted at the left[9]. The address a_{fixed} is generated by combining the I_{msb} and Q_{shift} fields using

[9] After the right shift, applying a logic & operation to mask the lower 11 bits does not modify the result. Therefore, the masking operation is not done.

a bitwise or:

$$a_{\text{fixed}} = I_{\text{msb}} | Q_{\text{shift}} \tag{4.33}$$

The calculation of I_{msb}, Q_{shift} and a_{fixed} can be implemented on one Montium ALU in a single clock cycle.

For both implementations, the actual demapping now only consists of a lookup in a memory. Since the Montium TP memories can be used as 1024 entry LUTs, where for the integer lookup the 10 LSBs of the address are used and for the fixed point lookup the 10 MSBs of the address are used. The lookup itself requires one clock cycle. Hence, the demapping of one cell can be done in 2 clock cycles (using a fixed point lookup) or in 3 clock cycles (using an integer lookup). The operations performed in these 2 or 3 clock cycles can be fully pipelined, such that one lookup can be started each clock cycle. Both solutions offer an identical accuracy as they both use the entire LUT. Its 1024 entries are used as 32 columns (for the I part of the cell) and 32 rows (for the Q part of the cell), where a column can be selected by the 5 upper bits and a row can be selected through the 5 lower bits. The 64-QAM demodulation required by the MSC channel uses an 8x8 constellation diagram, which is encoded using 3 bits for the I part and 3 bits for the Q part of a cell. Because 5 bits are available for both the I and Q parts, the resolution provided by the LUT is high enough to allow for accurate demodulation. The number of cells to be demapped depends on the transmission mode and spectrum occupancy [124, section 7.1]. Therefore, at most N_{m} cells per symbol have to be demapped. This can be done in $N_{\text{m}} + 1$ clock cycles for the fixed-point lookup implementation. Since reading the input and storing the demapped result requires two additional clock cycles, the total calculation costs $N_{\text{m}} + 3$ clock cycles for the fixed-point mapping and $N_{\text{m}} + 4$ clock cycles for the integer mapping. The communication time for the block-mode operation equals $T_{\text{comm}} = 2\frac{2N_{\text{m}}}{L}$ and for the streaming mode all communication is done while the computation takes place, hence $T_{\text{comm}} = 0$. Therefore, the C/C ratio for block-mode equals $\frac{4\frac{N_{\text{m}}}{L}}{N_{\text{m}}+3} \approx \frac{4}{L}$ and for the streaming-mode it equals $\frac{0}{N_{\text{m}}+3} = 0$.

The sequence of bits obtained after QAM demapping is further processed, for example in a Viterbi decoder and MPEG-4 decoder.

4.2.4 DRM implementation overview

An overview of the Montium TP implementation costs for each of the baseband processing blocks presented in Figure 4.11, is given in Table 4.8. Note that the numbers given for the DDC are independent of the transmission mode. For the FFT implementation costs, refer to section 4.1.1.

Looking at the C/C ratio columns in Table 4.8, it can be concluded that the streaming-mode implementation for most operations introduces a considerably lower overhead in communication.

The combination of all blocks discussed in the previous sections is presented in an SDF model depicted in Figure 4.15. It is based on the implementation costs for a mode A receiver; for the other modes, the model is identical while only the execution times differ slightly.

Table 4.8 – Implementation costs for the DRM baseband processing operations, expressed in clock cycles per OFDM symbol for each of the DRM transmission modes. Cell demapping is assumed to be performed by a GPP as its operations cannot be mapped on the Montium TP efficiently.

	Mode				C/C ratio ‡	
	A	B	C	D	block mode	streaming mode
DDC	2668	2668	2668	2668	$1/L$	0
GTR	1604	1604	1204	1004	$0.68/L$	$0.68/L^*$
FOC	290	256	176	112	$4/L$	0
FFT	1450	1040	960	472	$0.675/L$	0.423
Channel eq.	225	205	137	87	$4/L$	0
Cell demapping†	–	–	–	–	–	–
QAM demapping	226	206	138	88	$4/L$	0
Total	6463	5979	5283	4431		

† Cell demapping was not implemented on the Montium TP and is assumed to be implemented by an external GPP.
‡ C/C ratios are based on the average of the 4 DRM modes.
* The GTR has only been implemented for block-mode operation. The streaming-mode implementation of the GTR is comparable, where first the entire input symbol is stored in memory, execution is enabled and after finishing the resulting symbol is streamed out.

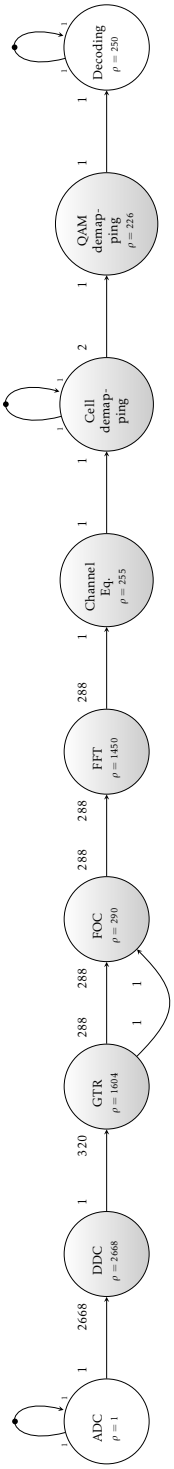


Figure 4.15 – SDF model of DRM receiver. Mode A is assumed.

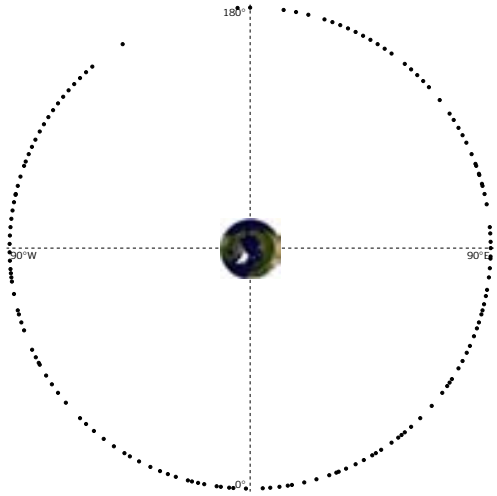


Figure 4.16 – DVB-S satellites in orbit

4.3 Mobile DVB-S receiver

The DVB-S [130, 131] standard uses the Ku-band (10.7 to 12.75 GHz) for broadcasting multi-media streams. Large dish based earth uplink stations transmit the DVB-S signal to the satellite. An uplink signal is received by the satellite and retransmitted by a *transponder*. Each transponder contains a receiver, amplifier, transmitter and an antenna that sends the received signal in a certain frequency band back to earth. In total, tens of signals can be transmitted simultaneously within the Ku-band, where each individual signal is transmitted by a separate transponder. To avoid interference between such transponders of satellites close to each other, the spectrum assignment for these transponders is done such that nearby satellites do not reuse the same spectrum. The shape of the transponder antenna determines how the Electro-Magnetic (EM) field is created. Such a field can be described by a *field vector* which consists of two polarization components [132]. When the phase difference between the components is $\pm 90°$ and the field components are equal in amplitude, the field is said to be *circularly polarized*. When the phase difference is $0°$ or $180°$, the field is said to be *linearly polarized*.

The type of antenna used determines the polarization type of the EM wave. For example, a vertically mounted antenna for terrestrial broadcasting typically transmits a vertically polarized signal. Hence, the receiver also uses an antenna that is sensitive to vertically polarized signals (for example, the antenna on a car). Another reason for choosing either linear or circular polarization is the effects of the channel. Due to reflection by rain drops, a Left-hand side (LHS) circularly polarized wave may be inverted to a Right-hand side (RHS) circularly polarized signal. Linear polarization is less affected by rain drops, hence this polarization technique is used for DVB-S.

The DVB-S standard does not define a minimum distance between the longitude of two satellites. Figure 4.16 shows an overview of satellites currently orbiting

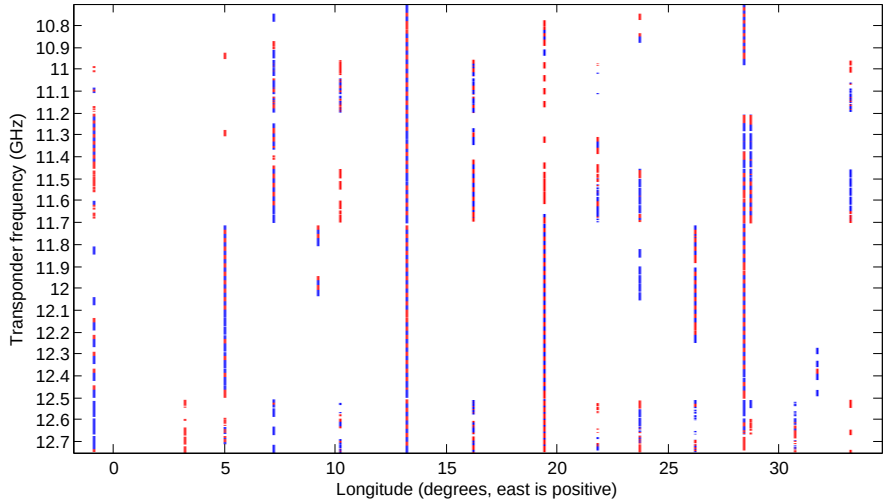


Figure 4.17 – Snapshot of 10.7 – 12.75 GHz spectrum usage. The horizontal axis indicates the east longitude of a satellite, while the vertical axis shows the transponder usage of that satellite.

within the Clarke belt[10]. For each satellite, it shows the transponders in a part of the spectrum range, where both horizontally and vertically polarized transponders are displayed. An observation of the spectrum usage by the satellites, depicted in Figure 4.17, shows that the typical spacing between two satellites broadcasting in the same frequency range with the same polarization is about 5° of latitude.

The bandwidth used for each transponder is about 54 MHz; however, the exact bandwidth and spacing between transponders can be chosen arbitrarily. DVB-S uses the full transponder bandwidth to modulate a single carrier, which on its turn can consist of a multiplex of one or multiple data streams. This is comparable to the cell multiplexing of multiple information channels into a DRM stream, as presented in section 4.2. The data streams in DVB-S can contain either MPEG-2 audio/video streams or it can be used for other services, like subtitling, electronic program guides or weather information. Although DVB-S requires a line-of-sight connection, it can cope with severe attenuation, since a Signal to Noise Ratio (SNR) of 16 dB is enough for correct demodulation[11].

Conventional DVB-S systems use dish antennas, which are pointed directly at the satellite. The geometry of the dish determines the signal quality. In order to have a sufficient SNR, the dish must be steered very accurately. Typically, this is done by hand. Therefore, the dish is unsuitable for mounting on vehicles that are moving, such as cars or yachts. A dynamically steered antenna can be used to ensure correct pointing while moving. For example, by moving the antenna using a servo motor, the pointing can be controlled [7]. However, using such mechanics

[10] Satellite information obtained from `http://joshyfun.cjb.net`.

[11] In contrast to DRM, for DVB-S no adaptive channel modulation techniques are applied. Its successor, DVB-S2, does support different modulation schemes to increase the channel utilization and data rates [133].

Figure 4.18 – Linear phased array mounted on the roof of a car

is expensive, heavy, energy consuming and slow. An electronically steered phased array system is more beneficial, since it consists of multiple antennas mounted at fixed positions. Moreover, it enables the reception of broadcasts from multiple satellites simultaneously by using two or three independent beams for a single phased array antenna. This is useful, when multiple users want to receive signals from different satellites simultaneously. The next sections deal with beamforming and beam steering using a phased array antenna based DVB-S receiver, mounted on a car as shown in Figure 4.18.

4.3.1 Phased array antenna processing

A phased array receiver consists of multiple antennas which are used to create a larger virtual single antenna. Phased array systems are based on coherent summation of signals from multiple antennas in an array layout to make a transceiver directional (see Figure 4.19). For in-phase signals, the waves add up constructively and for 180° out-of-phase signals the waves add up destructively. Assume a single omni-directional wave source, emitting a spherical waveform in time and space. Equation 4.34 describes the wave observed from a distance l:

$$s(t,l) = A \cdot \cos(\omega t \pm kl) \tag{4.34}$$

where A is the amplitude, ω the frequency, k the wave number, t time and l the path length from the source.

For a source in the far field perpendicular to the array, the wavefront is considered planar (l is equal for all antennas) and the received signals add up constructively. If the plane of the array is not perpendicular to the direction of the source, the wavefront arrives at different times at the antennas.

Assume the phased array consists of a sequence of antennas placed at a fixed distance d apart on a straight line. Such an antenna is referred to as a Uniform Linear Array (ULA). A wavefront arriving at an angle θ incident to the array then travels a

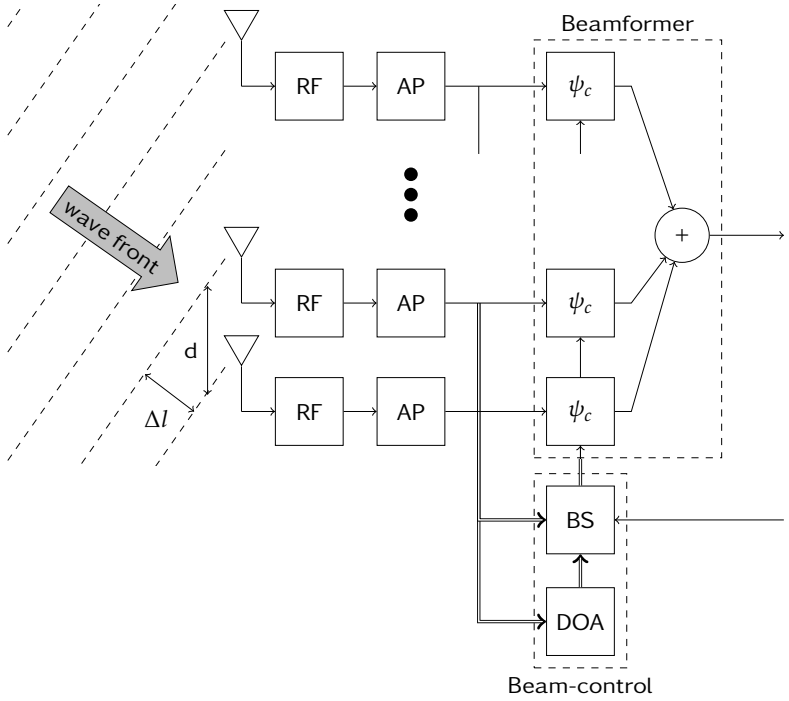


Figure 4.19 – Generic phased array receiver

distance $\Delta l = d \cdot \sin(\theta)$ further to each next antenna, which results in a time delay $\Delta t = \frac{\Delta l}{c}$ between the signals (where c is the propagation speed of radio waves).

Definition 15 (Narrowband)**.** A signal in the band between f_l and f_h is said to be a *narrowband* signal if $\frac{f_\mathrm{h}-f_\mathrm{l}}{(f_\mathrm{h}+f_\mathrm{l})/2} < 0.01$ [134].

If the signal is a narrowband signal, a time delay is considered to result in a phase shift ($\Delta\varphi = \omega \cdot \Delta t$) giving rise to the term *phased array*. The result of the addition of all $\mathcal{N}$ antenna signals, called *array factor*[12], then can be described as follows:

$$S_\mathrm{a}(\theta) = \sum_{i=1}^{\mathcal{N}} a_i e^{j\Phi_i} = \sum_{i=1}^{\mathcal{N}} a_i e^{j\frac{2\pi}{\lambda}(\mathcal{N}-i)d\sin(\theta)} \tag{4.35}$$

where θ is the so-called *scan angle*, a_i denotes the amplitude of the signal received at the i^{th} antenna, $\Phi_i = (1-i)\,\Delta\varphi$ equals the phase difference for antenna i with respect to antenna 1 and $\lambda = \frac{c}{f}$ is the wave length of the signal carrier.

The solid line in Figure 4.20 shows an example array factor of an array consisting of 8 antenna elements. It shows the maximum sensitivity (the *main beam* at 0°),

[12] Different definitions of the array factor are given in literature [132, 135–137]. For this thesis, we adopt the definition given by Visser [132].

