Design of Reconfigurable Hardware Architectures for Real-time Applications

Modeling and Implementation

Thomas Lenart

Lund 2008

The Department of Electrical and Information Technology
Lund University
Box 118, S-221 00 LUND
SWEDEN

This thesis is set in Computer Modern 10pt,
with the LaTeX Documentation System
using Pontus Åströms thesis template.

Series of licentiate and doctoral thesis
No. 5
ISSN 1654-790X

Abstract

This thesis discusses modeling and implementation of reconfigurable hardware architectures for real-time applications. The target application in this work is digital holographic imaging, where visible images are to be reconstructed based on holographic recordings. The reconstruction process is computationally demanding and requires hardware acceleration to achieve real-time performance. Thus, this work presents two design approaches, with different levels of reconfigurability, to accelerate the image reconstruction process and related computationally demanding applications.

The first approach is based on application-specific hardware accelerators, which are usually required in systems with high constraints on processing performance, physical size, or power consumption, and are tailored for a certain application to achieve high performance. Hence, an acceleration platform is proposed and designed to enable real-time image reconstruction in digital holographic imaging, constituting a set of hardware accelerators that are connected in a flexible and reconfigurable pipeline. Hardware accelerators are optimized for high computational performance and low memory requirements. The application-specific design has been integrated into an embedded system consisting of a microprocessor, a high-performance memory controller, a digital image sensor, and a video output device. The system has been prototyped using an FPGA platform and synthesized for a $0.13\,\mu$m standard cell library, achieving a reconstruction rate of 30 frames per second running at 400 MHz.

The second approach is based on a dynamically reconfigurable architecture to accelerate arbitrary applications, which presents a trade-off between versatileness and hardware cost. The proposed reconfigurable architecture is constructed from processing and memory cells, which communicate using a combination of local interconnects and a global network. High-performance local interconnects generate a high communication bandwidth between neighboring cells, while the global network provides flexibility and access to external memory. The processing and memory cells are run-time reconfigurable to enable flexible application mapping. Proposed reconfigurable architectures are modeled and evaluated using Scenic, which is a system-level exploration environment developed in this work. A design with 16 cells is implemented and synthesized for a $0.13\,\mu$m standard cell library, resulting in low area overhead when compared with application-specific solutions. It is shown that the proposed reconfigurable architecture achieves high computation performance compared to traditional DSP processors.

Science may set limits to knowledge,

but should not set limits to imagination

Bertrand Russell (1872 - 1970)

Contents

Preface

This thesis summarizes my academic work in the digital ASIC group at the department of Electrical and Information Technology. The main contributions to the thesis are derived from the following publications:

- Thomas Lenart, Henrik Svensson, and Viktor Öwall, "Modeling and Exploration of a Reconfigurable Architecture for Digital Holographic Imaging," in *Proceedings of IEEE International Symposium on Circuits and Systems*, Seattle, USA, May 2008.

- Henrik Svensson, Thomas Lenart, and Viktor Öwall, "Modelling and Exploration of a Reconfigurable Array using SystemC TLM," in *Proceedings of Reconfigurable Architectures Workshop*, Miami, Florida, USA, April 2008.

- Thomas Lenart, Henrik Svensson, and Viktor Öwall, "A Hybrid Interconnect Network-on-Chip and a Transaction Level Modeling approach for Reconfigurable Computing," in *Proceedings of IEEE International Symposium on Electronic Design, Test and Applications*, Hong Kong, China, January 2008, pp. 398–404.

- Thomas Lenart, Mats Gustafsson, and Viktor Öwall, "A Hardware Acceleration Platform for Digital Holographic Imaging," *Springer Journal of Signal Processing Systems*, DOI: 10.1007/s11265-008-0161-2, January 2008.

- Thomas Lenart and Viktor Öwall, "Architectures for Dynamic Data Scaling in 2/4/8K Pipeline FFT Cores," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 14, no. 11, November 2006, pp. 1286–1290.

- Thomas Lenart and Viktor Öwall, "XStream - A Hardware Accelerator for Digital Holographic Imaging," in *Proceedings of IEEE International Conference on Electronics, Circuits, and Systems*, Gammarth, Tunisia, December 2005.

- Mats Gustafsson, Mikael Sebesta, Bengt Bengtsson, Sven-Göran Pettersson, Peter Egelberg, and Thomas Lenart, "High Resolution Digital Transmission Microscopy - a Fourier Holography Approach," in *Optics and Lasers in Engineering*, vol. 41, issue 3, March 2004, pp. 553–563.

- Thomas Lenart, Viktor Öwall, Mats Gustafsson, Mikael Sebesta, and Peter Egelberg, "Accelerating Signal Processing Algorithms in Digital Holography using an FPGA Platform," in *Proceedings of IEEE International Conference on Field-Programmable Technology*, Tokyo, Japan, December 2003, pp. 387–390.

- Thomas Lenart and Viktor Öwall, "A 2048 Complex Point FFT Processor using a Novel Data Scaling Approach," in *Proceedings of IEEE International Symposium on Circuits and Systems*, vol. 4, Bangkok, Thailand, May 2003, pp. 45–48.

- Thomas Lenart and Viktor Öwall, "A Pipelined FFT Processor using Data Scaling with Reduced Memory Requirements," in *Proceedings of Norchip*, Copenhagen, Denmark, November 2002, pp. 74–79.

The following papers concerning education are also published, but are not considered part of this thesis:

- Hugo Hedberg, Thomas Lenart, Henrik Svensson, "A Complete MP3 Decoder on a Chip," in *Proceedings of Microelectronic Systems Education*, Anaheim, California, USA, June 2005, pp. 103–104.

- Hugo Hedberg, Thomas Lenart, Henrik Svensson, Peter Nilsson and Viktor Öwall, "Teaching Digital HW-Design by Implementing a Complete MP3 Decoder," in *Proceedings of Microelectronic Systems Education*, Anaheim, California, USA, June 2003, pp. 31–32.

Acknowledgment

I have many people to thank for a memorable time during my PhD studies. It has been the most challenging and fruitful experience in my life, and given me the opportunity to work with something that I really enjoy!

I would first of all like to extend my sincere gratitude to my supervisor associate professor Viktor Öwall. He has not only been guiding, reading, and commenting my work over the years, but also given me the opportunity and encouragement to work abroad during my PhD studies and gain experience from industrial research. I also would like to sincerely thank associate professor Mats Gustafsson, who has been supervising parts of this work, for his profound theoretical knowledge and continuous support during this time.

I would like to thank my colleagues and friends at the department of Electrical and Information Technology (former Electroscience) for an eventful and enjoyable time. It has been a pleasure to work with you all. I will always remember our social events and interesting discussions on any topic. My gratitude goes to Henrik Svensson, Hugo Hedberg, Fredrik Kristensen, Matthias Kamuf, Joachim Neves Rodrigues, and Hongtu Jiang for friendship and support over the years, and to my new colleagues Johan Löfgren and Deepak Dasalukunte for accompanying me during my last year of PhD studies. I especially would like to thank Henrik Svensson and Joachim Neves Rodrigues for reading and commenting parts of the thesis, which was highly appreciated. The administrative and practical support at the department has been more than excellent, and naturally I would like to thank Pia Bruhn, Elsbieta Szybicka, Erik Jonsson, Stefan Molund, Bengt Bengtsson, and Lars Hedenstjerna for all their help.

I am so grateful that my family has always been there for me, and encouraged my interest in electronics from an early age. At that time, my curiosity and research work constituted of dividing all kind of electronic devices into smaller pieces, i.e. components, which were unfortunately never returned back into their original state again. This experience has been extremely valuable for me, though the focus has slightly shifted from system separation to system integration.

My dear fiancée Yiran, I am so grateful for your love and patient support. During my PhD studies I visited many countries, and I am so glad and fortunate that I got the chance to meet you in the most unexpected place. Thank you for being who you are, for always giving me new perspectives and angles of life, and for inspiring me to new challenges.

The initial part of this work was initiated as a multi-disciplinary project on digital holography, which later became Phase Holographic Imaging AB. I have had the opportunity to take part in this project for a long time, and it has been a valuable experience in many ways. From our weekly meetings in the department library, to the first prototype of the holographic microscope, what an exciting time! I would like to thank all the people who have been involved in this work, especially Mikael Sebesta for our valuable discussions since the very beginning, and for your enduring enthusiasm to take this project forward.

In 2004 and 2006, I was invited for internships at Xilinx in San José California, for 3 and 9 months, respectively. I gained a lot of experience and knowledge during that time, and I would like to thank Dr. Ivo Bolsens for giving me this opportunity. I would like to extend my gratitude to the colleagues and friends in the Xilinx Research Labs, especially my supervisors Dr. Adam Donlin and Dr. Jörn Janneck for their encouragement, support, and valuable discussions. I have a lot of good memories from the visit, and I would like to thank my local and international friends for all the unforgettable adventures we experienced together in the US.

This work has been funded by the Competence Center for Circuit Design (CCCD), and sponsored with hardware equipment from the Xilinx University Program (XUP).

Lund, May 2008

Thomas Lenart

List of Acronyms

AHB	Advanced High-performance Bus
AGU	Address Generation Unit
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction Processor
CCD	Charge-Coupled Device
CGRA	Coarse-Grained Reconfigurable Architecture
CMOS	Complementary Metal Oxide Semiconductor
CORDIC	Coordinate Rotation Digital Computer
CPU	Central Processing Unit
DAB	Digital Audio Broadcasting
DAC	Digital-to-Analog Converter
DCT	Discrete Cosine Transform
DDR	Double Data-Rate
DFT	Discrete Fourier Transform
DIF	Decimation-In-Frequency
DIT	Decimation-In-Time
DRA	Dynamically Reconfigurable Architecture
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processor *or* Digital Signal Processing

DVB	Digital Video Broadcasting
EDA	Electronic Design Automation
EMIF	External Memory Interface
ESL	Electronic System Level
FFT	Fast Fourier Transform
FIFO	First In, First Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPP	General-Purpose Processor
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDL	Hardware Description Language
IFFT	Inverse Fast Fourier Transform
IP	Intellectual Property
ISS	Instruction Set Simulator
LSB	Least Significant Bit
LUT	Lookup Table
MAC	Multiply-Accumulate
MC	Memory Cell
MPMC	Multi-Port Memory Controller
MSB	Most Significant Bit
NoC	Network-on-Chip
OFDM	Orthogonal Frequency Division Multiplexing
OSCI	Open SystemC Initiative

PE	Processing Element
PIM	Processor-In-Memory
RAM	Random-Access Memory
PC	Processing Cell
RC	Resource Cell
RGB	Red-Green-Blue
ROM	Read-Only Memory
RPA	Reconfigurable Processing Array
RTL	Register Transfer Level
SAR	Synthetic Aperture Radar
SCENIC	SystemC Environment with Interactive Control
SCV	SystemC Verification (library)
SDF	Single-path Delay Feedback
SDRAM	Synchronous DRAM
SNR	Signal-to-Noise Ratio
SOC	System-On-Chip
SQNR	Signal-to-Quantization-Noise Ratio
SRAM	Static Random-Access Memory
SRF	Stream Register File
TLM	Transaction Level Modeling
VGA	Video Graphics Array
VHDL	Very high-speed integrated circuit HDL
VLIW	Very Long Instruction Word
XML	eXtensible Markup Language

List of Definitions and Mathematical Operators

$\mathbb{Z}^+$	Positive integer space
i, j	Imaginary unit
∞	Infinity
$\propto$	Proportional
$\approx$	Approximation
$\mathcal{O}$	Computational complexity
$\lceil x \rceil$	Ceiling function. Rounds x to nearest upper integer value towars ∞
$\lfloor x \rfloor$	Floor function. Rounds x to nearest lower integer value towars $-\infty$
$x \in A$	The element x belongs to the set A
$x \notin A$	The element x does not belong to the set A
$\mathcal{F}(x(t))$	Fourier transform of $x(t)$
$\mathcal{F}^{-1}(X(f))$	Inverse Fourier transform of $X(f)$
$x * h$	The convolution of x and h
$\boldsymbol{r}$	Spatial vector
$\lvert\boldsymbol{r}\rvert$	Spatial vector length
$\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}, \hat{\boldsymbol{z}}$	Cartesian unit vectors

Chapter 1

Introduction

This thesis discusses modeling and implementation of both application-specific and dynamically reconfigurable hardware architectures. Application-specific hardware architectures are usually found in systems with high constraints on processing performance, physical size, or power consumption, and are tailored for a certain application or algorithm to achieve high performance. However, an application-specific architecture provides limited flexibility and reuse. It is also likely that a system platform may require a large number of specific hardware accelerators, but only a few units will operate concurrently. In contrast, a reconfigurable architecture trades performance for increased flexibility. It allows the hardware architecture to be changed during run-time in order to accelerate arbitrary algorithms, hence extends the application domain and versatileness of the device.

In this work, two approaches to reconfigurable hardware design are presented. In the first approach, a high-performance application-specific hardware accelerator is proposed, which is designed to enable real-time image reconstruction in digital holographic imaging [1]. The hardware accelerator provides reconfigurability on *block level*, using a flexible pipeline architecture. In the second approach, a reconfigurable platform is designed and modeled by combining a conventional processor based solution with a dynamically reconfigurable architecture. Hence, the platform can be reused to map arbitrary applications, which is a trade-off between versatileness and hardware cost. A reconfigurable approach enables flexibility in the algorithm design work, and also the possibility to dynamically reconfigure the accelerator for additional processing steps. An example is post-processing operations to enhance the quality and perception of reconstructed images.

1.1 Challenges in Digital Hardware Design

The hardware design space is multi-dimensional and constrained by many physical and practical boundaries. Hence, designing hardware is a trade-off between system performance, resource and development costs, power consumption, and other parameters related to the implementation. Hence, a design goal is to find an optimal balance to efficiently utilize available system resources to avoid performance bottlenecks.

However, performance bottlenecks do not disappear – they move around. Improving a part of the design is in practice to push the performance issues into another part of the design, or even to another part of the design space. For example, increasing the processing performance traditionally requires a higher clock frequency f_{clk}, which will increase the power consumption and therefore also the heat dissipation. Heat dissipation is physically constrained to avoid damage due to overheating of the device. One way to prevent this situation is to lower the supply voltage V_{DD}, since it has a quadratic effect on the dynamic power as

$$P_{\mathrm{dyn}} = \alpha C_{\mathrm{L}} V_{\mathrm{DD}}^2 f_{\mathrm{clk}},$$

where α is the switching activity and C_{L} is the load capacitance [2]. However, the supply voltage also affects the propagation time t_{p} in logic gates as

$$t_{\mathrm{p}} \propto \frac{V_{\mathrm{DD}}}{(V_{\mathrm{DD}} - V_{\mathrm{T}})^{\beta}},$$

where V_{T} is the threshold voltage and β is a technology specific parameter. Hence, lowering the supply voltage means that the system becomes slower and counteracts the initial goal. This is the situation for most sequential processing units, e.g. microprocessors, where the processing power heavily depends on the system clock frequency.

The current trend in the computer industry is to address this situation using multiple parallel processing units, which shows the beginning of a paradigm shift towards parallel hardware architectures [3]. However, conventional computer programs are described as sequentially executed instructions and can not easily be adapted for a multi-processor environment. This situation prevents a potential speed-up due to the problem of finding enough parallelism in the software. The speedup S from using N parallel processing units is calculated using Amdahl's law as

$$S = \frac{1}{(1-p) + \frac{p}{N}},$$

where p is the fraction of the sequential program that can be parallelized [4]. Assuming that 50% of the sequential code can be executed on parallel processors, the speedup will still be limited to a modest factor of 2, no matter how many parallel processors that are used. Parallel architectures have a promising future, but will require new design approaches and programming methodologies to enable high system utilization.

1.1.1 Towards Parallel Architectures

In 1965, Intel's co-founder Gordon E. Moore made an observation that the number of transistors on an integrated circuit is increasing exponentially [5]. Since then, Moore's *law* has been used more than four decades to describe trends in computer electronics and integrated circuits. Niklaus E. Wirth, the developer of the Pascal programming language, more recently stated another "law" that *software gets slower faster than hardware gets faster.* This is only an ironic observation that software developers rely on Moore's law when the software complexity increase, and expect continuous exponential growth in performance as well as in complexity (number of transistors). However, there is a more serious aspect to this observation related to the current trend towards multi-processor and highly parallel systems. In the past, software abstraction has been a way to hide complex behavior. However, parallel systems require more from the software developer in terms of knowledge about the underlying architecture to fully utilize the hardware. From a software perspective, this would imply the use of programming languages that describe concurrent systems, to expose and utilize the system hardware resourced in order to efficiently program future parallel computer systems, or the use of programming interfaces for parallel computing [6, 7].

In addition, increasing hardware design complexity advocates tools for high-level modeling and virtual prototyping for hardware and software co-design. This is becoming an emergent requirement when designing multi-processor based systems, to understand and analyze system-level behavior and performance.

1.1.2 Application-Specific vs. Reconfigurable Architectures

Despite the rapid development pace of modern computer systems, there will always be situations where more customized solution will have a better fit. Examples are in mobile applications where power is a crucial factor, or in embedded systems that require real-time processing. These systems are usually composed by a generic processor-based hardware platform that includes one or more *application-specific* hardware accelerators. Application-specific architectures results in the highest performance with the lowest power consumption, which may seem like an ideal situation. However, application-specific hardware accelerators provide limited flexibility and reusability compared to more general-purpose processing units. They are also a physical part of device, which increase the manufacturing cost and the power consumption even if the hardware accelerator is only active for a fraction of the user-time.

In contrast, the design of reusable and *reconfigurable architectures* is an alternative to combine the flexibility of generic processing units with the processing power of application-specific architectures. Today, proposed reconfigurable architectures and processor arrays range from clusters of homogeneous full-scale multiprocessors to arrays of weakly programmable elements or configurable logic blocks, which communicate over a network of interconnects. By observing current trends, the question for the future will probably not be *if* highly parallel and reconfigurable architectures will become mainstream, but rather *how* they can be efficiently explored, constructed, and programmed [8]. A likely scenario for the future is processing platforms combining application-specific architectures, reconfigurable architectures, and generic processors to support a wide range of applications.

1.2 Contributions and Thesis Outline

The thesis is divided into four parts, which cover modeling, design, and implementation of application-specific and reconfigurable hardware architectures. Parts I and II present an application-specific hardware acceleration platform for digital holographic imaging. Part I proposes a system architecture, where a set of hardware accelerations are connected in a reconfigurable communication pipeline, to enable flexibility and real-time image reconstruction. Part II presents a high-performance and scalable FFT core, which is the main building block in the hardware acceleration platform in Part I. Furthermore, proposed ideas on index scaling in pipeline FFT architectures, originally presented in [9], has been further developed by other research groups [10].

In Parts III and IV, the research is generalized towards dynamically reconfigurable architectures based on scalable processing arrays. Part III presents an exploration framework and simulation models, which are used for constructing and evaluating reconfigurable architectures. In Part IV, a coarse-grained reconfigurable architecture is proposed, which consists of an array of processing and memory cells for accelerating arbitrary applications. An intended target application for this work is again digital holographic imaging.

A general overview of hardware design is given in Chapter 2, discussing processing alternatives and memory concerns. This chapter also presents the design flow from high-level specification down to physical layout, with the emphasis on system modeling and the use of virtual platforms for design exploration and hardware/software co-design. Chapter 3 presents an introduction to digital holographic imaging, which has been the driver application for the first part of this work, and as a motivation for the second part of the work. Algorithmic requirements and hardware design trade-offs are also discussed.

The research on reconfigurable architectures is a collaboration between the author of this thesis and PhD student Henrik Svensson. The work is however based on an individually developed set of reconfigurable models, using the presented Scenic exploration environment as a common development platform. For published material, the first named author indicates which reconfigurable modules that are evaluated. The Scenic core library has been mainly developed by the author, while the environment has been ported to additional operating systems and integrated with ArchC [11], which is an academic architectural description language for processors, by Henrik Svensson.

Some of the VHDL models presented in Part IV have been developed in collaboration with students in the *IC project and verification* course, according to the authors specification. The author especially wants to acknowledge the work of Chenxin Zhang.

Part I: A Hardware Acceleration Platform for Digital Holographic Imaging

Many biomedical imaging systems have high computational demands but still require real-time performance. When desktop computers are not able to satisfy the real-time requirements, dedicated or application-specific solutions are necessary. In this part, a hardware acceleration platform for digital holographic imaging is presented, which transforms an interference pattern captured on a digital image sensor into visible images. A computationally and memory efficient pipeline, referred to as Xstream, is presented together with simulations and implementation results for the final design. The work shows significant reductions in memory requirements and high speedup due to improved memory organization and efficient processing. The accelerator has been embedded into an FPGA system platform, which is integrated into a prototype of the holographic system.

Publications:

- Thomas Lenart, Mats Gustafsson, and Viktor Öwall, “A Hardware Acceleration Platform for Digital Holographic Imaging,” *Springer Journal of Signal Processing Systems*, DOI: 10.1007/s11265-008-0161-2, January 2008.
- Thomas Lenart and Viktor Öwall, “XStream - A Hardware Accelerator for Digital Holographic Imaging,” in *Proceedings of IEEE International Conference on Electronics, Circuits, and Systems*, Gammarth, Tunisia, December 2005.

- Thomas Lenart, Viktor Öwall, Mats Gustafsson, Mikael Sebesta, and Peter Egelberg, "Accelerating Signal Processing Algorithms in Digital Holography using an FPGA Platform," in *Proceedings of IEEE International Conference on Field-Programmable Technology*, Tokyo, Japan, December 2003, pp. 387–390.
- Mats Gustafsson, Mikael Sebesta, Bengt Bengtsson, Sven-Göran Pettersson, Peter Egelberg, and Thomas Lenart, "High Resolution Digital Transmission Microscopy - a Fourier Holography Approach," in *Optics and Lasers in Engineering*, vol. 41, issue 3, March 2004, pp. 553–563.

Part II: A High-performance FFT Core for Digital Holographic Imaging

The main building block in the holographic imaging platform is a large size two-dimensional FFT core. A high-performance FFT core is presented, where dynamic data scaling is used to increase the accuracy in the computational blocks, and to improve the quality of the reconstructed holographic image. A hybrid floating-point scheme with tailored exponent datapath is proposed together with a co-optimized architecture between hybrid floating-point and block floating-point. A pipeline FFT architecture using dynamic data scaling has been fabricated in a standard CMOS process, and is used as a building block in the FPGA prototype presented in Part I.

Publications:

- Thomas Lenart and Viktor Öwall, "Architectures for Dynamic Data Scaling in 2/4/8K Pipeline FFT Cores," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 14, no. 11, November 2006, pp. 1286–1290.
- Thomas Lenart and Viktor Öwall, "A 2048 Complex Point FFT Processor using a Novel Data Scaling Approach," in *Proceedings of IEEE International Symposium on Circuits and Systems*, vol. 4, Bangkok, Thailand, May 2003, pp. 45–48.
- Thomas Lenart and Viktor Öwall, "A Pipelined FFT Processor using Data Scaling with Reduced Memory Requirements," in *Proceedings of Norchip*, Copenhagen, Denmark, November 2002, pp. 74–79.

Part III: A Design Environment and Models for Reconfigurable Computing

System-level simulation and exploration tools and models are required to rapidly evaluate system performance in the early design phase. The use of virtual platforms enables hardware modeling as well as early software development. The exploration tool SCENIC is developed, which introduces functionality to access and extract performance related information during simulation. A set of model generators and architectural generators are also proposed to create custom processor, memory, and system architectures that facilitate design exploration. The Scenic exploration tool is used in Part IV to evaluate dynamically reconfigurable architectures.

Publications:

- Henrik Svensson, Thomas Lenart, and Viktor Öwall, "Modelling and Exploration of a Reconfigurable Array using SystemC TLM," in *Proceedings of Reconfigurable Architectures Workshop*, Miami, Florida, USA, April 2008.
- Thomas Lenart, Henrik Svensson, and Viktor Öwall, "A Hybrid Interconnect Network-on-Chip and a Transaction Level Modeling approach for Reconfigurable Computing," in *Proceedings of IEEE International Symposium on Electronic Design, Test and Applications*, Hong Kong, China, January 2008, pp. 398–404.

Part IV: A Run-time Reconfigurable Computing Platform

Reconfigurable hardware architectures are emerging as a suitable approach to combine high performance with flexibility and programmability. While fine-grained architectures are capable of bit-level reconfiguration, more recent work focus on more coarse-grained architectures that enable higher performance using word-level data processing. A coarse-grained dynamically reconfigurable architecture is presented, and constructed from an array of processing and memory cells. The cells communicate using local interconnects and a hierarchical network using routers. The design has been modeled using the Scenic exploration environment and simulation models, and implemented in VHDL and synthesized for a $0.13\,\mu$m cell library.

Publications:

- Thomas Lenart, Henrik Svensson, and Viktor Öwall, "Modeling and Exploration of a Reconfigurable Architecture for Digital Holographic Imaging," in *Proceedings of IEEE International Symposium on Circuits and Systems*, Seattle, USA, May 2008.
- Henrik Svensson, Thomas Lenart, and Viktor Öwall, "Modelling and Exploration of a Reconfigurable Array using SystemC TLM," in *Proceedings of Reconfigurable Architectures Workshop*, Miami, Florida, USA, April 2008.
- Thomas Lenart, Henrik Svensson, and Viktor Öwall, "A Hybrid Interconnect Network-on-Chip and a Transaction Level Modeling approach for Reconfigurable Computing," in *Proceedings of IEEE International Symposium on Electronic Design, Test and Applications*, Hong Kong, China, January 2008, pp. 398–404.

Chapter 2

Digital System Design

Digital systems range from desktop computers for general-purpose processing to embedded systems with application-specific functionality. A digital system platform comprises of elements for data *processing* and *storage*, which are connected and integrated to form a design-specific architecture.

In this chapter, elements for data processing are further divided into three classes, namely *programmable*, *reconfigurable*, and *application-specific* architectures. Programmable architectures include for example the general-purpose processor (GPP), the application-specific instruction processor (ASIP), and the digital signal processor (DSP). The reconfigurable architectures range from bit-level configuration, such as the field-programmable gate array (FPGA), to word-level configuration in coarse-grained reconfigurable architectures (CGRA). Application-specific architectures are required when there are high constraints on power consumption or processing performance, such as in hand-held devices and real-time systems, where tailored hardware may be the only feasible solution [12]. Figure 2.1 illustrates how different architectures trade flexibility for higher efficiency. The term *flexibility* includes programmability and versatility, while the term *efficiency* relates to processing performance and energy efficiency. General-purpose devices provide a high level of flexibility and are located in one end of the of the design space, while the specialized and optimized application-specific architectures are located in the other end [13].

Efficient memory and storage elements are required to supply the processing elements with data. Memory performance depends on how the memory system is *organized*, as well as how data is *stored*. The memory organization includes the memory hierarchy and configuration, while storage aspects relate to how

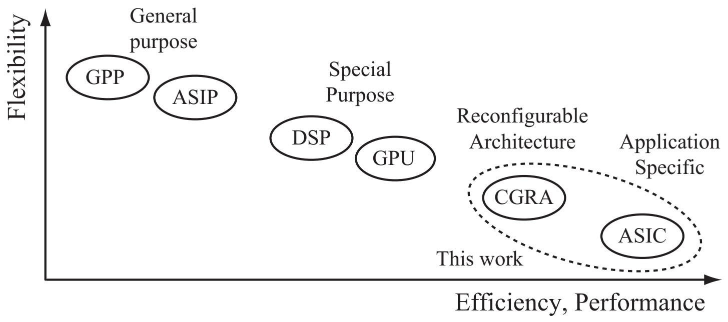


Figure 2.1: Efficiency and performance as a function of flexibility for various architectures. Architectures from different domains can not be directly relates, hence they are grouped in different domains.

well data is re-used through caching, and the memory access pattern to non-cacheable data [14]. Hence, system performance is a balance between efficient processing and efficient memory management.

Constructing digital systems not only require a set of hardware building blocks, but also a methodology and design flow to generate a functional circuit from a high-level specification. In addition, the current trends towards high-level modeling and system-level exploration require more advanced design steps to enable software and hardware co-development from a common *virtual platform* [15]. For complex system design, exploration tools require the use of abstraction levels to trade modeling accuracy for simulation speed [16], which is further discussed in Part III.

2.1 Programmable Architectures

Programmable architectures are further divided into three groups: general-purpose processors (GPP), configurable instruction-set processors, and special-purpose processors. The GPP has a general instruction set that has not been optimized for any specific application domain. Configurable instruction-set processors offer the possibility to extend the instruction set and to tune the processor for a user-defined application [17]. The special-purpose processors include instructions and additional hardware resources for primitives commonly used in its field of operation, for example signal or graphics processing. The processor architectures are trade-offs between computational efficiency, flexibility, and hardware requirements.

2.1.1 General-Purpose Processors

The microprocessor is a generic computational unit, designed to support any algorithm that can be converted into a computer program. Microprocessors suffer from an obvious performance bottleneck: the sequential nature of program execution. Instructions are processed one after the other, with the upside being high instruction locality and the downside being high instruction dependency.

There are several ways to improve the performance of microprocessors, i.e. increase the number of executed instructions per second. By dividing the instruction execution into a set of stages, several instructions can execute concurrently in an instruction *pipeline*. An extension to the pipelining concept is to identify independent instruction in run-time that can be sent to different instruction pipelines, commonly known as a *superscalar* architecture. However, evaluating instruction dependency during run-time is a costly technique due to the increased hardware complexity. Another approach is to let *software* analyze the dependencies and provide a program where several instructions are given in parallel, which is known as a very long instruction word (VLIW) processor. A comparison between different processor architectures are presented in [18].

In recent computer systems, multi-core processors are introduced to address the limitation of pure sequential processing. Independent applications, or parts of the same application, are parallelized over a plurality of processing units. This means that processing speed may potentially increase with a factor equal to the number of processing cores. In reality, speed-up for a single thread is limited by the *instruction level parallelism* (ILP) in the program code, which indicates to what extent a sequential program can be parallelized [19]. In the situation of a single-threaded application with limited ILP, multi-core will not be able to provide any noticeable speed-up. However, if the programmer divides the application into multiple parallel threads of execution, a multi-core system can provide *thread level parallelism* (TLP).

2.1.2 Configurable Instruction-Set Processors

One way to improve processing performance is to adapt and extend the processor's instruction set for a given application [20]. Profiling tools are first used analyze the application data-flow graph to find and extract frequently used computational kernels, which constitute the main part of the execution time [21]. The kernels are groups of basic instructions that often occur in conjunction, hence it is possible to merge them into a specialized instruction [22]. By extending the processor with support for such operations, the overall performance of the system is significantly improved. However, from a system perspective the performance improvement has to be weighted against resource, area, and power requirements for introducing specialized instructions.

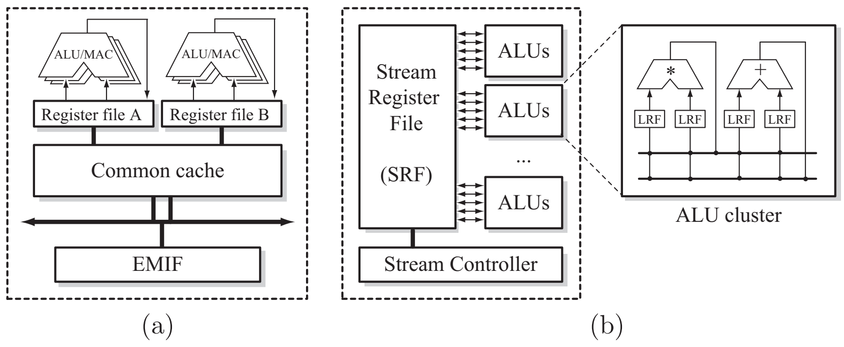


Figure 2.2: (a) A DSP architecture with two separate register files and multiple functional units on each data path. (b) A stream processor with a high-performance register file connected to an array of ALU clusters. Each cluster contains multiple functional units and local register files. The stream controller handles data transfers to external memory.

In addition to extending the processor with new instructions, the compiler tools have to be modified to support the extended instruction set. Frameworks for building extendable processors can be found in industry as well as in academia [23]. Advanced tools also provide generation of processor models for simulation, and tools for system tuning and exploring architectural trade-offs.

2.1.3 Special-Purpose Processors

An example of a special-purpose device is the digital signal processor (DSP). A digital signal processor has much in common with a general-purpose microprocessor but contains additional features and usually include more than one computation unit and register bank, as illustrated in Figure 2.2(a). In signal processing algorithms, a set of frequently used operations can be isolated. An example is multiplication followed by accumulation, present in for instance FIR filters [24]. In a DSP, this operation is executed in a single clock cycle using dedicated multiply-accumulate (MAC) hardware. Another useful feature is multiple addressing modes such as modulo, ring-buffer, and bit-reversed addressing. In signal processing, overflow causes serious problems when the values exceed the maximum wordlength. In a conventional CPU, integer values wrap around on overflow and corrupt the result value. DSPs often include saturation logic for integer arithmetic, preventing the value to wrap and cause less damage to the final result. Despite the fact that DSPs are referred to as a special-purpose processors, their application domain is fairly wide.

Stream processors combine a highly parallel processing architecture with a memory controller that supervise the data streams from external and local memories, and an architectural overview is shown in Figure 2.2(b) [25]. A stream register file (SRF) manages the exchange of data between computational clusters and the external memory. The clusters are based on parallel processing elements that are programmed using VLIW instructions. Examples of stream processors are Imagine and Merrimac from Stanford [26, 27].

Another special-purpose processor with a more narrow application domain is the graphic processing units (GPU), which resembles the stream processor. Traditionally, accelerators for graphic processing have been application-specific to be able to handle an extreme amount of operations per second. As a result, GPUs provided none or limited programmability. However, the trend is changing towards general purpose GPUs (GPGPU), which widens the application domain for this kind of special-purpose processors [28]. Due to the massively parallel architecture, the GPGPU is suitable for mapping parallel and streaming applications. The use of high-performance graphic cards for general processing is especially interesting for applications with either real-time requirements or long execution times.

A common characteristic for special-purpose processors is the ability to operate on sets of data. Execution in a generic processor is divided into *which* data to process and *how* to process that data, referred to as control-flow and data-flow, respectively. The control flow is expressed using control statements such as *if*, *for*, *while*. When applying a single operation to a large data set or a data stream, the control statements will dominate the execution time. A solution is to use a single operation or kernel that is directly applied to a larger data set. This is referred to as a single-instruction multiple-data (SIMD) or multiple-instruction multiple-data (MIMD) processing unit, according to Flynn's taxonomy [29]. Hence, the control overhead is reduced and the processing throughput increased.

2.2 Reconfigurable Architectures

In contrast to programmable architectures, the *reconfigurable architectures* enable hardware programmability. It means that not only the software that runs on a platform is modified, but also how the architecture operates and communicates [13, 30–32]. Hence, an application is accelerated by allocating a set of required processing, memory, and routing resource.

Many presented reconfigurable architectures consist of an array of processing and storage elements [33]. Architectures are either *homogeneous*, meaning that all elements implement the same functionality, or *heterogeneous* where elements provide different functionality. Processing elements communicate using

Table 2.1: A comparison between fine-grained and coarse-grained architectures. Development time and design specification refers to *applications* running on the platform.

Properties	Fine-grained	Coarse-grained
Granularity	bit-level (LUT)	word-level (ALU)
Versatility/Flexibility	high	medium/high
Performance	medium	high
Interconnect overhead	large	small
Reconfiguration time	long (ms)	short (μs)
Development time	long	medium
Design specification	hardware	software
Application domain	Prototyping HPC	RTR systems HPC

either statically defined interconnects or programmable switch boxes, which are dynamically configured during run-time. Physical communication channels for transferring data are reserved for a duration of time, referred to as *circuit switching*, or shared between data transfers using *packet switching* to dynamically route the traffic [34].

The size of the reconfigurable elements is referred to as the *granularity* of the device. Fine-grained devices are usually based on small look-up tables (LUT) to enable bit-level manipulations. These devices are extremely versatile and can be used to map virtually any algorithm. However, fine-grained architectures are inefficient in terms of hardware utilization of logic and routing resources. In contrast, coarse-grained architectures use building blocks in a size ranging from arithmetic logic units (ALU) to full-scale processors. This yields a higher performance when constructing standard datapaths, since the arithmetic units are constructed more efficiently, but the device becomes less versatile. The properties of fine-grained and coarse-grained architectures are summarized in Table 2.1, and are further discussed in the following sections.

Reconfigurable architectures are used for multiple purposes and for a wide application domain, which includes:

Prototyping - Versatile fine-grain architectures, i.e. FPGAs, are commonly uses for functional verification prior to ASIC manufacturing. This is of high importance when software simulations are not powerful enough to verify that the design operates correctly during exhaustive runs.

Reconfigurable computing - As an alternative to develop application-specific hardware for low-power and compute-intensive operation, coarse-grained reconfigurable architectures are used to combine efficiency with reconfigurability. This enables post-manufacturing programmability and a reduced development time and risk. Application mapping becomes flexible, where all hardware resources are either used for a single task, or shared to handle parallel task simultaneously.

High-Performance Computing - A reconfigurable architecture can accelerate and off-load compute-intensive tasks in a conventional computer system. The reconfigurable architecture is either placed inside the processor datapath, or as an external accelerator that communicates over the system bus or directly on the processor front side bus (FSB). Both fine-grained and coarse-grained architectures are candidates for high-performance computing [35].

2.2.1 Fine-Grained Architectures

Fine-grained reconfigurable architectures have been available for almost three decades, and is a natural part of digital system design and verification. The fine-grained devices include programmable array logic (PAL), complex programmable logic devices (CPLD), and field-programmable gate arrays (FPGA). The devices are listed in increasing complexity, and also differ in architecture and how the configuration data is stored. Smaller devices use internal non-volatile memory, while larger devices are volatile and programmed from an external source. However, since FPGAs have the most versatile architecture, it will here be used as a reference for fine-grained reconfigurable architectures.

The traditional FPGA is a fine-grained logic device that can be reconfigured after manufacturing, i.e. *field programmable*. The FPGA architecture is illustrated in Figure 2.3(a), and is constructed from an array of configurable logic blocks (CLB) that emulate boolean logic and basic arithmetic using look-up tables (LUT). An interconnect network, which is configured using switchboxes, enables the logic blocks to communicate.

The realization of memories using logic blocks is an inefficient approach, hence FPGAs include small dedicated block RAMs to implement single and dual port memories. Block RAMs may be connected to emulate larger memories, and also enables data exchange between two clock domains on the device. Furthermore, a current industry trend is to include more special-purpose structures in the FPGA, referred to as macro-blocks. Available macro-blocks are multipliers and DSP units, which lead to significant performance improvement over implementations using logic blocks. Larger devices may include one or more GPP, which are directly connected to the FPGA *fabric*. This enables rapid system design using a single FPGA platform.

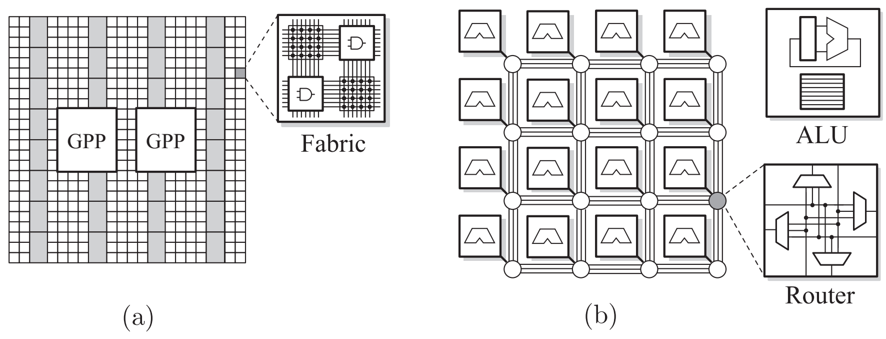


Figure 2.3: (a) Overview of the FPGA architecture with embedded memory (in gray) and GPP macro-blocks. The FPGA fabric containing logic blocks and switchboxes are magnified. (b) An example of a coarse-grained reconfigurable architecture, with an array of processing elements (ALU) and a routing network.

The FPGA design flow starts with a specification, either from a hardware description language (HDL) or from a more high-level description. HDL code is synthesized to gate level and the functionality is mapped onto the logic blocks in the FPGA. The final step is to route the signals over the interconnect network and to generate a device specific *configuration file* (bit-file). This file contains information on how to configure logic blocks, interconnect network, and IO-pads inside the FPGA. The configuration file is static and can not alter the functionality during run-time, but some FPGAs support run-time reconfiguration (RTR) and the possibility to download partial configuration files while the device is operating [36]. However, the reconfiguration time is in the range of milliseconds since the FPGA is configured at bit-level.

2.2.2 Coarse-Grained Architectures

Coarse-grained reconfigurable architectures are arrays constructed from larger computational elements, usually in the size of ALUs or smaller programmable kernels and state-machines. The computational elements communicate using a routing network, and an architectural overview is illustrated in Figure 2.3(b). In this way, the coarse-grained architecture requires less configuration data, which improves the reconfiguration time, while the routing resources generate a lower hardware overhead.

In contrast to a fine-grained FPGA, course-grained architectures are designed for partial and run-time reconfiguration. This is an important aspect due to situations when hardware acceleration is required for short time durations or only during the device initialization phase. Instead of developing an application-specific hardware accelerator for each individual situation, a reconfigurable architecture may be reused to accelerate arbitrary algorithms. Once the execution of one algorithm completes, the architecture is reconfigured for other tasks. The work on a dynamically reconfigurable architecture is further discussed in Part IV.

The possibility to support algorithmic scaling is also an important aspect. Algorithmic scaling means that an algorithm can be mapped to the reconfigurable array in multiple ways, which could be a trade-off between processing performance and area requirements. A library containing different algorithm mappings would enable the programmer or the mapping tool to select a suitable architecture for each situation. A low complexity algorithm mapping may be suitable for non-critical processing, while a parallel mapping may be required for high performance computing.

From an algorithm development perspective, the coarse-grained architectures differ considerably from the design methodology used for FPGA development. While FPGAs use a hardware-centric methodology to map functionality into gates, the coarse-grained architectures enable a more software-centric and high-level approach. Hence, it allows hardware accelerators to be developed *on-demand*, and potentially in the same language used for software development. Unified programming environments enhance productivity by simplifying system integration and verification.

2.3 Application-Specific Architectures

In some cases it is not feasible to use programmable solutions, where the reasons can be constraints related to power consumption, area requirements, and real-time performance. Instead, an application-specific architecture that satisfies the constraints is tailored for this situation. Many digital signal processing (DSP) algorithms, such as filters and transforms, allow efficient mapping to an application-specific architecture due to their highly regular structure. DSP algorithms are also easy to scale, using folding and un-folding, to find a balance between throughput and area requirements.

Application-specific architectures are found in most embedded systems, often in parts of the design that has hard real-time constraints. In cellular applications, the most time-critical parts of the digital transceiver require dedicated hardware to handle the high real-time and the throughput requirements. Application-specific hardware is also required when communicating with external components using fixed protocols, such as interfaces to memories and digital image sensors.

For embedded systems, application-specific hardware accelerators are commonly used to off-load computationally intensive software operations, in order to save power or to gain processing speed. First, profiling is used to reveal and identify performance critical sections that consume most processing power. Secondly, a hardware accelerator is designed and connected to the processing unit, either as a *tightly-coupled* co-processor or as a *loosely-coupled* accelerator directly connected to the system bus [37].

Since application-specific architectures are only restricted by user-defined constraints, the architectural mapping can take several forms. For design constrained by area, the hardware can be *time-multiplexed* to re-use a single processing element. The idea is the same as for the microprocessor: to process data sequentially. The difference is that the program is replaced by a state-machine, which reduces the control-flow overhead and increases the efficiency. In contrast to time-multiplexed implementations, *pipeline* and *parallel* architectures are designed to maximize throughput. Parallel architectures are often referred to as *direct-mapped*, which relates to the data-flow graph of the parallel algorithm.

Even application-specific architectures are often designed with flexibility in mind, but are still often limited to fixed algorithms. In datapath designs, a flexible dataflow enables algorithmic changes on a structural level. Examples are algorithms with similar dataflow, such as the discrete cosine transform (DCT) and the fast Fourier transform (FFT), hence having small structural differences. In control-based designs, programmable state-machines provide flexibility to change algorithmic parameters using the same structural datapath. Examples in filter design are variable filter length and ability to update filter coefficients.

2.4 Memory and Storage

In addition to the computation elements, efficient memory and storage elements are required to store and supply the system with data. However, for decades the speed of processors has improved at a much higher rate than for memories, causing a processor-memory *performance gap* [38]. The point where computational elements are limited by the memory performance is referred to as the *memory wall*. For high-performance systems using dedicated hardware accelerators or highly parallel architectures, this problem becomes even more critical.

2.4.1 Data Caching

There are several techniques to hide the processor-memory gap, both in terms of latency and throughput. The most common approach is caching of data,

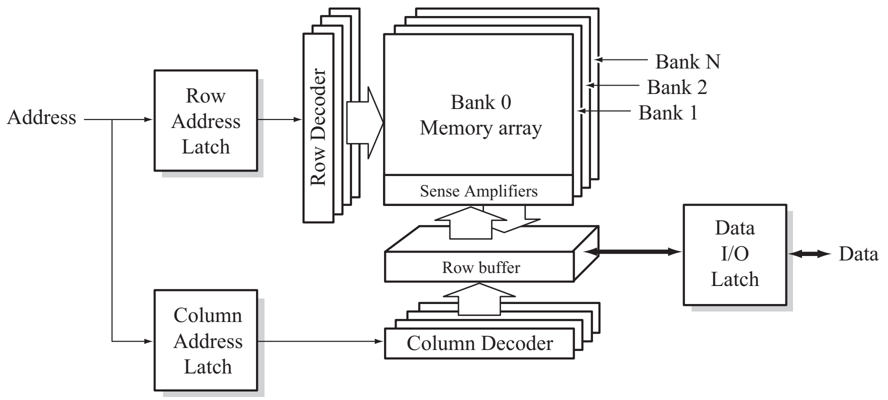


Figure 2.4: Modern SDRAM is divided into banks, rows and columns. Accessing consequtive elements result in a high memory bandwidth and must be considered to achieve high performance.

which means that frequently accessed information is stored in a smaller but faster memory close to the processing unit. Caches use static random access memory (SRAM), which has the desirable property of fast uniform access time to all memory elements. Thus, random access to data has low latency, assuming that the data is available in the cache. In contrast, accessing data that is currently not available in the cache generates a *cache miss*. For state-of-the-art processors, a cache miss is extremely expensive in terms of processor clock cycles. It is not uncommon that a cache miss stalls the processor for hundreds of clock cycles until data becomes available. The reasons are long round-trip latency and the difference in clock frequency between external memory and the processor.

Unfortunately, caching is not always helpful. In streaming applications each data value is often only referenced once, hence resulting in continuous access to new data in external memory. In this situation, the memory access pattern is of high importance as well as hiding the round-trip latency. In this case, the use of burst transfers with consecutive data elements is necessary to minimize the communication overhead, and to enable streaming applications to utilize the memory more efficiently.

2.4.2 Data Streaming

Many signal processing algorithms are based on applying one or more kernel functions to one or more data sets. If a data set can be described as a sequential stream, the kernel function can efficiently be applied to all elements in the stream. From an external memory perspective, streaming data generates a higher bandwidth than random access operations. This is mainly from the fact that random access generates a high round-trip latency, but also due to the physical organization of burst-oriented memories. While high-performance processors rely on caches to avoid or hide the external memory access latency, streaming application often use each data value only once and can not be cached. In other words the stream processor is optimized for streams, while the general purpose microprocessor handles random access by assuming a high locality of reference.

Modern DRAMs are based on several individual *memory banks*, and the memory address is separated into *rows* and *columns*, as shown in Figure 2.4. The three-dimensional organization of the memory device results in non-uniform access time. The memory banks are accessed independently, but the two-dimensional memory array in each bank is more complicated. To access a memory element, the corresponding row needs to be selected. Data in the selected row is then transferred to the *row buffer*. From the row buffer, data is accessed at high-speed and with uniform access time for any access pattern. When data from a different row is requested, the current row has to be closed, by pre-charging the bank, before the next row can be activated and transferred to the row buffer. Therefore, the bandwidth of burst-oriented memories highly depends on the access pattern. Accessing different banks or columns inside a row has a low latency, while accessing data in different rows has a high latency. When processing several memory streams, a centralized memory access scheduler can optimize the overall performance by reordering the memory access requests [39]. Latency for individual transfers may increase, but the goal is to minimize average latency and maximize overall throughput. Additional aspects on efficient memory access is discussed in Part I.

2.5 Hardware Design Flow

Hardware developers follow a design flow to systematically handle the design steps from an abstract specification or model to a functional hardware platform. A simplified design flow is illustrated in Figure 2.5 and involves the following steps:

Specification - The initial design decisions are outlined during the specification phase. This is usually in the form of block diagrams and hierarchical schematics, design constraints, and design requirements.

Virtual prototype - A virtual hardware prototype is constructed to verify the specification and to allow hardware and software co-design [40]. It is usually based on a high-level description using sequential languages,

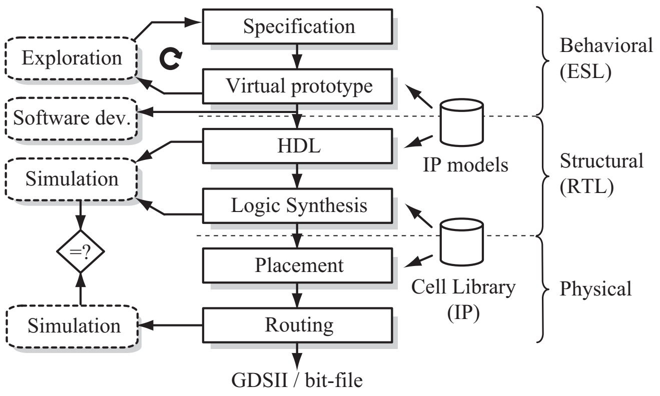


Figure 2.5: Design flow for system hardware development, starting from behavioral modeling, followed by hardware implementation, and finally physical placement and routing.

such as C/C++ or Matlab, or a high-level hardware description language such as SystemC [41]. More accurate descriptions yield better estimates of system performance, but results in increased simulation time [42]. High-level simulation models from vendor specific IP are also integrated if available.

Hardware description - Converting the high-level description into a hardware description can be done automatically or manually. Manual translation is still the most common approach, where the system is described at register-transfer level (RTL) using a hardware description language such as VHDL or Verilog.

Logic synthesis - The hardware description is compiled to gate level using logic synthesis. Gate primitives are provided in form of a standard cell library, describing the functional and physical properties of each cell. The output from logic synthesis is a netlist containing standard cells, macro-cells, and interconnects. The netlist is verified against the RTL description.

Physical placement and routing - The netlist generated after logic synthesis contains logic cells, where the physical properties are extracted from the standard cell library. A floorplan is specified, which describes the physical placement of IO-cells, IP macro-cells, and standard cells as well as power planning. The cells are then placed according to the floorplan and finally the design is routed.

The design flow is an iterative process, where changes to one step may require the designer to go back to a previous step in the flow. The virtual hardware prototype is one such example since it performs design exploration based on parameters given by the initial specification. Therefore, architectural problems discovered during virtual prototyping may require the specification to be revised. Furthermore, the virtual prototype is not only reference design for the structural RTL generation, but also a simulation and verification platform for software development. Changing the RTL implementation requires that the virtual platform is updated correspondingly, to avoid inconsistency between the embedded software and the final hardware.

2.5.1 **Architectural Exploration**

For decades, RTL has been a natural abstraction level for hardware designers, which is completely different from how software abstractions have developed over the same time. The increasing demands towards unified platforms for design exploration and early software development is pushing for higher abstractions in electronic design automation (EDA). To bridge the hardware-software gap, electronic system level (ESL) is a recent design methodology towards modeling and implementation of systems on higher abstraction levels [43]. The motivation is to raise the abstraction level, to increase productivity and to manage increasing design complexity. Hence, efficient architectural exploration and system performance analysis require *virtual prototyping* combined with models described at appropriate abstraction levels. Virtual prototyping is the emulation of an existing or non-existing hardware platform in order to analyze the system performance and estimate hardware requirements. Hence, models for virtual prototyping need to be abstracted and simplified to achieve high-performance exploration. An exploration environment and models for virtual prototyping is presented in Part III.

Modeling Abstraction Levels

RTL simulation is over-detailed when it comes to functional verification, which results in long run-times, limited coverage of system behavior, and poor observability about how the system executes. In contrast, the ESL design methodology promotes the use of appropriate abstractions to increase the knowledge and understanding of how a complex system operates [44]. The required level of detail is hence defined by the current modeling objectives. In addition, large system may require simulations described as a mix of different abstraction levels [45]. Common modeling abstractions are presented in Figure 2.6, which illustrates how simulation accuracy is traded for higher simulation speed, and can be divided into the following categories:

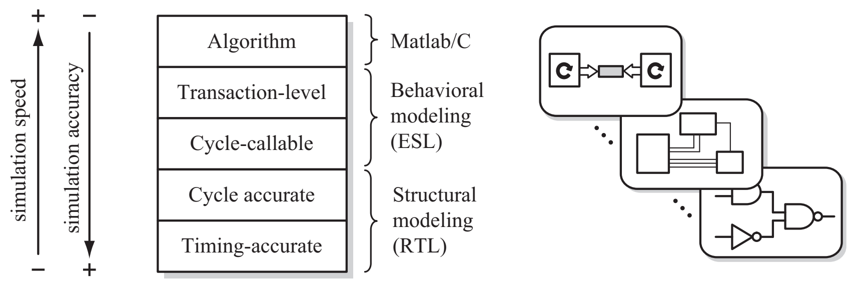


Figure 2.6: Abstraction levels is the ESL design space, ranging from abstract transaction-level modeling to pin and timing accurate modeling.

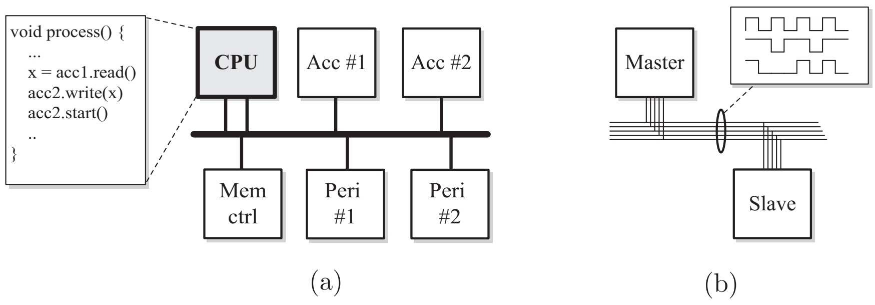
Algorithm - The algorithm design work is the high level implementation of a system, usually developed as Matlab or C code. At this design level, communication is modeled using shared variables and processing is based on sequential execution.

Transaction level - Transaction level modeling (TLM) is an abstraction level to separate functionality from communication [46]. Transactions, e.g. the communication primitives, are initiated using function calls, which results in a higher simulation speed than traditional pin-level modeling. TLM also enables simplified co-simulation between different systems since communication is handled by more abstract mechanisms. In this case, *adapters* are used to translate between different TLM and pin-accurate interfaces.

Cycle callable - A cycle callable (CC) model is also referred to as a *cycle approximate* or *loosely timed* model. A cycle callable model emulates time only on its interface, hence gaining speed by using untimed simulation internally. Simulations of embedded systems may use cycle callable models for accurate modeling of bus-transactions, but at the same time avoid the simulation overhead from the internal behavior inside each bus model.

Cycle accurate - A cycle accurate (CA) model uses the same modeling accuracy at clock cycle level as a conventional RTL implementation. Cycle accurate models are usually also pin-accurate.

Timing accurate - A timing accurate simulation is required to model delays in logic and wires. Timing can be annotated to the netlist produced after logic synthesis or after place and route, but is sometimes required



```
void process() {
  ...
  x = acc1.read()
  acc2.write(x)
  acc2.start()
  ..
}
```

Figure 2.7: (a) The programmer's view of an embedded system, where the hardware is register-true, but not clock cycle accurate. (b) Hardware architect's view of an embedded system. The design is cycle- and pin accurate.

during the RTL design phase to simulate for example the complex timing constraints in memory models.

During the implementation phase, models sometimes need to be *refined* by providing more details to an initial high-level description [42]. For example, at algorithmic level there is no information about how a computational block receives data. This is the system view for the *algorithmic designer*. When the model is converted into a hardware description, the communication is initially modeled using transactions on a bus, which is the *programmer's view* of a system [47]. Software development only requires the register map and memory addressing to be accurate, but depends less on the actual hardware timing. In contrast, the *hardware architect's view* requires clock cycle accurate simulation to verify handshaking protocols and data synchronization. The software and hardware views are illustrated in Figure 2.7(a-b), respectively.

2.5.2 Virtual Platforms

A virtual platform is a software simulation environment to emulate a complete hardware system [48]. It is a complement to functional verification on programmable logic devices, i.e. FPGAs, but introduces additional features that can not be addressed using hardware prototyping systems. Since a virtual platform is available *before* the actual hardware has been developed, it enables early software development and the possibility to locate system bottlenecks and other design issues in the initial design process. Hence, exploration of the hardware architecture using a virtual platform has several advantages over hardware-based prototyping, including:

- **Precise control** - In contrast to hardware-based verification, software simulation enables the possibility to freeze the execution in all simulation modules simultaneously at clock cycle resolution.
- **Full visibility** - All parts of virtual platform can be completely visible and analyzed using unlimited traces, while hardware-based verification is limited to hardware probing and standardized debug interfaces.
- **Deterministic execution** - A software simulation is completely deterministic, where all errors and malfunctions are reproducible. A reproducible simulation is a requirement for efficient system development and debugging.

However, controllability and visibility come with the price of longer execution times. Hence, hardware-based prototyping using FPGA platforms is still an important part of ASIC verification, but will be introduced later in the design flow.

2.5.3 Instruction Set Simulators

A majority of digital systems include one or more embedded processor, which require additional simulation models for the virtual platform. A processor simulation model is referred to as an *instruction-set simulator* (ISS), and is a functional model for a specific processor architecture. The ISS executes binary instructions in the same way as the target processor, hence all transactions on the bus correspond to the actual hardware implementation. An ISS based virtual platform normally reaches a simulation performance in the range of 1-10 MIPS, and academic work targeting automatic ISS generation is presented in [49]. Faster simulation is possible using a compiled ISS, where the target processor instructions are converted into native processor instructions [50]. In contrast to an interpreted instruction-set simulator, a compiled ISS can typically reach a simulation performance of 10-100 MIPS.

Given this background, the virtual platform and ISS concepts are further discussed in Part III, which proposes a design environment and a set of scalable simulation models to evaluate reconfigurable architectures.

Chapter 3

Digital Holographic Imaging

In 1947, the Hungarian scientist Dennis Gabor developed a method and theory to photographically create a three-dimensional recording of a scene, commonly known as holography [51]. For his work on holographic methods, Dennis Gabor was awarded the Nobel Prize in physics 1971. A holographic setup is shown in Figure 3.1, and is based on a coherent light source. The *interference* pattern between light from a reference wave and reflected light from an object illuminated with the same light source is captured on a photographic film (holographic plate). Interference between two wave fronts cancels or amplifies the light in each point on the holographic film. This is called *constructive* and *destructive* interference, respectively.

A recorded hologram has certain properties that distinguish it from a conventional photograph. In a normal camera, the intensity (amplitude) of the light is captured and the developed photography is directly visible. The photographic film in a holographic setup captures the interference, or phase difference, between two waves [52]. Hence, both amplitude and phase information are stored in the hologram. By illuminating the developed photographic film with the same reference light as used during the recording phase, the original image is *reconstructed* and appears three-dimensional.

The use of photographic film in conventional holography makes it a time-consuming, expensive, and inflexible process. In digital holography, the photographic film is replaced by a high-resolution digital image sensor to capture the interference pattern. The interference pattern is recorded in a similar way as in conventional holography, but the reconstruction process is completely different. Reconstruction requires the use of a signal processing algorithm to

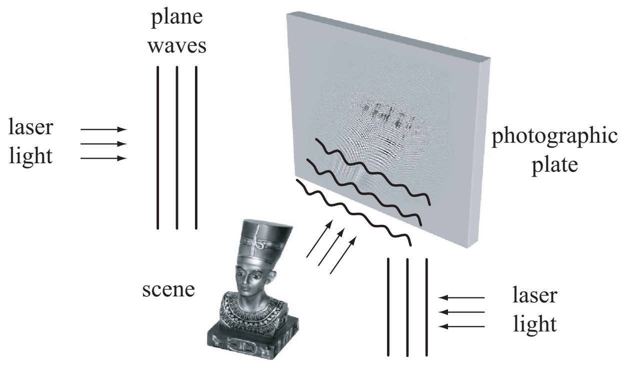


Figure 3.1: The reflected light from the scene and the reference light creates an interference pattern on a photographic film.

transform the digital recordings into visible images [1,53]. The advantage over conventional holography is that it only takes a fraction of a second to capture and digitally store the image, instead of developing a photographic film. The downside is that current sensor resolution is in the range 300-500 pixels/mm, whereas the photographic film contains 3000 lines/mm.

A holographic setup with a digital sensor is illustrated in Figure 3.2. The light source is a 633 nm Helium-Neon laser, which is divided into separate reference and object lights using a beam splitter. The object light passes through the object and creates an interference pattern with the reference light on the digital image sensor. The interference pattern (hologram) is used to digitally reconstruct the original object image.

3.1 Microscopy using Digital Holography

Digital holography has a number of interesting properties that has shown useful in the field of microscopy. The study of transparent specimens, such as organic cells, conventionally requires either the use of non-quantitative measurement techniques, or the use of contrast-increasing staining methods that could be harmful for the cells. This issue can be addressed with a microscope based on digital holography, introducing a non-destructive and quantitative measurement technique to study living cells over time [54].

Most biological specimens, such as cells and organisms, are almost completely colorless and transparent [55]. In conventional optical microscopy, this results in low contrast images of objects that are nearly invisible to the human eye, as illustrated in Figure 3.3(a). The reason is that the human eye only mea-

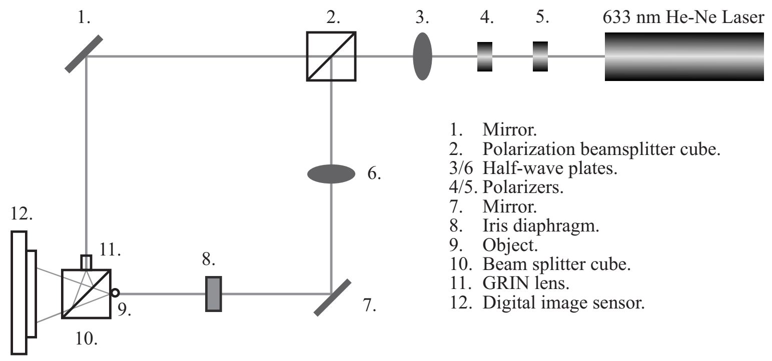


Figure 3.2: The experimental holographic setup. The reference, object, and interference pattern are captured on a high-resolution digital image sensor, instead of using a photographic film as in conventional holography. The images are captured by blocking the reference or object beam.

sures the light amplitude (energy), but the phase shift still carries important information. In the beginning of the 1930s, the Dutch physicist Frits Zernike invented a technique to enhance the image contrast in microscopy, known as phase contrast, for which he received the Nobel Prize in physics 1953. The invention is a technique to convert the phase shift in a transparent specimen into amplitude changes, in other words making invisible images visible [56]. However, the generated image does not provide the true phase of the light, only improved contrast as illustrated in Figure 3.3(b).

An alternative to phase contrast microscopy is to instead increase the cells contrast. A common approach is to stain the cells using various dyes, such as Trypan blue or Eosin. However, staining is not only a cumbersome and time-consuming procedure, but could also be harmful to the cells. Therefore, experiments to study growth, viability, and other cell characteristics over time require multiple sets of cell samples. This prevents the study of individual cells over time, and is instead based on the assumption that each cell sample has similar growth-rate and properties. In addition, the staining itself can generate artifacts that affect the result. In Figure 3.4(a), staining with Trypan blue has been used to identify dead cells.

In contrast, digital holography provides a non-invasive method and does not require any special cell preparation techniques [57]. The following sections discuss the main advantages of digital holography in the field of microscopy, namely the true phase-shift, the software autofocus, and the possibility to generate three-dimensional images.

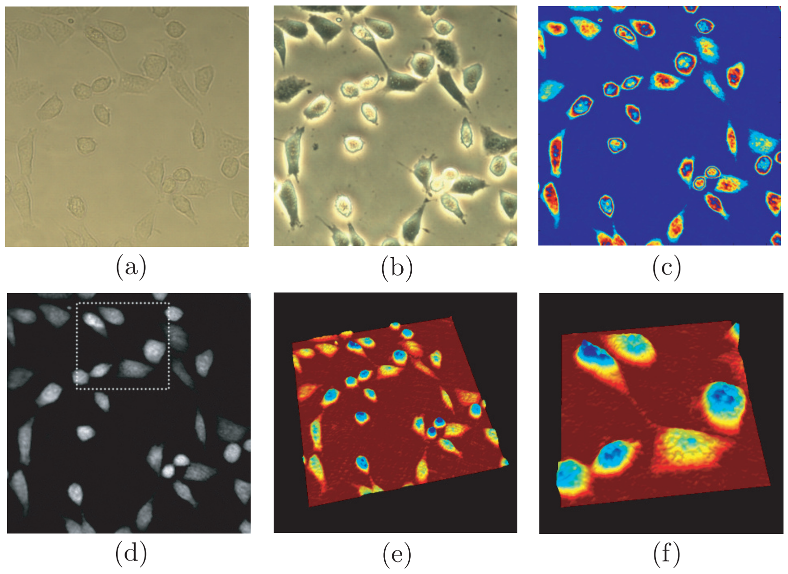


Figure 3.3: Images of transparent mouse fibroblast cells. (a) Conventional microscopy. (b) Phase-contrast microscopy. (c) Reconstructed phase information from the holographic microscope. (d) Unwrapped and background processed holographic image with true phase. (e) 3D view from the holographic microscope. (f) Close-up from image (d).

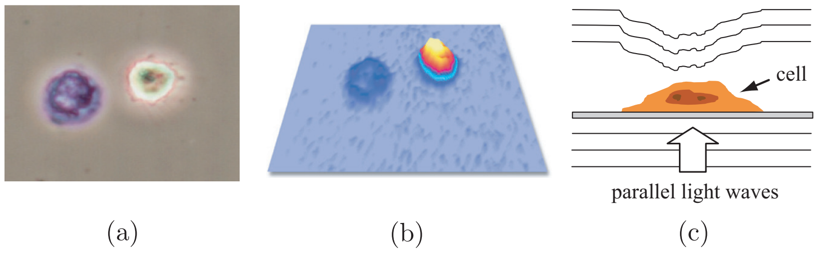


Figure 3.4: To the left a dead cell and to the right a living cell. (a) Using Trypan blue staining in a phase contrast microscope to identify dead cells. Dead cells absorb the dye from the surrounding fluid. (b) Using phase information in non-invasive digital holography to identify dead cells. (c) Phase-shift when parallel light pass through a cell containing regions with different refractive index.

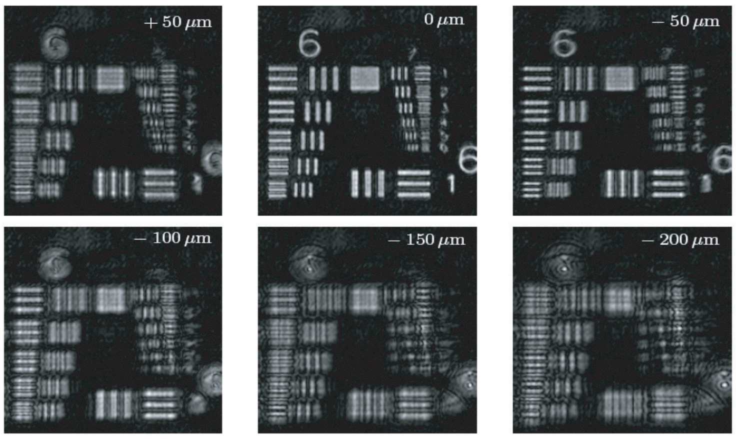


Figure 3.5: Reconstructed images of an USAF resolution test chart evaluated at the distances $-200\,\mu$m to $50\,\mu$m in step of $50\,\mu$m from the original position.

3.1.1 Phase Information

The true phase-shift is of great importance for quantitative analysis. Since the phase reveals changes to the light path, it can be used to determine either the integral refractive index of a material with known thickness, or the thickness in a known medium. The phase shift is illustrated in Figure 3.4(c). In digital holography, the obtained interference pattern contains both amplitude and phase information. The amplitude is what can be seen in a conventional microscope, while phase information can be used to generate high-contrast images and to enable quantitative analysis. The analysis can yield information about refractive index, object thickness, object volume, or volume distribution. Figure 3.4(a-b) show images obtained with a phase contrast microscope and a holographic microscope, respectively.

3.1.2 Software Autofocus

In conventional microscopes, the focus position is manually controlled by a mechanical lens system. The location of the lens determines the focus position, and the characteristics of the lens determine the *depth of field*. The image is sharp only at a certain distance, limited by the properties of the lens, and the objects that are currently in focus are defined to be in the *focal plane*. Naturally, images captured in conventional microscopy only show a single focal plane, which may cause parts of the image to be out of focus.

In digital holography, individual focal planes are obtained from a single recording by changing a software parameter inside the reconstruction algorithm. Hence, no mechanical movements are required to change the focus position. As an example, Figure 3.5 shows the reconstruction of a resolution test chart in different focal planes, in this case either *in focus* or *out of focus*. Microscope recordings can thus be stored and transmitted as *holographic images*, which simplifies remote analysis with the ability to interactively change focal plane and magnification.

3.1.3 Three-Dimensional Visualization

Phase and amplitude information enable visualization of captured images in three dimensions. The phase information reveals the optical thickness of an object, and can be directly used to visualize the cell topology. To avoid visual artifacts, it is assumed that cells grow on a flat surface, and that there is minimal overlap between adjoining cells. Examples of topology images are shown in Figure 3.3(e-f).

Extraction of accurate three-dimensional information requires more complex image processing, and involves the analysis of a stack of reconstructed amplitude images obtained at different focal planes [58]. Each reconstructed image is processed to identify and extract the region that is currently in focus. The resulting stack of two-dimensional shapes is joined into a three-dimensional object. The same technique can be used to merge different focal planes to produce an image where all objects are in focus [59].

3.2 Processing of Holographic Images

This section presents the *numerical reconstruction*, *phase unwrapping*, and *image analysis* required to enable non-invasive biomedical imaging. Numerical reconstruction is the process to convert holographic recordings into visible images, and is the main focus of this work. Reconstruction is followed by phase unwrapping, which is a method to extract true phase information from images that contain natural discontinuities. Finally, image analysis is briefly discussed to illustrate the potential of the holographic technology, especially to study cell morphology [55].

3.2.1 Numerical Reconstruction

Three different images are required to reconstruct one visible image. In consecutive exposures, the hologram ψ_h, object ψ_o, and reference light ψ_r are captured on the digital image sensor. First, the hologram is captured by recording the interference light between the object and reference source. Then, by block-

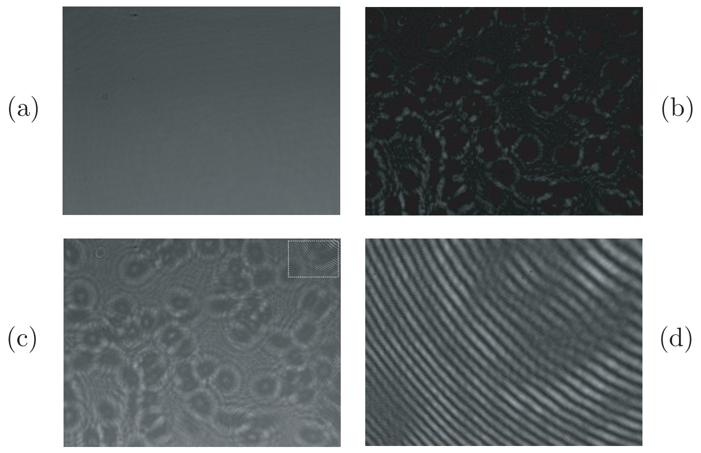


Figure 3.6: (a) Reference image ψ_{r}. (b) Object image ψ_{o}. (c) Hologram or interference image ψ_{h}. (d) Close-up of interference fringes in a holographic image.

ing either the object or the reference beam, the reference and object light are captured respectively. An example of captured holographic images is shown in Figure 3.6 together with a close-up on the recorded interference fringes in the hologram.

The images captured on the digital image sensor need to be processed using a signal processing algorithm, which reconstructs the original and visible image. The reference and object light is first subtracted from the hologram, and the remaining component is referred to as the *modified interference pattern* ψ

$$\psi(\boldsymbol{\rho}) = \psi_{\mathrm{h}}(\boldsymbol{\rho}) - \psi_{\mathrm{o}}(\boldsymbol{\rho}) - \psi_{\mathrm{r}}(\boldsymbol{\rho}), \tag{3.1}$$

where $\boldsymbol{\rho}$ represents the (x, y) position on the sensor. A visible image of an object (the object plane) is reconstructed from $\psi(\boldsymbol{\rho})$, which is the object field captured by the digital sensor (the sensor plane), as illustrated in Figure 3.7. The reconstruction algorithm, or inversion algorithm, can retrofocus the light captured on the sensor to an arbitrary object plane, which makes it possible to change focus position in the object. This is equivalent to manually moving the object up and down in a conventional microscope.

An image is reconstructed by computing the Rayleigh-Sommerfeld diffraction integral as

$$\Psi(\boldsymbol{\rho}') = \int_{\mathrm{sensor}} \psi(\boldsymbol{\rho}) \mathrm{e}^{-\mathrm{i}k|\boldsymbol{r}-\boldsymbol{r}_{\mathrm{r}}|} \mathrm{e}^{\mathrm{i}k|\boldsymbol{r}-\boldsymbol{r}'|} \, \mathrm{dS}_{\rho}, \tag{3.2}$$

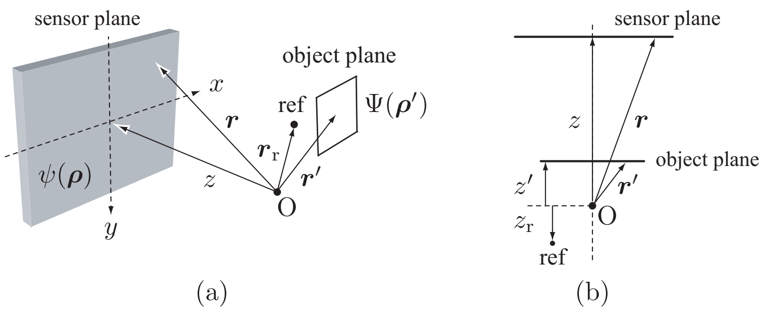


Figure 3.7: (a) Definition of vectors from origin of coordinates to the sensor surface, reference point, and object plane. (b) Sensor and object planes. Modifying the z-position of the object plane changes focus position.

where $k = 2\pi/\lambda$ is the angular wave number for a wavelength λ, and the reference field is assumed to be spherical for simplicity [60]. By specifying a point of origin, $\boldsymbol{r}'$ represents the vector to any point in the object plane and $\boldsymbol{r}$ represents the vector to any point in the sensor plan, as illustrated in Figure 3.7. The three-dimensional vectors can be divided into a z component and an orthogonal two-dimensional vector $\boldsymbol{\rho}$ representing the x and y positions as $\boldsymbol{r} = \boldsymbol{\rho} + z\hat{\boldsymbol{z}}$. The distance z' specifies the location of the image plane to be reconstructed, whereas z is the distance to the sensor. The integral in (3.2) can be expressed as a convolution

$$\Psi(\boldsymbol{\rho}') = \psi_1 * G, \tag{3.3}$$

where

$$\psi_1(\boldsymbol{\rho}) = \psi(\boldsymbol{\rho})\mathrm{e}^{-\mathrm{i}k\sqrt{|z-z_\mathrm{r}|^2+|\boldsymbol{\rho}-\boldsymbol{\rho}_\mathrm{r}|^2}}$$

and

$$G(\boldsymbol{\rho}) = \mathrm{e}^{\mathrm{i}k\sqrt{|z-z'|^2+|\boldsymbol{\rho}|^2}}.$$

The discrete version of the integral with an equidistant grid, equal to the sensor pixel size (Δx, Δy), generates a discrete convolution of (3.3) that can be evaluated with the FFT [61] as

$$\Psi(\boldsymbol{\rho}') = \mathcal{F}^{-1}(\mathcal{F}(\psi_1) \cdot \mathcal{F}(G)). \tag{3.4}$$

The size of the two-dimensional FFT needs to be at least the sum of the sensor size and the object size in each dimension. Higher resolution is achieved by shifting the coordinates a fraction of a pixel size and combining the partial

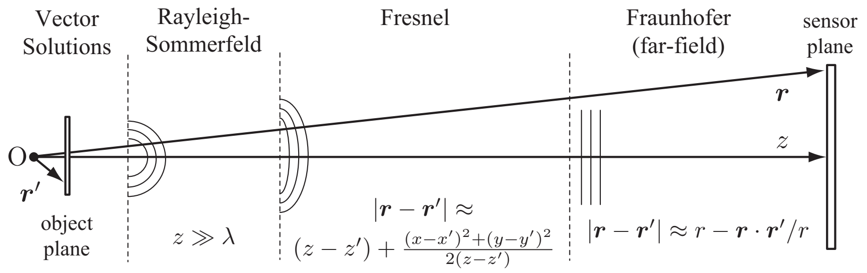

Vector Solutions
Rayleigh-Sommerfeld
Fresnel
Fraunhofer (far-field)
sensor plane
O
$\boldsymbol{r}'$
object plane
$\boldsymbol{r}$
z
$z \gg \lambda$
$|\boldsymbol{r}-\boldsymbol{r}'| \approx (z-z') + \frac{(x-x')^2+(y-y')^2}{2(z-z')}$
$|\boldsymbol{r}-\boldsymbol{r}'| \approx r - \boldsymbol{r}\cdot\boldsymbol{r}'/r$

Figure 3.8: Approximations to reduce the computational requirements.

results. Hence, an image with higher resolution requires several processing iterations, which increases the reconstruction time with a factor N^2 for an interpolation of N times in each dimension. The image reconstruction requires three FFT calculations (3.4), while each additional interpolation only requires two FFT calculations, since only $G(\boldsymbol{\rho})$ changes.

Several approximations to simplify the reconstruction process exist, based on assumptions about distance and angle between the image plane and the sensor plane. Figure 3.8 illustrates how different approximations can be applied when the distance between the object and sensor plane increases. The Fresnel approximation simplifies the relation to an arbitrary point in the image plane to an arbitrary point in the sensor plane [62]. In the *far-field*, the Fraunhofer approximation is applicable, which further simplifies the distance equation [63]. The approximations are further discussed in Section 3.3.2, and used in Part I to reduce the computational requirements of an application specific hardware accelerator.

3.2.2 **Image Processing**

The output data from the reconstruction algorithm is complex valued, represented as magnitude and phase. However, this means that the phase is constrained to an interval $(-\pi, \pi]$, which is referred to as the *wrapped phase*. A wrapped phase image contains 2π discontinuities, and requires further processing to extract the *unwrapped* phase image. Figure 3.3(c) shows a wrapped phase image where changes in the cell thickness have resulted in discontinuities. Figure 3.3(d) shows the same image after phase unwrapping.

Several algorithms to compute phase unwrapping exist, with various complexity in terms of processing speed and accuracy [64]. Phase-unwrapping algorithms are mainly divided into two groups, *path-following* and *minimum norm*. Path-following algorithms, such as Goldstein's and Flynn's algorithm [65], identifies paths on which to integrate. The algorithms are less sensitive to noise but the run-time is not deterministic. Minimum norm algorithms, based on the discrete cosine transform (DCT) or discrete Fourier transform (DFT), are faster but more sensitive to noise. Algorithm selection is a trade-off between run-time and accuracy, where faster algorithms are useful for real-time visualization, while more accurate algorithms are suitable for quantitative analysis.

Image reconstruction followed by phase unwrapping produces amplitude and phase images from the obtained hologram. Hence, the images are ready to be further processed for more advanced analysis. The imaging operations commonly desired to be able to study, quantify, classify, and track cells and other biological specimens are:

Cell visualization - Providing intuitive ways of visualizing cell information is desirable, and has previously been discussed in Section 3.1.3. Three-dimensional imaging provides depth information, and the use of two-dimensional image superposition of amplitude and phase may illustrate and highlight different cell properties.

Cell quantification - The possibility to automatically measure cells and cell samples is favorable over manual time-consuming and error-prone methods. Cell counting and viability studies are today a highly manual procedure, but which can be automated using holographic technology. Other desirable measurements are cell area, cell coverage, cell/total volume, and cell density.

Cell classification - Classification is a statistical procedure of placing individual cells into groups, based on quantitative information and feature extraction. The cell is matched against a database, containing training sets of previously classified cells, to find the most accurate classification. In biomedical imaging, this can be used to analyse the distribution of different cell types in a sample.

Cell morphology - Morphology refers to the change in a cells appearance, including properties such as shape and structure. Morphology can be used to study the development of cells, and to determine the cells condition and internal health [55]. Due to the non-invasive properties of a holographic microscope, it has shown to be a suitable candidate for *time-lapse studies*, i.e. the study of cells over time.

Cell tracking - Morphology requires each cell to be studied individually, hence the location of each cell must be well-defined. However, it is assumed that cells will not only change their physical properties over time, but also their individual location. Therefore, tracking is required to study cells over time. In biomedical imaging, this information can also be used to study cell trajectories.