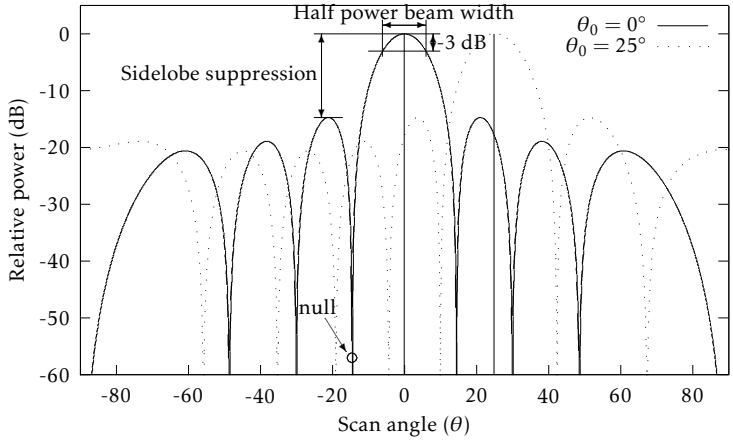


Figure 4.20 – Effect of beam steering in direction θ_0 on the array factor $S_a(\theta)$ (using 8 antenna elements). The solid line shows the pattern when all antenna samples are added without compensation, while the dashed line shows the array pattern when a linear increasing phase shift is added to all elements.

maximum suppression (*null*s, for example at 14° and 30°) and *sidelobe*s (local maxima at 21°, 38°, etcetera)[13]. When the signals received by the individual antenna elements are multiplied by different $e^{j\varphi_i}$ values, the main beam can be steered into another direction θ_0 [132, 137]. The result of this multiplication is shown in Equation 4.36:

$$S_a(\theta) = \sum_{i=1}^{\mathcal{N}} a_i e^{j\left[\frac{2\pi}{\lambda}(\mathcal{N}-i)d\sin(\theta)+\varphi_i\right]} \tag{4.36}$$

For example, the dashed line in Figure 4.20 shows the directional sensitivity of the same array where the main beam is steered ($\theta_0 = 25°$) by applying a linear *phase taper* to all antenna elements such that $\varphi_i = -\frac{2\pi}{\lambda}(N-i)\,d\sin(\theta_0)$. Then, Equation 4.36 can be written as follows:

$$S_a(\theta) = \sum_{i=1}^{\mathcal{N}} a_i e^{j\frac{2\pi}{\lambda}(\mathcal{N}-i)d[\sin(\theta)-\sin(\theta_0)]} \tag{4.37}$$

It is also possible to change the shape of the beams by applying an *amplitude taper*, resulting in more suppression in the side beams as shown in Figure 4.21. This is done by modifying the gain a_i applied to individual antenna elements. By choosing $a_i = 1$ for $i = 1, 2, \ldots, \mathcal{N}$ (*uniform taper*) the main beam is as small as possible, but the sidelobe suppression is limited (-13 dB for the first sidelobe). Each tapering function has different characteristics; an overview of many different functions can be found

[13] Note that only the *front face* of the array antenna (-90° to 90°) is shown. The range between -90° to -270° (the *rear face*) has an identical, mirrored array pattern. For brevity reasons, figures in this thesis only show the front face of the array pattern.

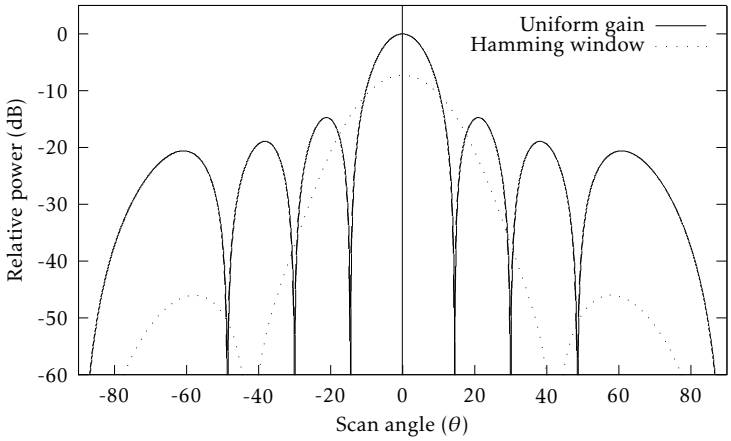


Figure 4.21 – Effect of gain tapers on the array factor (using 8 antenna elements). The solid line shows an uniform gain taper and the thin line shows a Hamming window based tapering.

in [137]. For example, a Hamming window can be used for sidelobe suppression by choosing the antenna gain elements as follows:

$$a_i = \frac{25}{46} + \frac{21}{46}\cos(2\pi p_i) \tag{4.38}$$

where $p_i = i - \frac{\mathcal{N}+1}{2}$ equals the position (in units of d) of antenna element i with respect to the center of the array [137]. If using such a window, the sidelobes are suppressed by -43 dB at the costs of an increasing main beam width (see Figure 4.21). Hence the array has become less sensitive to strong interferers far from the targeted direction but more sensitive to interferers close to the main beam. In this thesis we assume a uniform taper, because the directivity of the main beam is considered very important and uniform tapering provides a small main beam.

The DVB-S requirements for beam width, array gain, and the location of the satellites can be used to determine the phased array requirements. The satellite location with respect to the receiver is determined by the receiver's location on earth. As an example, for a receiver located in Europe, the satellite's altitude is about 43° (for Athens, Greece) to 68° (for Oslo, Norway) with respect to the orthogonal direction to the horizontal plane at that location. Since the array is mounted horizontally on a car roof, it should be designed such that the beam can be pointed in the region of $40° < \theta_0 < 70°$ (and in the mirrored direction $-70° < \theta_0 < -40°$) in order to support satellite reception in the whole continent with a varying car orientation. The antenna requirements for DVB-S are determined by the main beam width and the total antenna gain. Figure 4.22 shows how increasing the number of antennas from 4 to 8 changes the resulting beam width. For estimating the minimum number of antennas required for DVB-S, Figure 4.23 can be used. It shows how the main beam width changes with the steering angle θ_0, for multiple array sizes. As can be seen, the 5° beamwidth required for DVB-S for pointing a beam into the region between −70°

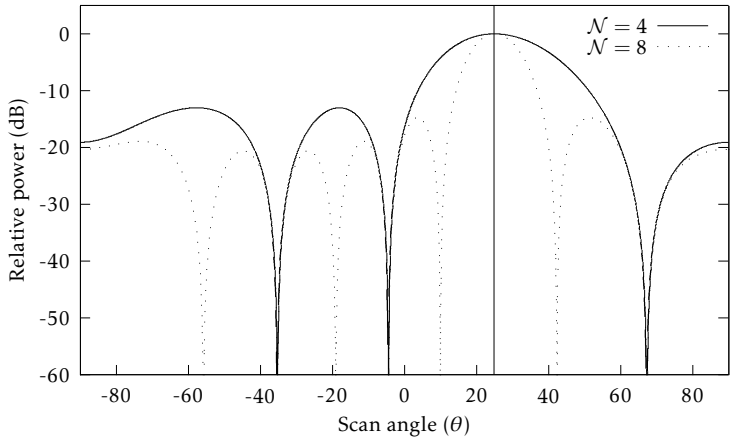


Figure 4.22 – Effect of the number of antenna elements on the resulting beam pattern. The main beam power is normalized to 0 dB, such that sidelobe suppression of both arrays can be compared. In this example, the main beam was steered to 25° for a 4-element array and an 8-element array.

$< \theta_0 < 70°$ can be obtained by using 64 antenna elements (indicated by the black square in Figure 4.23). Therefore, for the remainder of this thesis we will assume $\mathcal{N} = 64$ for the DVB-S receiver[14].

Using a phased array antenna for the reception of satellite signals enables multi-channel reception. This can be done by either pointing multiple beams at different satellites or by receiving multiple channels from a single satellite using a single beam. The latter case requires a wideband receiver that covers at least the bandwidth range in which both channels are transmitted. Hence, for example if $f_{c_1} = 10.79$ GHz and $f_{c_2} = 11.23$ GHz, the minimum bandwidth range should be 440 MHz. Because of the relatively large frequency difference the narrowband assumption is is not valid and the beamformer can not be controlled with a phase shifter solution. As a result, the digital implementation of such a receiver is very expensive. Therefore, we assume a multi-channel DVB-S receiver that uses 3 beams (denoted $\mathcal{B} = 3$ during the remainder of this thesis), each of which can be tuned individually.

Figure 4.24 shows a generic phased array receiver structure. After the RF front-end, the analog signal is quantized and sampled by an ADC. Then, using an equalization filter, the received sample stream is corrected for electrical or mechanical distortions of the antenna, the front-end and the wireless channel. The signals are then combined by the beamforming processing (beamformer) to create a resulting signal, such that the main beam is pointed into a direction of interest.

Definition 16 (snapshot)**.** A snapshot consists of the values obtained when sampling

[14] A ULA based array can only be used to receive signals in one plane. Therefore, in reality the array should be a 2-dimensional structure consisting of 64×64 antenna elements. However, in this thesis we assume a 1-dimensional array can be used, since this decreases the modeling complexity of the receiver and processing while its concepts remain the same.

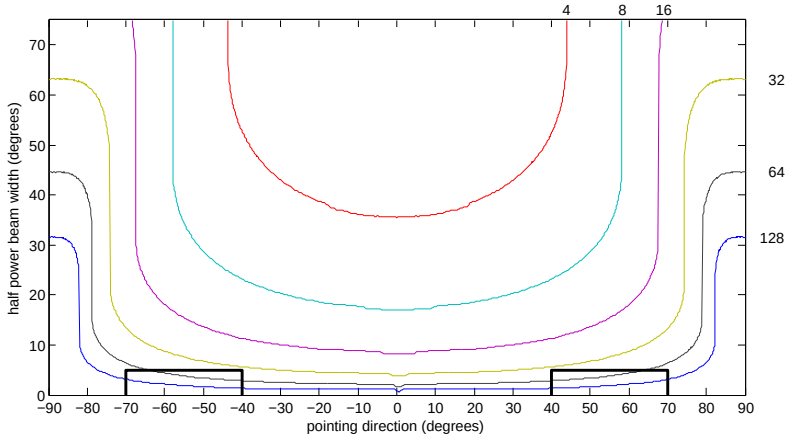


Figure 4.23 – Main beam width varying over steering direction for several phased array sizes. For all arrays, $d = \lambda/2$, using a constant gain taper and linear phase taper. For all array sizes, the front main beam and rear main beam join when the pointing direction nears $\pm 90^\circ$, resulting in a sudden increase of the main beam width.

all antenna elements of a phased array at the same time instant.

The Beamsteering (BS) block controls the steering angle and shape of the formed beam by generating a so-called *steering vector* (denoted by $\vec{\phi}$), which is in fact a combined gain and phase tapered weighting vector that is used by the beamforming block. Note that the same antenna signals can be used to form multiple beams in different directions simultaneously. For each beam, the antenna signals are combined with a different steering vector. In a dynamic environment where transmitters and/or the receiver are continuously moving, adaptive processing is required for the receiver to detect and follow transmitters. One example would be the application of the DVB-S satellite receiver mentioned in section 4.3 mounted on a vehicle, for example on the roof of a car or on the cabin of a yacht. Figure 4.24 shows the phased array processing chain that is used for DVB-S reception, in case beam steering is required. For each of the blocks in the chain after the ADC, we will shortly describe the functionality and the number of clock cycles needed when it is implemented on a Montium TP.

4.3.1.1 Calibration and equalization

For accurate beamforming, it is important that the gain (over the whole frequency band) and delay of each antenna up to the beamforming is equal. To realize this, antenna *calibration* and/or *channel equalization* has to be applied.

Calibration refers to the correction of antenna signals, required to adapt amplitudes and phases so the individual antenna signals can be used by the beamformer. The RF frontend response is sensitive to changes of the *thermal noise* caused by temperature changes of the environment [135], resulting in a time dependent error. Another distortion in the frontend is *jitter* in the Local Oscillator (LO) distribution path [137], which can be most accurately described as the period frequency displace-

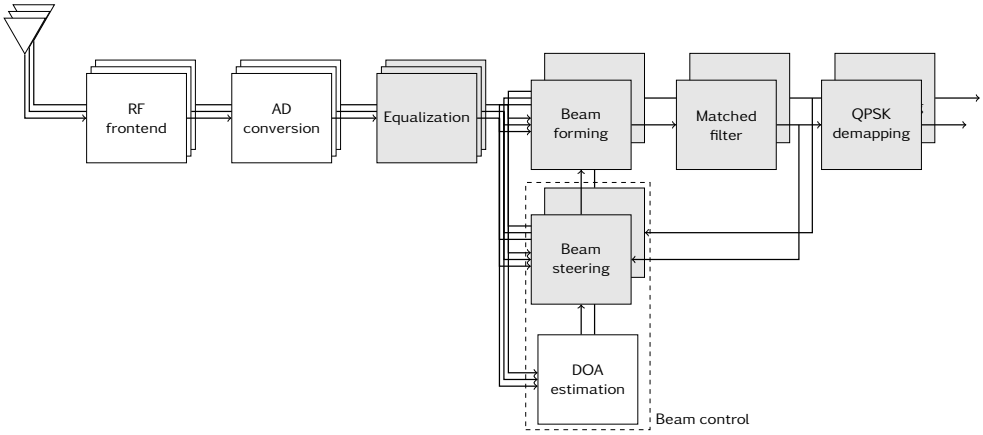


Figure 4.24 – Main system blocks in the phased array receiver used for DVB-S reception

ment of the signal from its ideal location. In the RF front-end, jitter introduces phase errors such that the ADC may sample wrong data. These effects, however, cannot be avoided as they vary quickly over time.

Manufacturing problems (for example, incorrect antenna placement or bad connections in the path between the antenna and the ADC) and temperature deviations are much less changing and therefore, these effects can be calibrated every few milliseconds to seconds to improve the overall SNR of a the phased array antenna. Such calibration can be done using an inline circuit [138] or by generating test data that is fed into the antennas via an internal network. However, for both solutions, additional hardware in the analog frontend is required, which is not covered in this thesis. Therefore, calibration is not within the scope of this thesis.

Equalization is done per antenna to correct for frequency dependent variations in the frontend. Such variations can be corrected by using an $\left(\mathcal{F}_{\text{eq}}-1\right)^{\text{th}}$ order complex FIR filter, where the minimum order can be determined based on the required SNR and filter roll-off characteristics. For this thesis, we consider an equalization filter using $\mathcal{F}_{\text{eq}}=5$ taps [162]. As such a filter is needed for each antenna, the amount of processing can easily become as large as the beamforming itself. However, it is independent of the number of beams that are formed. As discussed in section 4.1.4.2, the implementation of a complex FIR filter on the Montium TP costs 1 clock cycle per filter tap per input sample. Therefore, the processing costs for all $\mathcal{N}=64$ antenna filters sum up to $\mathcal{N}*\mathcal{F}_{\text{eq}}=320$ clock cycles (averaged per input sample).

For the block-mode implementation, the processing costs depend on the size of the block that is filtered. Using the FIR filter mentioned above, filtering of one block of length $5*L_b$ requires $5*L_b+1$ cycles. Loading the block of input data into the Montium TP using a DMA transfer via L channels then takes $\frac{2*L_b}{L}$ cycles. Retrieving the filtered antenna data takes another $\frac{2*L_b}{L}$ cycles. The total C/C ratio for the block-mode equalization filter therefore becomes $\frac{4*L_b/L}{5*L_b+1}=0.8/L$.

A streaming-mode implementation of the same FIR filter allows for reading a sample, filtering the sample in 2 cycles and writing the sample in another cycle. Pipelining can be used to continuously read samples while processing the previous sample and storing a previous result, such that the communication does not slow down the processing. After processing L_b samples, the communication only adds $2/L$ clock cycles delay due to the last write action. Therefore, the streaming-mode C/C ratio equals $\frac{2/L}{5*L_b} \approx 0$ (assuming the input stream with length L_b consists of at least tens of samples).

4.3.2 Beamformer

To implement phase shifting in the digital domain, a FIR filter can be used, which consists of complex multiplications followed by addition. In the narrowband case, a 1-tap FIR filter per antenna is sufficient, which costs 1 complex multiplication per antenna. The phase correction with a complex multiplication has the advantage that it is flexible with respect to the beam-shape and its angle (see figures 4.20 and 4.21). In case of multiple targets, multiple beamformers are required and each beamformer can be pointed in a different direction independently, while its shape may differ from other beams. However, the costs for this flexibility are considerable as processing costs increase linearly with the number of beams. The calculation of the i^{th} beam is described by Equation 4.39:

$$b_i[t] = \sum_{n=1}^{\mathcal{N}} \vec{\psi}_i^n[k] \cdot \vec{x}_n[t] \tag{4.39}$$

where $\vec{x}_n$ denotes the sample stream received from the n^{th} antenna and $\vec{\psi}_i^n[k]$ denotes the steering coefficient for that antenna to form the i^{th} beam during the k^{th} beam update interval. The update frequency depends on the dynamics of the device the antenna is mounted on (for example, a car or yacht).

Definition 17 (beam update interval)**.** The beam direction is updated once during each beam update interval, which consists of k_u antenna samples.

Equation 4.39 can be implemented with one complex multiplication and one accumulation per beam per sample per antenna. This can be executed in 1 clock cycle for the Montium TP, so for $\mathcal{N}$ antennas and $\mathcal{B}$ beams, the beamforming costs $\mathcal{N} * \mathcal{B}$ clock cycles per snapshot. For the DVB-S receiver discussed before ($\mathcal{N} = 64$ and $\mathcal{B} = 3$), the beamformer costs are 192 clock cycles per snapshot. In block-mode operation, first the snapshot is transported into the Montium TP using a DMA transfer in $2\mathcal{N}/L$ clock cycles (factor 2 because the antenna data is complex). The $\mathcal{B}$ resulting beam values are retrieved using $2\mathcal{B}/L$ cycles. Hence, the C/C ratio for block-mode beamforming is $\frac{2*(\mathcal{N}+\mathcal{B})/L}{\mathcal{N}*\mathcal{B}} = \frac{2*67/L}{192} = 0.70/L$. In streaming-mode operation, the snapshot is loaded simultaneously with the multiplication by the steering vector for the 3 beams. After the processing stage, the 3 results are communicated. Therefore, the streaming-mode C/C ratio equals $\frac{2*\mathcal{B}/L}{\mathcal{N}*\mathcal{B}} = \frac{2/L}{\mathcal{N}} = 0.03/L$.

4.3.3 Beamsteering

The pointing direction and shape of the beam are controlled by the beam control part in Figure 4.24. Since the receiver might be continuously moving, an adaptive beamsteering algorithm is required. There are 3 classes of adaptive beamsteering algorithms [134]. *Spatial beamforming* algorithms use correlation between the data streams received by individual antennas. These algorithms require a considerable amount of processing, as the correlation needs to be done over long data streams and over multiple antennas. Algorithms of the *temporal beamforming* class rely on correlation between the received data stream and a known reference stream. For example, when multi-path effects can occur, often pilot symbols are added to synchronize with the received signal. The third class consists of so-called *blind beamforming* algorithms. These algorithms use structural or statistical properties of the received signal to correct the beam direction.

In the initial situation where the satellites have not been detected yet, a search action has to be done to find the location of possible transmitters. This can be done with a so-called Direction of Arrival (DOA) estimation algorithm. Since, in the case of DVB-S reception in a moving vehicle, there is no reference signal available in the initial situation, only a spatial beamforming algorithm can be used for DOA estimation. Examples of suitable DOA algorithms are Estimation of Signal Parameters via Rotational Invariance Techniques (ESPRIT) [139, 140] and MUltiple SIgnal Classification (MUSIC) [141]. The disadvantage of these algorithms is their high complexity of $O\left(\mathcal{N}^4\right)$ (where $\mathcal{N}$ is the number of antennas) due to the correlation operations calculated for all possible antenna pairs. Therefore, a real-time implementation of such algorithms is very computationally intensive and should be avoided. Direction of Arrival estimation can be done initially to find the location of transmitters, and then it can be replaced by a less costly tracking algorithm. Hence, although its implementation is very computationally intensive, the effect of the processing costs on the longer period is very small. Therefore, for the remainder of this thesis we will assume that initial locations of transmitters are known. For more information on DOA estimation, we refer to [142].

Once the initial locations of the satellites are known, a tracking algorithm is enabled. As mentioned in the previous section, QPSK is used for transmitting DVB-S symbols. This modulation technique has well-defined structural and statistical properties. The signal is modulated in phase only, which is a strict structural property. The highest utilization of the channel can be reached when the usage of all constellation points is uniformly distributed, so transmitted symbols have a clear statistical property. Since the gain is assumed to be constant, a so-called Constant Modulus Algorithm (CMA) can be used efficiently [143]. Xu proposed an extension to CMA that allows for correction of phase deviations [144]. CMA uses both the antenna samples ($\vec{x}$) as well as the output y of the beamformer to adjust the current steering vector ($\vec{\phi}$, see Figure 4.25).

The beam steering block determines the steering vector $\vec{\phi}$ such that the error of the beamformer output y, with respect to the expected output, is minimized for both the magnitude and phase. The expected output has a magnitude of $\left|y\right| = 1$ and its

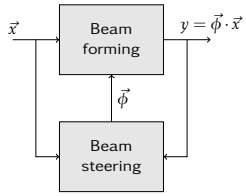


Figure 4.25 – Beamformer and beam steering blocks

phase matches the phases of the QPSK constellation points, shown in Figure 4.30(a). This is done by applying an iterative gradient descent method to decrease a cost function J that estimates the error of the received signal (using the steering vector $\vec{\phi}$) compared to the expected output. This is described by Equation 4.40, taken from [144]:

$$J\left(\vec{\phi}\right) = E\left(\left|y\right|^2 - 1\right)^2 + E\left(\sin^2\left(2\angle y\right)\right) \tag{4.40}$$

where $E\left(x\right)$ indicates the expected value of x. Xu [144] constructs a recurrent gradient descent equivalent of Equation 4.40, using an instantaneous approximation for the estimated costs. As a result, Equation 4.40 is rewritten to:

$$\begin{aligned}\vec{\phi}\left[n+1\right] &= \vec{\phi}\left[n\right] - \mu\nabla_{\vec{\phi}} J \\ &= \vec{\phi}\left[n\right] - \mu \cdot \frac{8j\left(\left|y\right|^4 - \left|y\right|^2\right) + 4\sin\left(4\angle y\right)}{4j \cdot y} \cdot \vec{x}\end{aligned} \tag{4.41}$$

which can be slightly simplified to:

$$\vec{\phi}\left[n+1\right] = \vec{\phi}\left[n\right] - \mu \cdot \frac{2\left(\left|y\right|^4 - \left|y\right|^2\right) - j\sin\left(4\angle y\right)}{y} \cdot \vec{x} \tag{4.42}$$

where $\angle x$ defines the angle of x (in radians).

The operations involved in calculating Equation 4.42 are graphically presented in Figure 4.26. The effort in calculating Equation 4.42 scales linearly with the number of antennas $\mathcal{N}$, because $\vec{x}$ and $\vec{\phi}$ are vectors of length $\mathcal{N}$. The calculation of $\mu \cdot \frac{2\left(\left|y\right|^4 - \left|y\right|^2\right) - j\sin\left(4\angle y\right)}{y}$ only consists of scalar operations and therefore requires a fixed number of operations independent of $\mathcal{N}$. Using the steering vector $\vec{\phi}$, the beam pattern tracks the transmitter. Multiple transmitters can be tracked by adding a beamformer and a CMA algorithm for each transmitter. As a result, the total complexity for tracking $\mathcal{B}$ beams with CMA equals $O\left(\mathcal{N}\mathcal{B}\right)$.

Most computations in Equation 4.42 are straightforward except for the following operations which need more attention: transformation from Cartesian coordinates to polar coordinates (section 4.3.3.1), sine calculation (section 4.3.3.2) and the complex division (section 4.3.3.3).

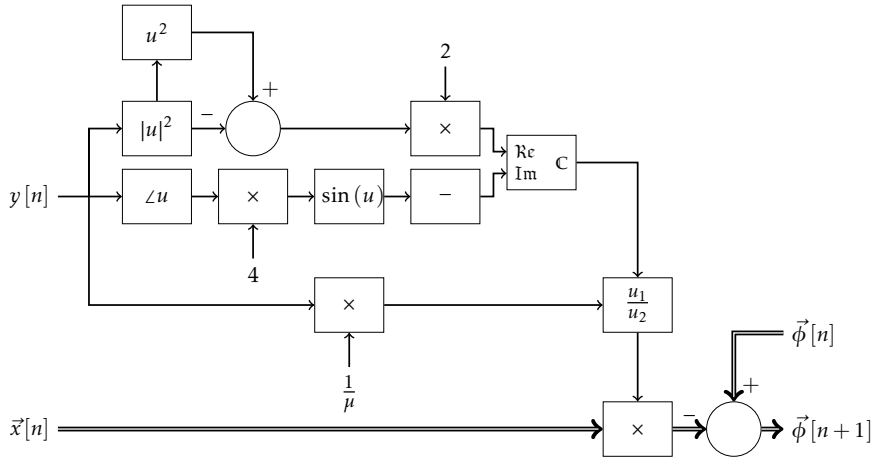


Figure 4.26 – Block diagram of the CMA adaptive beamsteering algorithm

4.3.3.1 Coordinate transformation

The conversion from Cartesian coordinates (x,y) to polar coordinates (r,θ) and back requires some goniometric operations. These could be implemented by a large LUT, at the cost of limited accuracy. A more efficient and accurate approach is the COordinate Rotation DIgital Computer (CORDIC) algorithm. It was originally proposed by Volder [145] as an iterative approach based on shift and add operations to apply coordinate transformations, which heavily rely on trigonometric functions. Extensions were proposed by Walther [146] to implement other operations like division, square root, Fourier transforms and many others. We mapped the algorithm, as described in [147], to the Montium TP. For the conversion from Cartesian to polar coordinates, the equations for the so-called *vectoring mode* are:

$$x_{i+1} = x_i - y_i \cdot d_i \cdot 2^{-i} \tag{4.43}$$

$$y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i} \tag{4.44}$$

$$z_{i+1} = z_i - d_i \cdot \tan^{-1}\left(2^{-i}\right) \tag{4.45}$$

where $d_i = +1$ if $y_i < 0$, -1 otherwise. When the number of iterations n is increased, the final values will converge to:

$$x_n = A_n \sqrt{x_0^2 + y_0^2} \tag{4.46}$$

$$y_n = 0 \tag{4.47}$$

$$z_n = z_0 + \tan^{-1}\left(\frac{y_0}{x_0}\right) \tag{4.48}$$

$$A_n = \prod_{i=1}^{n} \sqrt{1 + 2^{-2i}} \tag{4.49}$$

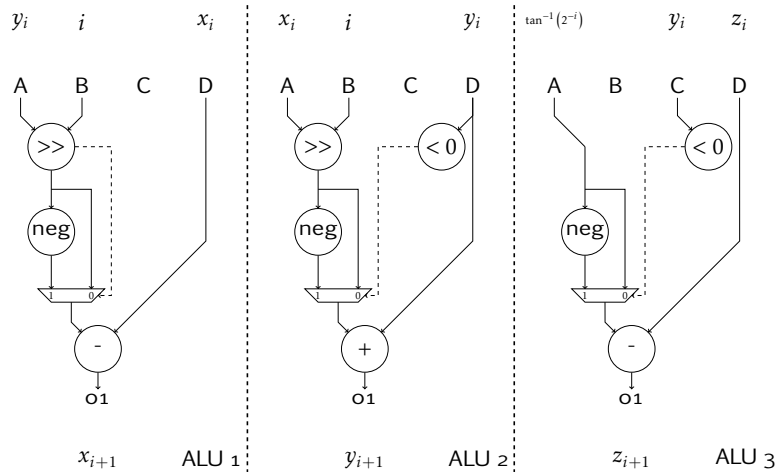


Figure 4.27 – Mapping of CORDIC equations on 3 Montium ALUs

such that $r = \frac{x_n}{A_n}$ and $\theta = z_n$.

The mapping of these equations onto Montium ALUs is shown in Figure 4.27. The decision variable d_i, which depends on the sign of y_i, is generated using the status bits of the function units. Then, based on the status bit, the new values of x_{i+1}, y_{i+1} and z_{i+1} can be calculated. For the calculation of x_{i+1}, the value of $y_i \cdot 2^{-i}$ is calculated by shifting y_i over i bits to the right. The calculation of x_{i+1}, y_{i+1} and z_{i+1} depends on d_i, the sign of y_i. For x_{i+1}, d_i is a by-product of the logic shift $y_i >> i$ $(= y_i \cdot 2^{-i})$ operation. For y_{i+1} and z_{i+1}, d_i is determined explicitly by the sign of y_i. Both the positive and negative values of the left operand are calculated. For example, for calculating x_{i+1} the value of $y_i \cdot 2^{-i}$ and its negative value are both calculated and based on the the decision variable d_i one of them is subtracted from x_i. The values for $\tan^{-1}\left(2^{-i}\right)$ are calculated offline and stored in a ROM. During the calculation of the CORDIC equations, an AGU reads the memory from an address based on the current iteration number i and writes the value to the register file for ALU 3. Hence, reading these constants does not require any additional clock cycles. Using this mapping, all three CORDIC equations can be calculated in a single clock cycle.

The accuracy of an example operation of the implemented algorithm is shown in Figure 4.28. It depicts the accuracy, expressed in number of fractional bits, of the CORDIC algorithm after each iteration for a typical case operation. As can be seen in Figure 4.28, during each iteration approximately one additional bit of precision is obtained. Due to the bit shift operations and the limited word-width of the Montium TP, the smallest possible error is reached after 14 iterations. If more than 15 iterations are done, the 16-bit word width limits the algorithm's accuracy as quantization errors due to arithmetic operations are larger than the gain in precision by the algorithm.

As identified by [145, 147] the CORDIC equations 4.43 to 4.45 are only valid for rotation angles between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$. For rotation angles between $-\frac{\pi}{2}$ and $-\pi$, an initial

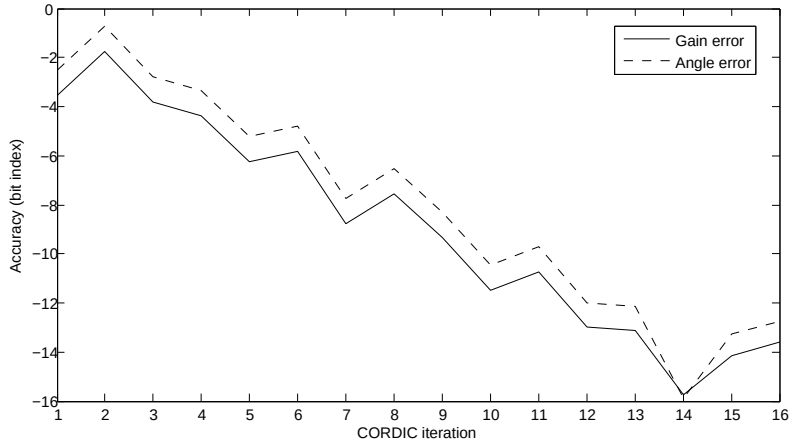


Figure 4.28 – Error between Montium result and input value (in bits) after each CORDIC iteration

rotation over $\frac{\pi}{2}$ is applied and for rotation angles between $\frac{\pi}{2}$ and π, an initial rotation over $-\frac{\pi}{2}$ is applied. These initial rotations can be realized with a set of equations 4.50 to 4.52.

$$x_0 = -d \cdot y \tag{4.50}$$

$$y_0 = d \cdot x \tag{4.51}$$

$$z_0 = z + d \cdot \frac{\pi}{2} \tag{4.52}$$

where $d = +1$ if $y < 0$, -1 otherwise.

The Montium implementation of these equations is comparable to the mapping presented for the regular CORDIC operations in Equation 4.43 to Equation 4.45, with identical processing requirements in terms of clock cycles. Hence, the calculation of the initial equations can be done in one additional iteration.

4.3.3.2 Sine calculation

The sine function (see Figure 4.26) could be calculated very accurately using CORDIC. However, this requires an additional CORDIC operation which is expensive in terms of clock cycles. Instead, we chose to map the sine function to a LUT which is stored inside one of the memories. The upper 10 bits of the 16-bit fixed point angle are used as address for the lookup. Such a lookup only requires 2 clock cycles, which is less compared to running a complete CORDIC operation with comparable accuracy. Due to the quantization of the lookup address, the accuracy of a lookup operation is limited to 10 bits. As can be seen in Figure 4.28, a comparable accuracy can be obtained by running 10 CORDIC iterations. If a higher precision is required, CORDIC is preferable to a lookup operation.

4.3.3.3 Complex division

The complex division could be implemented by using 2 CORDIC operations, one real division and 2 multiplications [148], which is useful for implementation on multiplier-limited architectures. The Montium TP, however, contains multipliers and therefore, the complex division can be implemented much more efficiently. Assume a division between the complex numbers $X = a + jb$ and $Y = c + jd$. The division can be rewritten as follows:

$$\frac{X}{Y} = \frac{a+jb}{c+jd} = \frac{a+jb}{c+jd} \cdot \frac{c-jd}{c-jd} = \frac{ac+bd}{c^2+d^2} + j\frac{bc-ad}{c^2+d^2} \tag{4.53}$$

Now define $e = \frac{1}{c^2+d^2}$. After substitution in Equation 4.53, we get:

$$\frac{a+jb}{c+jd} = \ldots = (ac+bd)\cdot e + j\,(bc-ad)\cdot e \tag{4.54}$$

which can be implemented by 6 multiplications, 2 additions and the costs for the calculation of e.

Note that $e = 1/\left(c^2+d^2\right) = 1/|Y|^2$. For the division used in Equation 4.42, X corresponds with the nominator and Y corresponds with the denominator which is y (the beamformer output), so $e = 1/\left|y\right|^2$. As can be seen in Figure 4.26, the calculation of $\left|y\right|^2$ is already done. Its inverse can be calculated efficiently by using a LUT, similar to the sine calculation. The values of $1/\left|y\right|^2$ with $\left|y\right|^2 \in [0,\ldots,1\rangle$ are in the range of $\langle 1,\ldots,\infty\rangle$, which cannot be represented in a 1.15 fixed point notation. A straight-forward LUT based implementation is therefore not useful. In order to solve this problem, the multiplication by a step factor μ (see Figure 4.26) is included in the LUT. Typically, $\mu = 0.005$ is used for the best tracking results. So, instead of using a LUT containing values $1/\left|y\right|^2$, the LUT consists of values $\mu/\left|y\right|^2$ which contains unsaturated values for all $\mu \leq \left|y\right|^2 < 1$ (see Figure 4.29)[15]. We use a LUT with 512 entries to calculate the inverse of $\left|y\right|^2$. For such a LUT, the first 3 entries are saturated (since $\frac{0\ldots2}{512} < \mu$). However, since the CMA algorithm is used to normalize $\left|y\right|^2$ to 1, the probability of a lookup of one of these saturated values is very low. Hence, the calculation of a complex division requires in total 6 multiplications, 2 additions and 2 clock cycles for one lookup operation.

4.3.3.4 CMA implementation costs

In total, the operations presented in the previous sections can be executed by the Montium TP in 288 clock cycles. The algorithm is executed once for each update of the beam direction. Using a DMA transfer, one antenna snapshot consisting of $\mathcal{N}$ complex samples can be stored in $2\mathcal{N}/L$ clock cycles. Loading the previous beam sample y takes $2/L$ cycles. The resulting steering vector, consisting of $\mathcal{N}$

[15] To emphasize the saturated positions in the LUT, μ has been slightly increased in the picture.