3.3 Acceleration of Image Reconstruction

As previously discussed, the reconstruction process is computationally demanding and the use of a standard computer is not a feasible solution to perform image reconstruction in *real-time*. In a microscope, a short visual response time is required to immediately reflect changes to the object position, illumination, or focus point. It is also useful to study real-time events, for example when studying a liquid flow. The human perception of real-time requires an update rate in a range of 20 to 60 frames per second (fps), depending on application and situation. However, for a biological application a frame rate around $f_{\text{rate}} = 25\,\text{fps}$ would be more than sufficient.

The sensor *frame rate* is one of the factors that affect the required system performance. Other parameters are the sensor *resolution* and the image *reconstruction time*. These properties are discussed in the following sections to specify the requirements for the hardware accelerator.

3.3.1 Digital Image Sensors

Digital image sensors are available in two technologies, either as charge-coupled devices (CCD) or as complementary metal oxide semiconductors (CMOS). The CCD sensor has superior image quality, but the CMOS sensor is more robust and reliable. The CCD is based on a technique to move charges from the sensor array and then convert the charge into voltage. In contrast, CMOS sensors directly convert the charge to voltage directly at each pixel, and also integrate control circuitry inside the device. A comparison between image sensor technologies is presented in [66].

The term *resolution* is often used to indicate the number of pixels on the image sensor array, while the term *pixel pitch* refers to the size of each active element in the array. Over the past few years the resolution of digital sensors has continuously increased, whereas the pixel pitch consequently has decreased to comply with standardized optical formats.

The frame rate for digital image sensors highly depends on the sensor resolution. In a digital video camera the real-time constraints require a high frame rate for the images to appear as motion video. However, sensors with high resolution are often not able to produce real-time video. This limitation is addressed by either lowering the resolution by reading out less pixels from the sensor array, or by using *binning* functionality to cluster neighboring pixels.

In digital holography, both high resolution and high frame rate are required.

Table 3.1: Algorithmic complexity (two-dimensional FFTs) and required throughtput for a holographic system with a resolution of 2048×2048 pixels and a frame rate of 25 fps.

Properties	Rayleigh-Sommerfeld	Fresnel/Fraunhofer approximation
Algorithmic complexity*	$2N^2 + 1$	1
Software run-time** (s)	≈ 82	≈ 1.1
Required throughput (samples/s)	15.3 G	210 M

* Number of 2D-FFTs required. The original algorithm requires refinement with $N = 6$.

** based on FFTW 2D-FFT [67] on a Pentium-4 2.0 GHz.

For the current application, a 1.3 Mpixel CMOS sensor with a video frame rate of 15 fps has been selected. This is a trade-off to be able to produce high quality images in close to video speed. However, it is likely that future image sensors would enable combinations of higher resolution with higher frame rate.

3.3.2 Algorithmic Complexity

Table 3.1 compares the Rayleigh-Sommerfeld diffraction integral and the approximations in terms of complexity, run-time and required throughput for real-time performance. The two-dimensional FFT is assumed to dominate the reconstruction time, hence the comparison has been limited to a part of the complete reconstruction process. For complexity analysis, the Rayleigh-Sommerfeld diffraction integral is evaluated using 3 FFT calculations, with additional 2 FFTs required for each refinement step. In contrast, only a single FFT is required when applying the Fresnel and Fraunhofer approximations, which is shown in Part I.

The two-dimensional FFT is evaluated on arrays that are larger than the actual sensor size. The FFT size must be at least the sum of the sensor size and the object size in each dimension. The digital image sensor used in this work has a resolution of 1312×1032 pixels. The object size can be at most the size of one quadrant on the sensor array, extending the processing dimensions to 1968×1548 pixels. However, the size of the FFT must be selected and zero-padded to a size of 2^n, where n is an integer number. The closest option is $n = 11$, which corresponds to an FFT of size 2048×2048 points. This defines $N_{\mathrm{FFT}} = 2048$.

The frame rate in this application is a *soft contraint*. Based on the current sensor device and the above definition of real-time video, the desired frame rate is in the range of 15-25 fps. However, for the algorithmic complexity analysis

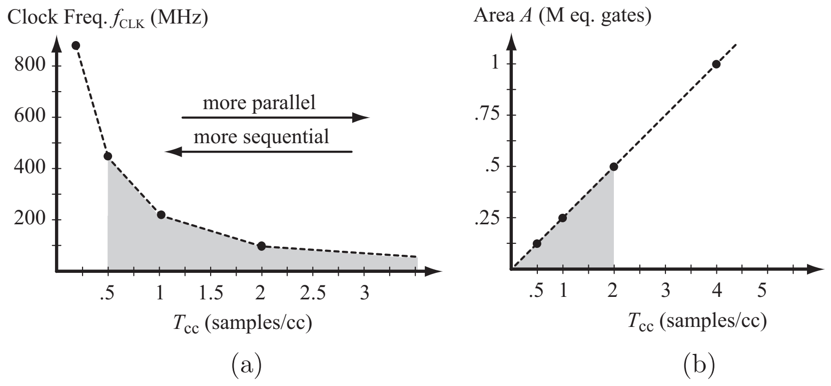


Figure 3.9: Feasible design space points (grey) that comply with the system requirements. Area is estimated based on similar designs.

in Table 3.1, a real-time video frame rate of 25 fps is assumed. The table shows the run-time in software and the required throughput to meet the real-time constraints.

3.3.3 Hardware Mapping

For hardware mapping, the approximations are applied and the complexity reduction is further explained in Part I. To estimate the hardware requirements, the throughput of the two-dimensional FFT is used. However, due to the large transform size, implementation of the two-dimensional FFT is separated into one-dimensional FFTs, first applied over rows and then over columns [68]. Hence, 2×2048 transforms of size 2048 must be computed for each frame, resulting in a required throughput of

$$2N_{\mathrm{FFT}}^2 \times f_{\mathrm{rate}} \quad \approx \quad 210\,\mathrm{Msample/s},$$

where FFT size N_{FFT} and frame rate f_{rate} are given by the specification. The choice of hardware architecture and the selected system clock frequency control the throughput, which results in the following relations

$$\begin{cases} f_{\mathrm{clk}} \times T_{\mathrm{cc}} & = & 2N_{\mathrm{FFT}}^2 \times f_{\mathrm{rate}} \\ A & \propto & T_{\mathrm{cc}}, \end{cases}$$

where f_{clk} is the clock frequency, and T_{cc} is the scaled throughput in *samples per clock cycle*. It is assumed that the required area (complexity) A is propositional to the throughput. An attempt to double the throughput for the

FFT algorithm, for given operational conditions such as frequency and supply voltage, would result in twice the amount of computational resources.

The equations enable design space exploration based on the variable parameters. Samples per clock cycle relate to architectural selection, while frequency corresponds to operational conditions. Figure 3.9 shows points in the design space that comply with the specification. Both graphs use the horizontal axis to represent the architectural scaling, ranging from time-multiplexed (folded) to more parallel architectures. Figure 3.9(a) shows the clock frequency required for a certain architecture. Since the clock frequency has an upper feasible limit, it puts a lower bound on T_{cc}. Figure 3.9(b) shows the area requirement, which is assumed to grow linearly with the architectural complexity, and puts an upper bound on T_{cc}. Hence, T_{cc} is constrained by system clock frequency and the hardware area requirement. The graphs can be used to select a suitable and feasible architecture, where $T_{cc} = 1$ seems to be reasonable candidate. Based on this estimated analysis, the architectural decisions are further discussed in Part I.

Part I

A Hardware Acceleration Platform for Digital Holographic Imaging

Abstract

A hardware acceleration platform for image reconstruction in digital holographic imaging is proposed. The hardware accelerator executes a computationally demanding reconstruction algorithm which transforms an interference pattern captured on a digital image sensor into visible images. Focus in this work is to maximize computational efficiency, and to minimize the external memory transfer overhead, as well as required internal buffering. We present an efficient processing datapath with a fast transpose unit and an interleaved memory storage scheme. The proposed architecture results in a speedup with a factor 3 compared with the traditional column/row approach for calculating the two-dimensional FFT. Memory sharing between the computational units reduces the on-chip memory requirements with over 50%. The custom hardware accelerator, extended with a microprocessor and a memory controller, has been implemented on a custom designed FPGA platform and integrated in a holographic microscope to reconstruct images. The proposed architecture targeting a 0.13 μm CMOS standard cell library achieves real-time image reconstruction with over 30 frames per second.

Based on: T. Lenart and M. Gustafsson and V. Öwall, “A Hardware Acceleration Platform for Digital Holographic Imaging,” *Springer Journal of Signal Processing Systems*, DOI: 10.1007/s11265-008-0161-2, Jan 2008.

1 Introduction

In digital holography, the photographic film used in conventional holography is replaced by a digital image sensor to obtain a digital hologram, as shown in Figure 1(a). A reconstruction algorithm processes the images captured by the digital sensor to create a visible image, which in this work is implemented using a hardware acceleration platform, as illustrated in Figure 1(b).

A hardware accelerator for image reconstruction in digital holographic imaging is presented. The hardware accelerator, referred to as XSTREAM, executes a computationally demanding reconstruction algorithm based on a 2048×2048 point two-dimensional FFT, which requires a substantial amount of signal processing. It will be shown that a general-purpose computer is not capable of meeting the real-time constraints, hence a custom solution is presented.

The following section presents how to reduce the complexity of the reconstruction algorithm, as presented in Chapter 3.2.1, and how it can be adapted for hardware implementation. System requirements and hardware estimates are discussed in Section 2, and Section 3 presents the proposed hardware architecture and the internal functional units. In Section 4, the hardware accelerator is integrated into an embedded system, which is followed by simulation results and proposed optimizations in Section 5. Results and comparisons are presented in Section 6 and finally conclusions are drawn in Section 7.

1.1 Reduced Complexity Image Reconstruction

As discussed in Chapter 3.2.1, the Rayleigh-Sommerfeld diffraction integral is compute-intensive since it requires three two-dimensional FFTs to be evaluated. However, by observing that $|\boldsymbol{r}'| \ll |\boldsymbol{r}|$, the Fraunhofer or *far-field* approximation [63] simplifies the relation between the sensor plane and object plane as

$$|\boldsymbol{r} - \boldsymbol{r}'| \approx r - \boldsymbol{r} \cdot \boldsymbol{r}'/r,$$

where $r = |\boldsymbol{r}|$. This changes the diffraction integral in (3.2) to

$$\begin{aligned}
\Psi(\boldsymbol{\rho}') &= \int_{\text{sensor}} \psi(\boldsymbol{\rho}) \mathrm{e}^{-\mathrm{i}k|\boldsymbol{r}-\boldsymbol{r}_\mathrm{r}|} \mathrm{e}^{\mathrm{i}k|\boldsymbol{r}-\boldsymbol{r}'|} \,\mathrm{dS}_{\boldsymbol{\rho}} \\
&\approx \int_{\text{sensor}} \psi(\boldsymbol{\rho}) \mathrm{e}^{-\mathrm{i}k(|\boldsymbol{r}-\boldsymbol{r}_\mathrm{r}|-r)} \mathrm{e}^{-\mathrm{i}k\boldsymbol{r}\cdot\boldsymbol{r}'/r} \,\mathrm{dS}_{\boldsymbol{\rho}} \\
&= \int_{\text{sensor}} \psi(\boldsymbol{\rho}) \mathrm{e}^{-\mathrm{i}k(|\boldsymbol{r}-\boldsymbol{r}_\mathrm{r}|-r)} \mathrm{e}^{-\mathrm{i}kzz'/r} \mathrm{e}^{-\mathrm{i}k\boldsymbol{\rho}\cdot\boldsymbol{\rho}'/r} \,\mathrm{dS}_{\boldsymbol{\rho}} \\
&= \int_{\text{sensor}} \psi_2(\boldsymbol{\rho}) \mathrm{e}^{-\mathrm{i}k\boldsymbol{\rho}\cdot\boldsymbol{\rho}'/r} \,\mathrm{dS}_{\boldsymbol{\rho}}. \qquad (1)
\end{aligned}$$

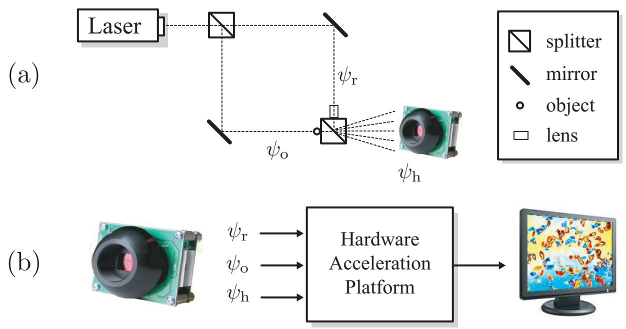


Figure 1: (a) Simplified holographic setup capturing the light from a laser source on a digital sensor. By blocking the reference or the object beam, totally three images are captured representing the reference, the object, and the hologram. (b) The three captured images are processed by the presented hardware accelerator to reconstruct visible images on a screen.

Assuming $r \approx z$, which is a constant, the integral in (1) is the definition of the two-dimensional FFT, where

$$\psi_2(\boldsymbol{\rho}) = \psi(\boldsymbol{\rho})\mathrm{e}^{-\mathrm{i}k(|\boldsymbol{r}-\boldsymbol{r}_\mathrm{r}|-r)}\mathrm{e}^{-\mathrm{i}kzz'/r}. \tag{2}$$

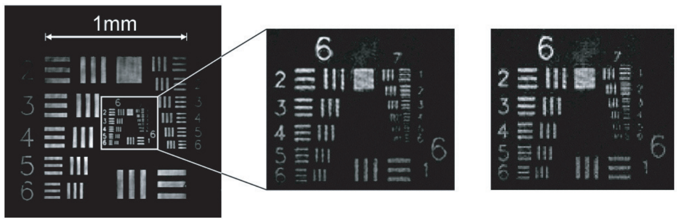
The first exponential term in (2) can be removed since it only affects the object location in the reconstructed image. Instead, the coordinates of the object location can be modified after reconstruction. The image reconstruction algorithm with reduced complexity requires only a single FFT as

$$\Psi(\boldsymbol{\rho}') \approx \mathcal{F}(\psi(\boldsymbol{\rho})\mathrm{e}^{-\mathrm{i}kzz'/r}), \tag{3}$$

where $\alpha(\boldsymbol{\rho}) = kzz'/r$ is referred to as the *phase factor*.

1.2 Algorithm Selection

The visible quality difference between using the convolution algorithm and applying the far-field approximation is shown in Figure 2(b-c). In this example, the convolution algorithm is interpolated $N = 6$ times in each dimension, which requires a total of $2(N^2) + 1 = 73$ FFTs to be evaluated, while the algorithm based on the far-field approximation requires only a single FFT. For the current application, the visual difference between the images is negligible. Hence, the algorithm with reduced complexity has been chosen due to the substantially lower computational effort. The possibility to evaluate convolutions is still



(a) (b) (c)

Figure 2: (a) Reconstructed images from the reference, object, and hologram captured by the digital sensor. (b) Close-up of the resulting image using the convolution reconstruction algorithm. (c) Close-up of the resulting image using the reduced complexity reconstruction algorithm (far-field approximation). The difference in visual quality is negligible.

supported by our proposed architecture, and a special bidirectional pipeline FFT is presented in Section 3.3 for this purpose.

2 System Requirements

As discussed in Chapter 3.3, the performance of the two-dimensional FFT will limit the number of frames that can be reconstructed per second. The selected digital image sensor has a resolution of 1312 × 1032 pixels, with a pixel pitch of $6 \times 6\,\mu$m, and a precision of 8 bits per pixel. The hardware accelerator is designed to satisfy the following specification:

$$\begin{cases} f_{\text{rate}} &= 25 \text{ fps} \\ N_{\text{FFT}} &= 2048 \text{ points} \\ T_{\text{cc}} &= 1 \text{ sample/cc} \end{cases}$$

Captured images and intermediate results during computation must be stored in external memory. Each captured image requiring $1312 \times 1032 \times 8 \approx 1.3$ Mbytes of storage space, and a table of the same size is required to store pre-computed phase factors. Two intermediate memory areas, in 32-bit precision and with the same size as the two-dimensional FFT, are needed during processing. Hence, an estimate of the required amount of external memory is $4 \cdot 1.3 + 2(4 \cdot N_{\text{FFT}}^2) \approx 37.2$ Mbytes. Due to the large amount of external memory, the use of static RAM (SRAM) is not a feasible solution, and instead two parallel 512-Mbit Synchronous Dynamic RAM (SDRAM) has been used [69].

An estimate of the required processing speed can be found by analyzing the two-dimensional FFT, which is the most compute intensive part of the

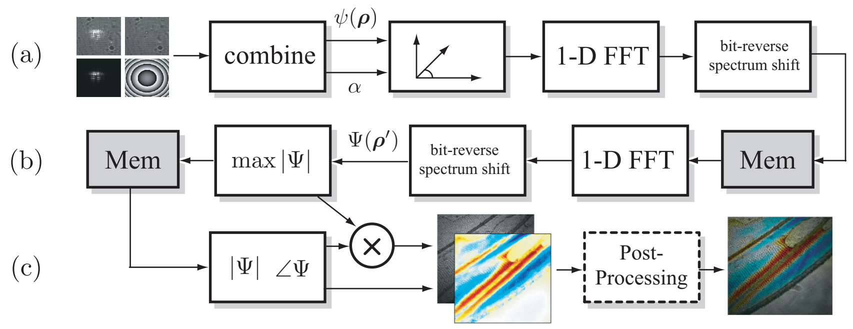

(a)
combine
$\psi(\boldsymbol{\rho})$
α
1-D FFT
bit-reverse spectrum shift
(b)
Mem
$\max|\Psi|$
$\Psi(\boldsymbol{\rho}')$
bit-reverse spectrum shift
1-D FFT
Mem
(c)
$|\Psi|$ $\angle\Psi$
Post-Processing

Figure 3: Dataflow divided into three processing steps, where grey boxes indicate memory transfers and white boxes indicate computation. (a) Combine, rotate, and one-dimensional FFT over rows. (b) One-dimensional FFT over columns and store peak amplitude. (c) Calculate magnitude or phase and scale result into pixels.

system. The FFT is a separable transform and can be calculated by consecutively applying one-dimensional transforms over rows and columns [68]. However, the three-dimensional organization of SDRAM devices into *banks*, *rows*, and *columns*, results in non-uniform access time to memory contents. Accessing different banks or consecutive elements inside a row has a low latency, while accessing data in different rows has a high latency. Therefore, the memory bandwidth is highly dependent on the access pattern, and data should be transferred in *bursts* accessing consecutive row elements [14]. Calculating the FFT over columns will consequently result in a longer latency for each memory access, resulting in a reduced throughput. An alternative, which is presented in Section 3.4, is to transpose the data as an intermediate step in the two-dimensional FFT. This improves the overall performance, but requires data to be transferred to and from memory three times to compute a single frame. With a dual memory system, supporting simultaneous reads and writes in burst mode, the system operating frequency must hence be in the range of $3 \cdot N_{\mathrm{FFT}}^2 \cdot 25\,\mathrm{fps} \approx 300\,\mathrm{MHz}$.

3 Proposed Architecture

Figure 3 shows the dataflow for the reconstruction algorithm and additional processing to obtain the result image. Images recorded with the digital sensor are transferred and stored in external memory. The processing is divided into three sequential steps, each streaming data from external memory. Storing images or intermediate data in on-chip memory is not feasible due to the size of

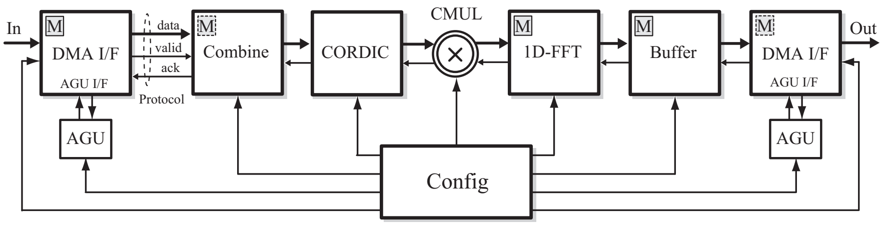


Figure 4: Functional units in the XSTREAM accelerator. Grey boxes (M) indicate units that contain local memories. All processing units communicate using a handshake protocol as illustrated between the DMA and the combine unit.

the two-dimensional FFT, which requires the processing flow to be sequential. Figure 3(a) shows the first processing step where images are combined and each pixel vector is rotated with a phase factor according to (3), and one-dimensional FFTs are calculated over the rows. Figure 3(b) shows the second processing step, which calculates one-dimensional FFTs over the columns. The peak amplitude $\max(|\Psi|)$ is stored internally before streaming the data back to memory. In the final processing step, shown in Figure 3(c), the vector magnitude or phase is calculated to produce a visible image with 8-bit dynamic range.

3.1 Algorithm Mapping

The dataflow in Figure 3 has been mapped onto a pipeline architecture containing several 32-bit processing units, referred to as XSTREAM. The name XSTREAM denotes that the accelerator operates on data streams with focus on maximizing the computational efficiency with concurrent execution, and optimizing global bandwidth using long burst transfers. Processing units communicate internally using a handshake protocol, with a *valid* signal from the producer and an *acknowledge* signal back from the receiver. Each of the units has local configuration registers that can be individually programmed to setup different operations, and a global register allows units to be enabled and disabled to bypass parts of the pipeline.

Figure 4 shows the XSTREAM pipeline, which streams data from external memories using DMA interfaces, where each interface can be connected to a separate bus or external memory to operate concurrently. In the first processing step, the recorded images are combined to compute the modified interference pattern. The combine unit is constructed from a reorder unit to extract pixels from the same position in each image, and a subtract unit. The resulting interference pattern is complex-valued with the imaginary part set to zero. This two-dimensional pixel vector is then rotated with the phase factor $\alpha(\boldsymbol{\rho})$, defined in Section 1.1, using a *coordinate rotation digital computer* (CORDIC) unit operating in rotation mode [70]. The CORDIC architecture is pipelined to produce one complex output value each clock cycle. Rotation is followed by a two-dimensional Fourier transformation, evaluated using a one-dimensional FFT separately applied over rows and columns. Considering the size of the two-dimensional FFT, data must be transferred to the main memory between row and column processing, which requires a second processing step. The buffer unit following the FFT is required to unscramble the FFT output since it is produced in bit-reversed order, and to perform a spectrum shift operation to center the zero-frequency component. In the third processing step, the CORDIC unit is reused to calculate magnitude or phase images from the complex-valued result produced by the FFT. The output from the CORDIC unit is scaled using the complex multiplier (CMUL) in the pipeline, from floating-point format back to fixed-point numbers that represent an 8-bit grayscale image. The following sections present architectures and simulations of the functional units in detail. The simulation results are further analyzed in Section 5 to select design parameters.

To efficiently supply the processing units with data, a high bandwidth to external memories is required. The external SDRAM is burst oriented and the bandwidth depends on how data is being accessed. Linear or row-wise access to the memory generates a high bandwidth since the read/write latency for accessing sequential element is low, while random and column-wise access significantly degrades the performance due to increased latency. Some algorithms, for example the two-dimensional FFT, require both row-wise and column-wise data access. Memory access problems are addressed in the following sections, where we propose modifications to the access pattern to optimize the memory throughput.

3.2 Image Combine Unit

The combine unit processes the three images captured by the digital sensor according to (3.1) in Chapter 3.2.1. An additional table (image) containing pre-calculated phase factors $\alpha(\boldsymbol{\rho})$ is stored in main memory, and is directly propagated through the combine unit to the CORDIC unit. Since all four images have to be accessed concurrently, the physical placement in memory is an important aspect. We first consider the images to be stored in separate memory regions, as shown in Figure 5(a). The read operation would then either be *single reads* accessing position (x, y) in each image, or *four burst transfers* from separate memory location. In the former case, simply accessing

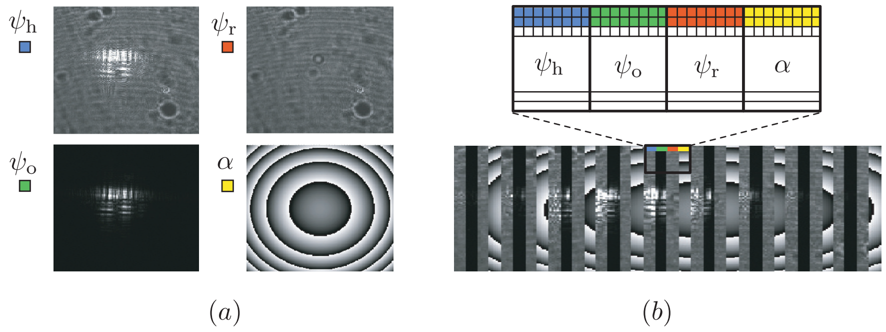


Figure 5: (a) The three captured images and the phase factor. (b) Example of how the captured images are stored interleaved in external memory. The buffer in the combine unit contains 4 groups holding 8 pixels each, requiring a total buffer size of 32 pixels. Grey indicates the amount of data copied to the input buffer in a single burst transfer.

the images pixel-by-pixel from a burst oriented memory is inefficient since it requires four single transfers for each pixel. For the latter approach the burst transfers will require four separate DMA transfers, and each transfer will cause latency when accessing a new memory region. Accessing the data in one single burst transfer would be desirable, and therefore our approach is to combine images by reordering the physical placement in memory during image capture.

Instead of storing the images separately, they can be stored *interleaved* in memory *directly* when data arrive from the sensor, as shown in Figure 5(b). Hence, partial data from each image is fetched in a *single burst transfer* and stored in a local buffer. Thereafter, the data sequence is reordered using a modified n-bit address counter from linear address mode $[a_{n-1}a_{n-2}\dots a_1a_0]$ to extract image information pixel-by-pixel using a reordered address mode $[a_1a_0a_{n-1}\dots a_2]$ as

$$\begin{array}{ll}\textbf{in}: & \{\{\psi_\mathrm{h}^{0\dots B-1}\}\{\psi_\mathrm{o}^{0\dots B-1}\}\{\psi_\mathrm{r}^{0\dots B-1}\}\{\psi_\alpha^{0\dots B-1}\}\} \\ \textbf{out}: & \{\{\psi_\mathrm{h}^{0}\psi_\mathrm{o}^{0}\psi_\mathrm{r}^{0}\psi_\alpha^{0}\}\{\dots\}\{\dots\}\{\psi_\mathrm{h}^{B-1}\psi_\mathrm{o}^{B-1}\psi_\mathrm{r}^{B-1}\psi_\alpha^{B-1}\}\},\end{array}$$

where B is the size of the block of pixels to read from each image. Hence, the internal buffer in the combine unit must be able to store $4B$ pixels. Using this scheme, both storing captured images and reading images from memory can be performed with single burst transfers. Figure 6 shows how the reorder operation depends on the burst size N_burst, a parameter that also defines the block size B and the required internal buffering. For a short burst length, it is more efficient to store the images separately since the write operation can be

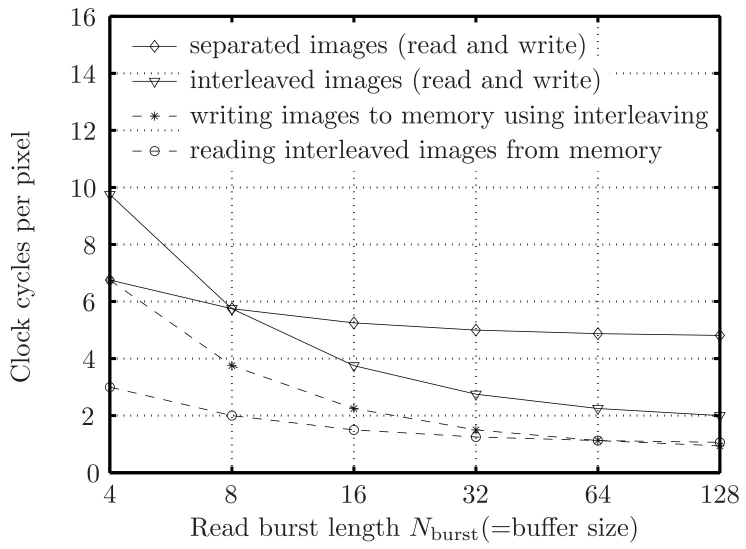


Figure 6: Average number of clock cycles for each produced pixel, including both the write and the read operation to and from memory. The dashed lines show the read and write part from the interleaved version. Simulation assumes that two memory banks are connected the XSTREAM accelerator. (Memory device: [69], 2×16 bit, $f_{\text{mem}} = f_{\text{bus}} = 133\,\text{MHz}$)

burst oriented. When the burst length increases above 8, interleaving becomes a suitable alternative but requires more buffer space. A trade-off between speed and memory is when the curve that represents interleaving flattens out, which suggests a burst length between 32 and 64. Parameter selection is further discussed in Section 5.

3.3 One-Dimensional Flexible FFT Core

The FFT is a separable transform, which means that a two-dimensional FFT can be calculated using a one-dimensional FFT applied consecutively over the x and y dimensions of a matrix. We present the implementation of a high-precision one-dimensional FFT using data scaling, that is used as the core component in a two-dimensional FFT presented in Section 3.4. The FFT implementation is based on a radix-2^2 single-path delay feedback architecture [71]. It supports data scaling to reduce the wordlength to only 10 bits, and supports input data in both linear and bit-reversed order.

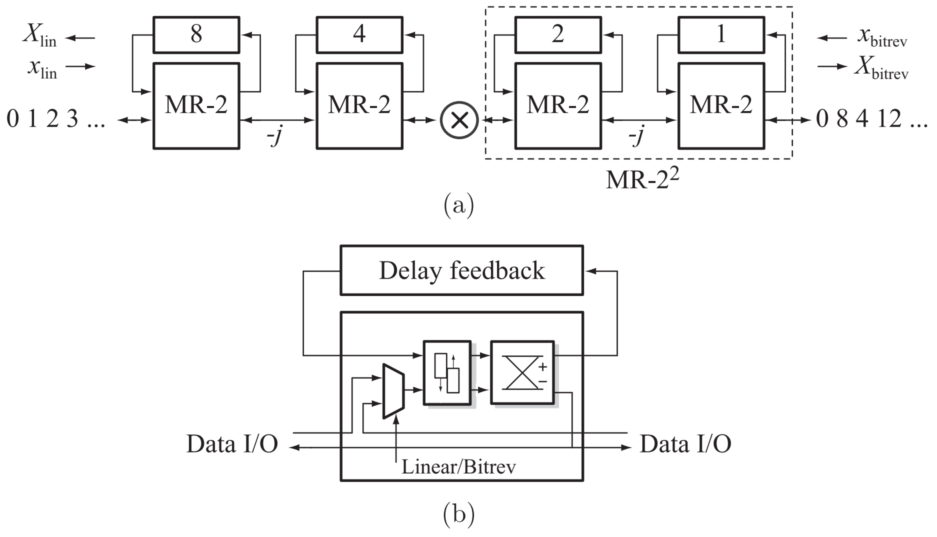


Figure 7: (a) The FFT pipeline is constructed from modified radix-2 units (MR-2). For a 2048-point FFT, totally 5 MR-2^2 and one MR-2 unit is required. (b) A modified radix-2 unit containing a selection unit for switching between linear and bit-reversed mode, an alignment unit for hybrid floating-point, and a butterfly unit.

Scaling alternatives - Various data scaling alternatives have been evaluated in this work and the approach selected for the current implementation is briefly presented below. A more detailed description and comparison is given in [72], with the focus on various types of data scaling, and related work is presented in [73]. We propose to use a reduced complexity floating-point representation for complex valued numbers called hybrid floating-point, which uses a shared exponent for the real and imaginary part. Besides reduced complexity in the arithmetic units, especially the complex multiplication, the total wordlength for a complex number is reduced. Figure 7(a) shows the pipeline architecture with cascaded processing units. The radix-2 units are modified to support data scaling by adding an alignment unit before the butterfly operation, presented in Figure 7(b). The output from the complex multipliers requires normalization, which is implemented using a barrel shifter. The proposed hybrid floating-point architecture has low memory requirements but still generates a high SQNR of 45.3 dB.

Bidirectional pipeline - An architecture with a bidirectional pipeline has been implemented to support the convolution-based algorithm. A bidirectional dataflow can be supported if the wordlength is constant in the pipeline, which enables the data to be propagated through the butterfly units in reverse order. Reversing the data flow means that it is possible to support both linear and bit-reversed address mode on the input port, as shown in Figure 7(a). Today, one-directional FFT with optimized wordlength in each butterfly unit is commonly used, increasing the wordlength through the pipeline to maintain accuracy. Implementations based on convergent block floating-point (CBFP) are also proposed in [74], but both of these architectures only work in one direction. Our bidirectional pipeline simplifies the memory access pattern when evaluating a one- or two-dimensional convolution using the FFT [75]. If the forward transform generates data in bit-reversed order, input to the reverse transform should also be bit-reversed. The result is that both input and the output from the convolution is in normal bit order, hence no reorder buffers are required. Other applications for a bidirectional pipeline is in OFDM transceivers to minimize the required buffering for inserting and removing the cyclic suffix, which has been proposed in [76].

3.4 Two-Dimensional FFT

The two-dimensional FFT is calculated using the one-dimensional FFT core from Section 3.3. Calculating the two-dimensional FFT requires both row-wise and column-wise access to data, which will cause a memory bottleneck as discussed in Section 2. The memory access pattern when processing columns will cause a serious performance loss since new rows are constantly accessed, preventing burst transfers. However, it can be observed that an equivalent procedure to the row/column approach is to transpose the data matrix between computations. This means that the FFT is actually applied two times over rows in the memory separated with an intermediate transpose operation. If the transpose operation combined with FFTs over rows is faster than FFTs over columns, then the total execution time is reduced. To evaluate this approach, a fast transpose unit is proposed.

Transpose unit - Considering the large size of the data matrix, on-chip storage during processing is not realistic and transfers to external memory is required. The access pattern for a transpose operation is normally reading rows and writing columns, or vice versa. To avoid column-wise memory access, a transpose operation can be broken down into a set of smaller transpose operations

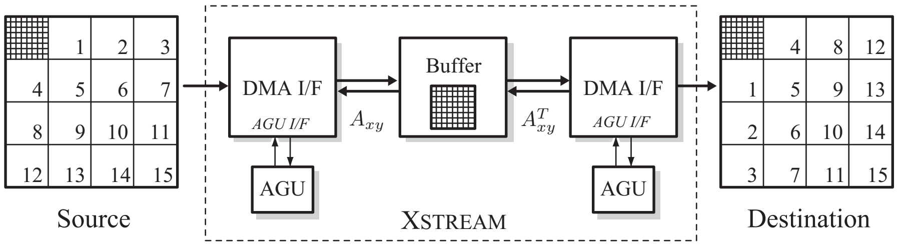


Figure 8: The transpose logic uses the DMA controllers, the AGU units and the internal buffer to transpose large matrices. In the figure, a 32×32 matrix is transposed by individually transposing and relocating 4×4 macro blocks. Each macro block contains an 8×8 matrix.

combined with block relocation as

$$\begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} \end{bmatrix}^T = \begin{bmatrix} \boldsymbol{A}_{11}^T & \boldsymbol{A}_{21}^T \\ \boldsymbol{A}_{12}^T & \boldsymbol{A}_{22}^T \end{bmatrix}.$$

The matrix is divided recursively using a divide-and-conquer approach, which can be performed all the way down to single elements. Breaking the matrix down into single elements is not desirable, since all accesses result in single read and write operations. However, if the matrix is divided into smaller macro blocks that fit into an on-chip buffer with a minimum size of $M \times M$ elements, burst transfers of size $N_{\text{burst}} \leq M$ can be applied for both read and write operations. Hence, data is always transferred to and from the buffer as consecutive elements, while the actual transpose operation is performed in the buffer by writing rows and reading columns. The transpose operation is illustrated in Figure 8. The macro blocks are transferred to the internal buffer using direct memory access (DMA) units and address generation units (AGU) provide a base address for each macro block as

$$\text{ADDR}_{\text{x,y}} = M(x + yN_{\text{FFT}}) \qquad x, y = 0, 1, \ldots, M - 1.$$

Input and output AGUs generate base addresses in reverse order by exchanging index counters x, y to relocate the macro blocks. Figure 9 shows a simulation of the row-column FFT and the row-transpose-row FFT. For the latter approach, the transpose operation is required, also shown separately in the graph. When the burst length is short, the transpose overhead dominates the computation time. When the burst length increases, the row-transpose-row FFT improves the overall performance. The graph representing row-transpose-row flattens out when the burst length is between 16 and 32 words, which is a good trade-

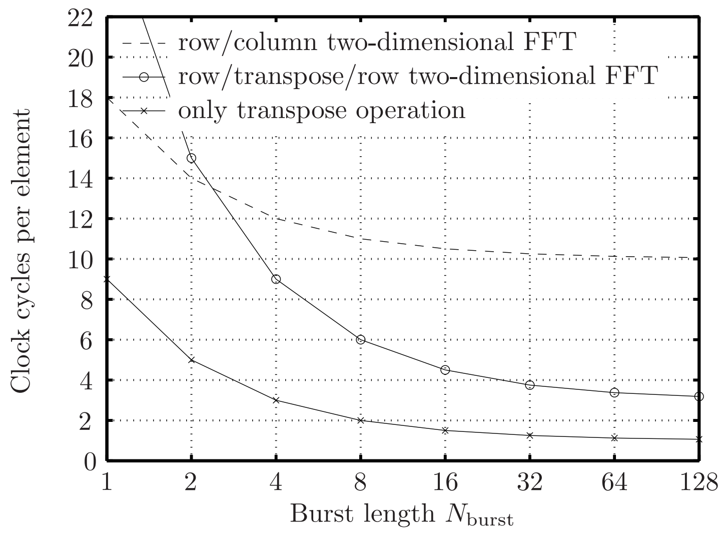


Figure 9: Number of clock cycles per element for calculating a two-dimensional FFT using the two methods. The transpose operation is also shown separately. Simulation assumes that two memory banks are connected the XSTREAM accelerator. (Memory device: [69], 2×16 bit, $f_{\text{mem}} = f_{\text{bus}} = 133\,\text{MHz}$)

off between speed and memory. Parameter selection is further discussed in Section 5.

4 System Design

This section presents the integration of the XSTREAM accelerator into an embedded system and a prototype of a holographic microscope. The system captures, reconstructs and presents holographic images. The architecture is based on the XSTREAM accelerator, extended with an embedded SPARC compatible microprocessor [77] and a memory controller for connecting to external memory. Only a single memory interface is supported in the prototype, but the XSTREAM accelerator supports streaming from multiple memories. Two additional interface blocks provide functionality for capturing images from an external sensor device and to present reconstructed images on a monitor. The complete system is shown in Figure 10.

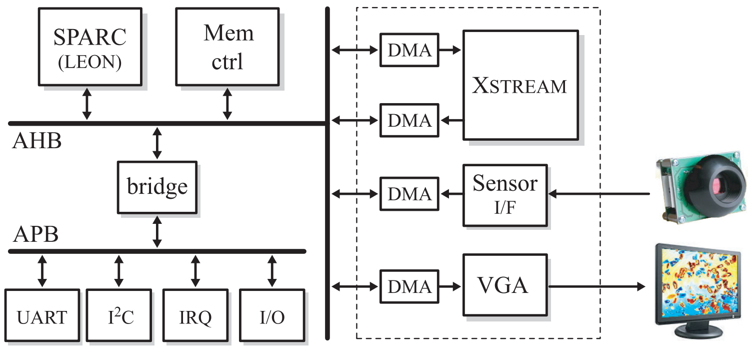


Figure 10: The system is based on a CPU (LEON), a hardware accelerator, a high-speed memory controller, an image sensor interface and a VGA controller.

4.1 External Interface

XSTREAM communicates using a standard bus interface to transfer data between the processing pipeline and an external memory, where data is transferred using DMA modules as shown in Figure 11. The custom designed DMA modules connect to the on-chip bus, and are constructed from three blocks: a bus interface, a buffer module and a configuration interface. One side connects to the external bus while the other side connects to the internal dataflow protocol.

The DMA bus interface is compatible with the advanced high-speed bus (AHB) protocol which is a part of the AMBA specification [78]. Configuration data is transferred over the advanced peripheral bus (APB). The bus interface reads and writes data using transfers of any length, which allows fast access to burst oriented memories. By specifying a base address (*addr*), the transfer size (*hsize, vsize*), and the space between individual rows (*skip*), the bus interface can access a two-dimensional matrix of any size inside a two-dimensional address space of any size. The DMA interface also supports an external address generation unit (AGU). This interface can be used to automatically restart the DMA transfer from a new base address when the previous transfer has completed. Hence, there is no latency between transfers. This is useful when processing or moving data inside a larger memory space, e.g. a matrix of blocks. An illustrating example is the transpose operation described in Section 3.4, which relocates blocks of data inside a matrix. The *block* is the actual transfer and *matrix* is the current address space.