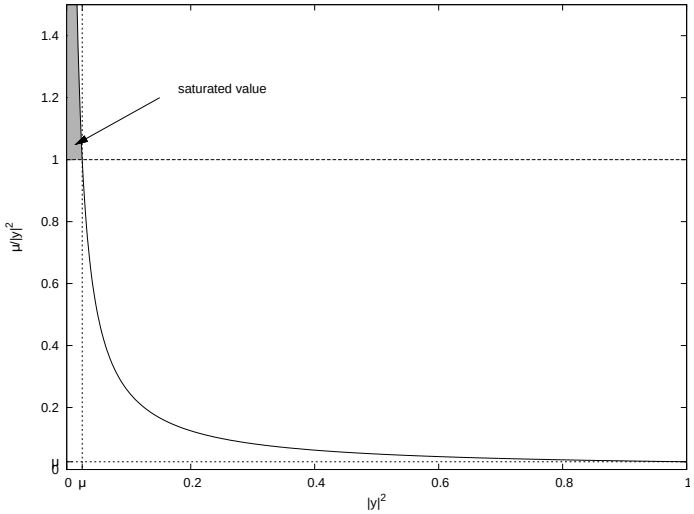


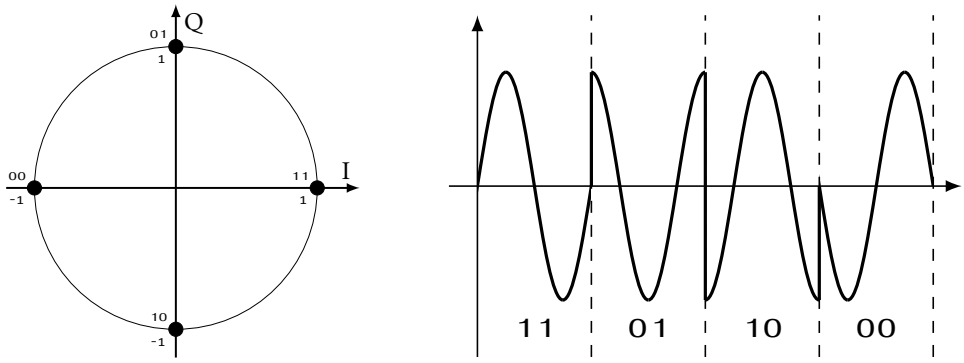
Figure 4.29 – Contents of the LUT for calculation of $\frac{\mu}{|y|^2}$

complex samples, is retrieved in as many clock cycles as the snapshot loading costs. Therefore, in block-mode operation the communication time for the CMA algorithm is $(4\mathcal{N}+2)/L$ cycles. Since the processing requires 288 clock cycles, the C/C ratio of CMA equals $\frac{4*64+2/L}{288} = 0.90/L$.

The scalar operations depicted in Figure 4.26 have to be calculated first, before the new steering vector can be calculated based on the current steering vector and the current antenna snapshot. A streaming-mode implementation of the latter part, where the current antenna snapshot is multiplied by a scalar value and added to the current steering vector (Equation 4.42), can be implemented in a streaming fashion. Due to pipelining, the communication overhead in this part is hidden in the processing, where only the storage of the last steering vector element is left as communication overhead. As a result, the current (complex) beam value y and the last (complex) steering vector element contribute to communication overhead, so the C/C ratio for the streaming-mode implementation becomes $\frac{4/L}{288} \approx 0.01$.

4.3.4 Baseband processing

The modulation technique used in DVB-S for transmitting symbols is QPSK [130]. QPSK modulation maintains a constant modulus, while the information is added by applying a multiple of $\frac{\pi}{2}$ shift in phase to the carrier frequency. When switching instantaneously between two symbols, high frequency components occur in the transmitted signal due to discontinuity in the phase. The transmitter uses a pulse shaping filter to suppress the high frequency components by spreading the signal over a slightly wider frequency band. At the receiving side, a *matched filter* is then used to restore the signal back into the original frequency band, such that the modulation information can be reconstructed.



(a) QPSK constellation diagram (b) Example QPSK modulated stream

Figure 4.30 – QPSK modulation

4.3.4.1 Matched filter

The matched filter can be implemented using an FIR filter consisting of $\mathcal{F}_{\text{mf}}$ taps. Since this filter is applied to the output of the beamformer, its complexity only depends on the number of beams $\mathcal{B}$. The matched filter is implemented using two real FIR filters of $\mathcal{F}_{\text{mf}} = 9$ taps [149], for the I and Q parts of the beamformer output. Hence, for 3 beams, this part of the baseband requires $\mathcal{B}*2*\left(\left\lceil \frac{\mathcal{F}_{\text{mf}}}{5} \right\rceil + 1\right) = 3*2*2 = 12$ clock cycles.

Since both filters are real FIR filters, the combined C/C ratio equals the C/C ratio of an individual filter. Therefore, the block-mode C/C ratio is $\frac{2*N/L}{N*\left\lceil \frac{9}{5} \right\rceil+1} \approx 1/L$. For the streaming-mode filter, the C/C ratio is $\frac{1}{N*\left\lceil \frac{9}{5} \right\rceil+1} \approx 0$.

4.3.4.2 QPSK demapping

The (I, Q) signal obtained from the matched filter is converted to bits via QPSK demapping. While QAM (presented in section 4.2.3.3) uses amplitude modulation for the I and Q parts individually, QPSK modulation only applies a phase shift to the combined (I, Q) signal. Hence, the amplitude of the (I, Q) pair is kept constant while its phase is modified. Figure 4.30(a) shows an example constellation diagram for QPSK, where for each of the constellation points the accompanying bit code is displayed.

The demapping is done identically to the QAM demapping presented in section 4.2.3.3. A LUT, containing the constellation point codes, is indexed using an address composed from the I and Q parts of a received signal. Therefore, the computational complexity for both the address generation and the lookup action is identical as presented in section 4.2.3.3.

Table 4.9 – Implementation costs for the DVB-S receiver, expressed in clock cycles per antenna snapshot. Here, we assumed $\mathcal{N} = 64$ and $\mathcal{B} = 3$.

	Clock cycles	C/C ratio	
		Block-mode	Streaming-mode
Channel Equalization	320	$0.8/L$	0
Beam forming	192	$0.70/L$	$0.03/L$
Beam steering[†]	$1/k_u * 288$	$1/k_u * 0.90/L$	$1/k_u * 0.01$
Matched filter	12	$1/L$	0
QPSK demapping	2	$4/L$	0
Total	526		

[†] Given CMA computation and communication costs are defined per beam update. However, for a realistic update interval $k_u = 500$, the average cost antenna snapshot is less than one clock cycle.

4.3.5 DVB-S implementation overview

In the previous sections, all blocks required for the reception and baseband processing of a DVB-S receiver have been presented. Table 4.9 summarizes the communication and computation numbers of all blocks. For all blocks, an implementation was presented for both operating modes. The complexity of the CMA algorithm is linear with the number of antenna elements. Although (parts of) the algorithm seem to be computationally intensive (for example, the CORDIC part for applying a coordinate transformation), the average processing requirements posed by the CMA algorithm are low, because the beam update interval is long. A considerable amount of processing is done at the antenna channel, because of the channel equalization filtering. However, its computational complexity is independent of the number of beams.

The combination of all blocks discussed in the previous sections is presented in an SDF model depicted in Figure 4.31. Duplicator blocks denote a copying action, such that a received token is duplicated and sent to different processes. For example, the beamformer process and the decimation process at the left receive the same tokens. A decimation process consuming k_u tokens and producing 1 token removes all tokens except for the first token received from the stream. The opposite operation, the upsampling process, is implemented as a sample-and-hold operation. A token received from its input stream is duplicated k_u times on the output edge of the upsampling process.

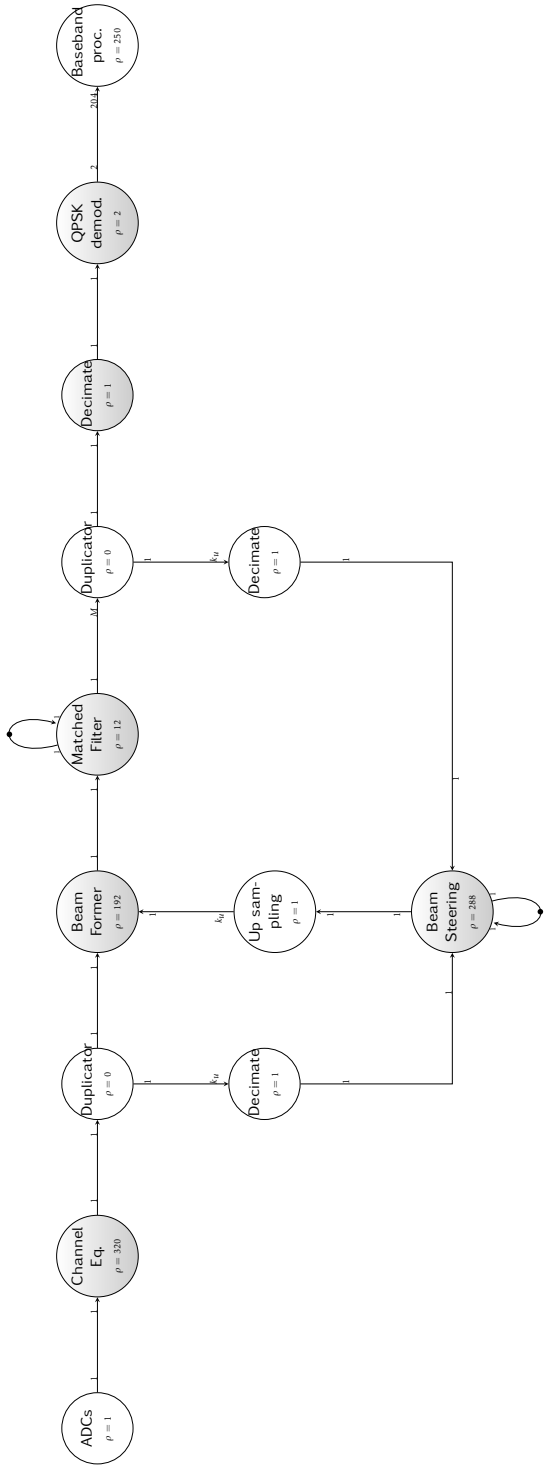


Figure 4.31 – SDF model of a beamformer used for tracking a DVB-S transmitter

4.4 Conclusion

The majority of the kernels in both case studies show a streaming behavior and they can be described mathematically. The streaming-mode implementation results in a low communication overhead for these kernels. Kernels that do not support streaming-mode communication can be implemented by a block-mode alternative, where the usage of parallel DMA transfers might reduce the communication overhead. Together, both communication modes allow for a low overhead in terms of clock cycles, enabling a stream processing architecture that can be programmed efficiently. For both applications in the case study, an SDF model was composed to show the resulting computational complexity and the resulting communication. We highlighted some of the Montium TP operations, to demonstrate that multiple operations can be mapped on the Montium ALUs simultaneously, such that the total number of clock cycles required for executing a kernel is reduced.

The Hydra NI, presented in this thesis, enables streaming-mode communication such that concurrency in computation and communication can be exploited. Moreover, the communication abstraction provided by the Hydra NI enabled most of the kernels to be implemented as streaming-mode algorithms, such that the kernels can be operated very efficiently because the communication overhead is avoided. For several basic operations we evaluated the communication to computation (C/C) ratio, which can be used to analyze the communication overhead. If the implementation of a kernel has a low C/C ratio, this indicates that the communication is cheap compared to the processing.

Chapter 5

Conclusion

The design, programming and use of a Multi-processor System-on-Chip (MPSoC) for streaming Digital Signal Processing (DSP) applications involves a complex integration of hardware building blocks and tools for the design flow. In MPSoC architectures, communication has become one of the most important factors because it enables the cooperation between different processors within the MPSoC, but it can also easily become one of the main bottlenecks if designed inaccurately. Efficient communication enables high utilization of individual processors and, therefore, a high performance of the MPSoC.

Latency and throughput of communication streams between processors determine the lower bound of processing performance, as processors are blocked as long as no input data is available. Hence, by minimizing the communication latency and increasing the communication bandwidth, processors can execute more efficiently. The use of a Network-on-Chip (NoC) enables multiple communication streams to exist in parallel, such that a large aggregated bandwidth is provided. By offering bandwidth guarantees to individual streams, the upper bound of latency of the stream can be determined.

Efficient computation of kernels within an application is required to design a low power/high performance MPSoC. The Montium TP is a reconfigurable tile processor that is suitable for application in Multi-Processor Systems-on-Chip, as it can execute complex instructions at a low energy budget. As it supports complex instructions, many operations can be executed in parallel such that the number of operations per clock cycle is high (typically, about 15 operations can be performed per instruction). As a result, the clock frequency of the Montium TP can be lowered considerably, such that low energy consumption is obtained. The Montium's instruction memory can be reconfigured very quickly, such that functionality can be replaced in only hundreds of clock cycles.

A crucial component in the MPSoC is the interface between a processor and the NoC. If the interface is inefficient, the bandwidth provided by the NoC cannot be used by the processor or the processor cannot be used efficiently and therefore latency of communication streams may become large. In this thesis we propose the Hydra Network Interface (NI), an energy-efficient and reconfigurable network

interface that connects the Montium TP to a NoC. It supports two different types of operating modes: block-mode communication is based on Direct Memory Access (DMA) transactions, such that other processors can read and write the Montium TP's memories and registers, and streaming-mode communication can be used to enable concurrency in processing and communication. Other processors in the MPSoC can control the Hydra NI by transmitting messages via the NoC to the Hydra NI. Using a light-weight message protocol, the Montium TP can be started and halted, the Montium TP can be reconfigured and the Montium memories and registers can be accessed via DMA transactions. Since the Montium TP can be operated at different clock frequencies, all data received by the Hydra from the NoC is stored in FIFO buffers that are responsible for synchronization between the clock domains. Data transmitted by the Montium TP to the NoC is stored in another set of FIFO buffers. The overhead of the Hydra NI, compared to the Montium TP, is low. If constrained to operate at a maximum clock frequency of 200 MHz, the Hydra NI has a total area of 19k gates or about 0.106 mm^2 in 0.13 µm ASIC technology, which is about 6% of the area of the Montium TP. For typical streaming DSP applications (see Table 2.11), the power distribution of one processing tile containing a Montium TP and a Hydra NI shows that the Hydra NI contributes to about 3% to 10% of the total power budget. Due to the message protocol, transmitted data is formatted in packets with a small overhead of typically 5% to 10%. Therefore, we claim that the general overhead of the Hydra NI, with respect to the Montium TP, is in the range between 5% and 10%.

Although efficient hardware architectures enable efficient processing, an inefficient programming model may destroy the performance of the architecture. A lot of effort has been (and is being) spent on automatic mapping of applications to multi-processor architectures. In chapter 3 of this thesis, we present new ideas on an application modeling technique that is suitable for modeling applications that are closely related to mathematics, like streaming DSP applications. While many proposed design flows focus on the parallelization of sequential (imperative) code to parallel threads that can be executed by a multi-processor architecture, our approach stays closely to the mathematical definition of an application (which is inherently parallel). An Embedded Domain Specific Language (EDSL), implemented in Haskell, is introduced in which applications are described. The EDSL can be used as a strongly typed programming language and is constructed as a single data type. This has the advantage that transformation functions can be defined for such a data type, to convert the application implemented in the EDSL into another (functionally equivalent) application. The correctness of these transformation rules can be proven, hence they can be applied safely to the application without changing the functional behavior and therefore the correctness. A special transformation rule is the partitioning rule, which can be applied to kernels to split them in multiple smaller kernels that can be mapped on a multi-processor system. The transformed application is converted to a Synchronous Data Flow (SDF) model, which models execution times of parts of the application (processes) and adds explicit communication between processes using communication channels. We propose an SDF simulator framework that can execute such a model, to simulate the behavior of an application and to test the synchronization between processes. Since communication is explicit, output values of processes can be analyzed to check the implementation correctness. To ease this

analysis, the simulator can generate a visualization during the simulation, to show the application structure and communication patterns. The effect of the operation mode (block-mode or streaming-mode), that is to be used for the mapping of the processes onto a processor, and the behavior of the interconnect (in terms of latency and throughput) can be added to the SDF model. This enables accurate simulation of the execution of the application on an MPSoC platform.

To evaluate the performance of applications mapped on the Montium TP, we introduced the communication-to-computation (C/C) ratio that expresses the communication overhead added to the processing, due to the different operating modes. With this ratio, in chapter 4 the kernel operations identified in two wireless communication receivers are analyzed. For both operating modes the kernels are analyzed and the C/C ratio is calculated. In general, block-mode operation adds more communication overhead than streaming-mode. The C/C ratio of the streaming-mode operations is lower than (or equal to) the C/C ratio for block-mode operation for almost all analyzed kernels.

5.1 Future work

The Annabelle MPSoC architecture presented in this thesis consists of an Advanced RISC Machine (ARM)-926 processor and 4 Montium TPs. The ARM processor is responsible for reconfiguring the Montium TPs and for configuring the NoC such that communication channels between Montium TPs are opened. With the current number of Montium TPs, the ARM processor can manage these tasks. However, if the number of processors in the MPSoC is increased, the ARM processor may not be capable of controlling all Montium TPs. Large Multi-Processor Systems-on-Chip require an advanced distributed control mechanism such that the available hardware resources (in terms of total processing power and aggregated interconnect bandwidth) can be used efficiently. Increasing the number of processors on the MPSoC will eventually require a new Complementary Metal Oxide Semiconductor (CMOS) technology. This will have a considerable influence on the energy demands of the entire chip, and therefore on the power management.

We presented a mathematic programming based design-flow for mapping applications to an MPSoC architecture. The EDSL, in which the application is described, is constructed such that transformations can be applied to modify the language but keep the functional behavior unchanged. Which transformation rules should be applied to the language, and in which order, is currently decided manually. A typical transformation rule that modifies the structure of the application is a partitioning rule. Before partitioning an application, knowledge about the targeted MPSoC architecture is required. We showed an example where partitioning of a large adder resulted in an adder tree consisting of adders with less inputs. For that particular example, the partitioning parameter was the number of operands that are added by an adder in the Arithmetic Logic Unit (ALU) of a processor. However, for more complex language constructions the partitioning rule is also much more complex.