The DMA buffer contains a format transformation unit that allows splitting of words into sub-word transfers on the read side and combining sub-words into

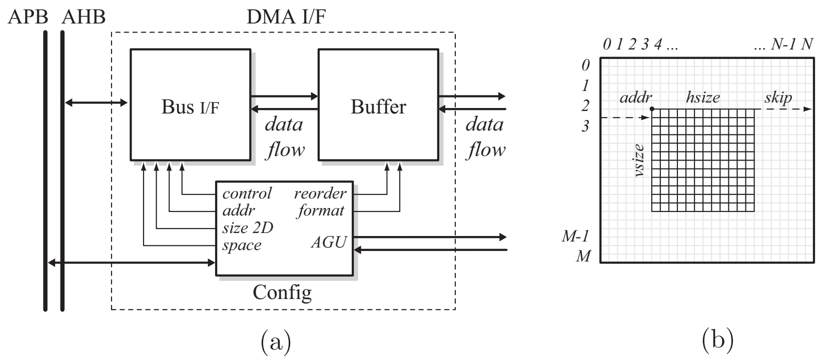


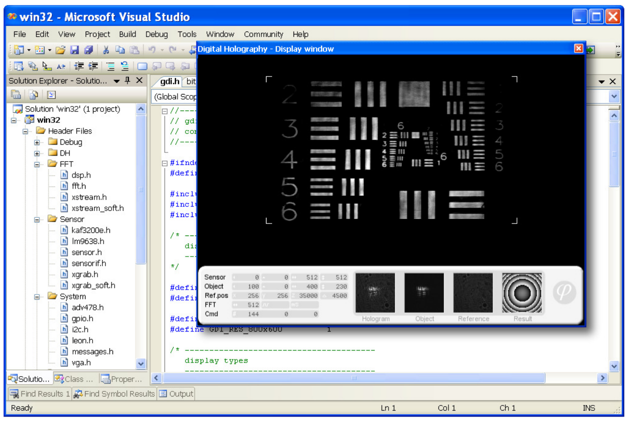
Figure 11: (a) The DMA interface connects the internal dataflow protocol to the external bus using a buffer and an AMBA bus interface. (b) The DMA interface can access a two-dimensional matrix inside a two-dimensional memory array, where *skip* is the distance to the next row.

words on the write side. An example is when calculating the vector magnitude of the resulting image and then rescaling the value into an 8-bit pixel. Pixels are processed individually, but combined into 32-bit words (groups of 4 pixels) in the buffer before transferred to the memory to reduce the transfer size. The buffer also has the ability to reorder the output data in various ways, which is further explained in Section 5.1.

4.2 **Software Design**

The XSTREAM accelerator is configured by an on-chip processor. A program is running on the embedded processor, utilizing the hardware accelerator to capture, process and present images. For development purpose, the software program also has the capability of emulating all the hardware resources in software, which means that the system can run on any computer with a C compiler. For evaluation purpose, the software also contains a full-precision floating-point model, which can be activated to evaluate the arithmetic accuracy of the hardware units.

On the embedded system, platform specific drivers control the underlying hardware through configuration registers. Each driver has two modes, one for controlling the hardware and one for controlling the corresponding hardware emulator. Switching between hardware and emulated hardware is done transparently during compile time, based on the current platform. For example, capturing images from the sensor is on the computer implemented by reading a bitmap image from a file. Output to a monitor is emulated with a graph-



Sensor	0	0	512	512
Object	100	0	400	230
Ref.pos	256	256	35000	4500
FFT	512			
Cmd	144	0	0	

```
//---
// gd
// co
//---

#ifnd
#defi

#incl
#incl
#incl

/* ---
   di
   ---
*/

#defi
#defi

#defi
#define GDI_RES_800x600          1

/* ---------------------------------------
   display types
   ---------------------------------------
```

Figure 12: The platform independent software running in Microsoft Visual Studio. The hardware accelerator, sensor and monitor are emulated in software.

ical window, as shown in Figure 12. From an application point of view, the behavior of the system is the same.

5 Simulation and Optimization

Design parameter values that affect the performance and on-chip memory requirements, such as the burst length N_{burst}, are chosen based on the Matlab simulations presented in previous sections. For an accurate simulation, memory timing parameters are extracted from the same memory device as used in the hardware prototype, which is two parallel 512-Mbit SDRAM devices from Micron Technology, with a combined wordlength of 32 bits [69].

5.1 Flexible Addressing Modes

Many of the functional units contain buffers to store and re-arrange data, where each buffer requires a special addressing mode as presented in Table 1. In the next section some of the buffers are merged to save storage space, and therefore it is important that each buffer supports a flexible addressing mode to maintain the original functionality after merging. Figure 13 illustrates how different addressing modes are used to rearrange input data. Data is always written to the buffer using linear addressing, and when reading from the buffer the address mode determines the order of the output sequence. Both bit-reverse and FFT spectrum shift depends on the transform size, i.e. the number of

Table 1: Required buffering inside each processing block for the original and optimized pipeline. The addressing mode for each buffer depends on the function that the block is evaluating.

Unit	Buffer	Original	Optimized	Addr mode
DMA	Input buffer	N_{burst}	N_{burst}	Linear
Combine	Reorder	N_{burst}	0	Interleaved
FFT	Delay feedback	$N_{\text{FFT}}-1$	$N_{\text{FFT}}-1$	FIFO
Buffer	Transpose	${N_{\text{burst}}}^2$	$\max({N_{\text{burst}}}^2, N_{\text{FFT}})$	Col-row swap
	Bit-reverse	N_{FFT}	0	Bit-reversed
	Spectrum shift	$N_{\text{FFT}}/2$	0	FFT shift
DMA	Output buffer	N_{burst}	0	Linear

address bits, which requires the addressing modes to be flexible with a dynamic address wordlength. Since bit-reverse and FFT spectrum shift is often used in conjunction, this addressing mode can be optimized. The spectrum shift inverts the MSB, as shown in Figure 13(d), and the location of the MSB depends on the transform size. However, in bit-reversed addressing the MSB is actually the LSB, and the LSB location is always constant. By reordering the operations, the cost for moving the spectrum is a single inverter.

5.2 **Memory Optimization**

Table 1 presents the required buffering in each functional unit, where some of the units can merge buffers if they support the flexible addressing mode presented in Section 5.1. The table also presents an optimized pipeline with merged buffers. Units without a buffer after optimization read data directly from the buffer located in the previous unit. The following optimizations are performed:

- The reorder operation in the image combine block is moved to the DMA input buffer. The only modification required is that the DMA input buffer must support both linear and interleaved addressing mode.

- The transpose buffer for two-dimensional FFT transforms is reused for bit-reversal and FFT spectrum shift. These are in fact only two different addressing modes, which are supported by the transpose buffer. Merging these operations save a large amount of memory, since both bit-reverse and FFT shift require the complete sequence and half the sequence to be stored, respectively.

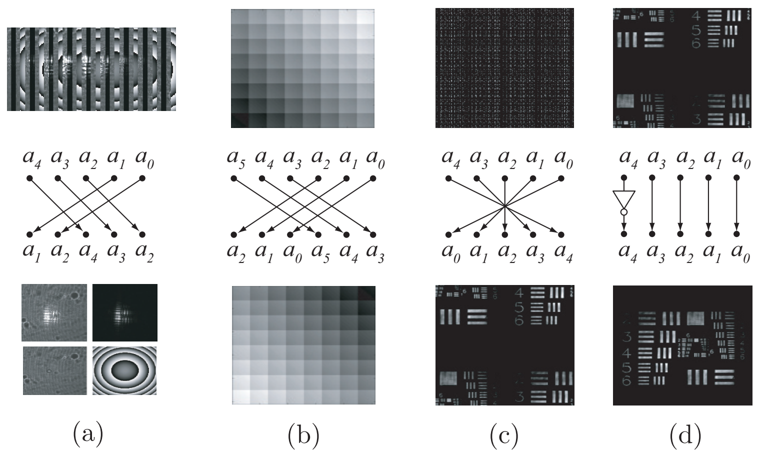

a_4 a_3 a_2 a_1 a_0
a_1 a_2 a_4 a_3 a_2
a_5 a_4 a_3 a_2 a_1 a_0
a_2 a_1 a_0 a_5 a_4 a_3
a_4 a_3 a_2 a_1 a_0
a_0 a_1 a_2 a_3 a_4
a_4 a_3 a_2 a_1 a_0
a_4 a_3 a_2 a_1 a_0
(a)
(b)
(c)
(d)

Figure 13: Buffers support multiple addressing modes to rearrange data. a_n represents the address bits, and the input data (top) is addressed with a linear counter. (a) Interleaving with 4 groups and 8 values in each group. (b) Transpose addressing for an 8×8 macro block matrix by swapping row and column address bits. (c) Bit-reversed addressing to unscramble FFT data. (d) FFT spectrum shift by inverting the MSB.

- The output DMA read data in linear mode directly from the main buffer unit.

Figure 14 shows the memory requirements before and after optimization and how it depends on N_{burst}. The delay feedback memory in the FFT can not be shared and is not included in the memory simulation. For short burst lengths, the memory requirements are reduced from $\approx 3\,\text{K}$ words to $\approx 2\,\text{K}$ words. The memory requirement then increases with the burst length for the unoptimized design, but stays constant for the optimized design up to $N_{\text{burst}} = 32$. After this point, the memory requirements for both architectures rapidly increase.

5.3 Parameter Selection

The shared buffer unit in the pipeline must be at least the size of N_{FFT} to support the bit-reverse operation. It is reused for the transpose operation, which requires ${N_{\text{burst}}}^2$ elements. The buffer size is hence the maximum of the two. Figure 14 shows how the memory requirements depend on the design parameter N_{burst}, which should be maximized under the condition that

$$N_{\text{burst}}^2 \leq N_{\text{FFT}} = 2048,$$

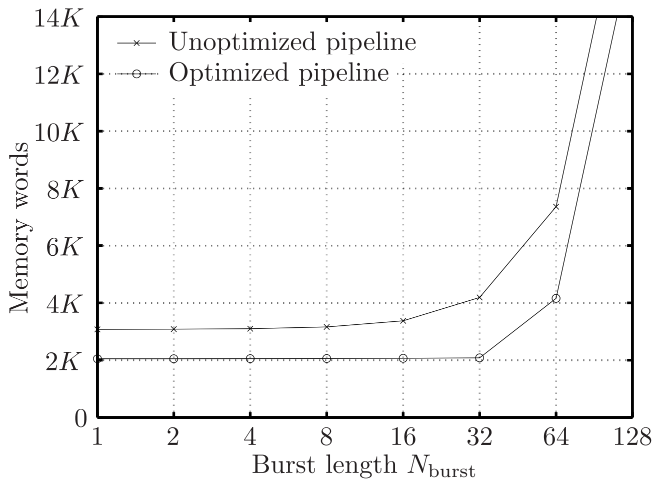


Figure 14: Memory requirements in 32-bit words for different values on N_{burst} when $N_{\text{FFT}} = 2048$. The delay feedback memory is not included since it can not be shared with other units.

and that N_{burst} is an integer power of 2. When ${N_{\text{burst}}}^2$ exceeds N_{FFT}, internal memory requirements rapidly increases, which leads to a high area cost according to Figure 14 and a relatively low performance improvement according to Figure 6 and Figure 9. Another condition is that the image read operation should generate one set of pixels per clock cycle, or at least close to this value, to supply the FFT with input data at full speed to balance the throughput. Selecting $N_{\text{burst}} = 32$ satisfy the conditions and results in:

- A total of 3.2 cc/element for calculating the two-dimensional FFT, compared with up to 10 clock cycles for the traditional row-column approach, as shown in Figure 9. This is a speed-up factor of approximately 3 for the two-dimensional FFT.

- The combine unit requires 2.8 cc/element to store and read the image data from external memory in interleaved mode. This is a speed-up factor of approximately 2 compared to storing images separately in memory, as shown in Figure 6.

- The combine unit is capable of constantly supplying the FFT unit with data. Hence, the total system speed-up factor is the same as for the two-dimensional FFT.

Table 2: Equivalent gates (NAND2) and memory cells (RAM and ROM) after synthesis to a 0.13 μm standard cell library. Slice count and block RAMs are presented for the FPGA design.

Module	Eq. gates (logic only)	RAM (Kbit)	ROM (Kbit)	FPGA (Slices)	# Block RAMs
DMA Input	3123 (3%)	1.0	0	223 (3%)	1
Combine	2489 (2%)	0	0	198 (2%)	0
CORDIC	10016 (8%)	0	0	403 (5%)	0
CMUL	5161 (4%)	0	0	362 (5%)	0
FFT	83430 (70%)	49.0	47.6	5826 (73%)	25
Buffer	1287 (1%)	65.5	0	166 (2%)	16
DMA Output	2592 (2%)	0	0	102 (1%)	0
AGU (2x)	2148 (2%)	0	0	110 (1%)	0
Sensor I/F	4574 (4%)	1.0	0	280 (4%)	1
VGA I/F	4469 (4%)	1.0	0	315 (4%)	1
Total	119289	117.5	47.6	7985	45

- The optimized pipeline requires less then 50% of the memory compared to the original pipeline, reduced from over 4 Kbit down to 2 Kbit as shown in Figure 14.

6 Results and Comparisons

The system presented in Section 4 has been synthesized for FPGA, targeting a Xilinx Virtex-1000E device, and integrated into a microscope based on digital holography to reconstruct images captured with a sensor device, shown in Figure 15(a). The microscope is shown to the right, and the screen in middle is connected to the FPGA platform to display the resulting image from the reconstruction. The computer to the left runs a graphical user interface to setup the hardware accelerator and to download reconstructed images for further processing. The FPGA prototyping board is shown in Figure 15(b) and contains a Virtex-1000E device, 128MB of SDRAM, a digital sensor interface, and a VGA monitor interface.

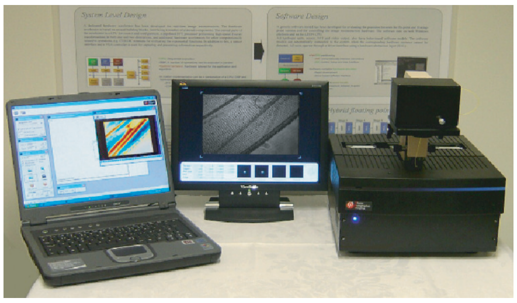
The design has also been synthesized to a high-speed 0.13 μm standard cell library from Faraday. Synthesis results from both implementations can be found in Table 2. The table shows the number of equivalent gates (NAND2) and the memory requirements including both RAM and ROM (twiddle factor tables in the FFT). For FPGA synthesis, the number of occupied slices and

Table 3: Comparison between related work and the proposed architecture. Performance is presented as fps/MHz, normalized to 1.0 for the proposed FPGA design.

Technology	Freq (MHz)	Mem I/F	Rate (fps)	FPGA Slices	Performance (relative)
Pentium-4 Dual Core	3 GHz	Dual	0.8	-	0.0044
$0.35\,\mu$m [79]	133	Single	2.6	-	0.3
XCV2000E [80]	35	Quad	2.0	8480	0.9
Proposed XCV1000E	24	Dual	1.5	7985	1.0
Proposed $0.13\,\mu$m	398	Dual	≈ 31	-	1.27

required block RAMs are presented, where the Xstream accelerator occupies approximately 65% of the FPGA resources. The largest part of the design is the 2048-point pipeline FFT, which is approximately 73% of the Xstream accelerator. The FPGA design runs at a clock frequency of 24 MHz, limited by the embedded processor, while the design synthesized for a 0.13μm cell library is capable of running up to 398 MHz. A floorplan of the Xstream accelerators is shown in Figure 16, with a core area of $1500 \times 1400\,\mu\text{m}^2$ containing 352 K equivalent gates.

In related work on two-dimensional FFT implementation, the problem with memory organization is not widely mentioned. Instead, a high bandwidth from memory with uniform access-time is assumed (SRAM). However, for computing a large size multi-dimensional FFT the memory properties and data organization must be taken into account, as discusses in this work. Table 3 shows a comparison between the proposed architecture, a modern desktop computer, and related work. In [79], an ASIC design of a 512-point two-dimensional FFT connected to a single memory interface has been presents. [80] presents an FPGA implementation with variable transform size storing data in four separate memory banks. To compare the processing efficiency between different architectures, a performance metric is defined as (fps / MHz) and normalized to 1.0 for the proposed FPGA implementation. The frame rate is estimated for a transform size of 2048×2048 points. The table shows the proposed architecture to be highly efficient, resulting in real-time image reconstruction with over 30 fps for the proposed ASIC design. The reason for increased efficiency when targeting ASIC is that the DMA transfers with fixed bandwidth requirements, such as the sensor and VGA interfaces from Figure 10, will have less impact on the total available bandwidth as the system clock frequency increases.

(a) 

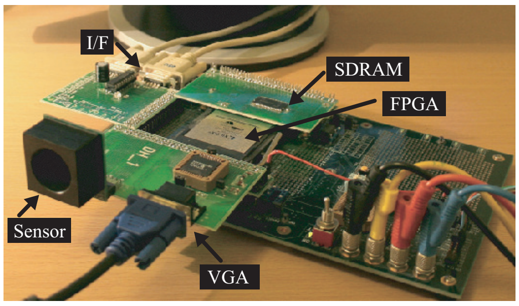
(b) 

Figure 15: (a) Microscope prototype connected to an external monitor. (b) The hardware platform containing a Virtex-1000E device, 128MB of SDRAM, a digital sensor interface, and a VGA monitor interface.

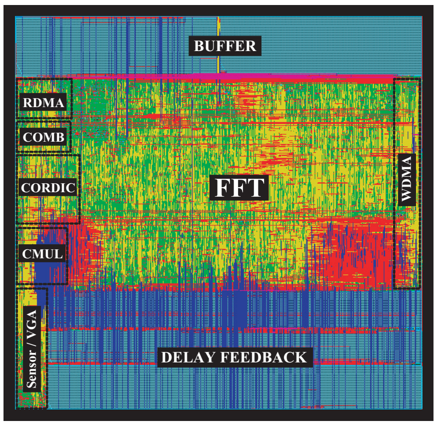


Figure 16: Final layout of the XSTREAM accelerator using a 0.13 μm cell library. The core size is $1500 \times 1400\,\mu\text{m}^2$.

7 Conclusion

A hardware acceleration platform for image processing in digital holography has been presented. The hardware accelerator contains an efficient datapath for calculating FFT and other required operations. A fast transpose unit is proposed to significantly improve the computation time for a two-dimensional FFT, which improves the computation time with a factor of 3 compared with the traditional row/column approach. To cope with the increased bandwidth and to balance the throughput of the computational units, a fast reorder unit is proposed to store captured images and read data in an interleaved fashion. This results in a speedup of 2 compared with accessing separately stored images in memory. It is also shown how to reduce the memory requirement in a pipelined design with over 50% by sharing buffers between modules. The design has been synthesized and integrated in an FPGA-based system for digital holography. The same architecture targeting a $0.13\,\mu$m CMOS standard cell library achieves real-time image reconstruction with over 30 frames per second.

Part II

A High-performance FFT Core for Digital Holographic Imaging

Abstract

Dynamic data scaling in pipeline FFTs is suitable when implementing large size FFTs in applications such as DVB and digital holographic imaging. In a pipeline FFT, data is continuously streaming and must hence be scaled without stalling the dataflow. We propose a hybrid floating-point scheme with tailored exponent datapath, and a co-optimized architecture between hybrid floating-point and block floating-point to reduce memory requirements for two-dimensional signal processing. The presented co-optimization generates a higher SQNR and requires less memory than for instance convergent block floating-point. A 2048 point pipeline FFT has been fabricated in a standard CMOS process from AMI Semiconductor [9], and an FPGA prototype integrating a two-dimensional FFT core in a larger design shows that the architecture is suitable for image reconstruction in digital holographic imaging.

Based on: T. Lenart and V. Öwall, “Architectures for dynamic data scaling in 2/4/8K pipeline FFT cores,” *IEEE Transaction on Very Large Scale Integration.* ISSN 1063-8210, vol. 14, no. 11, Nov 2006.

and: T. Lenart and V. Öwall, “A 2048 Complex Point FFT Processor using a Novel Data Scaling Approach,” in *Proceedings of IEEE International Symposium on Circuits and Systems*, vol. 4, Bangkok, Thailand, May 2003, pp. 45–48.

1 Introduction

The discrete Fourier transform is a commonly used operation in digital signal processing, where typical applications are linear filtering, correlation, and spectrum analysis [68]. The Fourier transform is also found in modern communication systems using digital modulation techniques, including wireless network standards such as 802.11a [81] and 802.11g [82], as well as in audio and video broadcasting using DAB and DVB.

The DFT is defined as

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \qquad 0 \leq k < N, \tag{1}$$

where

$$W_N = \mathrm{e}^{-\mathrm{i}2\pi/N}. \tag{2}$$

Evaluating (1) requires N MAC operations for each transformed value in X, or N^2 operations for the complete DFT. Changing transform size significantly affects computation time, e.g. calculating a 1024-point Fourier transform requires three orders of magnitude more work than a 32-point DFT.

A more efficient way to compute the DFT is to use the fast Fourier transform (FFT) [83]. The FFT is a decomposition of an N-point DFT into successively smaller DFT transforms. The concept of breaking down the original problem into smaller sub-problems is known as a divide-and-conquer approach. The original sequence can for example be divided into $N = r_1 \cdot r_2 \cdot \ldots \cdot r_q$ where each r is a prime. For practical reasons, the r values are often chosen equal, creating a more regular structure. As a result, the DFT size is restricted to $N = r^q$, where r is called *radix* or decomposition factor. Most decompositions are based on a radix value of 2, 4 or even 8 [84]. Consider the following decomposition of (1), known as radix-2

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n) W_N^{kn} \\ &= \sum_{n=0}^{N/2-1} x(2n) W_N^{k(2n)} + \sum_{n=0}^{N/2-1} x(2n+1) W_N^{k(2n+1)} \\ &= \underbrace{\sum_{n=0}^{N/2-1} x_{even}(n) W_{N/2}^{kn}}_{DFT_{N/2}(x_{even})} + W_N^k \underbrace{\sum_{n=0}^{N/2-1} x_{odd}(n) W_{N/2}^{kn}}_{DFT_{N/2}(x_{odd})} . \end{aligned} \tag{3}$$

The original N-point DFT has been divided into two $N/2$ DFTs, a procedure that can be repeated over again on the smaller transforms. The complexity is thus reduced from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log_2 N)$. The decomposition in (3) is called *decimation-in-time* (DIT), since the input $x(n)$ is decimated with a factor of 2 when divided into an even and odd sequence. Combining the result from each transform requires a scaling and add operation. Another common approach is known as *decimation-in-frequency* (DIF), splitting the input sequence into $x_1 = \{x(0), x(1), ..., x(N/2-1)\}$ and $x_2 = \{x(N/2), x(N/2+1), ..., x(N-1)\}$. The summation now yields

$$\begin{aligned} X(k) &= \sum_{n=0}^{N/2-1} x(n)W_N^{kn} + \sum_{n=N/2}^{N-1} x(n)W_N^{kn} \qquad (4) \\ &= \sum_{n=0}^{N/2-1} x_1(n)W_N^{kn} + \underbrace{W_N^{kN/2}}_{(-1)^k} \sum_{n=0}^{N/2-1} x_2(n)W_N^{kn}, \end{aligned}$$

where $W_N^{kN/2}$ can be extracted from the summation since it only depends on the value of k, and is expressed as $(-1)^k$. This expression divides, or decimates, $X(k)$ into two groups depending on whether $(-1)^k$ is positive or negative. That is, one equation calculate the even values and one calculate the odd values as in

$$\begin{aligned} X(2k) &= \sum_{n=0}^{N/2-1} \Big(x_1(n) + x_2(n)\Big) W_{N/2}^{kn} \qquad (5) \\ &= DFT_{N/2}(x_1(n) + x_2(n)) \end{aligned}$$

and

$$\begin{aligned} X(2k+1) &= \sum_{n=0}^{N/2-1} \Big[\Big(x_1(n) - x_2(n)\Big) W_N^n\Big] W_{N/2}^{kn} \qquad (6) \\ &= DFT_{N/2}((x_1(n) - x_2(n))W_N^n). \end{aligned}$$

(5) calculates the sum of two sequences, while (6) calculates the difference and then scales the result. This kind of operation, adding and subtracting the same two values, is commonly referred to as *butterfly* due to its butterfly-like shape in the flow graph, shown in Figure 1(a). Sometimes, scaling is also considered to be a part of the butterfly operation. The flow graph in Figure 1(b) represents

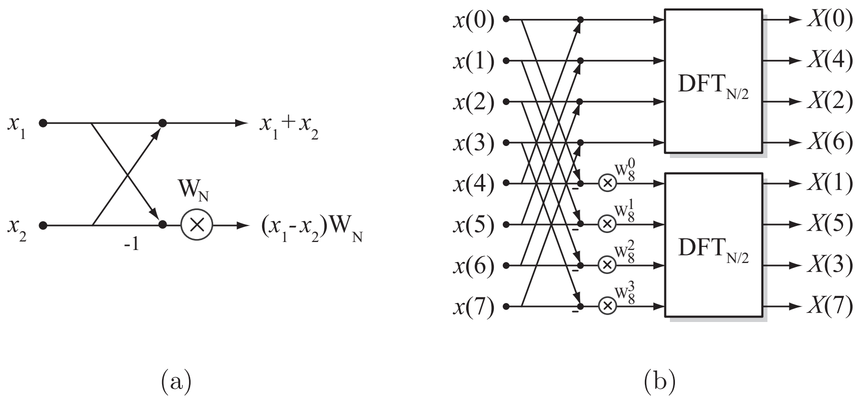


Figure 1: (a) Butterfly operation and scaling. (b) The radix-2 decimation-in-frequency FFT algorithm divides an N-point DFT into two separate $N/2$-point DFTs.

the computations from (5) and (6), where each decomposition step requires $N/2$ butterfly operations.

In Figure 1(b), the output sequence from the FFT appears scrambled. The binary output index is *bit-reversed*, i.e. the most significant bits (MSB) have changed place with the least significant bits (LSB), e.g. 11001 becomes 10011. To unscramble the sequence, bit-reversed indexing is required.

1.1 **Algorithmic Aspects**

From an algorithmic perspective, several possible decomposition algorithms exist [85]. The radix-2 algorithm can be used when the size of the transform is $N = 2^q$, where $q \in \mathbb{Z}^+$. The algorithm requires q decomposition steps, each computing $N/2$ butterfly operation using a radix-2 (R-2) butterfly, shown in Figure 1(a).

However, when the size is $N = 4^q$, more hardware efficient decomposition algorithms exist. One possible alternative is the radix-4 decomposition, which reduces the number of complex multiplications with the penalty of increasing the number of complex additions. The more complex radix-4 butterfly is shown in Figure 2(a). Another decomposition similar to radix-4 is the radix-2^2 algorithm, which simplifies the complex radix-4 butterfly into four radix-2 butterflies [71]. On a flow graph level, radix-4 and radix-2^2 requires the same number of resources. The difference between the algorithms become evident when folding is applied, which is shown later. The radix-2^2 (R-2^2) butterfly

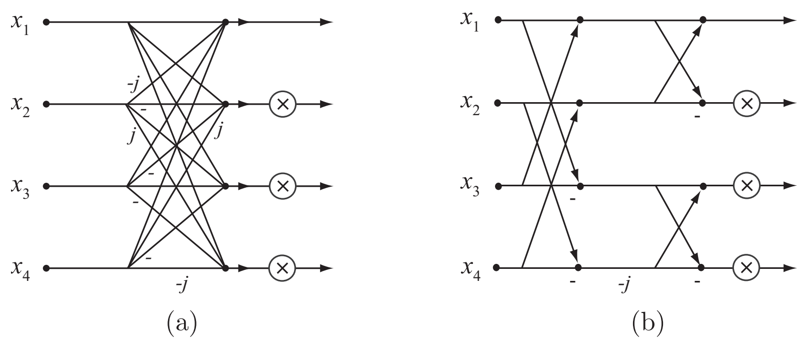


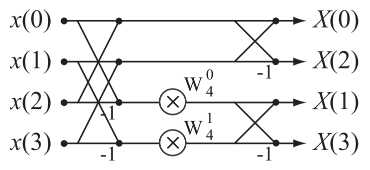
Figure 2: (a) Radix-4 butterfly. (b) Radix-2^2 butterfly.

is found in Figure 2(b). To calculate an FFT size not supported by radix-4 and radix-2^2, for example 2048, both radix-4 decompositions and a radix-2 decomposition is needed since $N = 2048 = 2^1 4^5$.

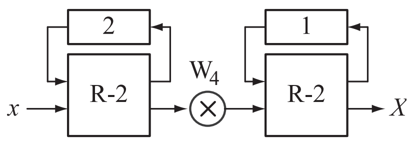
1.2 **Architectural Aspects**

There are several ways of mapping an algorithm to hardware. Three approaches are discussed and evaluated in this section: direct-mapped hardware, a pipeline structure, and time-multiplexing using a single butterfly unit. Direct-mapped hardware basically means that each processing unit in the flow graph is implemented using a unique arithmetic unit. Normally, using this approach in a large and complex algorithm is not desirable due to the huge amount of hardware resources required. The alternative is to fold operations onto the same block of hardware, an approach that saves resources but to the cost of increased computation time.

Figure 3(a) shows a 4-point radix-2 FFT. Each stage consists of two butterfly operations, hence the direct-mapped hardware implementation requires 4 butterfly units and 2 complex multiplication units, numbers that will increase with the size of the transform. Folding the algorithm vertically, as shown in Figure 3(b), reduces hardware complexity by reusing computational units, a structure often referred to as a *pipeline* FFT [86]. A pipeline structure of the FFT is constructed from cascaded butterfly blocks. When the input is in sequential order, each butterfly operates on sample x_n and $x_{n+N/2}$, hence a delay buffer of size $N/2$ is required in the first stage. This is referred to as a single-path delay feedback (SDF). In the second stage, the transform is $N/2$, hence the delay feedback memory is $N/4$. In total, this sums up to $N - 1$ words in the delay buffers.



(a)



(b)

Figure 3: (a) Flow graph of a 4-point radix-2 FFT (b) Mapping of the 4-point FFT using two radix-2 butterfly units with delay feedback memories, where the number represents the FIFO depth.

Folding the pipeline architecture horizontally as well reduces the hardware to a single time-multiplexed butterfly and complex multiplier. This approach saves arithmetic resources, but still requires the same amount of storage space as a pipeline architecture. The penalty is further reduced calculation speed, since all decompositions are mapped onto a single computational unit.

To summarize, which architecture to use is closely linked to the required computational speed and available hardware and memory resources. A higher computational speed normally requires more hardware resources, a trade-off that has to be decided before the actual implementation work begins.

Another important parameter associated with the architectural decisions is the wordlength, which affects the computational accuracy and the hardware requirements. An increased wordlength improves the computational quality, measured in signal-to-quantization-noise-ratio (SQNR), but also increases the hardware cost and the latency in arithmetic units. The trade-off between low hardware cost and high SQNR is referred to as *wordlength optimization*. The SQNR is defined as

$$\mathrm{SQNR}_{\mathrm{dB}} = 10 \cdot \log_{10} \frac{P_x}{P_q}, \tag{7}$$

where P_q is the quantization energy and P_x is the energy in the input signal. A way to increase the SQNR for signals with large dynamic range is to use data scaling, which is discussed in the next section.

2 Proposed Architectures

Currently the demands increase towards larger and multidimensional transforms for use in synthetic aperture radar (SAR) and scientific computing, including biomedical imaging, seismic analysis, and radio astronomy. Larger transforms require more processing on each data sample, which increases the total quantization noise. This can be avoided by gradually increasing the

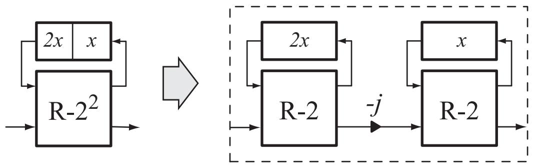


Figure 4: The radix-2^2 (R-2^2) butterfly is constructed from two radix-2 (R-2) butterflies, separated with a trivial multiplication.

wordlength inside the pipeline, but will also increase memory requirements as well as the critical path in arithmetic components. For large size FFTs, dynamic scaling is therefore a suitable trade-off between arithmetic complexity and memory requirements. The following architectures have been evaluated and compared with related work:

(A) A hybrid floating-point pipeline with fixed-point input and tailored exponent datapath for one-dimensional (1D) FFT computation.

(B) A hybrid floating-point pipeline for two-dimensional (2D) FFT computation, which also requires the input format to be hybrid floating-point. Hence, the hardware cost is slightly higher than in (A).

(C) A co-optimized design based on a hybrid floating-point pipeline combined with block floating-point for 2D FFT computation. This architecture has the processing abilities of (B) with hardware requirements comparable to (A).

The primary application for the implemented FFT core is a microscope based on digital holography, where visible images are to be digitally reconstructed from a recorded interference pattern [87]. The pattern is recorded on a large digital image sensor with a resolution of 2048×2048 pixels and processed by a reconstruction algorithm based on a 2D Fourier transformation. Hence, the architectures outlined in (B) and (C) are suitable for this application. Another area of interest is in wireless communication systems based on orthogonal frequency division multiplexing (OFDM). The OFDM scheme is used in for example digital video broadcasting (DVB) [88], including DVB-T with 2/8K FFT modes and DVB-H with an additional 4K FFT mode. The architecture described in (A) is suitable for this field of application.

Fixed-point is a widely used format in real-time and low power applications due to the simple implementation of arithmetic units. In fixed-point arithmetic, a result from a multiplication is usually rounded or truncated to avoid a significantly increased wordlength, hence generating a quantization error. The

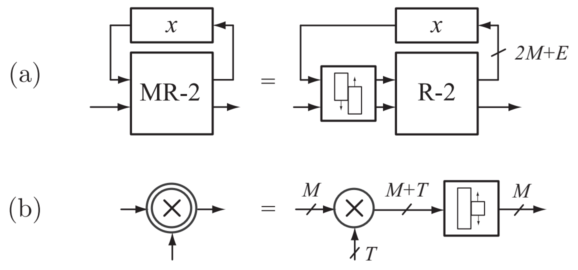


Figure 5: Building blocks for a hybrid floating-point implementation. (a) Symbol of a modified butterfly containing an align unit on the input. (b) Symbol for a modified complex multiplier containing a normalization unit.

quantization energy caused by rounding is relatively constant due to the fixed location of the binary point, whereas the total energy depends on how the input signal utilizes the available dynamic range. Therefore, precision in the calculations depends on properties of the input signal, caused by uniform resolution over the total dynamic range. Fixed-point arithmetic usually requires an increased wordlength due to the trade-off between dynamic range and precision. By using floating-point and dynamically changing the quantization steps, the energy in the error signal will follow the energy in the input signal and the resulting SQNR will remain relatively constant over a large dynamic range. This is desirable to generate a high signal quality, less dependent on the transform length. However, floating-point arithmetic is considerably more expensive in terms of chip area and power consumption, and alternatives are presented in the following subsections followed by a comparison in Section 4.

2.1 **Hybrid Floating-Point**

Floating-point arithmetic increases the dynamic range by expressing numbers with a mantissa m and an exponent e, represented with M and E bits, respectively. A hybrid and simplified scheme for floating-point representation of complex numbers is to use a single exponent for the real and imaginary part. Besides reduced complexity in the arithmetic units the total wordlength for a complex number is reduced from $2 \times (M + E)$ to $2 \times M + E$ bits. Supporting hybrid floating-point requires pre- and post-processing units in the arithmetic building blocks, and Figure 5 defines symbols used for representing these units. The FFT twiddle factors are represented with T bits.

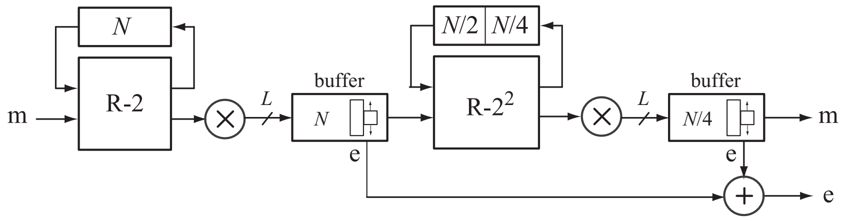


Figure 6: An example of convergent block floating-point. The buffer after each complex multiplier selects a common exponent for a group of values, allowing fixed-point butterfly units. The first buffer is often omitted to save storage space, but will have a negative impact on signal quality.

2.2 Block Floating-Point

Block floating-point (BFP) combines the advantages of simple fixed-point arithmetic with floating-point dynamic range. A single exponent is assigned to a group of values to reduce memory requirements and arithmetic complexity. However, output signal quality depends on the block size and characteristics of the input signal [89]. Finding a common exponent requires processing of the complete block. This information is directly available in a *parallel* FFT architecture, but for *pipeline* FFT architectures scaling becomes more complicated since data is continuously streaming. A scheme known as convergent block floating-point (CBFP) has been proposed for pipeline architectures [73]. By placing buffers between intermediate stages, data can be rescaled using block floating-point, as shown in Figure 6. The block size will decrease as data propagates through the pipeline until each value has its own exponent. Intermediate buffering of data between each stage requires a large amount of memory, and in practical applications the first intermediate buffer is often omitted to save storage space. However, this leads to a reduced SQNR as will be shown in Section 4 and referred to as $\mathrm{CBFP}_{\mathrm{low}}$ due to the lower memory requirements.

2.3 Co-Optimization

In this section a co-optimized architecture that combines hybrid floating-point and BFP is proposed. By extending the hybrid floating-point architecture with small intermediate buffers, the size of the delay feedback memory can be reduced. Figure 7(a-c) show dynamic data scaling for hybrid floating-point, CBFP, and the proposed co-optimization architecture. Figure 7(c) is a combined architecture with an intermediate buffer to apply block scaling on D elements, which reduces the storage space for exponents in the delay feed-

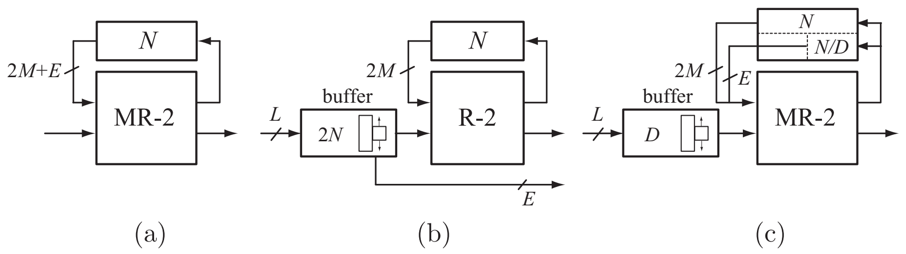


Figure 7: (a) Hybrid floating-point. (b) Convergent block floating-point with $D = 2N - 1$ using a large buffer and fixed-point butterfly. (c) A small buffer reduces the exponent storage space in the delay feedback memory.

back memory with a factor D. We will derive an expression to find optimum values for the block size in each butterfly stage i to minimize the memory requirements for supporting dynamic scaling. The equations can be used for all configurations in Figure 7(a-c) by specifying $D = 1$ for hybrid floating-point and $D = 2N_i$ for CBFP. The length of the delay feedback memory, or FIFO, at stage i is

$$N_i = 2^i \qquad 0 \le i \le i_{max} = \log_2 N_{\text{FFT}} - 1$$

and the number of exponent bits for the same stage is denoted E_i. The block size D spans from single elements to $2N_i$, which can be expressed as

$$D(\alpha_i) = 2^{\alpha_i} \qquad 0 \le \alpha_i \le i + 1.$$

The total bits required for supporting dynamic scaling is the sum of exponent bits in the delay feedback unit and the total size of the intermediate buffer. This can be expressed as

$$\text{Mem}_i = \underbrace{E_i \left\lfloor \frac{\gamma N_i}{D(\alpha_i)} \right\rfloor}_{\text{delay feedback}} + \underbrace{L(D(\alpha_i) - 1)}_{\text{buffer}}, \tag{8}$$

where

$$\gamma = \begin{cases} 1 & \text{Radix-2} \\ 3/2 & \text{Radix-}2^2 \end{cases}$$

and

$$L = \begin{cases} 2M + E_i & i = i_{max} \\ 2(M + T) & 0 \le i < i_{max} \end{cases}$$

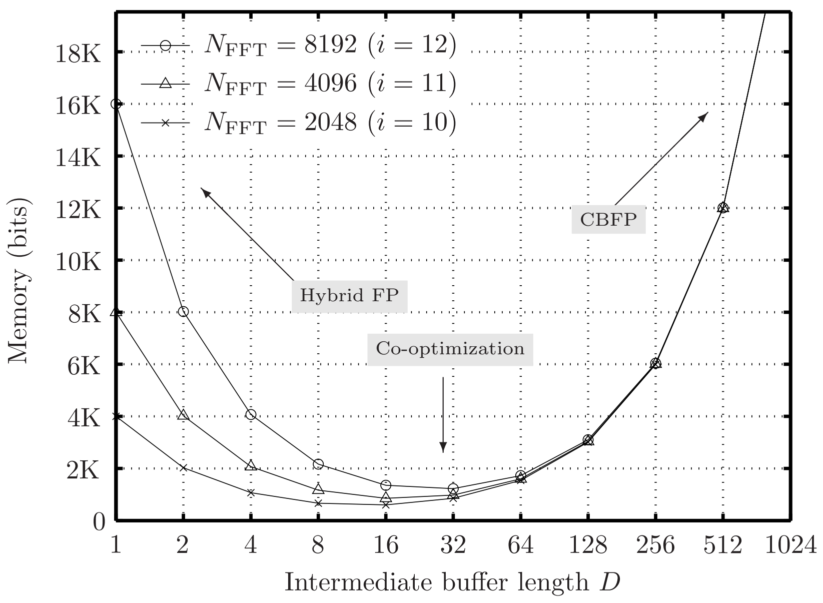


Figure 8: Memory requirements for supporting dynamic scaling as a function of D for the initial butterfly in an N_{FFT} point FFT using data format $2 \times 10 + 4$. $D = 1$ represent a hybrid floating-point architecture, whereas $D \to N_{\mathrm{FFT}}$ approaches the CBFP architecture. An optimal value can be found in between these architectures.