The SDF simulator framework, presented in this thesis, is used for functional and structural simulation of streaming DSP applications. An extension to Cyclo-static

Data Flow (CSDF) simulations could be made to offer a more intuitive interface to the application programmer, as CSDF models the individual phases of a process. The current execution model in the simulator uses a naive scheduling mechanism, where for all processes during each simulation cycle is checked whether the process is waiting, executing or finished. If the execution times of all processes are large, this results in many simulation cycles where nothing is actually calculated and therefore, those cycles could be skipped. Replacing the current scheduling mechanism with a more efficient scheduler may improve simulation times, but will not change functional and temporal behavior of the application.

In the Digital Video Broadcast for Satellite (DVB-S) application, where a phased array is used as a steerable antenna, the quality of the received signal is very sensitive to environmental distortions. For example, such distortions may be caused by objects like buildings and trees, or radio waves transmitted by other devices. In situations where the signal is distorted, channel equalization may improve the signal quality after reception. In this thesis, the design of the antenna processing blocks like calibration and equalization has been mentioned but was considered out of scope. Further research on the use of these DSP operations is required to compensate for deviations between simulation and real-life operation.

Appendix A

Hydra NI timing diagrams

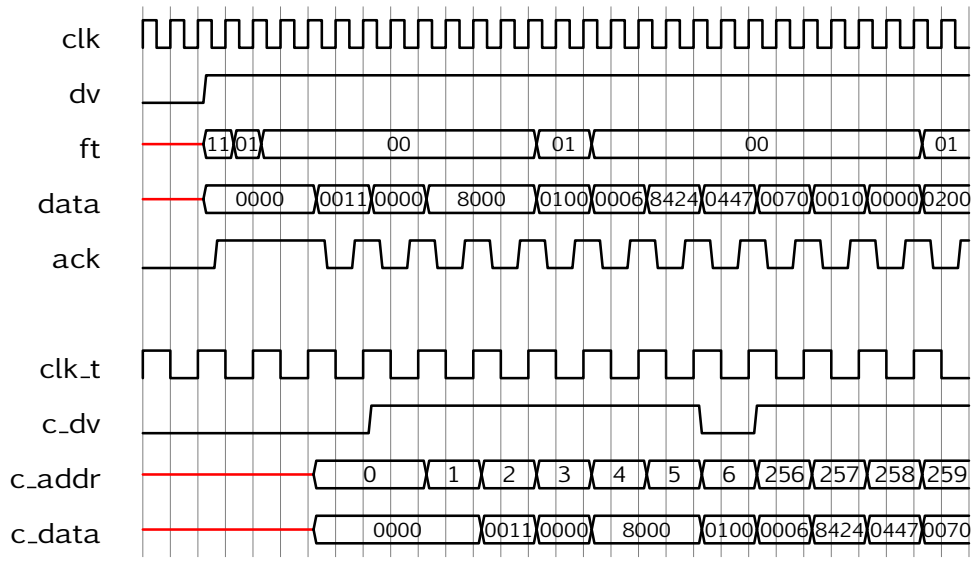


Figure A.1 – Timing diagram of the execution of a configuration packet

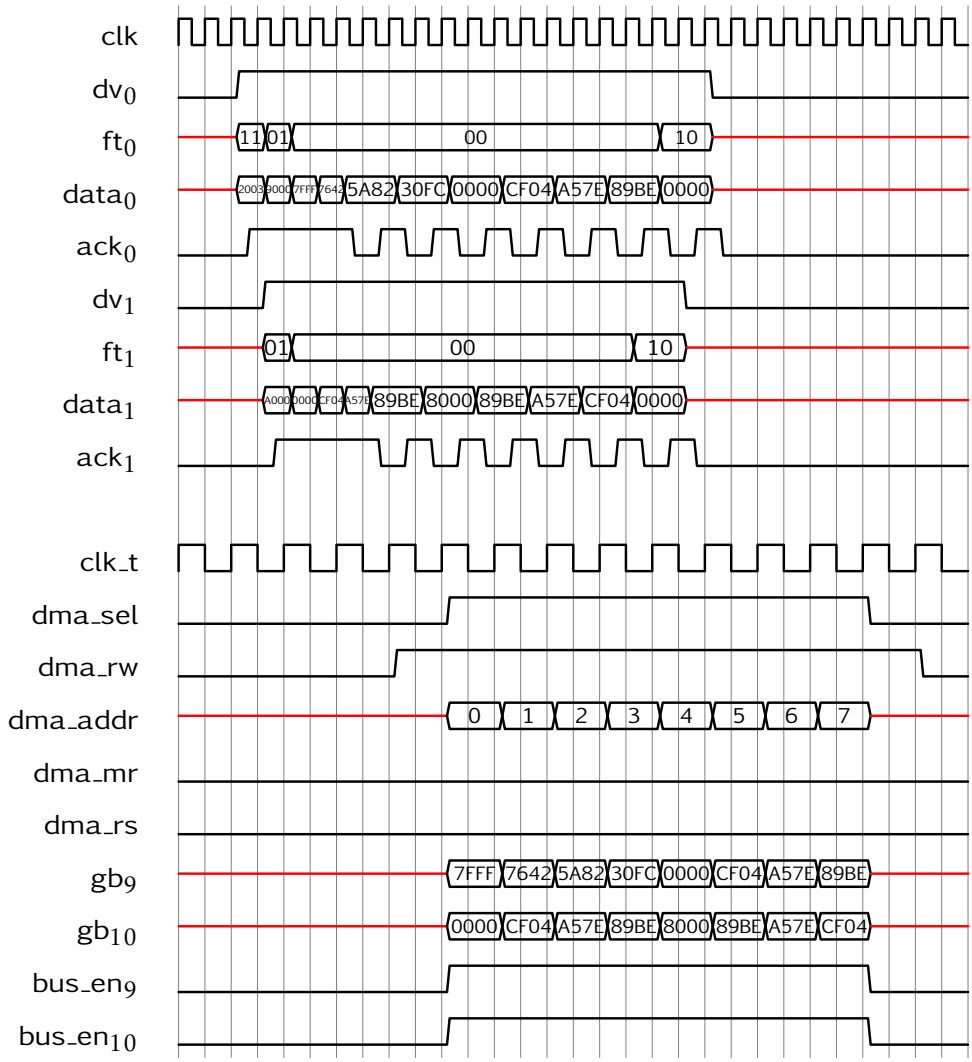


Figure A.2 – Timing diagram of a DMA load transaction for the Montium TP memories

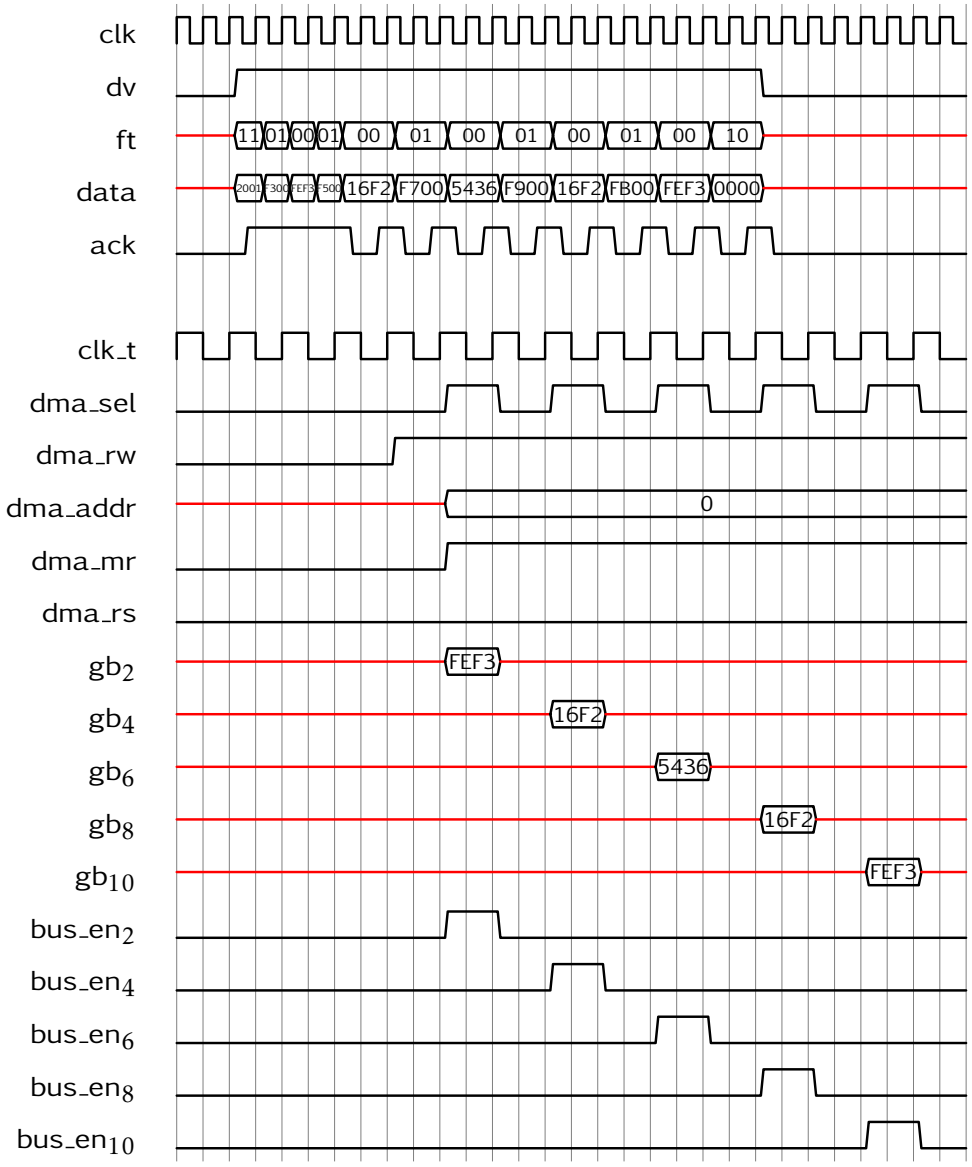


Figure A.3 – Timing diagram of a DMA load transaction for the Montium TP register files

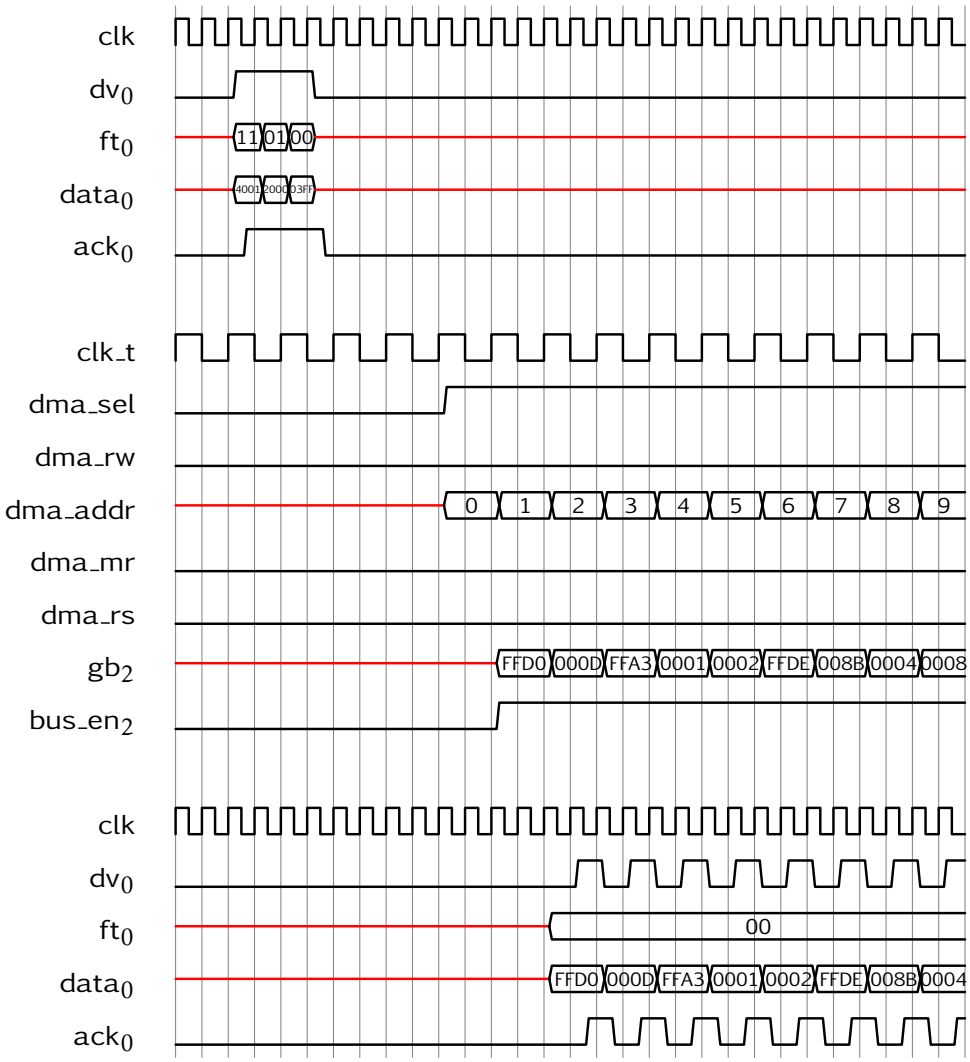


Figure A.4 – Timing diagram of a DMA retrieve transaction for the Montium TP memories

Appendix B

Data flow simulator

B.1 Haskell

Several examples of Haskell constructions are given in Listing B.1. More information can be found on the Haskell website [150].

B.2 Simulator data types

A token is a black box container in which anything can be stored using the `toToken()` function and which can be opened with the function `fromToken()`. The `fromToken()` function includes error reporting in case the conversion did not succeed (for brevity reasons not shown in Listing B.2).

The simulator is recursively operated at a `SimState` item, that is constructed as defined in Listing B.3.

```haskell
-- Data types
tuple :: (Int,Char)
list  :: [Double]

-- Abstract data types using constructors
data Color = Red | Green | Blue | Cyan | Magenta | Yellow   -- and more

-- Record type "Rec" is defined using a constructor "R"
data Rec = R { field1 :: Int, field2 :: Char }
r1 = R {field1 = 1, field2 = 'a'}

-- field names can be used as functions on the record
a = field1 r1 -- evaluates to integer value '1'

-- Definition of a function "f" with an argument of type "a", resulting in a
-- value of type "b"
f :: a -> b

-- Quadratic formula, showing the use of a "where clause".
-- Expressions in the where clause are unordered.
quadratic :: Float -> Float -> Float -> (Float, Float)
quadratic a b c = ( x1, x2)
                  where
                    x1 = (-b + sqrt(d)) / (2*a)
                    x2 = ( b - sqrt(d)) / (2*a)
                    d  = b*b - 4*a*c

-- Lists
l = [1, 2, 3, 4, 5]

-- List concatenation "++" glues two lists together
twice_l = l ++ l

-- application of an element-wise function to two lists, resulting in a new list
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys

-- List comprehension
doubleList :: [Int] -> [Int]
doubleList xs = [ 2*x | x <- xs ]

-- Recursion
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)

-- Pattern matching using specific values (0, 1 or any other number)
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

-- The Maybe abstract data type is used to indicate a possible value t
-- (written as "Just t") or the absence of a value (written as "Nothing")
data Maybe t = Nothing
             | Just t

-- Pattern matching using abstract data type constructors ("Nothing" or "Just")
addOne :: Maybe Int -> Int
addOne (Nothing) = 0
addOne (Just x)  = x+1
```

Listing B.1 – Haskell examples

```
type Token = Dynamic

fromToken :: (Typeable t) => Token -> t
fromToken t = fromJust (fromDynamic t)

toToken :: (Typeable t) => t -> Token
toToken t = toDyn t

isEmpty :: Token -> Bool
isEmpty t = fromDynamic t == Just ()

empty :: Token
empty = toToken ()
```

Listing B.2 – Token definition

```
-- Simulator types
data Stream = Stream [Token]

data Exec
        = Waiting
        | Running Int
        | Finished

type ProcessExec  = (Exec, Buffer)
type Buffer = [Stream]

data SimState
        = SS { ps   :: [Process]
             , cs   :: [Channel]
             , exec :: [ProcessExec]
             }

-- Process types
type ProcessID = Int
type Fun  = (State -> [Stream] -> (State, [Stream]))
type State  = Token

type PortID = Int
data Port
        = Data PortID
        | Sync PortID

data Process
        = PS { pid   :: ProcessID
             , f     :: Fun
             , state :: State
             , is    :: [(Port,Int)]
             , os    :: [(Port,Int)]
             , wcet  :: Int
             }

-- Channel types
type ChannelID = String
type ChannelContent = Stream

data Channel
        = C { chid :: ChannelID
            , from :: (ProcessID, Port)
            , to   :: (ProcessID, Port)
            , cont :: ChannelContent
            }
```

Listing B.3 – Simulator types

Appendix C

PFA address calculation

This example explains the output ordering for streaming out the complex sample $X[k]$ in linear order. The Chinese Remainder Theorem (CRT) mapping on the Montium TP's memories can be described as follows (only the real part of X_{CRT} is shown):

$$X_{\mathrm{CRT}}[k_1,k_2] = \begin{cases} m_{1.2}\left[\langle k_1\rangle_{\frac{N_2}{2}} + k_2\frac{N_2}{2}\right], & \text{if } k_1 < \frac{N_2}{2} \\ m_{2.2}\left[\langle k_1\rangle_{\frac{N_2}{2}} + k_2\frac{N_2}{2}\right], & \text{otherwise} \end{cases} \tag{C.1}$$

When equations 4.8 and C.1 are combined, the mapping of $X[k]$ can be described as follows (again only showing the real part):

$$X[k] = \begin{cases} m_{1.2}\left[\langle p[k]\rangle_{\frac{N}{2}}\right], & \text{if } p[k] < \frac{N}{2} \\ m_{2.2}\left[\langle p[k]\rangle_{\frac{N}{2}}\right], & \text{otherwise} \end{cases} \tag{C.2}$$

where $p[k]$ indicates the indirection address that is calculated as follows:

$$p[k] = \left\langle 65\cdot k + 896\cdot\left\lfloor\frac{k}{64}\right\rfloor + (1024 + (16-N_1)\cdot 64)\cdot\left\lfloor\frac{k}{15}\right\rfloor\right\rangle_N \tag{C.3}$$

For example, to obtain the location of $X[3]$ from Equation C.2, one has to calculate $p[3]$ using Equation C.3. As $p[3] = 65\cdot 3 = 195 < \frac{1920}{2}$, it is stored in $m_{1.2}$ at address position 195.

A full recalculation of the indexing address for each update is expensive. By writing Equation C.3 in a differential form, it can be shown that the new address can

be based on the current address such that it can be calculated efficiently:

$$
\begin{aligned}
c_1[k] &= \begin{cases} 896, & \text{if } \langle k \rangle_{64} = 0 \\ 0, & \text{otherwise} \end{cases} \\
c_2[k] &= \begin{cases} 1024 + (16 - N_1) \cdot 64, & \text{if } \langle k \rangle_{15} = 0 \\ 0, & \text{otherwise} \end{cases} \\
\Delta[k] &= p[k-1] + c_1[k] + c_2[k] + 65 \\
p[k] &= \begin{cases} \Delta[k], & \text{if } \Delta[k] < 1920 \\ \Delta[k] - 1920, & \text{otherwise} \end{cases}
\end{aligned}
\tag{C.4}
$$

One ALU is used for determining the values of $c_1[k]$ and $c_2[k]$, which depend on the current value of k. The calculation of the new $p[k]$ requires a pseudo-modulo operation. As mentioned before, an ALU has four inputs which can be used by four function units. Two function units are used for the calculation of the two cases of $p[k]$. A third function unit performs the test $\Delta[k] < 1920$ which can be mapped on the conditional Compare/Select unit mentioned in section 2.1.1.6 to select the result of one of the first two function units depending on the test performed by the third. All these operations can be executed on one ALU in one clock cycle. By subtracting $\Delta[k] - 1920$, the sign is used to select the correct value for the output $p[k]$. This equals the operation $p[k] = \langle p[k-1] + c_1[k] + c_2[k] + 65 \rangle_N$ because $p[k-1]$ and $c_1[k] + c_2[k]$ both are larger than 0 and smaller than 1920.

List of Acronyms

Acronyms

4S Smart Chips for Smart Surroundings. vi, viii, 2, 40, 50, 51, 79

ADC Analog-to-Digital Converter. 98, 99, 103, 116–118
ADT Algebraic Data Type. 55, 58
AGU Address Generation Unit. 13, 83, 89, 95, 106, 123
AHB Advanced High-performance Bus. 8, 40
ALU Arithmetic Logic Unit. 12–16, 82, 83, 88, 89, 92, 95, 96, 102, 107, 123, 130, 133, 144
AM Amplitude Modulation. 77, 97
AMBA Advanced Microcontroller Bus Architecture. 8
ARM Advanced RISC Machine. 10, 40, 42, 92, 105, 133
ASIC Application Specific Integrated Circuit. 2, 10–12, 45, 132
AWGN Additive White Gaussian Noise. 86

BE Best Effort. 19, 20
BS Beamsteering. 117

CIC Cascading Integrating Comb. 100
CMA Constant Modulus Algorithm. 120–122, 125, 126, 128
CMOS Complementary Metal Oxide Semiconductor. 2, 4, 7, 54, 133
CORDIC COordinate Rotation DIgital Computer. 122–125, 128
CRT Chinese Remainder Theorem. 81, 89, 93, 143
CSDF Cyclo-static Data Flow. 49, 53, 54, 73–76, 133, 134

DAB Digital Audio Broadcast. 1
DCT Discrete Cosine Transform. 10
DDC Digital Down Converter. 98–101, 107
DFS Dynamic Frequency Scaling. 24
DFT Discrete Fourier Transform. 80, 81, 85, 88, 94, 95, 98, 99, 104
DMA Direct Memory Access. 21, 33, 34, 36–38, 43, 73, 74, 83, 84, 118, 119, 125, 130, 132, 136–138
DOA Direction of Arrival. 120
DRM Digital Radio Mondiale. 1, 2, 4, 5, 78, 79, 85–87, 89–91, 94, 97–100, 102–105, 108, 109, 111

DSP Digital Signal Processing. 5, 10–13, 45, 48, 50, 55, 56, 61, 76, 79, 99, 131–134
DSP Digital Signal Processor. 2, 10, 11, 50, 89
DSRA Domain Specific Reconfigurable Architecture. 10, 12
DVB-S Digital Video Broadcast for Satellite. 1, 4, 5, 79, 110–112, 115–120, 126, 128, 129, 134
DVS Dynamic Voltage Scaling. 24, 25

EDSL Embedded Domain Specific Language. 55–57, 60–62, 65, 76, 132, 133
EM Electro-Magnetic. 110
ESPRIT Estimation of Signal Parameters via Rotational Invariance Techniques. 120

FAC Fast Access Channel. 97, 105
FFT Fast Fourier Transform. 10, 15, 43–45, 79–95, 104, 107
FIFO First In, First Out. 29–33, 35, 43, 44, 52, 132
FIR Finite Impulse Response. 10, 37, 44, 45, 61–64, 71, 72, 79, 95–97, 100, 118, 119, 127
FM Frequency Modulation. 77, 97
FOC Frequency Offset Correction. 98, 102, 103
FPGA Field Programmable Gate Array. 10–12

GALS Globally Asynchronous Locally Synchronous. 23
GB Global Bus. 13, 15, 31, 32, 36, 37, 40, 43
GP General Purpose. 38, 39
GPP General Purpose Processor. 10, 11, 42, 50, 108
GT Guaranteed Throughput. 19, 20
GTR Guard Time Removal. 98, 99, 101–103, 108
GUI Graphical User Interface. 72, 76

IC Integrated Circuit. 2, 7, 25
iDFT Inverse Discrete Fourier Transform. 80, 85, 104
iFFT Inverse Fast Fourier Transform. 86, 94
IO Input/Output. 8, 11, 33, 40, 46
ISI Inter Symbol Interference. 100

KPN Kahn Process Network. 48, 49, 51, 52

LAN Local Area Network. 20
LHS Left-hand side. 110
LM Local Memory. 8
LO Local Oscillator. 117
LSB Least Significant Bit. 33, 92, 106, 107
LUT Lookup Table. 11, 89, 102, 103, 105–107, 122, 124, 125, 127

MAC Multiply Accumulate. 11, 14
MoC Model of Computation. 50
MPEG MPEG. 2, 97, 105, 107, 111

MPSoC Multi-processor System-on-Chip. v, vi, 1–5, 7–10, 16, 17, 20, 21, 23, 40–44, 48, 50, 54, 55, 64, 73, 75, 76, 79, 131–133
MSB Most Significant Bit. 33, 106, 107
MSC Main Service Channel. 97, 104, 105, 107
MUSIC MUltiple SIgnal Classification. 120

NCO Numerically Controlled Oscillator. 100
NI Network Interface. v, vi, 4, 5, 8, 9, 12, 13, 20–23, 28, 29, 31, 35, 36, 39–46, 73, 76, 84, 90, 93, 130–132
NoC Network-on-Chip. v, vi, 1, 4, 8–10, 12, 16–21, 23, 27–29, 32, 33, 35, 38, 40, 42, 45, 64, 73, 75, 90, 131–133

OFDM Orthogonal Frequency Division Multiplexing. 78, 86, 97–99, 101, 103, 104, 108
OSYRES Operating System for Real-Time Embedded Systems. 50

PC Program Counter. 15
PCB Printed Circuit Board. 11
PFA Prime Factor Algorithm. 80–82, 86–89
PLL Phase Locked Loop. 24
PP Processing Part. 13, 37
PPA Processing Part Array. 12

QAM Quadrature Amplitude Modulation. 86, 105–107, 127
QoS Quality of Service. 2, 50
QPSK Quadrature Phase Shift Keying. 78, 120, 121, 126, 127

RC Ruritanian Correspondence. 80, 81, 89, 92
RF Radio Frequency. 97, 98, 102, 116, 117
RHS Right-hand side. 110
RISC Reduced Instruction Set Computer. 10
ROM Read-only Memory. 11, 32, 33, 40, 42, 123
RTOS Real-Time Operating System. 50

SDC Service Description Channel. 97, 104, 105
SDF Synchronous Data Flow. vi, 5, 50, 52, 53, 55, 63–65, 68, 72, 73, 75, 76, 107, 109, 128–130, 132, 133
SDF3 SDF for Free. 50
SESAME Simulation of Embedded Systems Architectures for Multi-level Exploration. 48
SIMD Single Instruction Multiple Data. 47
SIO Streaming IO. 40, 43
SNR Signal to Noise Ratio. 111, 118
SRAM Static Random Access Memory. 13, 36, 38

TDMA Time Divison Multiple Access. 17
TP Tile Processor. vi, 8, 9, 33, 39

ULA Uniform Linear Array. 112, 116

VBR Variable Bit Rate. 53
VC Virtual Channel. v, vi, 17, 19, 28, 29, 31, 32, 37, 40, 43
VCO Voltage-controlled oscillator. 24
VHDL VHSIC (Very High Speed Integrated Circuit) Hardware Description Language. x, 12, 41
VLSI Very-Large-Scale Integration. 7, 24
VPDF Variable Phase Data Flow. 54

XML Extensible Markup Language. 75
XPP Extreme Processing Platform. 12

YAPI Y-chart Application Programmer's Interface. 50

Bibliography

[1] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *IEEE Computer*, 36(8):54–62, August 2003. ISSN 0018-9162. doi: 10.1109/MC.2003.1220582.

[2] 4S. Smart chips for smart surroundings, October 2009. URL http://www.smart-chips.net.

[3] Eric A. M. Klumperink, Bram Nauta, André B. J. Kokkeler, and Gerard J. M. Smit. CMOS Beamforming Techniques. STW project proposal, University of Twente, February 2006.

[4] Gerard J. M. Smit, Eberhard Schüler, Jürgen Becker, Jérôme Quévremont, and Werner Brugger. Overview of the 4S project. In Jari Nurmi, Jarmo Takala, and Timo D. Hämäläinen, editors, *Proceedings of the International Symposium on System-on-Chip, 2005*, pages 70–73, Piscataway, NJ, USA, November 2005. IEEE Computer Society. ISBN 0-7803-9294-9. doi: 10.1109/ISSOC.2005.1595647.

[5] Parbhu D. Patel, Dion W. Kant, Erik van der Wal, and Arnold van Ardenne. Phased array antennas demonstrator as a radio telescope - EMBRACE. In *Proceedings of the IEEE International Antennas and Propagation Society Symposium (AP-S 2008)*, pages 1–4, July 2008. doi: 10.1109/APS.2008.4619303.

[6] Thales Nederland B.V. APAR - active phased array multifunction radar, February 2010. URL http://www.thalesgroup.com/Portfolio/Defence/Air_Systems_Product_-_APAR/?pid=6902.

[7] KVH Industries, Inc. Making the world as mobile as you are, January 2010. URL http://www.kvh.com.

[8] *AMBA Specification (Rev 2.0)*. ARM, 1999. URL http://www.arm.com. IHI 0011A.

[9] *Multi-layer AHB - Overview*. ARM, 2004. URL http://www.arm.com. DVI 0045B.

[10] John von Neumann. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993. ISSN 1058-6180. doi: 10.1109/85.238389.

[11] John Bayko. Great microprocessors of the past and present, December 2003. URL http://jbayko.sasktelwebsite.net/cpu.html.

[12] Howard H. Aiken and Grace M. Hopper. The automatic sequence controlled calculator. *Electrical Engineering*, 65:384–391, 449–454, 522–528, 1946.

[13] Edwin J. Tan and Wendi B. Heinzelman. DSP architectures: past, present and futures. *SIGARCH Computer Architecture News*, 31(3):6–19, 2003. ISSN 0163-5964. doi: 10.1145/882105.882108.

[14] David G. Chinnery and Kurt Keutzer. Closing the gap between ASIC and custom: an ASIC perspective. In Giovanni de Micheli, editor, *Proceedings of the 37th Conference on Design Automation, 2000*, pages 637–642, Los Alamitos, CA, USA, June 2000. IEEE Computer Society. ISBN 1-58113-1897-9. doi: 10.1109/DAC.2000.855391.

[15] Farzad Nekoogar and Faranak Nekoogar. *From ASICs to SOCs: a practical approach*. Prentice Hall, Upper Saddle River, NJ, USA, May 2003. ISBN 0-13-033857-5.

[16] W. James MacLean. An evaluation of the suitability of FPGAs for embedded vision systems. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, pages 131–137, Washington, DC, USA, June 2005. IEEE Computer Society. ISBN 0-7695-2372-2-3. doi: 10.1109/CVPR.2005.408.

[17] John V. Oldfield and Richard C. Dorf. *Field-Programmable Gate Arrays: Reconfigurable Logic for Rapid Prototyping and Implementation of Digital Systems*. Wiley-Interscience, New York, NY, USA, March 1995. ISBN 0-471-55665-3.

[18] *IEEE standard for VHDL Register Transfer Level (RTL) Synthesis*. VHDL Analysis and Standardization Group, 2004. IEEE standard 1076.6-2004, IEC 62050-2005.

[19] *IEEE standard for Verilog hardware description language*. IEEE-SA Standards Board, New York, NY, USA, September 2001. URL http://ieeexplore.ieee.org/servlet/opac?punumber=7578. IEEE standard 1364-2001.

[20] *IEEE Standard SystemC Language Reference Manual*. IEEE-SA Standards Board, New York, NY, USA, December 2005. URL http://standards.ieee.org/getieee/1666/download/1666-2005.pdf. IEEE standard 1666-2005.

[21] Richard Wain, Ian J. Bush, Martyn F. Guest, Miles J. Deegan, Igor N. Kozin, and Christine A. Kitchen. An overview of FPGAs and FPGA programming; initial experiences at Daresbury. Technical Report DL-TR-2006-010, Council for the Central Laboratory of the Research Councils, Warrington, Cheshire, UK, November 2006. URL http://epubs.cclrc.ac.uk/bitstream/1167/DL-TR-2006-010.pdf.

[22] Karen A. Tomko and Anurag Tiwari. Hardware/software co-debugging for reconfigurable computing. In *Proceedings of the IEEE International High-Level Design Validation and Test Workshop (HLDVT'00)*, pages 59–63, Los Alamitos, CA, USA, November 2000. IEEE Computer Society. ISBN 0-7695-0786-7. doi: 10.1109/HLDVT.2000.889560.

[23] Martin Rozkovec and Ondřej Novák. Structural test of programmed FPGA circuits. In *Proceedings of the 12th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS'09)*, pages 136–139, Los Alamitos, CA, USA, April 2009. IEEE Computer Society. doi: 10.1109/DDECS.2009.5012114.

[24] Philip J. Garcia, Katherine L. Compton, Michael J. Schulte, Emily R. Blem, and Wenyin Fu. An overview of reconfigurable hardware in embedded systems. *EURASIP Journal on Embedded Systems*, 2006(56320), June 2006. ISSN 1687-3955. doi: 10.1155/ES/2006/56320.

[25] Paul M. Heysters, Gerard J. M. Smit, and Egbert Molenkamp. A flexible and energy-efficient coarse-grained reconfigurable architecture for mobile systems. *The Journal of Supercomputing*, 26(3):283–308, November 2003. ISSN 0920-8542. doi: 10.1023/A:1025699015398.

[26] PACT. XPP-III processor overview, July 2006. URL `http://www.pactxpp.com/main/download/XPP-III_overview_WP.pdf`. White Paper.

[27] Jeroen A. J. Leijten, Geoffrey Burns, Jos O. Huisken, Erwin Waterlander, and Antoine van Wel. AVISPA: a massively parallel reconfigurable accelerator. In Jari Nurmi, Jarmo Takala, and Timo D. Hämäläinen, editors, *Proceedings of the International Symposium on System-on-Chip, 2003*, pages 165–168, Piscataway, NJ, USA, November 2003. IEEE. ISBN 0-7803-8160-2. doi: 10.1109/ISSOC.2003.1267747.

[28] Arthur Abnous. *Low-Power Domain Specific Processors for Digital Signal Processing*. PhD thesis, University of California, Berkeley, Berkeley, CA, USA, 2001. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2001/8190.html`.

[29] Paul M. Heysters. *Coarse-Grained Reconfigurable Processors – Flexibility meets Efficiency*. PhD thesis, University of Twente, Enschede, The Netherlands, September 2004.

[30] Chameleon. Reconfigurable computing in hand-held multimedia computers, October 2009. URL `http://chameleon.ctit.utwente.nl`.

[31] Recore Systems. Flexible computing - efficient solutions, October 2009. URL `http://www.recoresystems.com`.