For radix-2^2 butterflies, (8) is only defined for odd values of i. This is compensated by a scale factor $\gamma = 3/2$ to include both delay feedback units in the radix-2^2 butterfly, as shown in Figure 4. The buffer input wordlength L differs between initial and internal butterflies. For every butterfly stage, α_i is chosen to minimize (8). For example, an 8192 point FFT using a hybrid floating-point format of $2 \times 10 + 4$ bits requires 16 Kb of memory in the initial butterfly for storing exponents, as shown in Figure 8. The number of memory elements for supporting dynamic scaling can be reduced to only 1256 bits by selecting a block size of 32, hence removing over 90% of the storage space for exponents. The hardware overhead is a counter to keep track of when to update the block exponent in the delay feedback, similar to the exponent control logic required in CBFP implementations. Thus the proposed co-optimization architecture supports hybrid floating-point on the input port at very low hardware cost.

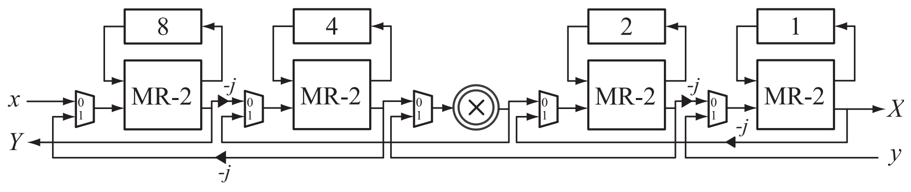


Figure 9: A bidirectional 16-point FFT pipeline. The input/output to the left is in sequential order. The input/output to the right is in bit-reversed order.

Since the input and output format is the same, this architecture then becomes suitable for 2D FFT computation.

3 Architectural Extensions

The architectures described in this paper have been extended with support for bidirectional processing, which is important for the intended application and also in many general applications. A pipeline FFT can support a bidirectional dataflow if all internal butterfly stages have the same wordlength. The advantage with a bidirectional pipeline is that input data can be supplied either in linear or bit-reversed sample order by changing the dataflow direction. One application for the bidirectional pipeline is to exchange the FFT/IFFT structure using reordering buffers in an OFDM transceiver to minimize the required buffering for inserting and removing the cyclic suffix, proposed in [76]. OFDM implementations based on CBFP have also been proposed in [74], but these solutions only operate in one direction since input and output format differ. Another application for a bidirectional pipeline is to evaluate 1D and 2D convolutions. Since the forward transform generates data in bit-reversed order, the architecture is more efficient if the inverse transform supports a bit-reversed input sequence as shown in Figure 9. Both input and output from the convolution are in linear sample order, hence no reorder buffers are required. The hardware requirement for a bidirectional pipeline is limited to multiplexers on the inputs of each butterfly and on each complex multiplier. Each unit requires 26 two-input muxes for internal $2 \times 11 + 4$ format, which is negligible compared to the size of an FFT stage.

4 Simulations

A simulation tool has been designed to evaluate different FFT architectures in terms of precision, dynamic range, memory requirements, and estimated chip size based on architectural descriptions. The user can specify the number of bits for representing mantissa M, exponents E, twiddle factors T, FFT size (N_{FFT}), rounding type and simulation stimuli. To make a fair comparison with related work, all architectures have been described and simulated in the developed tool.

First, we compare the proposed architectures with CBFP in terms of memory requirements and signal quality. In addition to the lower memory requirements, we will show how the co-optimized architecture produces a higher SQNR than CBFP. Secondly, we will compare the fabricated design with related work in terms of chip size and data throughput.

Table 1 shows a comparison of memory distribution between delay feedback units and intermediate buffers. 1D architectures have fixed-point input, whereas 2D architectures support hybrid floating-point input. The table shows that the intermediate buffers used in CBFP consume a large amount of memory, which puts the co-optimized architecture in favor for 1D processing. For 2D processing, the co-optimized architecture also has lower memory requirements than hybrid floating-point due to the buffer optimization. Figures 10 and 11 present simulation results for the 1D architectures in Table 1. Figure 10 is a simulation to compare SQNR when changing energy level in the input signal. In this case, the variations only affect CBFP$_{\text{low}}$ since scaling is applied later in the pipeline. Figure 11 shows the result when applying signals with a large crest factor, i.e. the ratio between peak and mean value of the input. In this case, both CBFP implementations are strongly affected due to the large block size in the beginning of the pipeline. Signal statistics have minor impact on the hybrid floating-point architecture since every value is scaled individually. The SQNR for the co-optimized solution is located between hybrid floating-point and CBFP since it uses a relatively small block size.

Table 1: Memory requirements in Kbits for pipeline architectures, based on a 2048 point radix-2^2 with $M = 10$ and $E = 4$.

Architecture	Delay feedback	Intermediate buffers	Total memory
1D Co-optimization	45.8	1.6	47.4
1D Hybrid FP (A)	49.0	-	49.0
1D CBFP$_{\text{low}}$	45.7	14.7	60.4
1D CBFP	45.7	60.0	105.7
2D Co-optimization(C)	50.0	0.4	50.4
2D Hybrid FP (B)	53.9	-	53.9

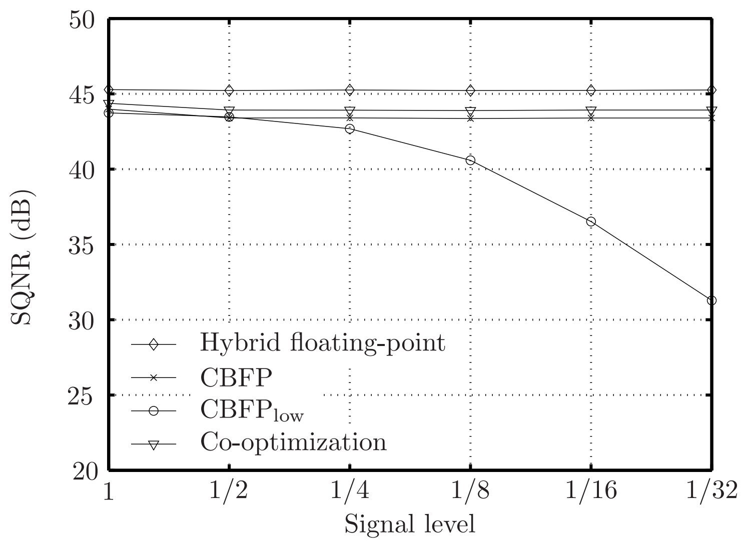


Figure 10: Decreasing the energy in a random value input signal affects only the architecture when scaling is not applied in the initial stage. Signal level=1 means utilizing the full dynamic range.

Table 2 shows an extended comparison between the proposed architectures and related work. The table includes two pipeline architectures using hybrid floating-point, for 1D signal processing (A) using a tailored datapath for exponent bits $E = 0 \ldots 4$, and for 2D signal processing (B) using a constant number of exponent bits $E = 4$. Then the proposed co-optimized architecture for 2D signal processing (C), with a reduced hardware cost more comparable to the 1D hybrid floating-point implementation. It uses block scaling in the initial butterfly unit and then hybrid floating-point in the internal butterfly to show the low hardware cost for extending architecture (A) with support for 2D processing.

The parallel architecture proposed by Lin *et al.* [90] uses block floating point with a block size of 64 elements. The large block size affects the signal quality, but with slightly lower memory requirements compared to pipeline architectures. A pipeline architecture proposed by Bidet *et al.* [73] uses convergent block floating point with a multi-path delay commutator. The memory requirements are high due to the intermediate storage of data in the pipeline, which significantly affects the chip area. However, CBFP generates a higher SQNR than traditional BFP. The pipeline architecture proposed by Wang *et al.* [91]

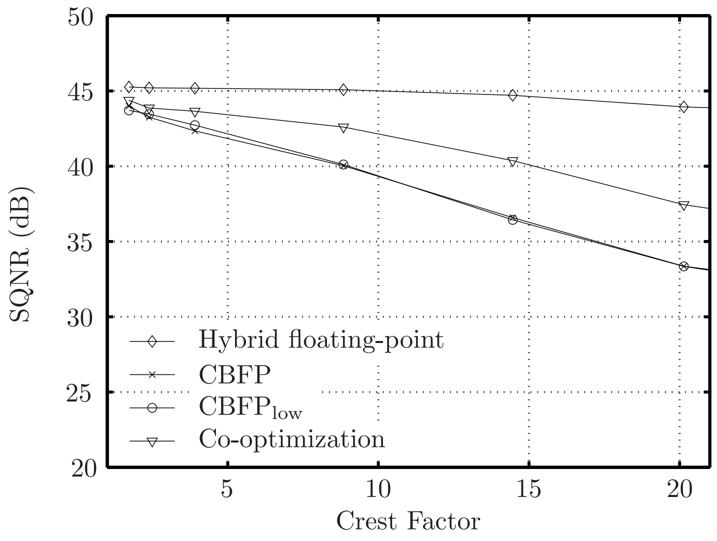


Figure 11: Decreasing the energy in a random value input signal with peak values utilizing the full dynamic range. This affects all block scaling architectures, and the SQNR depends on the block size. The co-optimized architecture performs better than convergent block floating-point, since it has a smaller block size through the pipeline.

does not support scaling and is not directly comparable in terms of precision since SQNR depends on the input signal. The wordlength increases gradually in the pipeline to minimize the quantization noise, but this increases the memory requirements or more important the wordlength in arithmetic components and therefore also the chip area.

The proposed architectures have low hardware requirements and produce high SQNR using dynamic data scaling. They can easily be adapted to 2D signal processing, in contrast to architectures without data scaling or using CBFP. The pipeline implementation results in a high throughput by continuous data streaming, which is shown as peak performance of 1D transforms in Table 2.

5 **VLSI Implementation**

A 2048 complex point pipeline FFT core using hybrid floating-point and based on the radix-2^2 decimation-in-frequency algorithm [71] has been designed, fabricated, and verified. This section presents internal building blocks and measurements on the fabricated ASIC prototype.

The butterfly units calculate the sum and the difference between the input sequence and the output sequence from the delay feedback. Output from the butterfly connects to the complex multiplier, and data is finally normalized and sent to the next FFT stage. The implementation of the delay feedbacks is a main consideration. For shorter delay sequences, serially connected flip-flops are used as delay elements. As the number of delay elements increases, this approach is no longer area and power efficient. One solution is to use SRAM and to continuously supply the computational units with data, one read and one write operation have to be performed in every clock cycle. A dual port memory approach allow simultaneous read and write operations, but is larger and consumes more energy per memory access than single port memories. Instead two single port memories, alternating between read and write each clock cycle could be used. This approach can be further simplified by using one single port memory with double wordlength to hold two consecutive values in a single location, alternating between reading two values in one cycle and writing two values in the next cycle. The latter approach has been used for delay feedback exceeding the length of eight values. An area comparison can be found in [9].

A 2048 point FFT chip based on architecture (A) has been fabricated in a $0.35\,\mu m$ 5ML CMOS process from AMI Semiconductor, and is shown in Figure 12. The core size is $2632 \times 2881\,\mu m^2$ connected to 58 I/O pads and 26 power pads. The implementation requires 11 delay feedback buffers, one for each butterfly unit. Seven on-chip RAMs are used as delay buffers (approximately 49 K bits), while the four smallest buffers are implemented using flip-flops. Twiddle factors are stored in three ROMs containing approximately 47 K bits. The memories can be seen along the sides of the chip. The number of equivalent gates (2-input NAND) is 45900 for combinatorial area and 78300 for non-combinatorial area (including memories). The power consumption of the core was measured to 526 mW when running at 50 MHz and using a supply voltage of 2.7 V. The pipeline architecture produces one output value each clock cycle, or 37K transforms per second running at maximum clock frequency. The 2D FFT architecture (B) has been implemented on FPGA in [92].

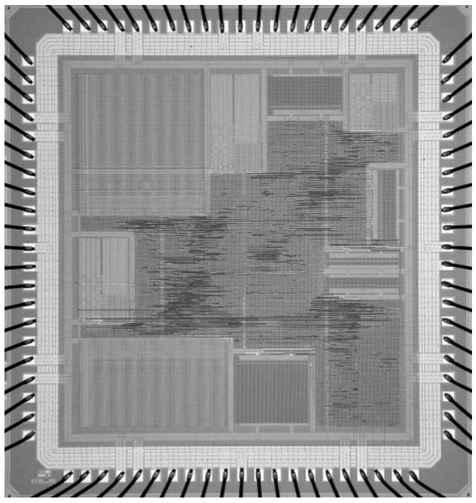


Figure 12: Chip photo of the 2048 complex point FFT core fabricated in a 0.35 μm 5ML CMOS process. The core size is $2632 \times 2881\,\mu m^2$.

6 Conclusion

Dynamic data scaling architectures for pipeline FFTs have been proposed for both 1D and 2D applications. Based on hybrid floating-point, a high-precision pipeline with low memory and arithmetic requirements has been constructed. A co-optimization between hybrid floating-point and block floating-point has been proposed, reducing the memory requirement further by adding small intermediate buffers. A 2048 complex point pipeline FFT core has been implemented and fabricated in a 0.35 μm 5ML CMOS process, based on the presented scaling architecture and a throughput of 1 complex point/cc. The bidirectional pipeline FFT core, intended for image reconstruction in digital holography, has also been integrated on a custom designed FPGA platform to create a complete hardware accelerator for digital holographic imaging.

Table 2: Comparison between proposed architectures and related work. The values based on simulated data are highlighted by grey fields.

	Proposed (A)		Proposed (B)	Proposed (C)	Lin [90]	Bidet [73]	Wang [91]
Architecture	Pipeline (1D)		Pipeline (2D)	Pipeline (2D)	Parallel	Pipeline	Pipeline
Dynamic scaling	Hybrid FP		Hybrid FP	Co-optimized	BFP $D = 64$	CBFP	No
Technology (μm)	0.35		Virtex-E	Virtex-E	0.18	0.5	0.35
Max Freq. (MHz)	76		50	50	56	22	16
Input wordlength	2×10		$2 \times 10 + 4$	$2 \times 10 + 4$	2×10	2×10	2×8
Internal wordlength	$2 \times 11 + (0 \ldots 4)$		$2 \times 11 + 4$	$2 \times 11 + 4$	$2 \times 11 + 4$	$2 \times 12 + 4$	$2 \times (19 \ldots 34)$
Transform size	2K	1/2/4/8K	2K	2K	8K	8K	2/8K
SQNR (dB)	45.3	44.0	45.3	44.3	41.2	42.4	
Memory (bits)	49K	196K	53.9K	50.4K	185K	350K	213K
Norm. Area (mm^2)[1]	7.58	≈ 16			18.3	49	33.75
1D Transform/s	37109	9277	24414	24414	3905	2686	7812/1953

[1] Area normalized to $0.35\,\mu$m technology.

Part III

A Design Environment and Models for Reconfigurable Computing

Abstract

System-level simulation and exploration tools are required to rapidly evaluate system performance early in the design phase. The use of virtual platforms enables hardware modeling as well as early software development. An exploration framework (Scenic) is proposed, which is based on OSCI SystemC and consists of a design exploration environment and a set of customizable simulation models. The exploration environment extends the SystemC library with features to construct and configure simulation models using an XML description, and to control and extract performance data using run-time reflection. A set of generic simulation models have been developed, and are annotated with performance monitors for interactive run-time access. The Scenic framework is developed to enable design exploration and performance analysis of reconfigurable architectures and embedded systems.

Based on: T. Lenart, H. Svensson, and V. Öwall, "Modeling and Exploration of a Reconfigurable Architecture for Digital Holographic Imaging," in *Proceedings of IEEE International Symposium on Circuits and Systems*, Seattle, USA, May 2008.

and: T. Lenart, H. Svensson, and V. Öwall, "A Hybrid Interconnect Network-on-Chip and a Transactional Level Modeling approach for Reconfigurable Computing," in *Proceedings of IEEE International Symposium on Electronic Design, Test and Applications*, Hong Kong, China, Jan. 2008.

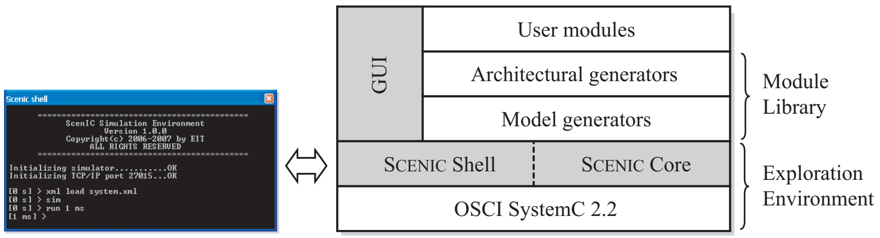
1 Introduction

Due to the continuous increase in design complexity, system-level exploration tools and methodologies are required to rapidly evaluate system behavior and performance. An important aspect for efficient design exploration is the design methodology, which involves the *construction* and *configuration* of the system to be simulated, and the *controllability* and *observability* of simulation models.

SystemC has shown to be a powerful system-level modeling language, mainly for exploring complex system architectures such as System-on-Chip (SOC), Network-on-Chip (NoC) [93], multi-processor systems [94], and run-time reconfigurable platforms [95]. The Open SystemC Initiative (OSCI) maintains an open-source simulator for SystemC, which is a C++ library containing routines and macros to simulate concurrent processes using an HDL like semantic [96]. Systems are constructed from SystemC *modules*, which are connected to form a design hierarchy. SystemC modules encapsulate processes, which describe behavior, and communicate through *ports* and *channels* with other SystemC modules. The advantages with SystemC, besides the well-known C++ syntax, include modeling at different abstraction levels, simplified hardware/software co-simulation, and a high simulation performance compared to traditional HDL. The abstraction levels range from cycle accurate (CA) to transaction level modeling (TLM), where abstract models trade modeling accuracy for a higher simulation speed [42].

SystemC supports observability by tracing signals and transactions using the SystemC verification library (SCV), but it only provides limited features using trace-files. Logging to trace-files is time-consuming and requires post-processing of extracted simulation data. Another drawback is that the OSCI simulation kernel does not support real-time control to allow users to start and stop the simulation interactively. Issues related to system construction, system configuration, and controllability are not addressed.

To cover the most important aspects on efficient design exploration, a SystemC Environment with Interactive Control (Scenic) is proposed. Scenic is based on OSCI SystemC 2.2 and extends the SystemC library with functionality to construct and configure simulations from eXtensible Markup Language (XML), possibilities to interact with simulation models during run-time, and the ability to control the SystemC simulation kernel using *micro-step* simulation. Scenic extends OSCI SystemC *without modifying* the core library, hence proposing a non-intrusive exploration approach. A command shell is provided to handle user interaction and a connection to a graphical user interface. In addition, a library of customizable simulation models is developed, which contains commonly used building blocks for modeling embedded systems. The models are used in Part IV to simulate and evaluate reconfigurable architec-



```
        ==========================================
               ScenIC Simulation Environment
                      Version 1.0.0
               Copyright(c) 2006-2007 by EIT
                   ALL RIGHTS RESERVED
        ==========================================

Initializing simulator...........OK
Initializing TCP/IP port 27015...OK

[0 s] > xml load system.xml
[0 s] > sim
[0 s] > run 1 ms
[1 ms] >
```

Figure 1: The Scenic environment extends OSCI SystemC with functionality for system exploration (core) and user interaction (shell). Model generators are basic building blocks for the architectural generators, which are used to construct complex simulations.

tures. The Scenic exploration framework is illustrated in Figure 1, where the Scenic *core* implements SystemC extensions and the Scenic *shell* provides an interface for user interaction.

Section 2 presents related work on existing platforms for design exploration. This is followed by the proposed Scenic exploration environment in Section 3, which is divided into scripting environment, simulation construction, simulation interaction, and code generation. Section 4 propose model generators and architectural generators to construct highly customizable processor and memory architectures. The use of the Scenic framework for system design is presented in Part IV, where a platform for reconfigurable computing is proposed. More detailed information on the Scenic exploration environment can be found in Appendix A.

2 Related Work

Performance analysis is an important part of design exploration, and is based on extracted performance data from a simulation. Extraction of performance data requires either the use of trace files, for *post-processing* of simulation data, or *run-time access* to performance data inside simulation models. The former approach is not suitable for interactive performance exploration due to the lack of observability during simulation, and also has a negative impact on simulation performance. The latter approach is a methodology referred to as data introspection. Data introspection is the ability to access run-time information using a reflection mechanism. The mechanism enables either *structural reflection*, to expose the design hierarchy, or *run-time reflection* to extract performance data and statistics for performance analysis.

General frameworks to automatically reflect run-time information in software are presented in [97] and [47]. While this approach works well for standard data types, performance data is highly *model-specific* and thus can not be automatically annotated and reflected. Dust is another approach to reflect structural and run-time information [98]. Dust is a Java visualization front-end, which captures run-time information about data transactions using SCV. However, performance data is not captured, and has to be evaluated and analyzed from recorded transactions. Design structure and recorded transactions are stored in XML format and visualized in a structural view and a message sequence chart, respectively. Frameworks have also been presented for modeling at application level, where models are annotated with performance data and trace files are used for performance analysis [15]. However, due to the use of more abstract simulation models, these environments are more suitable for software evaluation than hardware exploration.

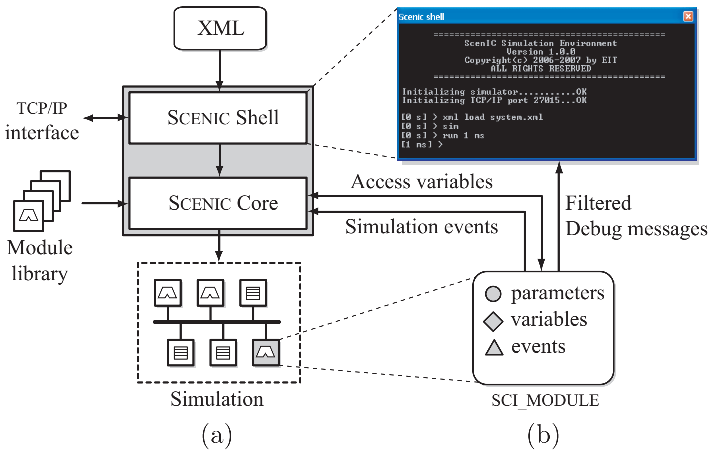
The use of structural reflection have been proposed in [99] and [100] to generate a visual representation of the simulated system. This can be useful to ensure structural correctness and provide greater understanding about the design hierarchy, but does not provide additional design information to enable performance analysis. In contrast, it is argued that structural reflection should instead be used to translate the design hierarchy into a user-specified format to enable automatic code generation.

Related work only covers a part of the functionality required for efficient design exploration. This work proposes an exploration environment, Scenic, that supports *simulation construction* using XML format, *simulation interaction* using run-time reflection, and *code generation* using structural reflection.

3 Scenic Exploration Environment

Design exploration is an important part of the hardware design flow, and requires tools and models that allow rapid simulation construction, configuration, and evaluation. The Scenic exploration environment is proposed to address these important aspects. Scenic is a system exploration tool for hardware modeling, and use simulation models annotated with *user-defined* performance monitors to capture and reflect *relevant* information during simulation-time.

Figure 2(a) illustrates the Scenic exploration flow from simulation construction, using XML format, to an interactive SystemC simulation model. The Scenic environment is divided into two tightly-coupled parts: the Scenic core and the Scenic shell. The Scenic core handles run-time interaction with simulation models, and the Scenic shell handles the user interface and scripting environment. Figure 2(b) illustrates user interaction during run-time, to configure and observe the behavior of simulation models.



```
        ============================================
              ScenIC Simulation Environment
                      Version 1.0.0
              Copyright(c) 2006-2007 by EIT
                   ALL RIGHTS RESERVED
        ============================================

Initializing simulator...........OK
Initializing TCP/IP port 27015...OK

[0 s] > xml load system.xml
[0 s] > sim
[0 s] > run 1 ms
[1 ms] >
```

Figure 2: (a) A simulation is created from an XML design specification using simulation models from a module library. (b) The models interact with the SCENIC shell using access variables, simulation events, and filtered debug messages.

SCENIC is characterized by the following modeling and implementation properties:

- SCENIC supports mixed application domains and abstraction levels, and is currently evaluated in the field of embedded system design and reconfigurable computing using transaction level modeling.

- SCENIC presents a non-intrusive approach that does not require the OSCI SystemC library to be changed. Furthermore, port or signal types are not modified or extended. This enables interoperability and allows integration of any simulation module developed in SystemC.

SCENIC interacts with simulation models during simulation-time, which is implemented by extending the OSCI SystemC *module* class with run-time reflection. To take advantage of the SCENIC extensions, the only required change is to use SCI_MODULE, which encapsulates functionality to access the simulation models from the SCENIC shell. SCENIC can still simulate standard OSCI SystemC modules without any required modifications to the source code, but they will not be visible from the SCENIC shell. Compatibility is an important aspect since it allows integration and co-simulation with existing SystemC models.

The class hierarchy and the extended functionality provided by SCI_MODULE is shown in Figure 3. SCI_MODULE is an extension of SCI_ACCESS and the

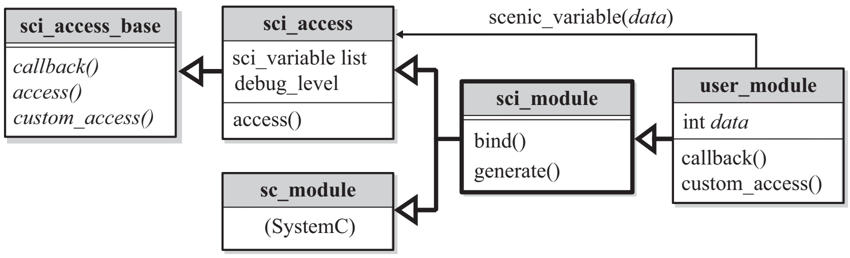


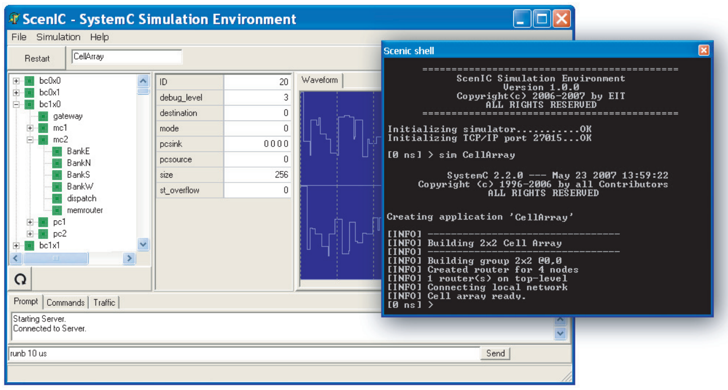
Figure 3: Class hierarchy for constructing simulation models. The access variables are stored in SCI_ACCESS and exported to the SCENIC shell. The custom access function enable user defined commands to be implemented.

SystemC SC_MODULE classes. SCI_MODULE supports automatically *binding* of simulation models and *code generation* in any structural language format. SCI_ACCESS handles *debug message filtering*, run-time reflection, and *custom access* functions. Debug message filtering enables the user to set a debug level on which messages are produced from each simulation model, and custom access methods provide an interface to implement *user-defined* shell commands for each simulation model. The SystemC extensions provide means for more effective simulation and exploration, and are briefly described below:

- **Scripting Environment** - The SCENIC shell provides a powerful scripting environment to control simulations and to interact with simulation models. The scripting environment supports features to define user scenarios by interactively configuring the simulation models. It can also be used to access and process performance data that is captured inside each model during simulation. The scripting environment is presented in Section 3.1.

- **Simulation Construction** - The simulation *construction* and static *configuration* is specified in XML format. Hence, simulation models are dynamically created, connected and configured from a single design specification language. This enables rapid simulation construction since no compilation processes is involved. Simulation construction is presented in Section 3.2.

- **Simulation Interaction** - An extension to access internal data structures inside simulation models is proposed. This allows simulation models to be dynamically *controlled*, to describe different simulation scenarios,



ID	20
debug_level	3
destination	0
mode	0
pcsink	0 0 0 0
pcsource	0
size	256
st_overflow	0

```
Starting Server.
Connected to Server.
```

```
runb 10 us
```

```
          ============================================
               ScenIC Simulation Environment
                       Version 1.0.0
               Copyright(c) 2006-2007 by EIT
                    ALL RIGHTS RESERVED
          ============================================

Initializing simulator...........OK
Initializing TCP/IP port 27015...OK

[0 ns] > sim CellArray

             SystemC 2.2.0 --- May 23 2007 13:59:22
        Copyright (c) 1996-2006 by all Contributors
                    ALL RIGHTS RESERVED

Creating application 'CellArray'

[INFO] ---------------------------------
[INFO] Building 2x2 Cell Array
[INFO] ---------------------------------
[INFO] Building group 2x2 @0,0
[INFO] Created router for 4 nodes
[INFO] 1 router(s) on top-level
[INFO] Connecting local network
[INFO] Cell array ready.
[0 ns] >
```

Figure 4: Graphical user interface connecting to SCENIC using a TCP/IP socket. It is used to navigate through the design hierarchy, modify access variables, and to view data captured over time. The SCENIC shell is shown on top of the graphical user interface.

and *observed* to extract performance data during simulation. Simulation interaction is presented in Section 3.3.

- **Code Generation** - Simulation models provide an interface to generate structural information in a user-specific language. This allows the design hierarchy to be visually represented [101], or user-developed hardware models to be constructed and configured in HDL syntax. Hence, it is possible to generate structural VHDL directly from the simulation models to support component instantiation and parameterization, port mapping, and signal binding. Code generation is presented in Section 3.4.

3.1 Scripting Environment

The SCENIC environment is a user-interface to construct, control, and interact with a simulation. It provides a command line interface (the SCENIC shell) that facilitates advanced scripting, and supports an optional TCP/IP connection to a graphical user interface. The SCENIC shell and graphical user interface are shown in Figure 4. The graphical user interface includes a tree view to navigate through the design hierarchy (structural reflection) and provides a list view with accessible variables and performance data inside each simulation module (run-time reflection). Variable tracing is used to capture trends over time, which is presented in a wave form window.

Built-in commands are executed from the Scenic shell command line interface by typing the command name followed by a parameter list. Built-in commands include for example setting environment variables, evaluate basic arithmetic or binary operations, and random number generation. If the *name* of a SystemC module is given instead of a Scenic shell command, then the command is forwarded and evaluated by that module's *custom access* method. The interface enables fast scripting to setup different scenarios in order to evaluate system architectures. The Scenic shell commands are presented in detail in Appendix A.

Simulation Control

OSCI SystemC currently lacks the possibility to enable the user to interactively control the SystemC simulation kernel, and simulation execution is instead directly described in the source code. Hence, changes to simulation execution require the source code to be recompiled, which is a non-desirable approach to control the simulation kernel. In contrast, Scenic addresses the issue of simulation control by introducing *micro-step* simulation. Micro-step simulation allows the user to interactively pause the simulation, to modify or view internal state, and then resume execution. A micro-step is a user-defined duration of simulation time, during which the SystemC kernel simulates in *blocking* mode. The micro-steps are repeated until the total simulation time is completed. From a user-perspective, the simulation appears to be *non-blocking* and fully interactive. The performance penalty for using micro-steps is evaluated to be negligible down to clock cycle resolution.

From the Scenic shell, simulations are controlled using either blocking or non-blocking *run* commands. Non-blocking commands are useful for interactive simulations, whereas blocking commands are useful for scripting. A *stop* command halts a non-blocking simulation at the beginning of the next micro-step.

3.2 Simulation Construction

SystemC provides a pre-defined C-function (*sc_main*) that is called during elaboration-time to instantiate and bind the top-level simulation models. A change to the simulation architecture requires the source files to be updated and re-compiled, which makes design exploration more difficult. In Scenic, the use of XML is proposed to construct and configure a simulation. XML is a textural language that is suitable to describe hierarchical designs, and has been proposed as an interchangeable format for ESL design by the Spirit Consortium [102]. The use of XML for design specification also enables the possibility to co-simulate multiple projects, since several XML files can be overloaded in

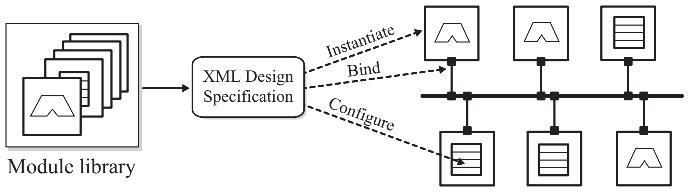


Figure 5: XML is used for design specification, to instantiate and configure simulation models from the Scenic module library. Automatic port binding enables models and designs to be connected.

Scenic to construct larger simulations. By using multiple design specifications, parts of the simulation may be substituted with abstract models to gain simulation performance during the exploration cycle. Once the individual parts of the system have been verified, a more detailed and accurate system simulation can be executed.

Simulation models are instantiated from the Scenic *module library*, as previously illustrated in Figure 2(a). User-models that are based on sci_module are automatically registered during start-up and stored in the module library. A design specification in XML format specifies instantiations, bindings, and configuration of simulation models from the module library, as illustrated in Figure 5. All models based on sci_module use the same constructor arguments, which simplify the instantiation and configuration process. Configuration parameters are transferred from the XML specification to the models using a data structure supporting *arbitrary data types*. Hence, models receive and extract XML parameters to obtain a user-defined *static* configuration.

3.3 Simulation Interaction

Run-time reflection is required to dynamically interact with simulation models and to access internal information inside each model during simulation time. It is desirable to be able to access internal member variables inside each simulation model, and it is also valuable to trace such information over time. Hence, *access variables* are proposed, which makes it possible to access member variables from the Scenic shell. Access variables may also be *recorded* during simulation, and the information can be retrieved from the Scenic shell to study trends over time. Finally, *simulation events* are proposed and provides the functionality to notify the simulator when user-defined events occur. Simulation events are introduced to allow a more interactive simulation.

Access Variables

Member variables inside an SCI_MODULE are exported to the Scenic shell as configurable and observable *access variables*. This reflects the variables data type, data size and value during simulation time. Access variables enable dynamic configuration (controllability) and extraction of performance data and statistics during simulation (observability). Hence, the user can configure models for specific simulation scenarios, and then observe the corresponding simulation effects.

Performance data and statistics that are represented by native and *trivial* data types can be directly reflected using an access variable. An example is a memory model that contains a counter to represents the total number of memory accesses. However, more complex operations are required when performance values are *data* or *time* dependent. This requires functional code to evaluate and calculate a performance value, and is supported in Scenic by using variable *callbacks*. When an access variable is requested, the associated callback function is evaluated to assign a new value to the member variable. This new value is then reflected in the Scenic shell. The callback functionality can also be used for the reverse operation of assigning values that affect the model functionality or configuration. An example is a variable that represents a memory size, which requires the memory array to be resized (reallocated) when the variable change. The following code is a trivial example on how to export member variables and how to implement a custom callback function:

```
class ScenicModule : public sci_module {                 /* extended module      */
private:
  void*   m_mem;                                         /* pointer to data      */
  short   m_size;                                        /* scalar data type     */
  double  m_average;                                     /* scalar data type     */
  int     m_data[5];                                     /* static vector        */
public:
  ScenicModule(sc_module_name nm, SCI_PARAMS& params) :
    sci_module(nm, params)
  {
    scenic_variable(m_data,    "data");                  /* variable "data"      */
    scenic_callback(m_size,    "size");                  /* callback "size"      */
    scenic_callback(m_average, "average");               /* callback "average"   */
  }

  virtual SCI_RETURN callback(SCI_OP op, string name) {  /* callback function    */
    if (name == "size")
      if (op == WRITE) m_mem = realloc(m_mem, m_size);   /* realloc "size" bytes */
    if (name == "average")
      if (op == READ)  m_average = average(m_data);      /* return average value */
  }
};
```

Modifying the value of the *size* variable triggers the callback function to be executed and the memory to be reallocated. The variable *average* is also

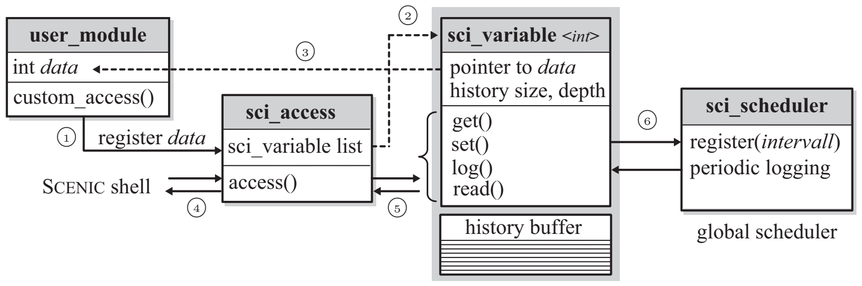


Figure 6: A local variable is first registered, which creates a SCI_VARIABLE object pointing to the variable. It contains features to create a history buffer and to register a periodic logging interval with a global scheduler to capture the value of a variable at constant intervals. The user interface can access a variable through the SCI_ACCESS object, which contains a list of all registered SCI_VARIABLE objects.

evaluated on demand, while the *data* vector is constant and reflects the current simulation value. Registered variables are accessed from the SCENIC shell as:

```
[0 ns]> ScenicModule set -var size -value 256
[0 ns]> ScenicModule set -var data -value "4 7 6 3 8"
[0 ns]> ScenicModule get
  data     5x4 [0]     { 4 7 6 3 8 }
  average  1x8 [0]     { 5.6 }
  size     1x2 [0]     { 256 }
```

Figure 6 shows how a user-defined simulation model exports a member variable named *data* ①, which is encapsulated by creating a new SCI_VARIABLE ② that contains a reference to the actual member variable ③. SCI_VARIABLE provides functionality to enable for example *get* and *set* operations on arbitrary data types using standard C++ `iostreams`. Hence, the stream operators (`<<` and `>>`) must be supported for all access variable types. The SCENIC shell obtains information about variables by calling the *access* method in SCI_ACCESS ④. The call is responded by the SCI_VARIABLE that is associated with the requested access variable ⑤.

Access Variable Tracing

It is desirable to be able to study trends over time and to trace the values of access variables. Hence, the value of an access variable may be periodically logged and stored into a *history buffer* during simulation. The periodical time *interval* and history buffer *depth* are configurable parameters from the SCENIC shell, which is also used to request and reflect data from history buffers. However,

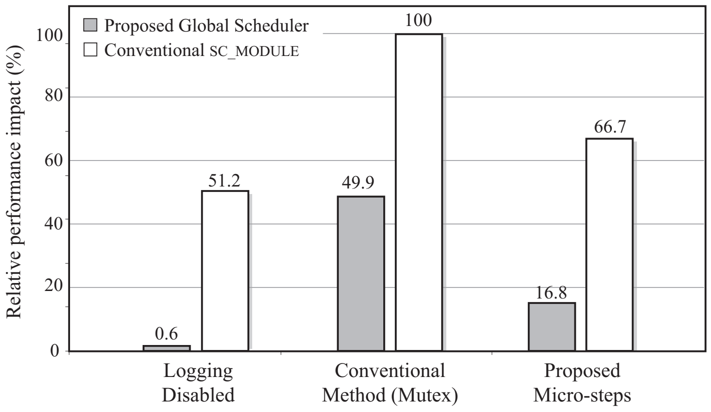


Figure 7: The impact on simulation performance using SC_MODULE to support SCI_VARIABLE logging and when using a global scheduler to reduce context switching in the simulation kernel. The graph represents relative performance overhead.

periodical logging requires the variable class to be aware of SystemC simulation time. Normally, this would require each SCI_VARIABLE to inherit from SC_MODULE and to create a separate SystemC process. This would consume a substantial amount of simulation time due to increased context switching in the SystemC kernel.

To reduce context switching, an implementation based on a *global scheduler* is proposed, which handles logging for all access variables inside all simulation models. The global scheduler, shown in Figure 6, is configured with a *periodical time intervall* from the SCENIC shell through a SCI_VARIABLE ⑥. A global scheduler improves the simulation performance since it is only invoked *once* for each time event that requires variable logging. Hence, multiple variables are logged in a single context switch.

To evaluate the performance improvement, a comparison between using a conventional SC_MODULE module and the proposed global scheduler is shown in Figure 7. In the former approach there is large performance penalty even if logging is disabled, due to the increased number of SystemC processes. For the latter approach, there is negligible simulation overhead when logging is turned off. When logging is enabled, two approaches are presented. The first simulation is based on a conventional approach when using standard macros (mutex) to enable mutually exclusive data access. These macros prevent simultaneous access to history buffers to protect the thread communication between the simulation kernel and the SCENIC shell. The second approach propose a method

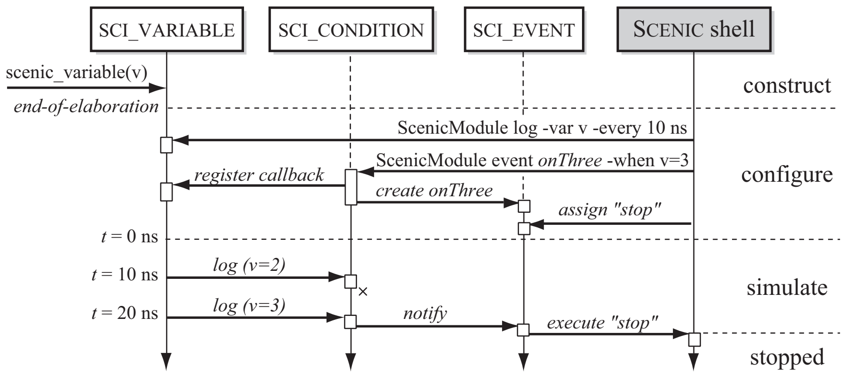


Figure 8: The function call flow to create a dynamic simulation event from the SCENIC shell. A condition is created and registered with an access variable. When a simulation condition is satisfied, the associated event is triggered.

based on atomic steps (micro-steps), discussed in Section 3.1, which prevents race conditions between the simulation thread and the SCENIC shell. As shown, the proposed implementation based on a global scheduler and atomic simulation steps results in lower overhead compared to conventional approaches.

Simulation Events

During simulations, it is valuable to automatically receive information from simulation models regarding simulation status. This information could be informative messages on when to read out generated performance data, or warning messages to enable the user to trace simulation problems at an exact time instance. Instead of printing debug messages in the SCENIC shell, it is desirable to be able to automatically halt the simulation once *user-defined events* occur.

An extension to the access variables provides the ability to combine data logging with conditional simulation events. Hence, dynamic *simulation events* are proposed, which can be configured to execute SCENIC commands when triggered. In this way, the simulation models can notify the simulator of specific events or conditions on which to observe, reconfigure, or halt the simulation.

A simulation event is a SCI_VARIABLE that is assigned a user-specified SCENIC shell command. Simulation events are created dynamically during runtime from the SCENIC shell to notify the simulator when a boolean condition associated with an access variable is satisfied. The condition is evaluated on data inside the history buffer, hence repeated assignments between clock cycles are effectively avoided.

The internal call graph for creating a simulation event is shown in Figure 8. An access variable is first configured with a periodical logging intervall on which to evaluate a *boolean condition*. A boolean condition is created from the Scenic shell, which registers itself with the access variable and creates a new *simulation event*. This simulation event is configured by the user to evaluate a Scenic shell command when triggered. Every time the access variable is logged, the condition is evaluated against the assigned boolean expression. If the condition is true, the associated event is triggered. The event executes the user command, which in this case is set to halt the simulation. Hence, the simulation events operate in a similar way as software assertions, but are dynamically created during simulation time.

3.4 **Code Generation**

Simulation models provide an interface to generate structural information in a user-specific language, where Scenic currently supports DOT and VHDL formats. The code generators are executed from the Scenic shell by specifying the *format* and a *top-level*. A code generator use multiple passes to traverse all hierarchical modules from the top-level to visit each simulation module. Modules can implement both *pre_visitors* and *post_visitors*, depending on in which order the structural information should be produced. In addition, the code generator divides a structural description into separate *files* and *sections*, which are user-defined to represent different parts of the generated code. For example, a VHDL file is divided into sections representing *entity*, *architecture*, *type declarations*, *processes*, and *concurrent statements*.

A DOT generator is implemented to create a visual representation of a system design and is used to verify the connectivity and design hierarchy for the complex array architectures presented in Part IV. DOT is a textural language describing *nodes* and *edges*, which is used to represent graphs and hierarchical relations [101]. A DOT specification is converted into a visible image using open-source tools.

A VHDL generator is implemented to generate structural information to instantiate and parameterize components, and to generate signal and port bindings. Hence, only the behavioral description of each simulation model has to be translated. The generator produces a VHDL netlist, where the modules are instantiated, parameterized, and connected according to a Scenic XML design specification and dynamic configuration generated by architectural generators, presented in Section 4.3. Hence, all manual design steps between system specification and code generation have been effectively removed.

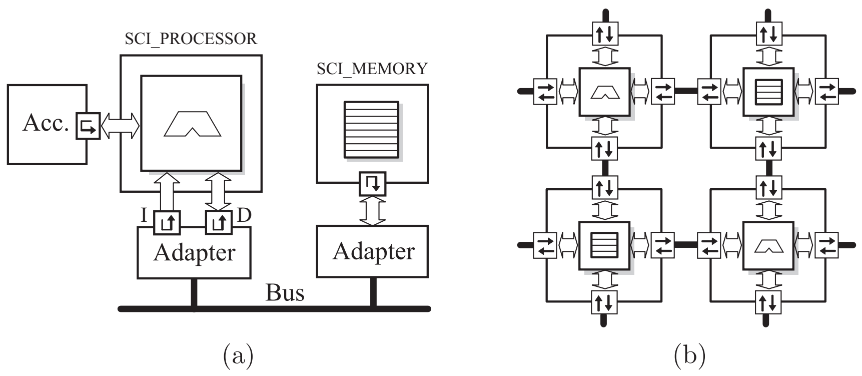


Figure 9: (a) Constructing an embedded system using the processor and memory generators. The processor uses the external memory interface to communicate with a system bus, and an I/O register to communicate with an external hardware accelerator. (b) Constructing a processor array using the model and architectural generators. The array elements are connected using bi-directional I/O registers.

4 Scenic Model and Architectural Generators

The Scenic exploration environment allows efficient hardware modeling by combining rapid simulation *construction* with simulation *interaction.* Simulation interaction requires simulation models that are highly configurable and that export performance data and statistics to the Scenic shell. Hence, this section presents simulation primitives developed to enable design exploration.

A set of generic models are developed to provide commonly used building blocks in digital system design. The models enable design exploration by exporting configuration parameters and exposing performance metric to the Scenic shell. Hence, models for design exploration are characterized by *scalability* (static parameterization), *configurability* (dynamic parameterization), and a high level of *observability.* Scalability enables model re-use in various parts of a system, while configurability allows system tuning. Finally, observability provides feedback about the system behavior and performance.

The module library contains highly configurable and reusable simulation models, referred to as *model generators.* Two model generators have been developed to simplify the modeling of digital systems, namely the *processor generator* and the *memory generator.* The processor generator constructs a custom processor model based on a user-defined architectural description, while the memory generator constructs a custom memory model based on user-defined timing parameters. The module library also contains several *architectural gen-*

Table 1: Highly customizable model generators for rapid design exploration.

Generator	Scalability	Configurability	Observability
Processor	- Wordlength - Instruction set - Registers - Memory I/F	Program code	- Execution flow - Registers & Ports - Memory access
Memory	- Wordlength - Memory size	Memory timing	- Bandwidth/Latency - Access pattern - Memory contents

erators. An architectural generator constructs more complex simulation objects, such as Network-on-Chip (NoC) or embedded system architectures.

The model and architectural generators are used in Part IV to construct a dynamically reconfigurable architecture, and are further described in the following sections. Figure 9(a) illustrates an embedded system, where the processor and memory generators have been used to construct the main simulation models. The models communicate with the system bus using adaptors, which translate a *model-specific* interface to a *user-specific* interface. An array of processors and memories are shown in Figure 9(b) and illustrates the architectures presented in Part IV.

4.1 Processor Generator

The Scenic module library includes a processor generator to create custom instruction-set simulators (ISS) from a user-defined architectural description. An architectural description specifies the processor data wordlength, instruction wordlength, internal registers, instruction set, and external memory interface, where the properties are summarized in Table 1. Furthermore, a generated processor automatically provides a code generator (assembler) that translates processor-specific assembly instructions into binary format.

Generated processors are annotated with access variables to extract performance data and statistics during simulation time. The performance data and statistics include processor utilization, number of executed and stalled instructions, and internal register values. Logging of performance data allows designers to study processor execution over time, and captured statistics are valuable for high-accuracy software profiling.

Processor Registers

The *register bank* is user-defined, thus each generated processor has a different register configuration. However, a register holding the program counter is automatically created to control the processor execution flow.

There are two types of processor registers: *internal* general-purpose registers and *external* I/O port registers. General-purpose registers hold data values for local processing, while I/O port registers are bi-directional interfaces for external communication. I/O port registers are useful for connecting external co-processors or hardware accelerators to a generated processor. Each uni-directional link uses flow control to prevent the processor from accessing an empty input link or a full output link. Similar port registers are found in many processor architectures, for example the IBM PowerPC [103].

Processor Instructions

The *instruction set* is also user-defined, to emulate arbitrary processor architectures. An instruction is constructed from the SCI_INSTRUCTION class, and is based on an *instruction template* that describes *operand fields* and *bitfields*. The operand fields specify the number of destination registers (D), source registers (S), and immediate values (I). The bitfields specifies the number of bits to represent opcode (OP), operand fields, and instruction flags (F). Figure 10 illustrates how a generic instruction format is used to represent different instruction templates, based on $\{\mathrm{OP}, D, S, I, \mathrm{F}\}$. An instruction template has the following format:

```
class TypeB : public sci_instruction {                /* instruction template */
public:
    TypeB(string nm) : sci_instruction(nm,            /* constructor          */
      "D=1,S=1,I=1",                                  /* operand fields       */
      "opcode=6,dst=5,src=5,imm=16,flags=0")          /* bitfields            */
    { }
};
```

In this example the opcode use 6 bits, source and destination registers use 5 bits each, the immediate value use 16 bits, and the instruction flags are not used. Templates are reused to group similar instructions, i.e. those represented with the same memory layout. For example, arithmetic instructions between *registers* are based on one template, while arithmetic instructions involving *immediate* values are based on another template.

A user-defined instruction inherit properties from an instruction template, and extends it with a *name* and an *implementation*. The name is used to reference the instruction from the assembly source code, and the implementation specifies the instructions functionality. An *execute* method describes the functionality by reading processor registers, performing a specific operation, and

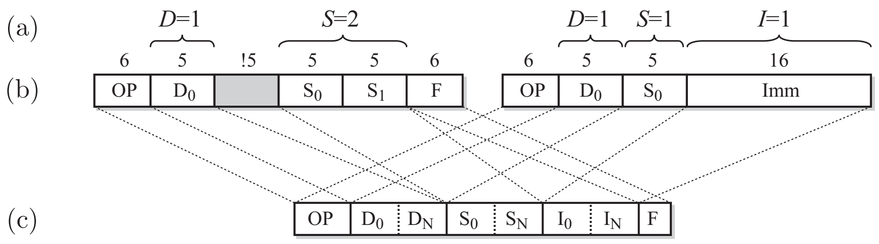


Figure 10: Instruction templates. (a) Specification of operand fields to set the number of source, destination, and immediate values. (b) Specification of bitfields to describe the instructions bit-accurate format. (c) General format used to derive instruction templates.

writing processor registers accordingly. An instruction can also control the program execution flow by modifying the program counter, and provides a method to specify the *instruction delay* for more accurate modeling. Instructions have the following format:

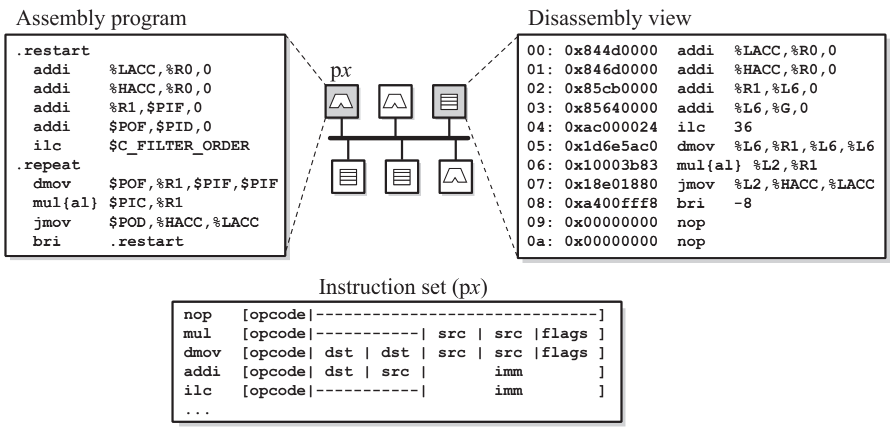
```
class subtract_immediate : public TypeB {               /* instruction class      */
public:
    subtract_immediate() : TypeB("subi") {}             /* a TypeB template       */
    int cycle_delay() { return 1; }                     /* instruction delay      */

    void execute(sci_registers& regs, sci_ctrl& ctrl)   /* implementation         */
    {
        dst(0)->write( src(0)->read() - imm(0) );       /* behavioral description */
    }
};
```

The *execute* method initiates a read from zero or more registers (I/O or general-purpose) and a write to zero or more registers (I/O or general-purpose). The instruction is only allowed to execute if all input values are available, and the instruction can only finish execution if all output values can be successfully written to result registers. Hence, an instruction performs *blocking* access to I/O port registers and to memory interfaces.

Memory Interfaces

The processor provides an interface to access external memory or to connect the processor to a system bus. External memory access is supported using virtual function calls, for which the user supply a suitable adaptor implementation. For the processor to execute properly, functionality to *load data* from memory, *store data* in memory, and *fetch instructions* from memory are required. The implementation of virtual memory access methods are illustrated in Figure 9(a),



```
.restart
  addi    %LACC,%R0,0
  addi    %HACC,%R0,0
  addi    %R1,$PIF,0
  addi    $POF,$PID,0
  ilc     $C_FILTER_ORDER
.repeat
  dmov    $POF,%R1,$PIF,$PIF
  mul{al} $PIC,%R1
  jmov    $POD,%HACC,%LACC
  bri     .restart
```

```
00: 0x844d0000  addi  %LACC,%R0,0
01: 0x846d0000  addi  %HACC,%R0,0
02: 0x85cb0000  addi  %R1,%L6,0
03: 0x85640000  addi  %L6,%G,0
04: 0xac000024  ilc   36
05: 0x1d6e5ac0  dmov  %L6,%R1,%L6,%L6
06: 0x10003b83  mul{al} %L2,%R1
07: 0x18e01880  jmov  %L2,%HACC,%LACC
08: 0xa400fff8  bri   -8
09: 0x00000000  nop
0a: 0x00000000  nop
```

```
nop   [opcode|------------------------------]
mul   [opcode|-----------| src | src |flags ]
dmov  [opcode| dst | dst | src | src |flags ]
addi  [opcode| dst | src |       imm        ]
ilc   [opcode|-----------|       imm        ]
...
```

Figure 11: The processor provides a code generator to convert user-developed assembly code into processor specific machine code. The machine code is stored in an external memory, and instructions are retrived using the memory interface.

and is performed by the *adapter* unit. The adapter translates the function calls to bus accesses, and returns requested data to the processor.

Programming and Code Generation

The architectural description provides the processor with information regarding the instructions binary format, the instruction and register names, and the register coding. Based on this information, the processor automatically provides a code generator to convert user-developed assembly code into processor specific machine code. Hence, when extending the instruction set with additional instruction, these can be immediately used in the assembly program without any manual development steps. The reverse process is also automatically supported and generates the disassembled source code from binary format. Figure 11 illustrates an embedded system, where an assembly program is downloaded to a processor. The processor translates the assembly program and stores the instructions in an external memory. The disassembly view of the external memory is also illustrated in Figure 11. Processors request instructions from external memory over the external memory interface connected to the bus.

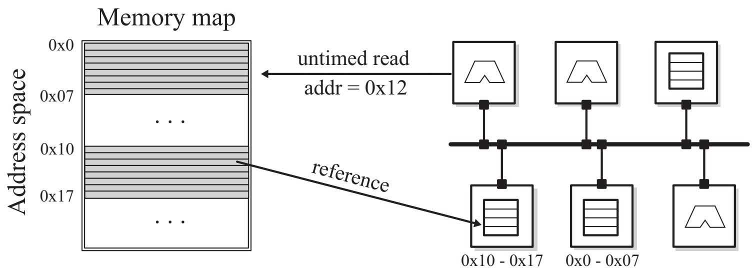


Figure 12: The memory models are accessed using either the simulation ports or thought the untimed interface. The untimed interface enables access to the memory contents from the Scenic shell.

4.2 Memory Generator

Memories are frequently used in digital system designs, operating as external memories, cache memories, or register banks. The memory generator enables the possibility to generate custom memory models for all these situations by allowing the memory wordlength, access latency, and timing parameters to be configurable. Each generated memory module implements a generic interface for reading and writing the memory contents, which can also be accessed from the Scenic shell. As shown in Figure 9(a), adaptors translate user requests into memory accesses. Hence, the memory can be connected to standard bus formats such as AMBA [78], by using appropriate adaptors.

The memory is not only active during simulation time, but also when downloading program code and when reading and writing application data from the Scenic environment. Therefore, an additional *untimed* interface is provided by the processors and memories, to be used even if the simulation is not running. The untimed interface is byte-oriented, and can hence be used for any memory regardless of the user-defined data type (abstraction). In addition, several memories can be grouped using a common *memory map.* A memory map is an address space that is shared by multiple memories, as shown in Figure 12. An assembly program can be translated by a processor and downloaded to any external memory through a global memory map.

The generated memories contain performance monitors to export statistics and implement a custom access method to access memory contents. The performance monitors provide information about the bandwidth utilization, average data latency, and the amount of data transferred. The memory *access pattern* can be periodically logged to identify memory locations that are frequently requested. Frequent access to data areas may indicate that data caching is required in another part of the design.

Table 2: Architectural generators used to construct reconfigurable architectures.

Generator	Scalability	Configurability
Tile	- Dimension ($W \times H$) - Static template	- Dynamic reconfiguration
Topology	- Topologies: - Mesh, Torus, - Ring, Custom	
Network	- Router fanout - Routing schemes - Network enhancements	- Router queue depth - Router latency - External connections

4.3 Architectural Generators

In addition to the model generators, three architectural generators are provided to construct more complex simulation platforms. The architectural generators construct customized arrays containing processor and memory models. Architectures are constructed in three steps using a *tile generator*, a *topology generator*, and a *network generator*. The tile generator creates arrays of simulation models, while the topology and network generators creates and configures the local and global communication networks, respectively. Table 2 presents a summary over the scalability and configurability of the proposed architectural generators. The generators are used in Part IV to model and evaluate dynamically reconfigurable architectures, using both the presented model generators and the architectural generators. The following architectural generators for reconfigurable computing have been designed and are briefly described below:

- **Tile Generator** - The tile generator use a *tile template* file to create a static array of *resource cells*, presented in Part IV, which are generic containers for any type of simulation models. When constructing larger arrays, the tile is used as the basic building block, as illustrated in Figure 13(a), and extended to arrays of arbitrary size. The resource cells are configured with a functional unit specified by the tile template, and can be either a processing cell or a memory cell.

- **Topology Generator** - The topology generator creates the local interconnects between resource cells, and supports *mesh*, *torus*, *ring*, and user-defined topologies. Figure 13(b) illustrates resource cells connected in a mesh topology. The local interconnects provide a high bandwidth between neighboring resource cells.

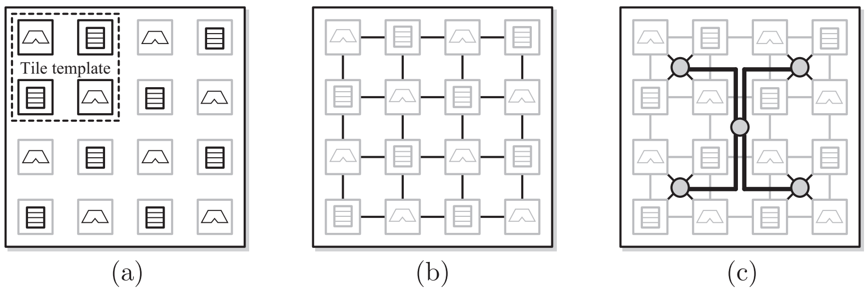


Figure 13: (a) Constructing the array from a tile description. (b) Building local communication topology, which can be mesh, ring, torus or custom topology. (c) Building global network with routers (in grey).

- **Network Generator** - The network generator inserts a global network, which connects to resource cells using routers, as illustrated in Figure 13(c). The network generator is highly configurable, including number of devices to connect to a single router (fanout), the router queue depth and latency, and the possibility to create user-defined links to enhance the global communication network. The routing tables are automatically configures to enable global communication.

5 Conclusion

SystemC enables rapid system development and high simulation performance. The proposed exploration environment, Scenic, is a SystemC environment with interactive control that addresses the issues on controllability and observability of SystemC models. It extends the SystemC functionality to enable system construction and configuration from XML, and provides access to simulation and performance data using run-time reflection. Scenic provides advanced scripting capabilities, which allow rapid design exploration and performance analysis in complex designs. In addition, a library of model generators and architectural generators is proposed. Model generators are used to construct designs based on customized processing and memory elements, and architectural generators provide capabilities to construct complex systems, such as reconfigurable architectures and Network-on-Chip designs.

Part IV

A Run-time Reconfigurable Computing Platform

Abstract

Reconfigurable hardware architectures are emerging as a suitable and feasible approach to achieve high performance combined with flexibility and programmability. While conventional fine-grained architectures are capable of bit-level reconfiguration, recent work focuses on medium-grained and coarse-grained architectures that result in higher performance using word-level data processing. In this work, a coarse-grained dynamically reconfigurable architecture is proposed. The system is constructed from an array of processing and memory cells, which communicate using local interconnects and a hierarchical routing network. Architectures are evaluated using the Scenic exploration environment and simulation models, and implemented VHDL modules have been synthesized for a $0.13\,\mu$m cell library. A reconfigurable architecture of size 4×4 has a core area of $2.48\,\text{mm}^2$ and runs up to 325 MHz. It is shown that mapping of a 256-point FFT generates 18 times higher throughput than for commercial embedded DSPs.

Based on: T. Lenart, H. Svensson, and V. Öwall, “Modeling and Exploration of a Reconfigurable Architecture for Digital Holographic Imaging,” in *Proceedings of IEEE International Symposium on Circuits and Systems*, Seattle, USA, May 2008.

and: T. Lenart, H. Svensson, and V. Öwall, “A Hybrid Interconnect Network-on-Chip and a Transactional Level Modeling approach for Reconfigurable Computing,” in *Proceedings of IEEE International Symposium on Electronic Design, Test and Applications*, Hong Kong, China, Jan. 2008.

1 Introduction

Platforms based on reconfigurable architectures combine high performance processing with flexibility and programmability [104, 105]. A reconfigurable architecture enables re-use in multiple design projects to allow rapid hardware development. This is an important aspect for developing consumer electronics, which are continuously required to include and support more functionality.

A dynamically reconfigurable architecture (DRA) can be reconfigured during run-time to adapt to the current operational and processing conditions. Using reconfigurable hardware platforms, radio transceivers dynamically adapt to radio protocols used by surrounding networks [106], whereas digital cameras adapt to the currently selected image or video compression format. Reconfigurable architectures provide numerous additional advantages over traditional application-specific hardware accelerators, such as resource sharing to provide more functionality than there is physical hardware. Hence, currently inactivated functional units do not occupy any physical resources, which are instead dynamically configured during run-time. Another advantage is that a reconfigurable architecture may enable mapping of future functionality without additional hardware or manufacturing costs, which could also extend the lifetime of the platform.

In this part, a DRA is proposed and modeled using the Scenic exploration framework and simulation models presented in Part III. By evaluating the platform at system level, multiple design aspects are considered during performance analysis. The proposed design flow for constructing, modeling, and implementing a DRA is presented in Figure 1. System construction is based on the Scenic architectural generators, which use the model library to customize simulation components. Related work is discussed in Section 2, and the proposed architecture is presented in Section 3. In Section 4, the Scenic exploration environment is used for system-level integration, and the platform is modeled and evaluated for application mapping in Section 5. However, automated application mapping is not part of the presented work, but is a natural extension. In Section 6, the exploration models are translated to VHDL, synthesized, and compared against existing architectures.

2 Related Work

A range of reconfigurable architectures have been proposed for a variety of application domains [33]. Presented architectures differ in granularity, processing and memory organization, communication strategy, and programming methodology. For example, the GARP project presents a generic MIPS processor with reconfigurable co-processors [107]. PipeRench is a programmable datapath of virtualized hardware units that is programmed through self-managed

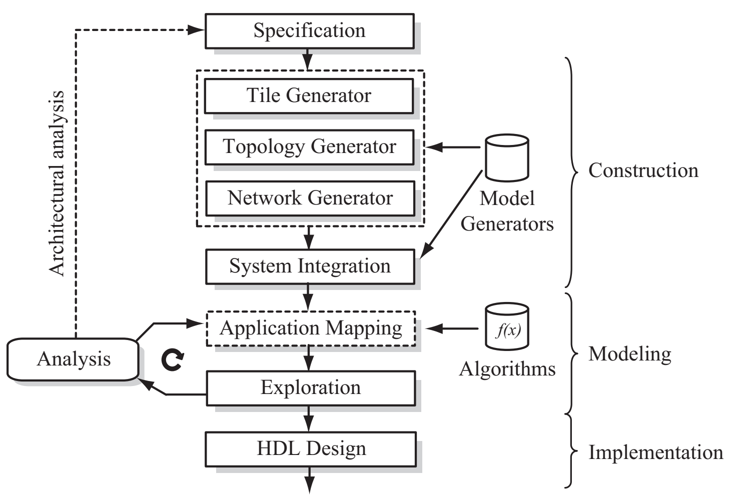


Figure 1: Design flow to construct a DRA platform. Modeling is based on the SCENIC model and architectural generators, and the hardware implementation is described using VHDL.

configurations [108]. Other proposed architectures are the MIT RAW processor array [109], the REMARC array of 16-bit nano processors [110], the Cimaera reconfigurable functional unit [111], the RICA instruction cell [112], weakly programmable processor arrays (WPPA) [113, 114], and architectures optimized for multimedia applications [115]. A medium-grained architecture is presented in [116], using a multi-level interconnection network similar to this work.

Examples of commercial dynamically reconfigurable architectures include the field programmable object array (FPOA) from MathStar [117], which is an array of 16-bit *objects* that contain local program and data memory. The adaptive computing machine (ACM) from QuickSilver Technology [118] is a 32-bit array of *nodes* that each contain an algorithmic core supporting arithmetic, bit-manipulation, general-purpose computing, or external memory access. The XPP platform from PACT Technologies is constructed from 24-bit processing array elements (PAE) and communication is based on self-synchronizing data flows [119, 120]. The Montium tile processor is a programmable architecture where each tile contains five processing units, each with a reconfigurable instruction set, connected to ten parallel memories [106].

A desirable programming approach for the architectures above is *software-centric*, where applications are described using a high-level design language to ease development work and to increase productivity. The high-level description is then partitioned, scheduled, and mapped using automated design steps. Many high-level design languages for automated hardware generation exist [121], but few for application mapping onto arrays containing coarse-grained reconfigurable resources. For these architectures, the most widely used programming approach today is *hardware-centric*, where the designer manually develop, map, and simulate applications. Processing elements are programmed using either a C-style syntax or directly in assembly code. Although being a more complex design approach, this is the situation for most commercial reconfigurable platforms.

3 Proposed Architecture

Based on recently proposed architectures, it is evident that coarse-grained designs are becoming increasingly complex, with heterogeneous processing units comprising a range of application-tuned and general-purpose processors. As a consequence, efficient and high-performance memories for internal and external data streaming are required, to supply the computational units with data.

However, increasing complexity is not a feasible approach for an embedded communication network. In fact, it has lately been argued that interconnect networks should only be sufficiently complex to be able to fully utilize the computational power of the processing units [122]. For example, the mesh-based network-on-chip (NoC) structure has been widely researched in both industry and academia, but suffers from inefficient global communication due to multi-path routing and long network latencies. Another drawback is the large amount of network routers required to construct a mesh. As a consequence, star-based and tree-based networks are being considered [123], as well as a range of hybrid network topologies [124].

This work proposes reconfigurable architectures based on the following statements:

- Coarse-grained architectures result in better performance/area trade-off than fine-grained and medium-grained architectures for the current application domain. Flavors of coarse-grained processing elements are proposed to efficiently map different applications.
- Streaming applications require more than a traditional load/store architecture. Hence, a combination of a RISC architecture and a streaming architecture is proposed. Furthermore, the instruction set of each processor is customized to extend the application domain.
- Multi-level communication is required to combine high bandwidth with flexibility to a reasonable hardware cost [116]. A separate local and global

communication network is proposed, which also simplifies the interface to external resources.

- Presented systems where a processor provides data to a tightly-coupled reconfigurable architecture is not a feasible solution for high-performance computing [110]. Hence, stream memory controllers are proposed to efficiently transfer data between the reconfigurable architecture and external memories.

3.1 Dynamically Reconfigurable Architecture

A hardware acceleration platform based on a dynamically reconfigurable architecture is proposed. The DRA is constructed from a tile of $W \times H$ resource cells and a communication network, which is shown in Figure 2(a). *Resource cell* is the common name for all types of functional units, which are divided into *processing cells* (PC) and *memory cells* (MC). Processing cells implement the processing functionality to map applications to the DRA, while memory cells are used to store data tables and intermediate results during processing. The resource cells are dynamically reconfigurable to support run-time mapping of arbitrary applications.

An array of resource cells is constructed from a *tile template*. A tile template is user-defined and contains the pattern in which processing and memory cells are distributed over the array. For example, the architecture presented in Figure 2(a) is based on a tile template of size 2×2, with two processing cells and two memory cells. The template is replicated to construct an array of arbitrary size.

The resource cells communicate over local interconnects and a global hierarchical network. The local network of dedicated wires enable high-speed communication between neighboring resource cells, while the global network provides communication flexibility to allow any two resource cells in the array to communicate [122].

3.2 Processing Cells

Processing cells are computational units on which applications are mapped, and the processing cells contain a configurable number of local input and output ports (L_x) to communicate with neighboring processing or memory cells, and one global port (G) to communicate on the hierarchical network.

Three different processing cells are presented and implemented: A 32-bit DSP processor with radix-2 butterfly support, a 16/32-bit MAC processor with multiplication support, and a 16-bit CORDIC processor for advanced function evaluation. The DSP and MAC processors are based on a similar architecture,

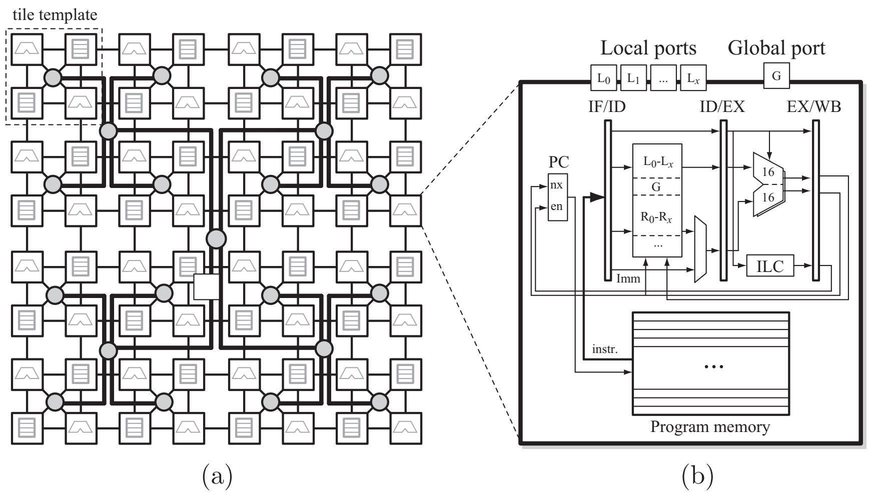


Figure 2: (a) Proposed architecture with an array of processing and memory cells, connected using a local and a global (hierarchical) network. $W = H = 8$. (b) Internal building blocks in a processing cell, which contain a programmable processing element.

with a customized instruction set and a program memory to hold processor instructions, as shown in Figure 2(b). Data is processed from a register bank that contains the general-purpose registers (R_0-R_x) and the I/O port registers (L_0-L_x,G). An instruction that access I/O port registers is automatically stalled until data becomes available. Hence, from a programming perspective there is no difference between general-purpose registers and I/O port register access. The DSP and MAC processors have the following functionality and special properties to efficiently process data streams:

- I/O *port registers* - The port registers connect to surrounding processing and memory cells. Port registers are directly accessible in the same way as general-purpose processor registers, to be used in arithmetic operations. Hence, no load/store operations are required to move data between registers and ports, which significantly increase the processing rate.

- *Dual ALU* - A conventional ALU takes two input operands and produces a single result value. In contrast, the DSP and MAC processors include two separate ALUs to produce two values in a single instruction. This is useful when computing a radix-2 butterfly or when moving two data values in parallel. The MAC processor uses a parallel move instruction

Table 1: Supported functionality by the configurable processing cells.

Processor	ALU format	Dual ALU (ALU_1/ALU_2)	Separable ALU (ALU_n^{hi}/ALU_n^{lo})	Special functions
DSP	32-bit 2×16-bit	$+/+$ 2×32-bit $-/-$ 2×32-bit $+/-$ 2×32-bit	$+/+$ 2×16-bit $-/-$ 2×16-bit $+/-$ 2×16-bit	Radix-2 Butterfly
MAC	16-bit	Split/Join: 2 GPR $\rightarrow$ I/O I/O $\rightarrow$ 2 GPR		Multiply Multiply/Add Multiply-Acc.
CORDIC	2×16-bit	16-stage pipeline		Multiply/Divide Trigonometric Magnitude/Phase

to split and join 16-bit internal registers and 32-bit I/O registers.

- *Separable ALU* - Each 32-bit ALU data path can be separated into two independent 16-bit fields, where arithmetic operations are applied to both fields in parallel. This is useful when operating on complex valued data, represented as a 2×16-bit value. Hence, complex values can be added and subtracted in a single instruction.

- *Inner loop counter* - A special set of registers are used to reduce control overhead in compute-intensive inner loops. The inner loop counter (ILC) register is loaded using a special instruction that stores the next program counter address. Each instruction contains a flag that indicates end-of-loop, which updates the ILC register and reloads the previously stored program counter.

The CORDIC processor is based on a 16-stage pipeline of adders and subtractors, with 2 input and 2 output values, capable of producing one set of output values each clock cycle. The CORDIC processor operates in either vectoring or rotation mode to support multiplication, division, trigonometric functions, and to calculate magnitude and phase of complex valued data [70].

The supported functionality is summarized in Table 1. As shown, the DSP processor is designed for standard and complex valued data processing, while the MAC processor supports multiply-accumulate and the CORDIC processor provides complex function evaluation. The instruction-set and a more detailed architectural schematic of the DSP and MAC processors are presented in Appendix B.

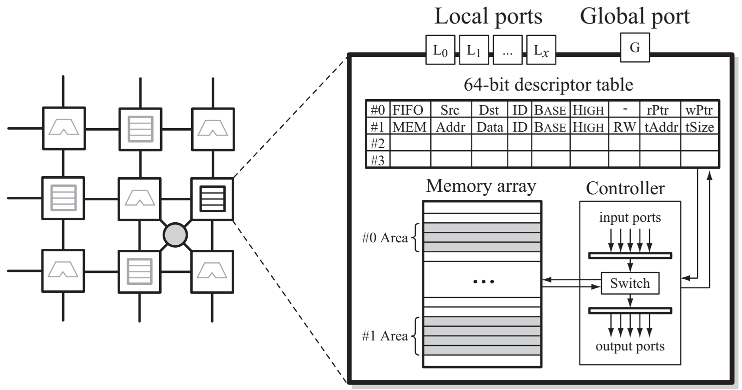


Figure 3: A memory cell contains a descriptor table, a memory array, and a controller. The descriptor table indicates how data is to be transferred between the ports and which part of the memory that is allocated for each descriptor.

3.3 Memory Cells

Most algorithms require intermediate storage space to buffer, reorder, or delay data. Memories can be *centralized* in the system, *internal* inside processing cells, or *distributed* over the array. However, to fully utilize the computational power of the processing cells, storage units are required close to the computational units. Naturally, an internal memory approach satisfies this requirement, but suffers from the drawback that it is difficult to share *between* processing cells [125]. In contrast, a distributed approach enables surrounding cells to share a single memory. Hence, shared *memory cells* are distributed in a user-defined pattern (tile template) over the array.

An overview of the memory cell architecture is shown in Figure 3, which is constructed from a descriptor table, a memory array, and a controller unit. The descriptor table contains *stream transfers* to be processed, and is dynamically configured during run-time. The memory array is a shared dual-port memory bank that stores data associated with each stream transfer. The controller unit manages and schedules the descriptor processing and initiates transfers between the memory array and the external ports.

Stream transfers are rules that define how the external ports communicate with neighboring cells, operating in either FIFO or RAM mode. A stream transfer in FIFO mode is configured with a *source* and *destination* port, and an allocated *memory area*. The source and destination reference either local ports (L_0-L_x) or the global port (G). The memory area is a part of the shared memory bank that is reserved for each stream transfer, which is indicated with a BASE address and a HIGH address. In FIFO mode, the reserved memory area operates as a circular buffer, and the controller unit handles address pointers to the current read and write locations. Input data is received from the source port and placed at the current write location inside the memory array. At the same time, the destination port receives the data stored in the memory array at the current read location.

In a similar fashion, a stream transfer in RAM mode is configured with an *address* port, a *data* port, and an allocated memory area. The controller unit is triggered when the address port receives either a READ or a WRITE request, that specifies a memory address and a transfer length. If the address port receives a READ request, data will start streaming from the memory array to the configured data port. Consequently, if the address port receives a WRITE request, data is fetched from the configured data port and stored in the memory array.

3.4 **Communication and Interconnects**

The resource cells communicate over local and global interconnects, where data transactions are synchronized using a handshake protocol. The local interconnects handle communication between neighboring resource cells, and provide a high communication bandwidth. Hence, it is assumed that the main part of the total communication is between neighboring cells and through local interconnects. However, a global hierarchical network enables non-neighboring cells to communicate and provides an interface to external memories. Global communication is supported by routers that forward network packets over a global network. In the proposed architecture, routers use a *static* lookup-table that is automatically generated at design-time. Since the routing network is static, there is only one valid path from each source to each destination. Static routing simplifies the hardware implementation, and enables each router instance to be optimized individually during logic synthesis. However, a drawback with static routing is network congestion, but mainly concerns networks with a high degree of routing freedom, for example a traditional mesh topology.

An overview of the network router architecture is shown in Figure 4(a), which is constructed from a *decision unit* with a static routing table, a *routing structure* to direct traffic, and a *packet queue* for each output port. The decision unit monitors the status of input ports and output queues, and configures the routing structure to transfer data accordingly. The complexity of the routing structure determines the routing capacity, as illustrated in Figure 4(b). A parallel routing structure is associated with a higher area cost and requires a more complex decision unit.

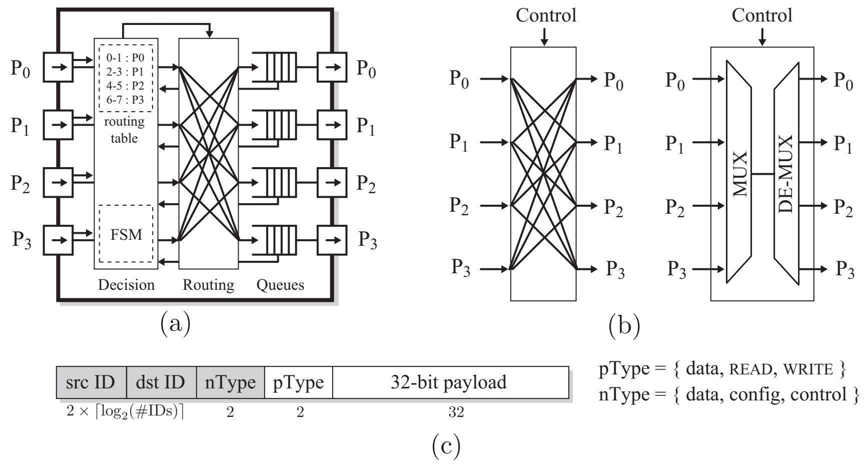


Figure 4: (a) Network router constructed from a decision unit, a routing structure, and packet queues for each output port. (b) Architectural options for routing structure implementation. (c) Local flit format (white) and extension for global routing (grey).

Network Packets

The routers forward *network packets* over the global interconnects. A network packet is a carrier of data and control information from a source to a destination cell, or between resource cells and an external memory. A data stream is a set of network packets, and for streaming data each individual packet is send as a network *flow control digit* (flit). A flit is an atomic element that is transferred as a single word on the network, as illustrated in Figure 4(c). A flit consists of a 32-bit payload and a 2-bit payload type identifier to indicate if the flit contains data, a read request, or a write request. For global routing, unique identification numbers are required and discusses in the next section. An additional 2-bit network type identifier indicates if the packet carries data, configuration, or control information. *Data* packets have the same size as a flit, and contain a payload to be either processed or stored in a resource cell. *Configuration* packets contain a functional description on how resource cells should be configured, and is further discussed in Section 4.2. Configurations are transferred with a header specifying the local target address and the payload size, to be able to handle *partial* configurations. *Control* packets are used to notify the host processor of the current processing status, and are reserved to exchange flow control information between resource cells.