[32] Gerard K. Rauwerda, Paul M. Heysters, and Gerard J. M. Smit. Towards software defined radios using coarse-grained reconfigurable hardware. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(1):3–13, January 2008. ISSN 1063-8210. doi: 10.1109/TVLSI.2007.912075.

[33] Jian Liang, Sriram Swaminathan, and Russell Tessier. aSOC: a scalable, single-chip communications architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 37–46, October 2000. doi: 10.1109/PACT.2000.888329.

[34] William J. Dally and Brian Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of the Design Automation Conference, 2001*, pages 684–689, Los Alamitos, CA, USA, June 2001. IEEE Computer Society. doi: 10.1109/DAC.2001.935594.

[35] Shashi Kumar, Axel Jantsch, Juha-Pekka Soininen, Martti J. Forsell, Mikael Millberg, Johnny Öberg, Kari Tiensyrjä, and Ahmed Hemani. A network on chip architecture and design methodology. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'02)*, pages 105–112, Los Alamitos, CA, USA, April 2002. IEEE Computer Society. ISBN 0-7695-1486-3. doi: 10.1109/ISVLSI.2002.1016885.

[36] Kees G. W. Goossens, Jef L. van Meerbergen, Ad Peeters, and Paul Wielage. Networks on silicon: combining best-effort and guaranteed services. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 423–425. ACM-SIGDA, March 2002. ISBN 0-7695-1471-5. doi: 10.1109/DATE.2002.998309.

[37] Théodore Marescaux, Jean-Yves Mignolet, Andrei Bartic, Will Moffat, Diederik Verkest, and Serge A. Vernalde. Networks on chip as hardware components of an os for reconfigurable systems. In Peter Y. K. Cheung, George A. Constantinides, and José T. de Sousa, editors, *Proceedings of the 13th International Conference on Field Programmable Logic and Applications*, volume 2778 of *Lecture Notes in Computer Science*, pages 595–605, Germany, September 2003. Springer-Verlag. ISBN 3-540-40822-3. doi: 10.1007/b12007.

[38] William J. Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, San Francisco, CA, USA, December 2003. ISBN 0-12-200751-4.

[39] John Dielissen, Andrei Rădulescu, Kees G. W. Goossens, and Edwin Rijpkema. Concepts and implementation of the Philips network-on-chip. In *Proceedings of the IP-Based SoC Design 2003*, November 2003. URL `http://homepages.inf.ed.ac.uk/kgoossen/2003-ipsoc.pdf`.

[40] Luciano Ost, Aline Mello, José Palma, Fernando Moraes, and Ney L.V. Calazans. MAIA - a framework for networks on chip generation and verification. In *Proceedings of the Asia South Pacific Design Automation Conference, 2005*, volume 1, pages 49–52, January 2005. doi: 10.1109/ASPDAC.2005.1466128.

[41] Pascal T. Wolkotte. *Exploration within the Network-on-Chip Paradigm*. PhD thesis, University of Twente, Enschede, The Netherlands, January 2009.

[42] Ankur Agarwal, Ravi D. Shankar, and Cyril D. Iskander. Survey of network on chip (NoC) architectures & contributions. *Journal of Engineering Computing and Architectures*, 3(1), June 2009. ISSN 1934-7197.

[43] Pascal T. Wolkotte, Gerard J. M. Smit, Gerard K. Rauwerda, and Lodewijk T. Smit. An energy-efficient reconfigurable circuit switched network-on-chip. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, pages 155–162, Los Alamitos, CA, USA, April 2005. IEEE Computer Society. ISBN 0-7695-2312-9. doi: 10.1109/IPDPS.2005.95.

[44] Pascal T. Wolkotte, Gerard J. M. Smit, Nikolay K. Kavaldjiev, Jens E. Becker, and Jürgen Becker. Energy model of networks-on-chip and a bus. In Jari Nurmi, Jarmo Takala, and Timo D. Hämäläinen, editors, *Proceedings of the International Symposium on System-on-Chip, 2005*, pages 82–85, Piscataway, NJ, USA, November 2005. IEEE Computer Society. ISBN 0-7803-9294-9. doi: 10.1109/ISSOC.2005.1595650.

[45] Nikolay K. Kavaldjiev, Gerard J. M. Smit, and Pierre G. Jansen. A virtual channel router for on-chip networks. In *Proceedings of the IEEE International SOC Conference, 2004*, pages 289–293, September 2004. ISBN 0-7803-8445-8. doi: 10.1109/SOCC.2004.1362438.

[46] Dana S. Henry and Christopher F. Joerg. A tightly-coupled processor-network interface. *ACM SIGPLAN Notices*, 27(9):111–122, 1992. ISSN 0362-1340. doi: 10.1145/143371.143497.

[47] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 473–484, New York, NY, USA, 1998. ACM. ISBN 1-58113-058-9. doi: 10.1145/285930.286006.

[48] Andrei Rădulescu, John Dielissen, Santiago González Pestana, Om Prakash Gangwal, Edwin Rijpkema, Paul Wielage, and Kees G. W. Goossens. An efficient on-chip NI offering guaranteed services, shared-memory abstraction, and flexible network configuration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(1):4–17, January 2005. doi: 10.1109/TCAD.2004.839493.

[49] Edith Beigné and Pascal Vivet. Design of on-chip and off-chip interfaces for a GALS NoC architecture. In *Proceedings of the 12th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 174–183, Los Alamitos, CA, USA, March 2006. IEEE Computer Society. ISBN 0-7695-2498-2. doi: 10.1109/ASYNC.2006.16.

[50] Open MPI. Open MPI: Open source high performacne computing, April 2010. URL `http://www.open-mpi.org`.

[51] Richard L. Graham, Galen M. Shipman, Brian W. Barrett, Ralph H. Castain, George Bosilca, and Andrew Lumsdaine. Open MPI: A high-performance, heterogeneous MPI. In *Proceedings of the Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (HeteroPar '06)*, Los Alamitos, CA, USA, September 2006. IEEE Computer Society.

[52] Rostislav Dobkin, Ran Ginosar, and Christos P. Sotiriou. High rate data synchronization in GALS SoCs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(10):1063–1074, October 2006. ISSN 1063-8210. doi: 10.1109/TVLSI.2006.884148.

[53] Johnny Öberg. Clocking strategies for networks-on-chip. In Axel Jantsch and Hannu Tenhunen, editors, *Networks on Chip*, volume Part II, chapter 8, pages 153–172. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2003. ISBN 1-4020-7392-5. doi: 10.1007/0-306-48727-6_8.

[54] Michael Keating, David W. Flynn, Robert C. Aitken, Alan M. Gibbons, and Kaijian Shi. *Low Power Methodology Manual For System-on-Chip Design*. Springer US, July 2007. ISBN 0-387-71818-4. doi: 10.1007/978-0-387-71819-4_4.

[55] Dimitrios J. Soudris, Christian Piguet, and Costas E. Goutis, editors. *Designing CMOS Circuits for Low Power*. European Low-Power Initiative for Electronic System Design. Springer-Verlag, New York, NY, USA, 2004. ISBN 1-4020-7234-1.

[56] Eric F. Weglarz, Kewal K. Saluja, and Mikko H. Lipasti. Minimizing energy consumption for high-performance processing. In *Proceedings of the Asia South Pacific Design Automation Conference, 2002*, pages 199–204, Washington, DC, USA, January 2002. IEEE Computer Society. ISBN 0-7695-1441-3. doi: 10.1109/ASPDAC.2002.994919.

[57] Alan B. Grebene and Hans R. Camenzind. Phase locking as a new approach for tuned integrated circuits. In Jack A. A. Raper and B. K. Winner, editors, *Digest of Technical Papers of the IEEE International Solid-State Circuits Conference, 1969*, volume 12, pages 100–101, New York, NY, USA, February 1969. Lewis Winner. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1154749`.

[58] Ricardo Gonzalez, Benjamin M. Gordon, and Mark A. Horowitz. Supply and threshold voltage scaling for low power CMOS. *IEEE Journal of Solid-State Circuits*, 32(8):1210–1216, August 1997. ISSN 0018-9200. doi: 10.1109/4.604077.

[59] Gustavo E. Téllez, Amir H. Farrahi, and Majid Sarrafzadeh. Activity-driven clock design for low power circuits. In Richard L. Rudell, editor, *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-95)*, pages 62–65, Los Alamitos, CA, USA, November 1995. IEEE Computer Society. ISBN 0-8186-7213-7. doi: 10.1109/ICCAD.1995.479992.

[60] Jerry Frenkil and Srini Venkatraman. Power gating design automation. In David G. Chinnery and Kurt Keutzer, editors, *Closing the Power Gap between ASIC & Custom: Tools and Techniques for Low Power Design*, chapter 10, pages 251–280. Springer US, 2007.

[61] Clifford E. Cummings. Simulation and synthesis techniques for asynchronous fifo design. In *Synopsys Users Group Conference (SNUG 2002)*. Synopsys, March 2002. URL http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf.

[62] Clifford E. Cummings and Peter H. Alfke. Simulation and synthesis techniques for asynchronous fifo design with asynchronous pointer comparisons. In *Synopsys Users Group Conference (SNUG 2002)*. Synopsys, March 2002. URL http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2.pdf.

[63] Ajanta Chakraborty and Mark R. Greenstreet. Efficient self-timed interfaces for crossing clock domains. In *Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC 2003)*, pages 78–88, Los Alamitos, CA, USA, May 2003. IEEE Computer Society. ISBN 0-7695-1898-2. doi: 10.1109/ASYNC.2003.1199168.

[64] ARTEMIS, 2005. URL http://ce.et.tudelft.nl/artemis.

[65] Andy D. Pimentel, Louis O. Hertzberger, Paul J. Lieverse, Pieter van der Wolf, and Ed F. Deprettere. Exploring embedded-systems architectures with Artemis. *IEEE Computer*, 34(11):57–63, 2001. ISSN 0018-9162. doi: 10.1109/2.963445.

[66] University of Amsterdam. SESAME - simulation of embedded systems architectures for multi-level exploration, 2008. URL http://sesamesim.sourceforge.net.

[67] Daedalus, 2008. URL http://daedalus.liacs.nl.

[68] Albert C. J. Kienhuis, Edwin Rijpkema, and Ed F. Deprettere. Compaan: deriving process networks from matlab for embedded signal processing architectures. In *Proceedings of the 8th international workshop on Hardware/software codesign (CODES '00)*, pages 13–17, New York, NY, USA, 2000. ACM. ISBN 1-58113-268-9. doi: 10.1145/334012.334015.

[69] Tjerk Bijlsma, Marco J. G. Bekooij, Pierre G. Jansen, and Gerard J. M. Smit. Communication between nested loop programs via circular buffers in an embedded multiprocessor system. In *Proceedings of the 11th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2008)*, pages 33–42, New York, NY, USA, March 2008. ACM. doi: 10.1145/1361096.1361104.

[70] Andreas Hansson. *A Composable and Predictable On-Chip Interconnect*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, June 2009.

[71] The MathWorks. The MathWorks - MATLAB and Simulink for technical computing, February 2010. URL http://www.mathworks.com.

[72] Arquimedes Canedo, Takeo Yoshizawa, and Hideaki Komatsu. Automatic parallelization of Simulink applications. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization (CGO '10)*, pages 151–159, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-635-9. doi: 10.1145/1772954.1772976.

[73] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003. ISSN 0018-9219. doi: 10.1109/JPROC.2002.805829.

[74] Albert C. J. Kienhuis. *Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools*. PhD thesis, Delft University of Technology, The Netherlands, January 1999. URL http://ptolemy.eecs.berkeley.edu/~kienhuis/ftp/thesis.pdf.

[75] Erwin A. de Kock, Wim J. M. Smits, Pieter van der Wolf, Jean-Yves Brunel, Wido M. Kruijtzer, Paul J. Lieverse, Kees A. Vissers, and Gerben Essink. YAPI: application modeling for signal processing systems. In Giovanni de Micheli, editor, *Proceedings of the 37th Conference on Design Automation, 2000*, pages 402–405, Los Alamitos, CA, USA, June 2000. IEEE Computer Society. ISBN 1-58113-1897-9. doi: 10.1145/337292.337511.

[76] Sander Stuijk, Marc Geilen, and Twan Basten. SDF^3: SDF for free. In Kees G. W. Goossens and Laure Petrucci, editors, *Proceedings of the sixth International Conference on Applicatoin of Concurrency to System Design (ACSD 2006)*, pages 276–278, Los Alamitos, CA, USA, June 2006. IEEE Computer Society. ISBN 0-7695-2556-3. doi: 10.1109/ACSD.2006.23.

[77] Twente Institute for Wireless and Mobile Communications. OSYRES - operating framework for heterogeneous multi-processor systems. White paper, May 2008. URL http://www.ti-wmc.nl/images/stories/downloads/product_sheet_osyres_1.1.pdf. PD-0010-1, Rev. 1v1.

[78] Raymond Reiter. Scheduling parallel computations. *Journal of the Association for Computing Machinery (ACM)*, 15(4):590–599, October 1968. ISSN 0004-5411. doi: 10.1145/321479.321485.

[79] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Proceedings of the IFIP Congress '74*, pages 471–475, New York, NY, USA, August 1974. North-Holland. ISBN 0-7204-2803-3.

[80] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987. ISSN 0018-9219. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1458143.

[81] Kazuhito Ito and Keshab K. Parhi. Determining the iteration bounds of single-rate and multi-rate data-flow graphs. In *Proceedings of the IEEE Asia-Pacific Conference on Circuits and Systems (APCCAS'94)*, pages 163–168, December 1994. ISBN 0-7803-2440-4. doi: 10.1109/APCCAS.1994.514543.

[82] Walid A. Najjar, Edward A. Lee, and Guang R. Gao. Advances in the dataflow computational model. *Parallel Computing*, 25(13-14):1907–1929, 1999. ISSN 0167-8191. doi: 10.1016/S0167-8191(99)00070-8.

[83] Maarten H. Wiggers, Nikolay K. Kavaldjiev, Gerard J. M. Smit, and Pierre G. Jansen. Architecture design space exploration for streaming applications through timing analysis. In Jan F. Broenink, Herman W. Roebbers, Johan P. E. Sunter, Peter H. Welch, and David C. Wood, editors, *Proceedings of the Communicating Process Architectures 2005 (WoTUG-28)*, volume 63 of *Concurrent Systems Engineering Series*, pages 219–233, Amsterdam, The Netherlands, September 2005. IOS Press. ISBN 1-58603-561-4. doi: http://purl.org/utwente/54540.

[84] Marco J. G. Bekooij, Sonali Parmar, and Jef L. van Meerbergen. Performance guarantees by simulation of process. In *Proceedings of the 9th International Workshop on Software and compilers for embedded systems (SCOPES 2005)*, pages 10–19, New York, NY, USA, September 2005. ACM. ISBN 1-59593-207-0. doi: 10.1145/1140389.1140391.

[85] Marco J. G. Bekooij, Maarten H. Wiggers, and Jef L. van Meerbergen. Throughput analysis of run-time scheduled multi-rate systems with backpressure. Technical Report TR-CTIT-06-71, University of Twente, Enschede, The Netherlands, November 2006.

[86] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean A. Peperstraete. Cyclo-static data flow. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP-95)*, volume 5, pages 3255–3258, Los Alamitos, CA, USA, 1995. IEEE Computer Society. ISBN 0-7803-2431-5. doi: 10.1109/ICASSP.1995.479579.

[87] Thomas M. Parks, José L. Pino, and Edward A. Lee. A comparison of synchronous and cyclo-static dataflow. In *Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers (2-Volume Set)*, pages 204–210, Washington, DC, USA, November 1995. IEEE Computer Society. ISBN 0-8186-7370-2. doi: 10.1109/ACSSC.1995.540541.

[88] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean A. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, February 1996. ISSN 053-587X. doi: 10.1109/78.485935.

[89] Maarten H. Wiggers. *Aperiodic Multiprocessor Scheduling for Real-Time Stream Processing Applications*. PhD thesis, University of Twente, Enschede, The Netherlands, June 2009.

[90] Philip K. F. Hölzenspies, Timon D. ter Braak, Jan Kuper, Gerard J. M. Smit, and Johann L. Hurink. Run-time spatial mapping of streaming applications to heterogeneous multi-processor systems. *International Journal of Parallel Programming*, 38(1):68–83, November 2009. ISSN 1573-7640. doi: 10.1007/s10766-009-0120-y.

[91] Timon D. ter Braak, Philip K. F. Hölzenspies, Jan Kuper, Johann L. Hurink, and Gerard J. M. Smit. Run-time spatial resource management for real-time applications on heterogeneous MPSoCs. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2010)*, pages 357–362. European Design and Automation Association, March 2010.

[92] Arthur J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, October 1966. ISSN 0367-7508. doi: 10.1109/PGEC.1966.264565.

[93] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, 1974. ISSN 0001-0782. doi: 10.1145/360827.360844.

[94] Simon L. Peyton-Jones. Taming effects - the next big challenge. Presentation at Ericsson Functional Programming seminar, February 2008. URL `http://ulf.wiger.net/weblog/2008/02/29/peyton-jones-taming-effects-the-next-big-challenge/`.

[95] Paul F. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28:6, December 1996. ISSN 0360-0300. doi: 10.1145/242224.242477.

[96] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989. ISSN 0010-4620. doi: 10.1093/comjnl/32.2.98.

[97] Peter J. Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, July 2005. ISSN 0001-0782. doi: 10.1145/1070838.1070856.