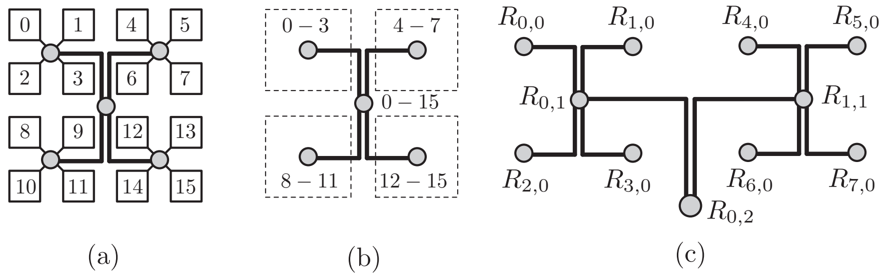


Figure 5: (a) Recursively assignment of network IDs. (b) A range of consecutive network IDs are assigned to each router table. (c) Hierarchical router naming as $R_{index,level}$.

Network Routing

Each resource cell allocates one or more network identifiers (ID), which are integer numbers to uniquely identify a resource cell. Each resource cell allocates one (or more) network ID as shown in Figure 5(a). The identification field is represented with $\lceil \log_2(W \times H + \mathrm{ID}_{\mathrm{ext}}) \rceil$ bits, where $\mathrm{ID}_{\mathrm{ext}}$ is the number of network IDs allocated for external communication.

A static routing table is stored inside the router and used to direct traffic over the network. At design-time, network IDs and routing tables are recursively assigned by traversing the global network from the top router. Recursive assignment results in that each entry in the routing table for a router $R_{i,l}$, where i is the router index number and l is the routers hierarchical level as defined in Figure 5(c), is a *continuous range* of network IDs as illustrated in Figure 5(b). Hence, network ID ranges are represented with a base address and a high address. The network connectivity C is defined by a function

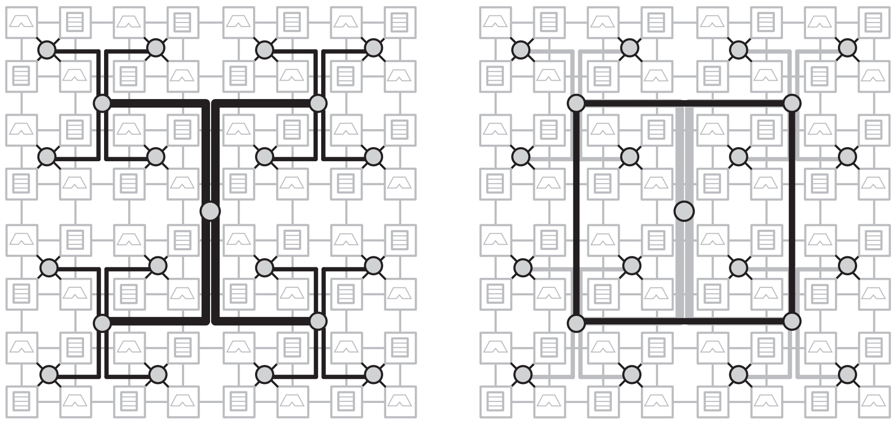
$$C(R_{i,l}, R_{m,n}) = \begin{cases} 1, & R_{i,l} \overset{\text{link}}{\rightarrow} R_{m,n} \\ 0, & \text{otherwise,} \end{cases}$$

where the value 1 indicates that there is a link from router $R_{i,l}$ to router $R_{m,n}$ ($R_{i,l} \neq R_{m,n}$) and 0 otherwise. Based on the network connectivity, the routing table Λ for a router $R_{i,l}$ is defined as

$$\Lambda(R_{i,l}) = \{\lambda(R_{i,l}), \kappa(R_{i,l})\}, \tag{1}$$

where $\lambda(R_{i,l})$ is the set of network IDs to reachable routers from $R_{i,l}$ to *lower* hierarchical levels, and $\kappa(R_{i,l})$ is the set of network IDs from $R_{i,l}$ to reachable routers on the *same* hierarchical level as

$$\lambda(R_{i,l}) = \{\lambda(R_{j,l-1}) : C(R_{i,l}, R_{j,l-1}) = 1, l > 0\}$$



(a) (b)

Figure 6: (a) Enhancing the router capacity when the hierarchical level increase. (b) Enhancing network capacity by connecting routers at the same hierarchical level.

and

$$\kappa(R_{i,l}) = \{\lambda(R_{j,l}) : C(R_{i,l}, R_{j,l}) = 1\}.$$

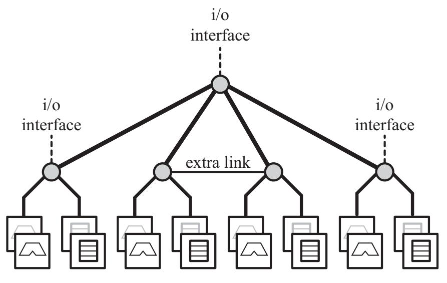
At the lowest hierarchical level, where $l = 0$, the reachable nodes in $\lambda(R_{i,0})$ is a set of network IDs that are reserved by the connected resource cells. At this level, λ is always represented as a continuous range of network IDs.

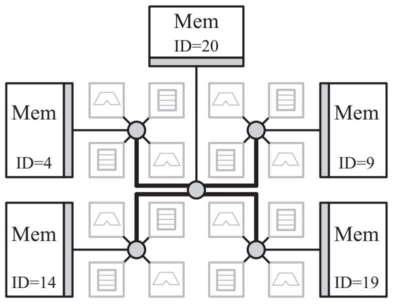
A link from a router $R_{i,l}$ to a router $R_{j,l+1}$ is referred to as an *uplink*. Any packet received by router R is forwarded to the uplink router if the packets network ID is not found in the router table $\Lambda(R)$. A router may only have a single uplink port, else the communication path could become non-deterministic.

In the SCENIC environment, routing tables can be extracted for any router, and contain the information below for top router $R_{0,1}$ and sub-router $R_{0,0}$ from Figure 5(b):

```
Routing table for R_01:              Routing table for R_00:
-------------------------------      -------------------------------------
  Port 0 -> ID 0-3   [R_00]            Port 0 -> ID 0-0 [resource cell #0]
  Port 1 -> ID 4-7   [R_10]            Port 1 -> ID 1-1 [resource cell #1]
  Port 2 -> ID 8-11  [R_20]            Port 2 -> ID 2-2 [resource cell #2]
  Port 3 -> ID 12-15 [R_30]            Port 3 -> ID 3-3 [resource cell #3]
  Port 4 -> ID 16-16 [Memory]          Uplink is port 4
```

The top router connects to an *external memory*, which is further discussed in Section 4.1.





(a) (b)

Figure 7: (a) Hierarchical view of the router interconnects and external interfaces. (b) External memories connected to the network routers. The memories are uniquely identified using the assigned network ID.

Network Capacity

When the size of a DRA increases, the global communication network is likely to handle more traffic, which requires network enhancements. A solution to improve the communication bandwidth is to increase the network capacity in the communication links, as shown in Figure 6(a). Since routers on a higher hierarchical level could become potential bottlenecks to the system, these routers and router links are candidates for network link capacity scaling. Thus, this means that a *single link* between two routers is replaced by *parallel links* to improve the network capacity. A drawback is increased complexity, since a more advanced router decision unit is required to avoid packet reordering. Otherwise, if packets from the same stream are divided onto different parallel links, this might result in that individual packets arrive out-of-order at the destination.

Another way to improve the communication bandwidth is to insert additional network paths to avoid routing congestion in higher level routers, referred to as *network balancing*. Figure 6(b) shows an example where all $R_{i,1}$ routers are connected to lower the network traffic through the top router. Additional links may be inserted between routers as long as the routing table in each network router is deterministic. When a network link is created between two routers, the destination router's reachable IDs (λ) is inserted in the routing table (Λ) for the source router. Links that do not satisfy the conditions above are not guaranteed to automatically generate deterministic routing tables, but could still represent a valid configuration.

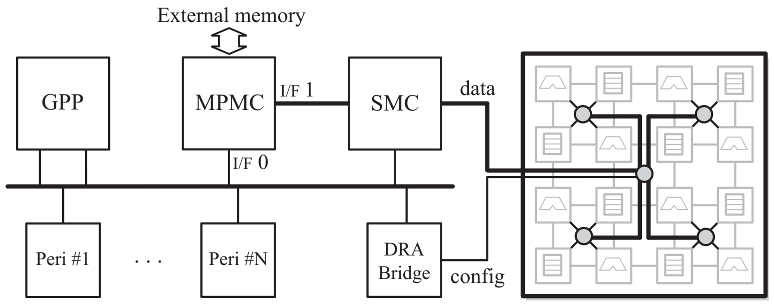


Figure 8: An embedded system containing a DRA, which is connected to a stream memory controller (SMC) to transfer streaming data between the DRA and external memory. Configurations are generated by the GPP and sent over the DRA bridge.

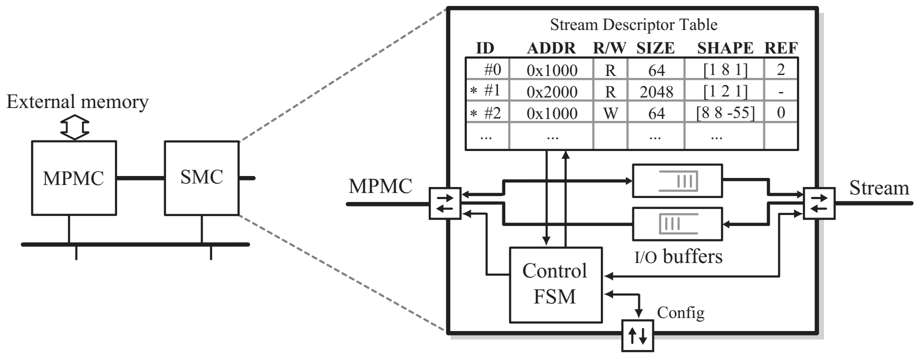
External Communication

The network routers are used for connecting external devices to the DRA, as illustrated in Figure 7(a). When binding an external device to a router, a unique network ID is assigned and the routing tables are automatically updated to support the new configuration. Examples of external devices are memories for data streaming, as illustrated in Figure 7(b), or an interface to receive configuration data from an embedded GPP. Hence, data streaming to and from external memories require the use of a global network. Details about external communication and how to enable efficient memory streaming is further discussed in Section 4.1.

4 System-Level Integration

Additional system components are required to dynamically configure resource cells, and to transfer data between resource cells and external memories. Therefore, the proposed DRA is integrated into an embedded system containing a general-purpose processor, a multi-port memory controller (MPMC), and a proposed stream memory controller (SMC) to efficiently supply the DRA with data. The GPP and the MPMC are based on the Scenic processor and memory generators, while the SMC is custom designed. The system architecture is shown in Figure 8, where the top network router is connected to the SMC to receive streaming data from an external memory. The top network router is also connected to a bridge, which allows the GPP to transmit configuration data over the system bus.

The MPMC contains multiple ports to access a shared external memory,



ID	ADDR	R/W	SIZE	SHAPE	REF
#0	0x1000	R	64	[1 8 1]	2
* #1	0x2000	R	2048	[1 2 1]	-
* #2	0x1000	W	64	[8 8 -55]	0
...	...		...	...	

Figure 9: A stream memory controller (SMC) that handles data transfers to and from a DRA. Data is transferred in blocks and the SMC handles memory addressing and double-buffering. An active transfer is indicated by a star sign ($*$).

were one port is connected to the system bus and one port to the SMC. For design exploration, the MPMC can be configured to emulate different memory types by changing operating frequency, wordlength, data-rate, and internal parameters for controlling the memory timing. This is useful when optimizing and tuning the memory system, which is presented in 5.2.

4.1 Stream Memory Controller

The SMC is connected to one of the memory ports on the MPMC, and manages data streams between resource cells and external memory. Hence, the DRA is only concerned with *data processing*, whereas the SMC provides *data streams* to be processed.

Internal building blocks in the SMC are shown in Figure 9, and includes a stream descriptor table, a control unit, and I/O stream buffers. The stream descriptor table is configured from the GPP, and each descriptor specifies how data is transferred [126]. The control unit manages the stream descriptors and initiates transfers between the external memory and the stream buffers. Data is transferred from the stream buffers to stream ports, which are connected to the DRA.

There is one stream descriptor for each allocated network ID, which are individually associated with the corresponding resource cells. A stream descriptor consists of a *memory address*, *direction*, *size*, *shape*, and additional *control* bits. The memory address is the allocated area in main memory that is associated

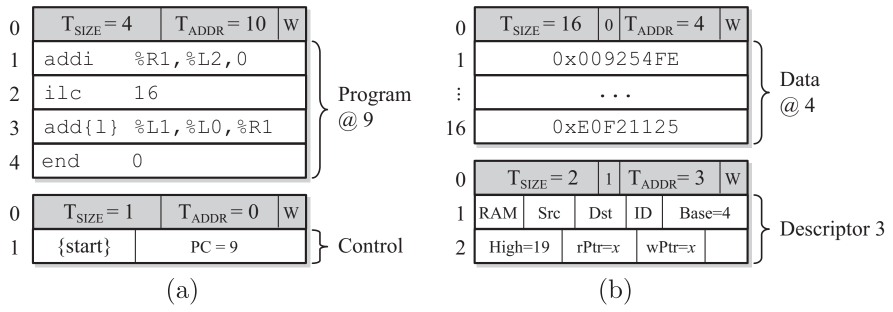


Figure 10: 32-bit configuration packets. (a) Configuration of a processing cell, including program memory and control register. (b) Configuration of a memory cell, including memory array and descriptor table.

with the stream transfer. The transfer direction is either READ or WRITE, and number of words to transfer is indicated by *size*. A *shape* describes how data is accessed inside the memory area, and consists of a three parameters: *stride*, *span*, and *skip* [127]. Stride is the memory distance to the next stream element, and span is the number of elements to transfer. Skip indicates the distance to the next start address, which restarts the stride and span counters. The use of programmable memory shapes is a flexible and efficient method to avoid address generation overhead in functional units.

Additional control bits enable resource cells to share buffers in an external memory. The reference field (REF) contains a pointer to an inactivated stream descriptor that shares the same memory area. When the current transfer completes, the associated stream is activated and allowed to access the memory. Hence, the memory area is used by two stream transfers, but the access to the memory is *mutually exclusive*. Alternatively, the descriptors are associated with different memory regions, and the data pointers are automatically interchanged when both stream transfers complete, which enables *double-buffering*. An example is shown in Figure 9, where stream descriptors 0 and 2 are configured to perform a 8×8 matrix transpose operation. Data is written column-wise, and then read row-wise once the write transaction completes.

4.2 Run-Time Reconfiguration

The resource cells are reconfigured during run-time to emulate different functionality. Configuration data is sent over a global network, and a destination cell is specified using the network ID. Hence, there is no additional hardware cost due to dedicated configuration interconnects. Furthermore, *partial* reconfiguration is supported, which means that only the currently required part of a processing or memory cell is reconfigured. This may significantly improves the reconfiguration time.

A resource cell is reconfigured by sending configuration packets to the allocated network ID. Each configuration packet contains a *header*, which specifies the target address and size of payload data, as well as the *payload*. Several partial configuration packets may be sent consecutively to configure different regions inside a resource cell. Configuration packet format for a processing cell is illustrated in Figure 10(a). A processing cell has two configurable parts, a *program memory* and a *control register*. The program memory is indexed from address 1 to M_{pc}, where M_{pc} is the size in words of the memory array. Address 0 is reserved for the control register, which contains configuration bits to reset, start, stop, and single-step the processor. Configuration packet format for a memory cell is illustrated in Figure 10(b). A memory cell is also divided into two configurable parts, a *memory array* and a *descriptor table*. The memory array contains 32-bit data, while descriptors are 64-bit wide, hence requiring two separate configuration words for each descriptor.

System configuration time depends on the required resources for an application mapping $\mathcal{A}$, which includes the size of the application data and programs to be downloaded. The reconfiguration time t_r is measured in clock cycles and is the sum of all the partial reconfigurations in $\mathcal{A}$. For the array in Figure 2(a) with $W = H = 8$, $M_{\mathrm{pc}} = 64$ words of program memory, and $M_{\mathrm{mc}} = 256$ words in the memory array, the configuration time is in the range of $t_r = 32M_{\mathrm{pc}} + 32M_{\mathrm{mc}} \approx 10K$ clock cycles for a complete system reconfiguration. At 300 MHz, this corresponds to a configuration delay of $34\,\mu$s. However, in most situations the partial reconfiguration time is only a fraction of the time required for full system reconfiguration.

5 Exploration and Analysis

This section presents simulation results and performance analysis of the proposed reconfigurable platform. It is divided into *network simulation*, *memory simulation* and *application mapping*. The SCENIC exploration framework enables configuration of resource cells to define different scenarios, and supports extraction of performance metric, such as throughput and latency, during simulation. For application mapping, SCENIC is used to configure resource cells using an XML-based description, and to analyze systems for potential performance bottlenecks during simulation.

5.1 Network Performance

To evaluate network performance, a DRA is configured as an 8×8 array consisting of *traffic generators*, which are resource cells that *randomly* send packets

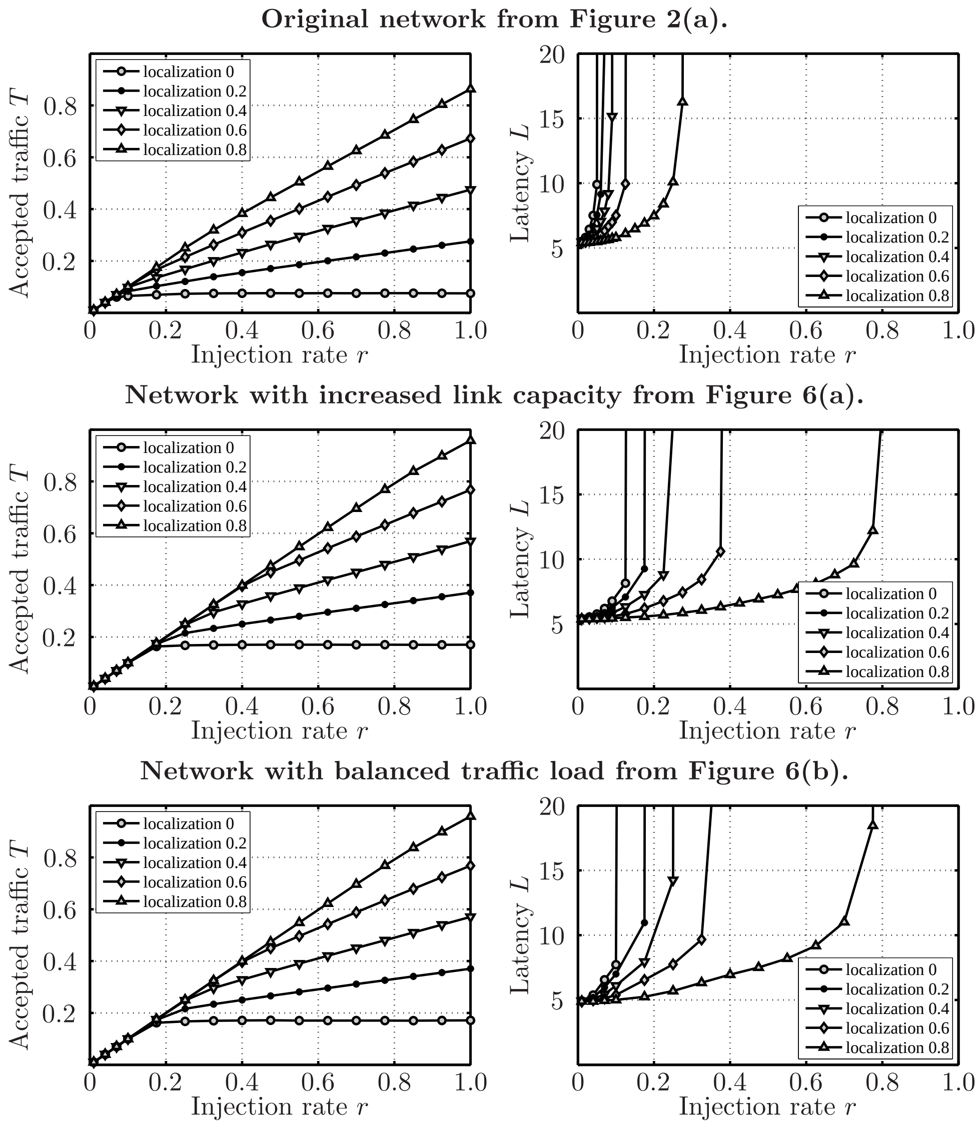


Figure 11: Network performance in terms of accepted traffic T and network latency L. Three simulations show original network, network with increased link capacity, and network with balanced traffic load.

to neighboring cells and to the global network. Packets are annotated with statistical information to monitor when each packet was produced, injected, received, and consumed. It also contains information about the number of network hops from the source node to the destination node, while the traffic generators monitor the number of received packets and the transport latency. Since communication is both local and global, a localization factor α is defined as the ratio between the amount of local and global communication, where $\alpha = 1$ corresponds to pure local communication and $\alpha = 0$ corresponds to fully global communication.

The traffic generators inject packets into the network according to the Bernoulli process, which is a commonly used injection process to characterize networks [128]. The *injection rate*, r, is the number of packets per clock cycle and per resource cell injected into the local or global network. The accepted traffic is the *network throughput* T, which is measured in packets per clock cycle and per resource cell. Ideally, accepted traffic should increase linearly with the injection rate. However, due to traffic congestion in the global network the amount of accepted traffic will saturate at a certain level. The average *transport latency* is defined as $L = \sum L_i/N$, where L_i is the transport latency for packet i and N is the total number of consumed packets. Transport latency is measured as the number of clock cycles between a successful injection into the network and consumption at the final destination.

The network performance depends on both the local and global network. As discussed in Section 3.4, the global network may be enhanced by either improving the link capacities or by inserting additional network links between routers. Figure 11 shows the simulation results from three network scenarios based on an 8×8 resource array. The first simulation is the original network configuration shown in Figure 2(a). The second and third scenarios are the enhanced routing networks shown in Figure 6(a-b). As shown, network enhancements generate a higher communication bandwidth with reduced latency. Performance is measured as the combined local and global communication, i.e. the overall performance, while latency is measured for global communication. The routers used in this simulation can manage up to two data transactions per clock cycle, but only for transactions that do not access the same physical ports.

The simulations in Figure 11 indicate how much traffic is injectable into the global network before saturation. Assuming an application with 80% local communication, i.e. $\alpha = 0.8$, the networks can manage an injection rate of around 0.3, 0.8 and 0.8, respectively. Thus, this illustrates the need for capacity enhancements in the global network. Since both enhancement techniques have similar performance, the network in Figure 6(b) is a better candidate since it is less complex and easier to scale with existing router models. However,

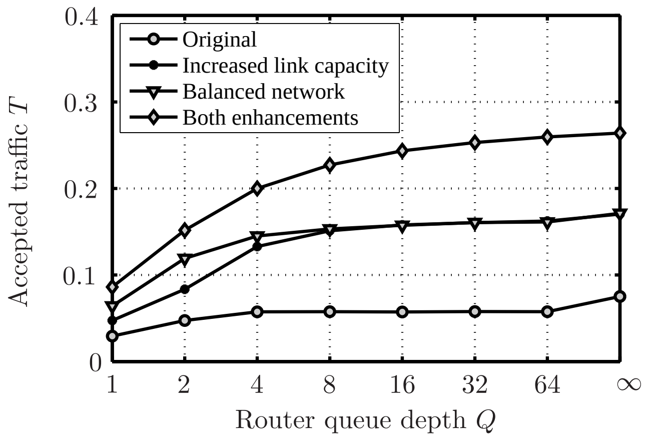


Figure 12: The impact on network performance for different router output queue depth Q when $\alpha = 0$. The curve start to flatten out around $Q = 8$.

if a shared memory is connected to the top router, more bandwidth will be required, which instead implies increased link capacity to the top router.

Another important system parameter is the router output queue depth, which affects the router complexity and the network performance. Network buffers are expensive, and avoiding large memory macros for each router is desirable. Figure 12 shows the global network performance when $\alpha = 0$ as a function of the router queue depth Q. The final simulation point shows the performance with unlimited buffer space, to illustrate the difference between feasible implementations and maximum theoretical performance. Around $Q = 8$, the curve start to flatten out with a performance of $80 - 90\%$ of $T_{\max}$ for each curve, hence a suitable trade-off between resource requirements and network performance. The hardware requirements for different router implementations are presented in Section 6.

5.2 Memory Simulation

Other important system aspects are the external memory configuration and the external memory interface. The functional units operate on streams of data, which are managed by the SMC. An *input* stream, i.e. from memory to the DRA, is fully controlled by the SMC unit and is transferred as a block of consecutive elements (assuming a trivial memory shape). Hence, the memory access pattern renders a high bandwidth to external memory. In contrast, an *output* stream, i.e. from the DRA to memory, can be managed in different ways, either as a *locally* controlled or as a *globally* controlled memory transfer. Local control means that each resource cell transfers data autonomously, while global control implies stream management by the SMC.

To evaluate the memory stream interface, two architectural mappings and five memory scenarios is evaluated. The first mapping is when an application executes on a single processing cell, and the second mapping is when the data processing is shared and divided onto two processing cells, as shown in Figure 13(a-b). This will illustrate that throughput is not only a matter of parallelism, but a balance between shared system resources such as the global network and the memory interface.

A processing cell is configured with an application to *post-process* images in digital holography, as illustrated in Part I Figure 3(c). The post-processing step generates a superposed image between magnitude and phase data after image reconstruction, to enhance the visual perception. A configuration is downloaded to combine the RGB-colors of two input streams, and write the result to an output stream. Four clock cycles are required to process one pixel, resulting in a throughput of 1/4 samples per clock cycle. The DRA and the external memory are assumed to operate at the same clock frequency, hence 3/4 of the theoretical memory bandwidth is required for three streams. The following Scenic script constructs a DRA platform, downloads two bitmap images through the MPMC, and inserts stream descriptors in the SMC to transfer data to and from the DRA:

```
xml load -file "system_4x4.xml"                              % load XML design
sim                                                          % create simulation platform
xml config -name "post-process"                              % load XML configuration

MPMC wr -addr 0x100000 -file "amplitude.bmp"                 % download bitmaps to memory
MPMC wr -addr 0x200000 -file "phase.bmp"
SMC insert -id 1 -addr 0x100000 -rnw TRUE  -size $IMSIZE     % insert stream descriptors
SMC insert -id 3 -addr 0x200000 -rnw TRUE  -size $IMSIZE
SMC insert -id 2 -addr 0x300000 -rnw FALSE -size $IMSIZE

runb 4 ms                                                    % run simulation

MPMC rd -addr 0x300000 -size $IMSIZE -file "result.bmp"      % save result image
set TRANSFERS   [ra.cell* get -var transfers]                % extract transfers
set SIM_CC      [SMC get -var last_transfer]                 % extract clock cycles
set UTILIZATION [eval [eval sum $TRANSFERS] / $SIM_CC]       % calculate utilization
```

The system is evaluated for different *transfer lengths*, which is the size in words of a transfer between the external memory and the processing cell. The transfer length is important for a realistic case study, since external memory is burst oriented with high initial latency to access the first word in a transfer. Consequently, the simulation in Figure 14 plot (a) shows improved throughput when the transfer length increases.

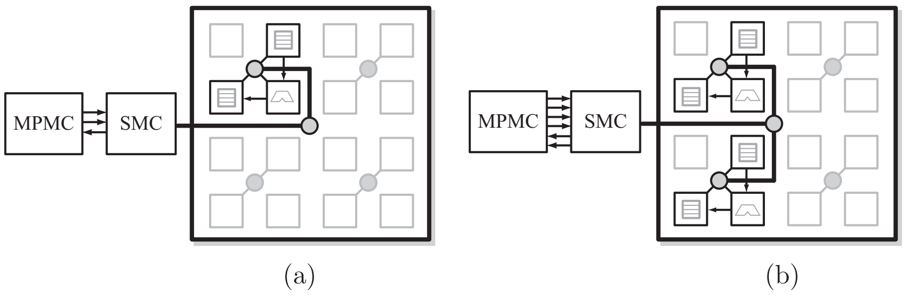


Figure 13: (a) Application mapping to a single processing cell, using two read and one write stream to external memory. (b) Dividing the computations onto two processing cells, which require two sets of I/O streams and double bandwidth to external memory.

The application mapping is modified to execute on two parallel processing cells, sharing the computational workload to increase throughput. Initially, the performance curve follows the result from mapping to a single processing cell, as shown in Figure 14 plot (b). However, for a certain burst length the performance suddenly decrease, presenting a *worse* result than for the previous application mapping. With further analysis, by inspecting the MPMC statistics using SCENIC, it can be seen in Figure 15 plot (b) that the memory access pattern change abruptly at this point, caused by interleaving of the result streams from the two processing cells. Interleaving results in shorter burst transfers to external memory, and therefore also a lower throughput.

The stream interleaving problem can be addressed by using globally controlled streaming, where the SMC operates as an arbitration unit and grants the resource cells access to external memory. To avoid transfer latency between two consecutive grants, two resource cells are granted access to the memory with an overlap in time, and the two streams are *reordered* inside the SMC to optimize the memory access pattern for burst oriented memories.

The result with globally controlled and reordered streams is shown in Figure 14 plot (c). Reordering solves the burst transfer issue, but the throughput has not increased. By inspecting the processing cells unit SCENIC, it can be seen that the utilization is only 50% in each processing element. Hence, more memory bandwidth is required to supply the processing elements with data.

To address the bandwidth issue, the MPMC is reconfigured to evaluate new scenarios, first emulating a double data-rate (DDR) memory and then increasing memory wordlength to 64 bits. Simulations with DDR and 64-bit

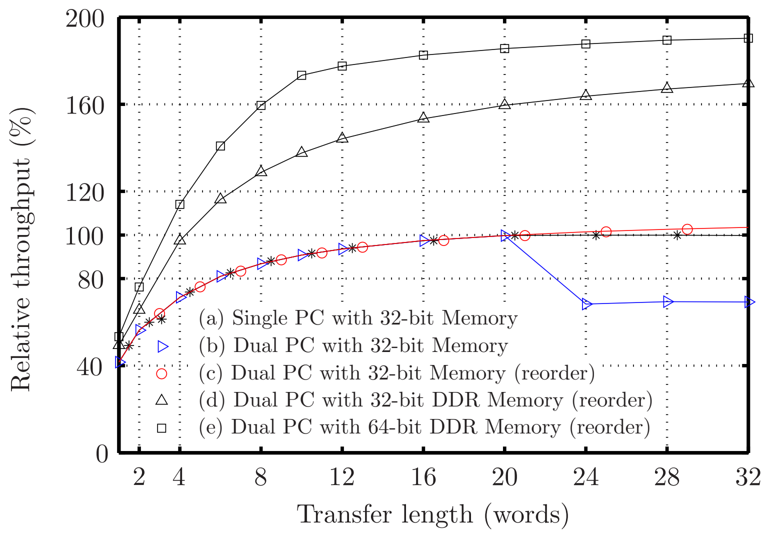


Figure 14: (a-e) Throughput relative to execution on a single processing cell. Simulations show mappings to single and dual processing cells for different memory configurations.

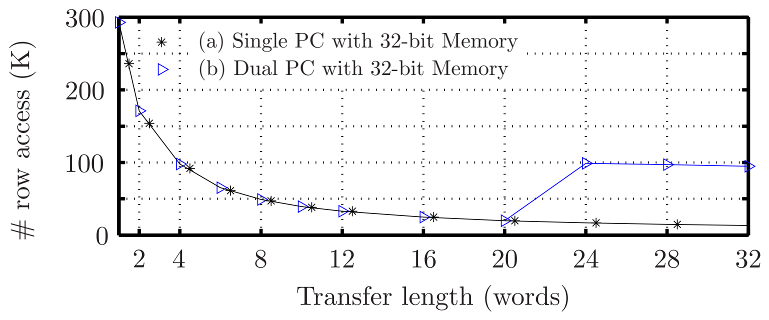
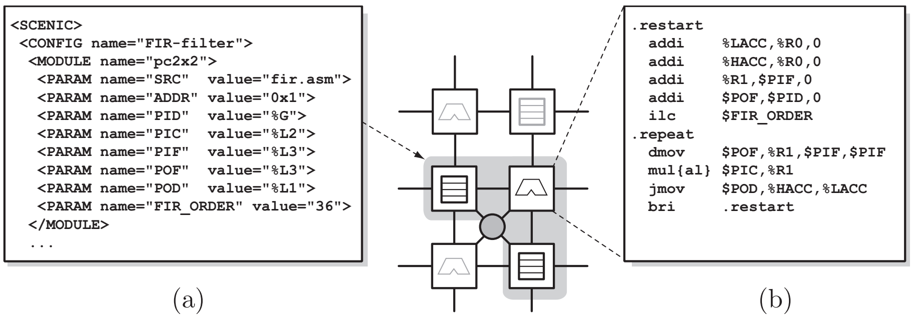


Figure 15: (a-b) Number of row address strobes to external memory. A high number indicates an irregular access pattern, which has negative impact on throughput.

memory are shown in Figure 14 plot (d-e), and presents dramatically increased data throughput as expected. Hence, it illustrates that system performance is a combination of many design parameters.



```
<SCENIC>
 <CONFIG name="FIR-filter">
  <MODULE name="pc2x2">
   <PARAM name="SRC"  value="fir.asm">
   <PARAM name="ADDR" value="0x1">
   <PARAM name="PID"  value="%G">
   <PARAM name="PIC"  value="%L2">
   <PARAM name="PIF"  value="%L3">
   <PARAM name="POF"  value="%L3">
   <PARAM name="POD"  value="%L1">
   <PARAM name="FIR_ORDER" value="36">
  </MODULE>
  ...
```

(a)

```
.restart
  addi    %LACC,%R0,0
  addi    %HACC,%R0,0
  addi    %R1,$PIF,0
  addi    $POF,$PID,0
  ilc     $FIR_ORDER
.repeat
  dmov    $POF,%R1,$PIF,$PIF
  mul{al} $PIC,%R1
  jmov    $POD,%HACC,%LACC
  bri     .restart
```

(b)

Figure 16: Application mapping (FIR filter) that allocates one processing cell and two memory cells (grey). (a) The XML configuration specifies a source file and parameters for code generation, and from which ports data are streaming. (b) A generic assembly program that use the XML parameters as port references.

5.3 Application Mapping

Currently, applications are manually mapped to the DRA using *configurations*. Configurations are specified in XML format, to allow rapid design exploration, which can be translated to binary files and reloaded by the embedded processor during run-time. A configuration allocates a set of resource cells, and contains parameters to control the application mapping, as shown in Figure 16(a). For memory cells, these parameters configure the memory descriptors and the initial memory contents. For processing cells, the parameters specify a program *source file* and *port mappings* to assist the code generator. The source file is described using processor-specific assembly code, as presented in Part III in Section 4.1, and an example is shown in Figure 16(b). Hence, algorithm source files only describe how to *process* data, while port mappings describe how to *stream* data.

The following applications have been evaluated for the proposed DRA, which are common digital signal processing algorithms and are also used in image reconstruction for digital holographic imaging:

- **FIR Filter -** The time-multiplexed and pipeline FIR filters are mapped to the DRA using MAC processors. The time-multiplexed design requires one MAC unit and two memory cells, one for coefficients and one operating as a circular buffer for data values. The inner loop counter is used to efficiently iterate over data values and coefficients, which are multiplied pair-wise and accumulated. When an iteration completes, the least recent value in the circular buffer is discarded and replaced with the value on the input port. At the same time, the result from accumulation is written to the output port. The time-multiplexed FIR implementation is illustrated in Figure 17(a), and the corresponding mapping to resource cells is shown in Figure 17(b). In contrast, the pipeline design requires one MAC unit for each filter coefficient, which are serially connected to form a pipeline. Each unit multiplies the input value with the coefficient, adds the partial sum from the preceding stage, and forwards the data value and result to the next stage.

- **Radix-2 FFT -** The time-multiplexed and pipeline FFT cores are mapped to the DRA using DSP and CORDIC processors. DSPs are used for the butterfly operations, while CORDIC units emulate complex multiplication using vector rotation. Delay feedback units are connected to each DSP butterfly, which are implemented using memory cells operating in FIFO mode. An example of an FFT stage is illustrated in Figure 17(c), and the corresponding mapping to resource cells is shown in Figure 17(d). For the time-multiplexed design, data is streamed through the DSP and CORDIC units n times for a 2^n-point FFT, where the butterfly size changes for every iteration. The pipeline radix-2 design is constructed from n DSP and $n-1$ CORDIC units, which significantly increases the throughput but also requires more hardware resources.

- **Matrix Transpose -** A matrix transpose operation is mapped to illustrate that the DSP processors may alternatively be used as address generation units (AGU). A fast transpose operation requires two DSP units to generate read and write addresses, and the data is double-buffered inside a memory cell. While one buffer is being filled linearly, the other is drained using an addressing mode to transpose the data. When both operations finish, the DSP units switch buffers to transpose the next block.

Table 2 presents the mapping results in terms of reconfiguration time, throughput, and resource requirements. It can be seen that the reconfiguration time is negligible for most practical applications. A configuration is assumed to be active for a much longer time than it takes to partially reconfigure the resource cells. It is also shown that both the pipeline FFT and the fast matrix transpose unit generates a throughput of close to 1 sample per clock cycle, hence these configurations are candidates for mapping the presented reconstruction algorithm in digital holographic imaging, as discussed in Part I.

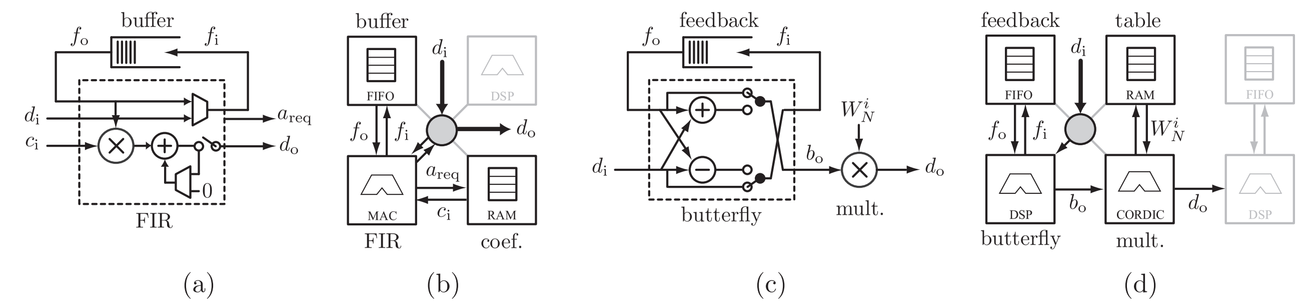


Figure 17: Examples of application mapping: (a) Data flow in a time-multiplexed FIR filter unit. (b) Mapping of MAC unit, buffer, and coefficients to the DRA. (c) Data flow graph of a radix-2 butterfly and complex multiplier. (d) Mapping of butterfly and complex multiplier to the DRA. The input data stream from the SMC is received from the global network.

Table 2: Application mapping results in terms of reconfiguration time, throughput, and resource requirements.

Application/Algorithm ($\mathcal{A}$)	Reconfiguration time t_r (cc)	Throughput (Samples/cc)	#PC (DSP)	#PC (MAC)	#PC (CORDIC)	#MC
FIR (N taps) TM	$12 + [8 + N]$	$1/(7 + 2N)$	0	1	0	2
FIR (N taps) pipeline	$10N$	$1/2$	0	N	0	0
Radix-2 FFT (2^n bins) TM	$24 + [2^n/2]$	$\approx 1/n$	1	0	1	2
Radix-2 FFT (2^n bins) pipeline	$9n + [6n + 2^n]$	≈ 1	n	0	$n - 1$	$\leq 2n$-1
Transpose ($N \times N$ block)	$7 + [3]$	$\approx 1/2$	1	0	0	1
Transpose ($N \times N$ block)	$14 + [3]$	$N/(N + 2) \approx 1$	2	0	0	1

6 Hardware Prototyping

The SCENIC models have been translated to VHDL, and verified using the same configuration as during design exploration and analysis. Currently available VHDL models are the DSP and MAC processors presented in Section 3.2, the memory cell presented in Section 3.3, and the router presented in Section 3.4. VHDL implementations have been individually synthesized to explore different parameter settings, and integrated to construct arrays of size 4×4 and 8×8. The results from logic synthesis in a $0.13\,\mu$m technology are presented in Table 3, where each design has been synthesized with the following configuration parameters:

R	Number of general purpose registers
L	Number of local ports
D	Number of descriptors in each memory cell
$\mathbf{M}_{\mathrm{pc}}$	Number of 32-bit words of program memory
$\mathbf{M}_{\mathrm{mc}}$	Number of 32-bit words in the memory array
Q	Router output queue depth

The table also presents the maximum frequency and the memory storage space inside each hardware unit. The router overhead illustrates how large part of the system resources that is spent on global routing. Synthesis results show how configuration parameters affect the area, frequency, and required storage space. The DSP and MAC units are in the same area range, but the memory cell is slightly larger. When constructing an array of cells, it is important to choose cells that have similar area requirements. Hence, a memory cell with $M_{\mathrm{mc}} = 256$ is a trade-off between area and memory space, since it is comparable is size with the processing cells. For routers, it can be seen that the output queue depth is associated with large hardware cost. To avoid overhead from routing resources, it is important to minimize the queue depth. Hence, a router with $Q = 4$ has been chosen as a trade-off between area and network performance. As an example, the floorplan and layout of a 4×4 array, with 8 processing cells and 8 memory cells, are shown in Figure 18 and 19, respectively. The floorplan size is $1660 \times 1660\,\mu\mathrm{m}^2$ (90% core utilization).