[98] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. S-Net: A typed stream processing language. In Zoltan Horváth and Viktória Zsók, editors, *Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages (IFL'06)*, pages 81–97, Budapest, Hungary, 2006. Eötvös Loránd University, Hungary.

[99] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, Cambridge, MA, USA, September 1996. ISBN 0-262-13321-0.

[100] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, USA, February 2002. ISBN 0-262-16209-1.

[101] Lodewijk T. Smit, Gerard J. M. Smit, Johann L. Hurink, Hajo J. Broersma, Daniël Paulusma, and Pascal T. Wolkotte. Run-time assignment of tasks to multiple heterogeneous processors. In *Proceedings of the 5th PROGRESS Symposium on Embedded Systems*, pages 185–192, October 2004. doi: http://purl.org/utwente/49442.

[102] Tadashi Shiomi and Mitsutoshi Hatori. *Digital Broadcasting*, volume Wave Summit Course. IOS Press, Amsterdam, The Netherlands, 2000.

[103] Robert W. Chang. Synthesis of band-limited orthogonal signals for multichannel data transmission. *Bell System Technical Journal*, 46:1775–1796, December 1966.

[104] David A. Vallado. *Fundamentals of Astrodynamics and Applications*, volume 12 of *Space Technology Library*. Springer-Verlag, New York, NY, USA, 2001.

[105] Arthur C. Clarke. Extra-terrestrial relays – can rocket stations give world-wide radio coverage? *Wireless World*, pages 305–308, October 1945. URL `http://www.clarkefoundation.org/docs/ClarkeWirelessWorldArticle.pdf`.

[106] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computations*, 19(90):297–301, April 1965. ISSN 0025-5718. doi: 10.2307/2003354.

[107] Irving J. Good. The interaction algorithm and practical Fourier analysis. *Journal of the Royal Statistical Society. Series B (Methodological)*, 20(2):361–372, 1958. ISSN 0035-9246. doi: http://www.jstor.org/stable/2983896.

[108] Clive Temperton. Implementation of a self-sorting in-place prime factor FFT algorithm. *Journal of Computational Physics*, 58:283–299, May 1985. doi: 10.1016/0021-9991(85)90164-0.

[109] R. Yavne. An economical method for calculating the discrete Fourier transform. In *Proceedings of the AFIPS Fall Joint Computer Conferences*, pages 115–125, New York, NY, USA, December 1968. ACM. doi: 10.1145/1476589.1476610.

[110] Pierre L. Duhamel and Henk D.L. Hollmann. Split radix FFT algorithm. *Electronics Letters*, 20(1):14–16, 1984. doi: 10.1049/el:19840012.

[111] Anthony T. Jacobson, Dean N. Truong, and Bevan M. Baas. The design of a reconfigurable continuous-flow mixed-radix FFT processor. In *Proceedings of the IEEE International Symposion on Circuits and Systems (ISCAS'09)*, pages 1133–1136, May 2009. ISBN 1-4244-3827-6. doi: 10.1109/ISCAS.2009.5117960.

[112] Paul M. Heysters and Gerard J. M. Smit. Mapping of DSP algorithms on the Montium architecture. In *Proceedings of the 17th IEEE International Parallel and Distributed Processing Symposium (IPDPS'03)*, page 6, Washington, DC, USA, April 2003. IEEE Computer Society. ISBN 0-7695-1926-1. doi: 10.1109/IPDPS.2003.1213333.

[113] Arnaud Rivaton, Jérôme Quévremont, Qiwei Zhang, Pascal T. Wolkotte, and Gerard J. M. Smit. Implementing non power-of-two FFTs on coarse-grain reconfigurable architectures. In Jari Nurmi, Jarmo Takala, and Timo D. Hämäläinen, editors, *Proceedings of the International Symposium on System-on-Chip, 2005*, pages 74–77, Piscataway, NJ, USA, November 2005. IEEE Computer Society. ISBN 0-7803-9294-9. doi: 10.1109/ISSOC.2005.1595648.

[114] Gerard K. Rauwerda. *Multi-Standard Adaptive Wireless Communication Receivers: adaptive applications mapped on heterogeneous dynamically reconfigurable hardware*. PhD thesis, University of Twente, Enschede, The Netherlands, January 2008.

[115] Richard C. Singleton. An algorithm for computing the mixed radix fast Fourier transform. *IEEE Transactions on Audio and Electroacoustics*, 17(2):93–103, June 1969. ISSN 0018-9278. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1162042.

[116] Thomas Dettbarn and Frank Mayer. Using divide-and-conquer on custom lengthened Fourier transforms. In *Proceedings of the International Conference on Consumer Electronics*, pages 333–334, January 2006. doi: 10.1109/ICCE.2006.1598446.

[117] Guoan Bi and Yan Qiu Chen. Fast DFT algorithms for length $n = q \times 2^m$. *IEEE Transactions on Circuits and Systems—Part II: Analog and Digital Signal Processing*, 45(6):685–690, June 1998.

[118] Saad Bouguezel, M. Omair Ahmad, and M. N. Srikanta Swamy. A new radix-2/8 FFT algorithm for length-$q \times 2^m$ DFTs. *IEEE Transactions on Circuits and Systems—Part I: Regular Papers*, 51(9):1723–1732, September 2004. ISSN 1057-7122. doi: 10.1109/TCSI.2004.834508.

[119] Shmuel Winograd. On computing the discrete Fourier transform. *Mathematics of Computations*, 32:175–199, January 1978.

[120] Richard W. Linderman, Carl G. Shephard, Kent Taylor, Paul W. Coutee, Paul C. Rossbach, James M. Collings, and Robert S. Hauser. A 70-MHz 1.2-µm CMOS 16-point DFT processor. *IEEE Journal of Solid-State Circuits*, 23(2):343–350, April 1988. ISSN 0018-9200. doi: 10.1109/4.994.

[121] Pierre Lavoie. A high-speed CMOS implementation of the Winograd Fourier transform algorithm. *IEEE Transactions on Signal Processing*, 44(8):2121–2126, August 1996. ISSN 1053-587X. doi: 10.1109/78.533738.

[122] RF Engines Ltd. Mixed-radix 'dual speed' FFT product specification, April 2004. URL http://www.rfel.com/download/D04022_Matrix_FFT_MR-DS_Product_Spec.pdf.

[123] G. Goertzel. An algorithm for the evaluation of finite Fourier series. *The American Mathematical Monthly*, 65(1):34–35, January 1958.

[124] *Digital Radio Mondiale (DRM); System Specification*. European Telecommunication Standard Institute (ETSI), Sophia Antipolis, France, August 2009. URL http://www.drm.org/uploads/media/es_201980v030101p.pdf. ES 201 980.

[125] Sjoerd F. Peerlkamp. Mapping DRM baseband processing on a heterogeneous architecture. Master's thesis, University of Twente, Enschede, The Netherlands, May 2006.

[126] Pascal T. Wolkotte, Gerard J. M. Smit, and Lodewijk T. Smit. Partitioning of a DRM receiver. In *Proceedings of the 9th International OFDM-Workshop*, pages 299–304, September 2004. doi: http://purl.org/utwente/49344.

[127] Tjerk Bijlsma, Pascal T. Wolkotte, and Gerard J. M. Smit. An optimal architecture for a DDC. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, page 183, Los Alamitos, CA, USA, April 2006. IEEE Computer Society. ISBN 1-4244-0054-6. doi: 10.1109/IPDPS.2006.1639440.

[128] Bart W. van der Wal. Mapping the baseband processing of a DRM receiver to the Montium architecture. Master's thesis, University of Twente, Enschede, The Netherlands, January 2006.

[129] Jérôme Quévremont, Matthieu Colet, Eberhard Schüler, P. Rao, J. Lebender, Philipp Kraetzer, Andreas Koerner, Gerard J. M. Smit, and Pascal T. Wolkotte. DRM and MPEG4 requirements. Technical Report Deliverable D1.1.1, 4S Consortium, April 2004.

[130] *Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for 11/12 GHz satellite services*. European Telecommunication Standard Institute (ETSI), Sophia Antipolis, France, August 1997. URL `http://www.etsi.org/deliver/etsi_en/300400_300499/300421/01.01.02_60/en_300421v010102p.pdf`. EN 300 421 v1.1.2.

[131] Hervé Benoit. *Digital Television: MPEG-1, MPEG-2 and principles of the DVB system*. Wiley & Sons, Inc., New York, NY, USA, first edition, 1997. ISBN 0-340-69190-5.

[132] Hubregt J. Visser. *Array and phased array antenna basics*. Wiley, Chichester, West Sussex, UK, September 2005. ISBN 0-4708-7117-2.

[133] *Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2)*. European Telecommunication Standard Institute (ETSI), Sophia Antipolis, France, August 2009. URL `http://www.etsi.org/deliver/etsi_en/302300_302399/302307/01.02.01_60/en_302307v010201p.pdf`. EN 302 307 v1.2.1.

[134] Ben H. Allen and Mohammad Ghavami. *Adaptive Array Systems, Fundamentals and Applications*. Wiley, Chichester, West Sussex, UK, 2005.

[135] Merrill I. Skolnik. *Introduction to Radar Systems*. McGraw-Hill, New York, NY, USA, third edition, 2001. ISBN 0-07-288138-0.

[136] Harry L. van Trees. *Optimum array processing*, volume Detection, estimation and modulation theory, Part IV. Wiley-Interscience, New York, NY, USA, 2002. ISBN 0-471-09390-4.

[137] Hamish D. Meikle. *Modern Radar Systems*. Artech House, Inc., Norwood, MA, USA, second edition, 2008.

[138] Cameron T. Charles. *A Calibrated Phase and Amplitude Control System for Phased-Array Transmitters*. PhD thesis, University of Washington, Seattle, WA, USA, 2006. URL `http://www.ece.utah.edu/~ccharles/dissertation.pdf`.

[139] Richard H. Roy, Arogyasvsami J. Paulraj, and Thomas Kailath. Esprit – a subspace rotation approach to estimation of parameters of cisoids in noise. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 34(5):1340–1342, October 1986. ISSN 0096-3518.

[140] Richard H. Roy and Thomas Kailath. Esprit – estimation of signal parameters via rotational invariance techniques. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37:984–995, July 1989.

[141] Ralph O. Schmidt. Multiple emitter location and signal parameter estimation. *IEEE Transactions on Antennas and Propagation*, AP-34(3):276–280, March 1986. ISSN 0018-926X. doi: 10.1109/TAP.1986.1143830.

[142] Jasper D. Vrielink. Phased array processing: Direction of arrival estimation on reconfigurable hardware. Master's thesis, University of Twente, Enschede, The Netherlands, January 2009.

[143] John R. Treichler and Brian G. Agee. A new approach to multipath correction of constant modulus signals. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 31(2):459–472, April 1983. ISSN 0096-3518.

[144] Zhengyuan Xu. New cost function for blind estimation of M-PSK signals. In *IEEE Wireless Communications and Networking Conference*, volume 3, pages 1501–1505, September 2000. doi: 10.1109/WCNC.2000.904857.

[145] Jack Volder. The CORDIC computing technique. In *Proceedings of the AFIPS Joint Computer Conferences*, pages 257–261, New York, NY, USA, March 1959. ACM. doi: 10.1145/1457838.1457886.

[146] John S. Walther. A unified algorithm for elementary functions. In *Proceedings of the AFIPS Spring Joint Computer Conferences*, pages 379–385, New York, NY, USA, May 1971. ACM. doi: 10.1145/1478786.1478840.

[147] Ray Andraka. A survey of CORDIC algorithms for FPGA based computers. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 191–200, New York, NY, USA, February 1998. ACM. ISBN 0-89791-978-5. doi: 10.1145/275107.275139.

[148] Sin Hitotumatu. Complex arithmetic through CORDIC. *Kōdai Mathematical Seminar Reports*, 26(2-3):176–186, March 1975. doi: 10.2996/kmj/1138846999.

[149] Koen C. H. Blom. DVB-S signal tracking techniques for mobile phased arrays. Master's thesis, University of Twente, Enschede, The Netherlands, December 2009.

[150] Haskell. Haskell, February 2010. URL `http://www.haskell.org`.

List of Publications

[151] Marcel D. van de Burgwal, Gerard J. M. Smit, Gerard K. Rauwerda, and Paul M. Heysters. Hydra: an energy-efficient and reconfigurable network interface. In Toomas P. Plaks, editor, *Proceedings of the International Conference on Engineering of Reconfigurable Systems & Algorithms (ERSA'06)*, pages 171–177, Las Vegas, NV, USA, June 2006. CSREA Press. ISBN 1-60132-011-6. doi: http://purl.org/utwente/62883.

[152] Gerard J. M. Smit, André B. J. Kokkeler, Pascal T. Wolkotte, Marcel D. van de Burgwal, and Paul M. Heysters. Efficient architectures for streaming applications. In Peter M. Athanas, Jürgen Becker, Gordon J. Brebner, and Jürgen Teich, editors, *Proceedings of the Dynamically Reconfigurable Architectures*, number 06141 in Dagstuhl Seminar Proceedings, pages 1–7, Schloss Dagstuhl, Germany, October 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI). doi: http://purl.org/utwente/66541.

[153] Gerard J. M. Smit, André B. J. Kokkeler, Pascal T. Wolkotte, and Marcel D. van de Burgwal. Multi-core architectures and streaming applications. In Ion I. Mandoiu and Andrew A. Kennings, editors, *Proceedings of the Tenth International Workshop on System-Level Interconnect Prediction (SLIP 2008)*, pages 35–42, New York, NY, USA, April 2008. ACM. ISBN 1-5959-3918-0. doi: 10.1145/1353610.1353618.

[154] Gerard J. M. Smit, André B. J. Kokkeler, Pascal T. Wolkotte, Philip K. F. Hölzenspies, Marcel D. van de Burgwal, and Paul M. Heysters. The Chameleon architecture for streaming DSP applications. *EURASIP Journal on Embedded Systems*, 2007(78082), January 2007. ISSN 1687-3955. doi: 10.1155/2007/78082.

[155] Marcel D. van de Burgwal. Hydra protocol specification. Technical Report v01.09.04, University of Twente, February 2007.

[156] Kenneth C. Rovers, Marcel D. van de Burgwal, André B. J. Kokkeler, and Gerard J. M. Smit. Rationale for and design of a generic tiled hierarchical phased array beamforming architecture. In *Proceedings of the 18th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2007)*, pages 160–168, Utrecht, The Netherlands, November 2007. Technology Foundation. doi: http://purl.org/utwente/64539.

[157] Marcel D. van de Burgwal and Gerard J. M. Smit. Communication costs in a multi-tiered MPSoC. In *Proceedings of the 19th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2008)*, pages 30–34, Utrecht, The Netherlands, November 2008. Technology Foundation. ISBN 90-73461-56-1. doi: http://purl.org/utwente/65247.

[158] Kenneth C. Rovers, Marcel D. van de Burgwal, Jan Kuper, André B. J. Kokkeler, and Gerard J. M. Smit. On reconfigurable tiled multi-core programming. In *Proceedings of the 20th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2009)*, pages 507–514, Utrecht, The Netherlands, November 2009. Technology Foundation.

[159] Kenneth C. Rovers, Marcel D. van de Burgwal, André B. J. Kokkeler, Jan Kuper, and Gerard J. M. Smit. Phased array beamforming processing – semantic & dataflow model based design. In *Scientific ICT Research Event Netherlands (SIREN 2009)*, Den Haag, The Netherlands, November 2009. Informatica Platform Nederland. URL http://www.ictonderzoek.net/3/assets/File/posters/2009_44/2009_44.pdf.

[160] Pascal T. Wolkotte, Marcel D. van de Burgwal, and Gerard J. M. Smit. Non-power-of-two FFTs: Exploring the flexibility of the Montium. In Jari Nurmi and Jarmo Takala, editors, *Proceedings of the International Symposium on System-on-Chip, 2006*, pages 167–170, Piscataway, NJ, USA, November 2006. IEEE Computer Society. ISBN 1-4244-0621-8. doi: 10.1109/ISSOC.2006.321993.

[161] Marcel D. van de Burgwal, Kenneth C. Rovers, André B. J. Kokkeler, Gerard J. M. Smit, S. Kasra Garakoui, Michiel C. M. Soer, Eric A. M. Klumperink, and Bram Nauta. CMOS Beamforming Techniques project overview. In *Scientific ICT Research Event Netherlands (SIREN 2007)*, Den Haag, The Netherlands, October 2007. Informatica Platform Nederland. URL http://www.ictonderzoek.net/3/assets/File/posters/2007_23/2007_23.pdf.

[162] Marcel D. van de Burgwal, Kenneth C. Rovers, Koen C. H. Blom, André B. J. Kokkeler, and Gerard J. M. Smit. Adaptive beamforming using the reconfigurable Montium TP. In Sebastián López, editor, *Proceedings of the 13th Euromicro Conference on Digital System Design (DSD)*, pages 301–308, Los Alamitos, CA, USA, September 2010. IEEE Computer Society. doi: 10.1109/DSD.2010.13.

[163] Marcel D. van de Burgwal, Pascal T. Wolkotte, and Gerard J. M. Smit. Non-power-of-two FFTs: exploring the flexibility of the Montium TP. *International Journal of Reconfigurable Computing*, 2009(678045), July 2009. ISSN 1687-7195. doi: 10.1155/2009/678045.

[164] Koen C. H. Blom, Marcel D. van de Burgwal, Kenneth C. Rovers, André B. J. Kokkeler, and Gerard J. M. Smit. DVB-S signal tracking techniques for mobile phased arrays. In *Proceedings of the IEEE 72nd Vehicular Technology Conference, 2010*, September 2010. Accepted for publication.