7 Comparison and Discussion

Table 4 presents a comparison between different platforms for computing a 256-point FFT, to relate the hardware cost and performance to existing architectures. The XSTREAM FFT core from Part II is presented as an *application-specific* alternative, with low area requirements and high performance. Hardware requirements for the time-multiplexed and pipelined FFT from Section 5.3

Table 3: Synthesis results for the processor, memory, router, and array. The results are based on a $0.13\,\mu$m cell library. $M_{\mathrm{pc}} = 64$ for all processing cells.

Architecture	Eq. Kgates (NAND2)	Area (mm^2)	$f_{\max}$ (MHz)	Storage (bytes)	Router overhead
DSP $(R = 4, L = 2)$	17.3	0.089	325	256	-
DSP $(R = 8, L = 4)$	20.7	0.106	325	256	-
DSP $(R = 16, L = 8)$	27.6	0.141	325	256	-
MAC $(R = 4, L = 2)$	17.4	0.089	325	256	-
MAC $(R = 8, L = 4)$	20.5	0.105	325	256	-
MAC $(R = 16, L = 8)$	25.0	0.128	325	256	-
CORDIC (16-bit)	24.7*	0.126	1100	0	-
MC $(D = 4, M_{\mathrm{mc}} = 256)$	30.2	0.155	460	1024	-
MC $(D = 4, M_{\mathrm{mc}} = 512)$	41.3	0.211	460	2048	-
MC $(D = 4, M_{\mathrm{mc}} = 1024)$	62.5	0.320	460	4096	-
4-port Router $(Q = 4)$	12.9	0.066	1000	4×16	-
4-port Router $(Q = 8)$	22.4	0.115	1000	4×32	-
4-port Router $(Q = 16)$	35.5	0.182	1000	4×64	-
5-port Router $(Q = 4)$	17.0	0.087	990	5×16	-
8-port Router $(Q = 4)$	28.4	0.145	890	8×16	-
4x4 Array (5 routers)	484	2.48	325	10 K	15.8%
8x8 Array (21 routers)	1790	9.18	325	40 K	17.7%

* CORDIC pipeline presented in Part I. Control overhead is not included.

are estimated based on how many resource cells that are allocation for each application. Table 4 also includes a recently proposed medium-grained architecture for mapping a 256-point FFT [116]. Finally, commercial CPU and DSP processors are presented to compare with *general-purpose* and *special-purpose* architectures.

Compared with an application-specific solution, a time-multiplexed version of the FFT can be mapped to the proposed DRA at an even lower cost, but with the penalty of reduced performance. In contrast, the proposed pipeline FFT generates the same throughput as the application specific solution, but with four times higher cost. Based on this application, this is the *price for flexible* for a reconfigurable architecture.

The area requirement for the proposed DRA of size 4×4 is $2.48\,\mathrm{mm}^2$, as shown in Table 3. As a comparison, the PowerPC 405F6 embedded processor has a core area of $4.53\,\mathrm{mm}^2$ (with caches) in the same implementation technology [103], while the Pentium 4 processor requires $305\,\mathrm{mm}^2$. Area numbers for the TMS320VC55 DSP are not available.

Table 4: Architectural comparison for computing a 256-point FFT. Latency is not considered. Required area is scaled to 0.13 μm technology. Performance/Area ratio is (samples per cc) / (required area).

Architecture	Required Area (mm^2)	$f_{\max}$ (MHz)	Cycles (cc)	Performance/ Area ratio
XSTREAM 256-point FFT	0.51	398	256	1.96
Proposed DRA (TM)	0.31	325	1536	0.54
Proposed DRA (pipeline)	2.04	325	256	0.49
Medium-Grain DRA [116]	32.08	267	256	0.03
Texas TMS320VC5502 [129]	-	300	4786	-
IBM PowerPC 405 F6 [103]	4.53	658	$> 20\text{K}^*$	< 0.0028
Intel Pentium 4 [130]	305	3000	20K	0.00004

* Estimated to require more clock cycles than an Intel Pentium 4.

Recent related work proposes a medium-grained architecture for mapping a 256-point FFT [116]. However, Table 4 shows that this architecture results in poor performance/area ratio due to only 4-bit granularity in processing elements. With the same throughput, the area requirement is 16 times higher than for the proposed DRA.

DSP architectures include MAC support, but still requires more than 18 times the number of clock cycles over pipelined solutions [129]. General purpose processors require even more processing, due to the lack of dedicated hardware for MAC operations [130]. General-purpose architectures gain performance through higher clock frequency, but this solution is not scalable. In contrast, presented architectures result in high performance and low area requirements.

8 Conclusion

Modeling and implementation of a dynamically reconfigurable architecture has been presented. The reconfigurable architecture is based on an array of processing and memory cells, communicating using local interconnects and a hierarchical network. The SCENIC exploration environment and models have been used to evaluate the architecture and to emulate application mapping. Various network, memory, and application scenarios have been evaluated using SCENIC, to facilitate system tuning during the design phase. A 4×4 array of processing cells, memory cells, and routers has been implemented in VHDL and synthesized for a 0.13 μm cell library. The design has a core size of 2.48 mm^2 and is capable of operating up to 325 MHz. It is shown that mapping of a 256-point FFT generate 18 times higher throughput than a traditional DSP solution.

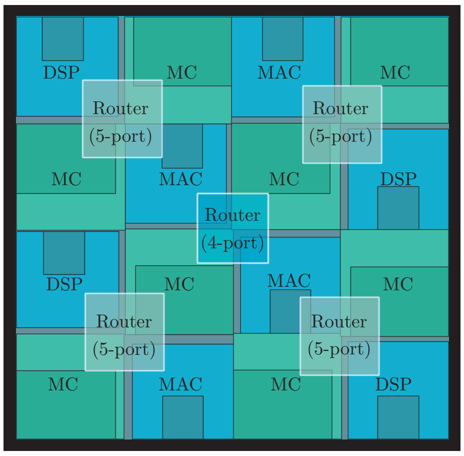


Figure 18: Floorplan of an 4×4 array of 8 processing cells (blue) and 8 memory cells (green) connected with 5 network routers. The internal memories in each resource cell are represented in slightly darker color. For this design $R = 8$, $L = 4$, $D = 4$, $M_{\mathrm{pc}} = 64$, $M_{\mathrm{mc}} = 256$, and $Q = 4$.

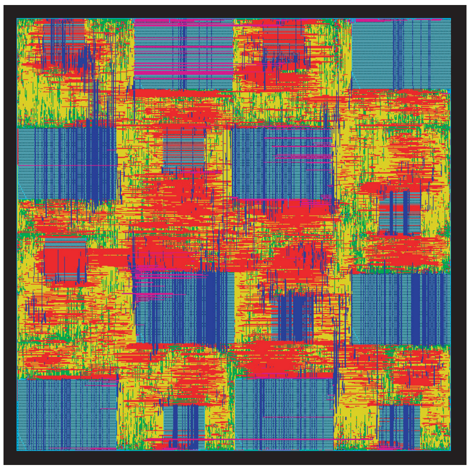


Figure 19: Final layout of a 4×4 array of processing and memory cells connected with network routers. The core size is $1660 \times 1660 \mu\mathrm{m}^2$.

Conclusion and Outlook

Flexibility and reconfigurability are becoming important aspects of hardware design, where the focus is the ability to dynamically adapt a system to a range of various applications. There are many ways to achieve reconfigurability, and two approaches are evaluated in this work. The target application is digital holographic imaging, where visible images are to be reconstructed based on holographic recordings.

In the first part of the thesis, an application-specific hardware accelerator and system platform for digital holographic imaging is proposed. A prototype is designed and constructed to demonstrate the potential of hardware acceleration in this application field. The presented result is a hardware platform that achieves real-time image reconstruction when implemented in a modern technology. It is concluded that application-specific architectures are required for real-time performance, and that block-level reconfiguration provides sufficient flexibility for this application.

In the second part of the thesis, the work is generalized towards dynamically reconfigurable architectures. Modeling and implementation results indicate the processing potential, and the cost of flexibility, of such architectures. An exploration framework and simulation modules for evaluating reconfigurable platforms are designed to study architectural trade-offs, system performance, and application mapping. An array of run-time reconfigurable processing and memory cells is proposed, and it is shown that arrays of simplified programmable processing cells, with regular interconnects and memory structures, result in increased performance over traditional DSP solutions. It is concluded that reconfigurable architectures provide a feasible solution to accelerate arbitrary applications.

Current trends towards parallel and reconfigurable architectures indicate the beginning of a paradigm shift for hardware design. Within a few years, we are likely to find massively parallel architectures in desktop computers and inside embedded systems. This paradigm shift will certainly also require us to change our mindset and our view on traditional software design, to allow efficient mapping onto such architectures.

Bibliography

[1] U. Schnars and W. Jueptner, *Digital Holography.* Springer-Verlag, Berlin, Heidelberg: Springer, 2005.

[2] J. M. Jabaey, A. Chandrakasan, and B. Nikolić, *Digital Integrated Circuits.* Upper Saddle River, New Jersey: Prentice-Hall, 2003.

[3] J. Shalf, "The New Landscape of Parallel Computer Architecture," *Journal of Physics: Conference Series 78*, pp. 1–15, 2007.

[4] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *AFIPS Conference Proceedings*, Vol. 30, Atlantic City, USA, Apr. 18-20 1967, pp. 483–485.

[5] G. E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, 1965.

[6] C. A. R. Hoare, *Communicating Sequential Processes.* Upper Saddle River, New Jersey: Prentice-Hall, 1985.

[7] A. Marowka, "Parallel Computing on Any Desktop," *Communications of the ACM*, vol. 50, no. 9, pp. 74–78, 2007.

[8] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and Automated Multiprocessor System Design, Programming, and Implementation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 3, pp. 542–555, 2008.

[9] T. Lenart and V. Öwall, "A 2048 complex point FFT processor using a novel data scaling approach," in *Proceedings of IEEE International Symposium on Circuits and Systems*, Bangkok, Thailand, May 25-28 2003, pp. 45–48.

[10] Y. Chen, Y.-C. Tsao, Y.-W. Lin, C.-H. Lin, and C.-Y. Lee, "An Indexed-Scaling Pipelined FFT Processor for OFDM-Based WPAN Applications," *IEEE Transactions on Circuits and Systems—Part II: Analog and Digital Signal Processing*, vol. 55, no. 2, pp. 146–150, Feb. 2008.

[11] Computer Systems Laboratory, University of Campinas , "The ArchC Architecture Description Language," http://archc.sourceforge.net.

[12] M. J. Flynn, "Area - Time - Power and Design Effort: The Basic Tradeoffs in Application Specific Systems," in *Proceedings of IEEE 16th International Conference on Application-specific Systems, Architectures and Processors*, Samos, Greece, July 23-25 2005, pp. 3–6.

[13] R. Tessier and W. Burleson, "Reconfigurable Computing for Digital Signal Processing: A Survey," *The Journal of VLSI Signal Processing*, vol. 28, no. 1-2, pp. 7–27, 2001.

[14] S.A. McKee *et al.*, "Smarter Memory: Improving Bandwidth for Streamed References," *Computer*, vol. 31, no. 7, pp. 54–63, July 1998.

[15] S. Mahadevan, M. Storgaard, J. Madsen, and K. Virk, "ARTS: A System-level Framework for Modeling MPSoC Components and Analysis of their Causality," in *Proceedings of 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Sept. 27-29 2005.

[16] G. Beltrame, D. Sciuto, and C. Silvano, "Multi-Accuracy Power and Performance Transaction-Level Modeling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 10, pp. 1830–1842, 2007.

[17] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "A Scalable Synthesis Methodology for Application-Specific Processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 11, pp. 1175–1188, 2006.

[18] D. Talla, L. K. John, V. Lapinskii, and B. L. Evans, "Evaluating Signal Processing and Multimedia Applications on SIMD, VLIW and Superscalar Architectures," in *Proceedings of IEEE International Conference on Computer Design*, Austin, Texas, USA, Sept. 11720 2000, pp. 163–172.

[19] A. Douillet and G. R. Gao, “Software-Pipelining on Multi-Core Architectures,” in *Proceedings of International Conference on Parallel Architecture and Compilation Techniques*, Brasov, Romania, Sept. 1519 2007, pp. 39–48.

[20] K. Keutzer, S. Malik, and R. Newton, “From ASIC to ASIP: The Next Design Discontinuity,” in *Proceedings of IEEE International Conference on Computer Design*, Freiburg, Germany, Sept. 16-18 2002, pp. 84–90.

[21] C. Wolinski and K. Kuchcinski, “Identification of Application Specific Instructions Based on Sub-Graph Isomorphism Constraints,” in *Proceedings of IEEE 14th International Conference on Application-specific Systems, Architectures and Processors*, Montréal, Canada, July 9-11 2007, pp. 328–333.

[22] A. C. Cheng, “A Software-to-Hardware Self-Mapping Technique to Enhance Program Throughput for Portable Multimedia Workloads,” in *Proceedings of IEEE International Symposium on Electronic Design, Test and Applications*, Hong Kong, China, Jan. 23-25 2008, pp. 356–361.

[23] Tensilica, “Configurable and Standard Processor Cores for SOC Design,” http://www.tensilica.com.

[24] K. K. Parhi, *VLSI Digital Signal Processing Systems.* 605 Third Avenue, New York 10158: John Wiley and Sons, 1999.

[25] U. Kapasi, S. Rixner, W. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. Owens, “Programmable Stream Processors,” *Computer*, vol. 36, no. 8, pp. 54–62, Aug. 2003.

[26] U. Kapasi, W. Dally, S. Rixner, J. Owens, and B. Khailany, “The Imagine Stream Processor,” in *Proceedings of IEEE International Conference on Computer Design*, Freiburg, Germany, Sept. 16-18 2002, pp. 282–288.

[27] W. J. Dally *et al.*, “Merrimac: Supercomputing with Streams,” in *Proceedings of ACM/IEEE Conference on Supercomputing*, Phoenix, AZ, USA, Nov. 15-21 2003, pp. 35–42.

[28] D. Manocha, “General-Purpose Computations Using Graphics Processors,” *Computer*, vol. 38, no. 8, pp. 85–88, 2005.

[29] M. J. Flynn, “Some Computer Organizations and Their Effectiveness,” *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.

[30] M. Gokhale and P. S. Graham, *Reconfigurable Computing.* Springer-Verlag, Berlin, Heidelberg: Springer, 2005.

[31] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys* , vol. 34, no. 2, pp. 171–210, 2002.

[32] T. Todman, G. Constantinides, S. Wilton, O. Mencer, W. Luk, and P. Cheung, "Reconfigurable Computing: Architectures and Design Methods," *IEE Proceedings of Computers and Digital Techniques*, vol. 152, no. 2, pp. 193–207, 2005.

[33] R. Hartenstein, "A Decade of Reconfigurable Computing: A Visionary Retrospective," in *Proceedings of IEEE Conference on Design, Automation and Test in Europe*, Munich, Germany, Mar. 10-14 2001, pp. 642–649.

[34] C. Hilton and B. Nelson, "PNoC: A Flexible Circuit-switched NoC for FPGA-based Systems," *IEE Proceedings on Computers and Digital Techniques*, vol. 153, pp. 181–188, May 2006.

[35] R. Baxter *et al.*, "The FPGA High-Performance Computing Alliance Parallel Toolkit ," in *Proceedings of NASA/ESA Conference on Adaptive Hardware and Systems*, Scotland, United Kingdom, Aug. 5-8 2007, pp. 301–310.

[36] S. Raaijmakers and S. Wong, "Run-time Partial Reconfiguration for Removal, Placement and Routing on the Virtex-II Pro," in *Proceedings of IEEE International Conference on Field Programmable Logic and Applications*, Amsterdam, Netherlands, Aug. 27-29 2007, pp. 679–683.

[37] S. Sirowy, Y. Wu, S. Lonardi, and F. Vahid, "Two-Level Microprocessor-Accelerator Partitioning," in *Proceedings of IEEE Conference on Design, Automation and Test in Europe*, Acropolis, Nice, France, Apr. 16-20 2007, pp. 313–318.

[38] D. Burger, J. Goodman, and A. Kagi, "Limited Bandwidth to Affect Processor Design," *IEEE Micro*, vol. 17, no. 6, pp. 55–62, Nov. 1997.

[39] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens, "Memory Access Scheduling," in *Proceedings of the 27th International Symposium on Computer Architecture*, Vancouver, BC Canada, June 10-14 2000, pp. 128–138.

[40] M. Brandenburg, A. Schöllhorn, S. Heinen, J. Eckmüller, and T. Eckart, "A Novel System Design Approach Based on Virtual Prototyping and its Consequences for Interdisciplinary System Design Teams," in *Proceedings of IEEE Conference on Design, Automation and Test in Europe*, Acropolis, Nice, France, Apr. 16-20 2007, pp. 1–3.

[41] D. C. Black and J. Donovan, *SystemC: From the Ground Up.* Springer-Verlag, Berlin, Heidelberg: Springer, 2005.

[42] A. Donlin, "Transaction Level Modeling: Flows and Use Models," in *Proceedings of IEEE International Conference on Hardware/Software Codesign and System Synthesis*, Stockholm, Sweden, Sept. 8-10 2004, pp. 75–80.

[43] G. Martin, B. Bailey, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology.* San Francisco, USA: Morgan Kaufmann, 2007.

[44] T. Kogel and M. Braun, "Virtual Prototyping of Embedded Platforms for Wireless and Multimedia," in *Proceedings of IEEE Conference on Design, Automation and Test in Europe*, Munich, Germany, Mar. 6-10 2006, pp. 1–6.

[45] T. Rissa, P. Y. Cheung, and W. Luk, "Mixed Abstraction Execution for the SoftSONIC Virtual Hardware Platform," in *Proceedings of Midwest Symposium on Circuits and Systems*, Cincinnati, Ohio, USA, Aug. 7-10 2005, pp. 976–979.

[46] L. Cai and D. Gajski, "Transaction Level Modeling: An Overview," in *Proceedings of IEEE International Conference on Hardware/Software Codesign and System Synthesis*, Newport Beach, CA, USA, Oct. 1-3 2003, pp. 19–24.

[47] T. Kempf *et al.*, "A SW Performance Estimation Framework for early System-Level-Design using Fine-grained Instrumentation," in *Proceedings of IEEE Conference on Design, Automation and Test in Europe*, Munich, Germany, Mar. 6-10 2006.

[48] M. Eteläperä, J. V. Anttila, and J. P. Soininen, "Architecture Exploration of 3D Video Recorder Using Virtual Platform Models," in *10th Euromicro Conference on Digital System Design*, Lübeck, Germany, Aug. 29-31 2007.

[49] S. Rigo, G. Araújo, M. Bartholomeu, and R. Azevedo, “ArchC: A SystemC-based Architecture Description Language,” in *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, Foz do Iguacu, Brazil, Oct. 27-29 2004, pp. 163–172.

[50] M. Reshadi, P. Mishra, and N. Dutt, “Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation,” in *Proceedings of the 40th Design Automation Conference*, Anaheim, CA, USA, June 2-6 2003, pp. 758–763.

[51] D. Gabor, “A New Microscopic Principle,” *Nature*, vol. 161, pp. 777–778, 1948.

[52] W. E. Kock, *Lasers and Holography.* 180 Varick Street, New York 10014: Dover Publications Inc., 1981.

[53] Wenbo Xu and M. H. Jericho and I. A. Meinertzhagen and H. J. Kreuzer, “Digital in-line Holography for Biological Applications,” *Cell Biology*, vol. 98, pp. 11 301–11 305, Sept. 2001.

[54] C. M. Do and B. Javidi, “Digital Holographic Microscopy for Live Cell Applications and Technical Inspection,” *Applied Optics*, vol. 47, no. 4, pp. 52–61, 2008.

[55] B. Alberts *et al.*, *Molecular Biology Of The Cell.* London, England: Taylor & Francis Inc, 2008.

[56] F. Zernike, “Phase Contrast, A new Method for the Microscopic Observation of Transparent Objects,” *Physica*, vol. 9, no. 7, pp. 686–698, 1942.

[57] P. Marquet, B. Rappaz, P. J. Magistretti, E. Cuche, Y. Emery, T. Colomb, and C. Depeursinge, “Digital Holographic Microscopy: A Noninvasive Contrast Imaging Technique allowing Quantitative Visualization of Living Cells with Subwavelength Axial Accuracy,” *Optics letters*, vol. 30, no. 7, pp. 468–470, 2005.

[58] A. Asundi and V. R. Singh, “Sectioning of Amplitude Images in Digital Holography,” *Measurement Science and Technology*, vol. 17, pp. 75–78, 2006.

[59] C. M. Do and B. Javidi, “Multifocus Holographic 3-D Image Fusion Using Independent Component Analysis,” *Journal of Display Technology*, vol. 3, no. 3, pp. 326–332, 2007.

[60] M. Born and E. Wolf, *Principles of Optics.* Cambridge, U.K.: Cambridge University Press, 1999.

[61] T.H. Demetrakopoulos and R. Mittra, "Digital and Optical Reconstruction of Suboptical Diffraction Patterns," *Applied Optics*, vol. 13, pp. 665–670, Mar. 1974.

[62] W. G. Rees, "The Validity of the Fresnel Approximation," *European Journal of Physics*, vol. 8, pp. 44–48, 1987.

[63] S. Mezouari and A. R. Harvey, "Validity of Fresnel and Fraunhofer Approximations in Scalar Diffraction," *Journal of Optics A: Pure and Applied Optics*, vol. 5, pp. 86–91, 2003.

[64] D. C. Ghiglia and M. D. Pritt, *Two-Dimensional Phase Unwrapping.* 605 Third Avenue, New York 10158: John Wiley and Sons, 1998.

[65] T. J. Flynn, "Two-dimensional Phase Unwrapping with Minimum Weighted Discontinuity," *Optical Society of America*, vol. 14, no. 10, pp. 2692–2701, 1997.

[66] D. Litwiller, "CCD vs. CMOS: Facts and Fiction," in *Photonics Spectra*, 2001.

[67] M. Frigo, "FFTW: An Adaptive Software Architecture for the FFT," in *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, Seattle, Washington, USA, May 12-15 1998, pp. 1381–1384.

[68] E. O. Brigham, *The Fast Fourier Transform and its Applications.* Upper Saddle River, New Jersey: Prentice-Hall, 1988.

[69] Micron Technology, "MT48LC32M16A2-75 - SDR Synchronous DRAM device," http://www.micron.com.

[70] B. Parhami, *Computer Arithmetic.* 198 Madison Avenue, New York 10016: Oxford University Press, 2000.

[71] S. He and M. Torkelson, "Designing Pipeline FFT Processor for OFDM (de)modulation," in *URSI International Symposium on Signals, Systems, and Electronics*, Pisa, Italy, Sept. 29-Oct. 2 1998, pp. 257–262.

[72] T. Lenart and V. Öwall, "Architectures for Dynamic Data Scaling in 2/4/8K Pipeline FFT Cores," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 11, pp. 1286–1290, Nov. 2006.

[73] E. Bidet, C. Joanblanq, and P. Senn, “A Fast Single-Chip Implementation of 8192 Complex Point FFT,” *IEEE Journal of Solid-State Circuits*, vol. 30, pp. 300–305, Mar. 1995.

[74] C.D. Toso *et al.*, “0.5-μm CMOS Circuits for Demodulation and Decoding of an OFDM-Based Digital TV Signal Conforming to the European DVB-T Standard,” *IEEE Journal of Solid-State Circuits*, vol. 33, no. 11, pp. 1781–1792, Nov. 1998.

[75] M. Wosnitza, M. Cavadini, M. Thaler, and G. Tröster, “A High Precision 1024-point FFT Processor for 2D Convolution,” in *Proceedings of IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, San Fransisco, CA, USA, Feb. 5-7 1998, pp. 118–119.

[76] F. Kristensen, P. Nilsson, and A. Olsson, “Reduced Transceiver-delay for OFDM Systems,” in *Proceedings of Vehicular Technology Conference*, vol. 3, Milan, Italy, May 17-19 2004, pp. 1242–1245.

[77] J. Gaisler, “A Portable and Fault-tolerant Microprocessor based on the SPARC v8 Architecture,” in *Proceedings of Dependable Systems and Networks*, Washington DC, USA, June 23-26 2002, pp. 409–415.

[78] ARM Ltd., “AMBA Specification - Advanced Microcontroller Bus Architecture,” http://www.arm.com, 1999.

[79] N. Miyamoto, L. Karnan, K. Maruo, K. Kotani, and T. Ohmi1, “A Small-Area High-Performance 512-Point 2-Dimensional FFT Single-Chip Processor,” in *Proceedings of European Solid-State Circuits*, Estoril, Portugal, Sept. 16-18 2003, pp. 603–606.

[80] I. Uzun, A. Amira, and F. Bensaali, “A Reconfigurable Coprocessor for High-resolution Image Filtering in Real time,” in *Proceedings of the 10th IEEE International Conference on Electronics, Circuits and Systems*, Sharjah, United Arab Emirates, Dec. 14-17 2003, pp. 192–195.

[81] IEEE 802.11a, “High-speed Physical Layer in 5 GHz Band,” http://ieee802.org, 1999.

[82] IEEE 802.11g, “High-speed Physical Layer in 2.4 GHz Band,” http://ieee802.org, 2003.

[83] J. Cooley and J. Tukey, “An Algorithm for Machine Calculation of Complex Fourier Series,” *IEEE Journal of Solid-State Circuits*, vol. 19, pp. 297–301, Apr. 1965.

[84] K. Zhong, H. He, and G. Zhu, “An Ultra High-speed FFT Processor,” in *International Symposium on Signals, Circuits and Systems*, Lasi, Romania, July 10-11 2003, pp. 37–40.

[85] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing - Principles, algorithms, and applications.* Upper Saddle River, New Jersey: Prentice-Hall, 1996.

[86] S. He, “Concurrent VLSI Architectures for DFT Computing and Algorithms for Multi-output Logic Decomposition,” Ph.D. dissertation, Ph.D. dissertation, Lund University, Department of Applied Electronics, 1995.

[87] M. Gustafsson *et al.*, “High resolution digital transmission microscopy - a Fourier holography approach,” *Optics and Lasers in Engineering*, vol. 41(3), pp. 553–563, Mar. 2004.

[88] ETSI, “Digital Video Broadcasting (DVB); Framing Structure, Channel Coding and Modulation for Digital Terrestrial Television,” ETSI EN 300 744 v1.4.1, 2001.

[89] K. Kalliojrvi and J. Astola, “Roundoff Errors is Block-Floating-Point Systems,” *IEEE Transactions on Signal Processing*, vol. 44, no. 4, pp. 783–790, Apr. 1996.

[90] Y.-W. Lin, H.-Y. Liu, and C.-Y. Lee, “A Dynamic Scaling FFT Processor for DVB-T Applications,” *IEEE Journal of Solid-State Circuits*, vol. 39, no. 11, pp. 2005–2013, Nov. 2004.

[91] C.-C. Wang, J.-M. Huang, and H.-C. Cheng, “A 2K/8K Mode Small-area FFT Processor for OFDM Demodulation of DVB-T Receivers,” *IEEE Transactions on Consumer Electronics*, vol. 51, no. 1, pp. 28–32, Feb. 2005.

[92] T. Lenart *et al.*, “Accelerating signal processing algorithms in digital holography using an FPGA platform,” in *Proceedings of IEEE International Conference on Field Programmable Technology*, Tokyo, Japan, Dec. 15-17 2003, pp. 387–390.

[93] X. Ningyi *et al.*, “A SystemC-based NoC Simulation Framework supporting Heterogeneous Communicators,” in *Proceedings IEEE International Conference on ASIC*, Shanghai, China, Sept. 24-27 2005, pp. 1032–1035.

[94] L. Yu, S. Abdi, and D. D. Gajski, “Transaction Level Platform Modeling in SystemC for Multi-Processor Designs,” UC Irvine, Tech. Rep., Jan. 2007. [Online]. Available: http://www.gigascale.org/pubs/988.html

[95] A.V. Brito *et al.*, "Modelling and Simulation of Dynamic and Partially Reconfigurable Systems using SystemC," in *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, Porto Alegra, Brazil, Mar. 9-11 2007, pp. 35–40.

[96] Open SystemC Initiative (OSCI), "OSCI SystemC 2.2 Open-source Library," http://www.systemc.org.

[97] F. Doucet, S. Shukla, and R. Gupta, "Introspection in System-Level Language Frameworks: Meta-level vs. Integrated," in *Proceedings of IEEE Conference on Design, Automation and Test in Europe*, Munich, Germany, Mar. 3-7 2003.

[98] W. Klingauf and M. Geffken, "Design Structure Analysis and Transaction Recording in SystemC Designs: A Minimal-Intrusive Approach," in *Proceedings of Forum on Specification and Design Languages*, TU Darmstadt, Germany, Sept. 19-22 2006.

[99] D. Berner *et al.*, "Automated Extraction of Structural Information from SystemC-based IP for Validation," in *Proceedings of IEEE International Workshop on Microprocessor Test and Verification*, Nov. 3-4 2005.

[100] F. Rogin, C. Genz, R. Drechsler, and S. Rulke, "An Integrated SystemC Debugging Environment," in *Proceedings of Forum on Specification and Design Languages*, Sept. 18-20 2007.

[101] E. Gansner *et al.*, "Dot user guide - Drawing graphs with dot," http://www.graphviz.org/Documentation/dotguide.pdf.

[102] The Spirit Consortium, "Enabling Innovative IP Re-use and Design Automation," http://www.spiritconsortium.org.

[103] IBM Microelectronics, "PowerPC 405 CPU Core," http://www.ibm.com.

[104] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–211, 2002.

[105] T. Todman, G. Constantinides, S. Wilton, O. Mencer, W. Luk, and P. Cheung, "Reconfigurable computing: architectures and design methods," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 2, pp. 193–207, 2005.

[106] G. K. Rauwerda, P. M. Heysters, and G. J. Smit, “Towards Software Defined Radios Using Coarse-Grained Reconfigurable Hardware,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 3–13, 2008.

[107] T. J. Callahan, J. R. Hauserand, and J. Wawrzynek, “The Garp architecture and C compiler,” *IEEE Computer*, vol. 33, no. 4, pp. 62–69, 2000.

[108] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, “PipeRench: A Reconfigurable Architecture and Compiler,” *IEEE Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[109] M. B. Taylor *et al.*, “The Raw microprocessor: A Computational Fabric for Software Circuits and General-purpose Programs,” *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.

[110] T. Miyamori and K. Olukotun, “A Quantitative Analysis of Reconfigurable Coprocessorsfor Multimedia Applications,” in *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, Apr. 15-17 1998, pp. 2–11.

[111] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, “The Chimaera Reconfigurable Functional Unit,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 2, pp. 206–217, 2004.

[112] S. Khawam, I. Nousias, M. Milward, Y. Ying, M. Muir, and T. Arslan, “The Reconfigurable Instruction Cell Array,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 75–85, 2008.

[113] D. Kissler, F. Hannig, A. Kupriyanov, and J. Teich, “A Highly Parameterizable Parallel Processor Array Architecture,” in *Proceedings of IEEE International Conference on Field Programmable Technology*, Bangkok, Thailand, Dec. 13-15 2006, pp. 105–112.

[114] ——, “Hardware Cost Analysis for Weakly Programmable Processor Arrays,” in *Proceedings of International Symposium on System-on-Chip*, Tampere, Finland, Nov. 13-16 2006, pp. 1–4.

[115] G. Zhou and X. Shen, “A Coarse-Grained Dynamically Reconfigurable Processing Array (RPA) for Multimedia Application,” in *Proceedings of third International Conference on Natural Computation*, 2007, pp. 157–161.

[116] M. J. Myjak and J. G. Delgado-Frias, "A Medium-Grain Reconfigurable Architecture for DSP: VLSI Design, Benchmark Mapping, and Performance," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 14–23, 2008.

[117] MathStar, "A High-performance Field Programmable Object Arrays," http://www.mathstar.com.

[118] QuickSilver Technology, "Adaptive Computing Machine: Adapt2400," http://www.qstech.com.

[119] PACT XPP Technologies, "XPP-III High Performance Media Processor," http://www.pactxpp.com.

[120] J. Becker and M. Vorbach, "Architecture, Memory and Interface Technology Integration of an Industrial/Academic Configurable System-on-Chip (CSoC)," in *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, Tampa, Florida, Feb. 20-21 2003, pp. 107–112.

[121] D. Rosenband and Arvind, "Hardware Synthesis from Guarded Atomic Actions with Performance Specifications," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, San Jose, CA, USA, Nov. 6-10 2005, pp. 784–791.

[122] W. J. Dally and B. Towles, "Route Packets, Not Wires: On-chip Interconnection Networks," in *Proceedings of the 38th Design Automation Conference*, Las Vegas, Nevada, USA, June 18-22 2001, pp. 684–689.

[123] P. Pratim *et al.*, "Performance Evaluation and Design Trade-Offs for Network-on-Chip Interconnect Architectures," *IEEE Transactions on Computers*, vol. 54, no. 8, pp. 1025–1040, Aug. 2005.

[124] L. Bononi and N. Concer, "Simulation and Analysis of Network on Chip Architectures: Ring, Spidergon and 2D Mesh," in *Proceedings of IEEE Conference on Design, Automation and Test in Europe*, Munich, Germany, Mar. 6-10 2006, pp. 154–159.

[125] M. Lanuzza, S. Perri, P. Corsonello, and M. Margala, "A New Reconfigurable Coarse-Grain Architecture for Multimedia Applications," in *Proceedings of NASA/ESA Conference on Adaptive Hardware and Systems*, Scotland, United Kingdom, Aug. 5-8 2007, pp. 119–126.

[126] A. López-Lagunas and S. M. Chai, "Compiler Manipulation of Stream Descriptors for Data Access Optimization," in *Proceedings of International Conference on Parallel Processing Workshops*, Columbus, Ohio, USA, Aug. 14-18 2006, pp. 337–344.

[127] S. Ciricescu *et al.*, “The Reconfigurable Streaming Vector Processor (RSVP),” in *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, San Diego, CA, USA, Dec. 3-5 2003, pp. 141–150.

[128] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, USA: Morgan Kaufmann, 2004.

[129] Texas Instruments, “TMS320C55x DSP Library Programmer’s Reference,” http://www.ti.com.

[130] D. Eliemble, “Optimizing DSP and Media Benchmarks for Pentium 4: Hardware and Software Issues,” in *Proceedings of IEEE International Conference on Multimedia and Expo*, Lusanne, Switzerland, Aug. 26-29 2002, pp. 109–112.

Appendix

Appendix A

The Scenic Shell

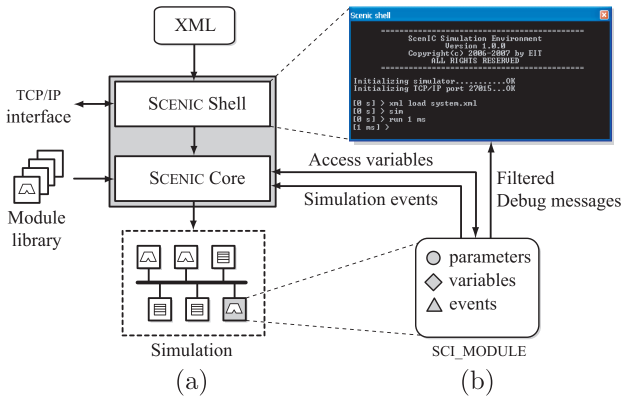
SCENIC is an extension to the OSCI SystemC library and enables rapid system prototyping and interactive simulations. This manual presents the built-in SCENIC shell commands, how to start and run a simulation, and how to interact with simulation models and access information during the simulation phase. An overview of the SCENIC functionality is presented in Figure 1. A module library contains simulation modules that can be instantiated, bound, and configured from an XML description. The SCENIC shell provides access to internal variables during simulation and enables extraction of performance data from simulation models.

1 Launching Scenic

SCENIC is started using the command *scenic.exe* from the windows command prompt, or using the command *./scenic* from a cygwin or linux shell.

```
scenic.exe <file> [-ip] [-port] [-exec] [-exit]
```

If a file named *scenic.ini* is present in the startup directory, it is executed before any other action is taken. SCENIC can be launched with optional arguments to execute a script file or to set environment variables. The first argument, or the flag *exec*, specifies a script to execute during startup, and the flag *exit* indicates with true or false if the program should terminate once the script has finished execution (for batch mode processing). The flags *ip* and *port* are used to open a socket from which the program can be interactively controlled, either from Matlab or from a custom application. All startup flags are registered as environment variables in the SCENIC shell to simplify parameter passing.



```
        ==========================================
              ScenIC Simulation Environment
                     Version 1.0.0
              Copyright(c) 2006-2007 by EIT
                  ALL RIGHTS RESERVED
        ==========================================

Initializing simulator...........OK
Initializing TCP/IP port 27015...OK

[0 s] > xml load system.xml
[0 s] > sim
[0 s] > run 1 ms
[1 ms] >
```

Figure 1: (a) A simulation is created from an XML description format using a library of simulation modules. (b) The modules can communicate with the SCENIC shell to access member variables, notify simulation events, and filter debug messages.

1.1 Command Syntax

The built-in SCENIC commands are presented in Table 2. Commands are executed from the SCENIC shell by typing a built-in command, or the hierarchical name of a simulation module. The command is followed by a list of *parameters*, i.e. single values, and *flag/value* pairs. The ordering of flag/value pairs does not matter, but parameters need to be specified in the correct order.

```
command/module {parameter}* {-flag value}* {-> filename} {;}
```

Variable substitution replaces environment variables with their corresponding value. Substitution is evaluated on all parameters and on the *value* for a flag/value pair. A parameter or value containing space characters must be enclosed by single or double quotation marks to be treated as a single value. Variable substitution is performed on a value enclosed by double quotation marks, but single quotation marks prevent variable substitution. Command substitution is expressed using square brackets, where a command inside a square bracket is evaluated before the parent command is executed.

A semicolon ";" at the end of a command prevents the return value to be printed on the screen. The return value can also be directed to a file by ending the line with "-> filename" to create a new file, or "->> filename" to append an existing file.

Comments are preceded by a "#" or a "%" token. The comment characters only have significance when they appear at the beginning of a new line.

```
[0 s]> set a 10                    % assigns variable a = 10
[0 s]> random -min 0 -max $a       % generated random number between 0 and 10
[0 s]> set b [eval $a - 2]         % calculate 10-2=8 and assign to b
[0 s]> set c "$a $b"               % assign c the concatinated string "10 8"
[0 s]> set d '$a $b'               % assign d the unmodified string "$a $b"
```

1.2 Number Representations

Integer values may be specified in decimal, hexadecimal, or binary representation. The decimal value 100 is represented in hexadecimal as 0x64 and in binary as b1100100. Negative and floating point numbers must be specified in decimal format. Boolean values are represented with TRUE or 1 for true, and FALSE or 0 for false.

1.3 Environment Variables

Environment variables are used to store global variables, intermediate results from calculations, and parameters during function calls. Variables are assigned using the SET command and can be removed using UNSET. Environment variables are treated as string literals and can represent any of the standard data types such as integer, float, boolean, and string.

```
[0 s]> set I 42
[0 s]> set S "Hello World"
```

Environment variable values are accessed by using the $ operator followed by the variable name.

```
[0 s]> set MSG "$S $I"
[0 s]> echo $MSG
  Hello World 42
```

1.4 System Variables

System variables are special environment variables that are set by the system, including the simulation time, execution time, and other global variables related to the simulation. The system variables are useful for calculating simulation performance or execution time.

```
[0 s]> set start $_TIME
  ...
[1 ms]> set time [eval $_TIME - $start]
[1 ms]> echo "execution time: $time ms"
  execution time: 5225 ms
```

Both environment and system variables can be listed using either the SET command without specifying any parameters or by typing LIST ENV.

```
[1 ms]> set

system variable(s) :
  _DELTACOUNT                            : "276635"
  _ELABDONE                              : "TRUE"
  _MICROSTEP                             : "25"
  _RESOLUTION                            : "1 ns"
  _SIMTIME                               : "1000000"
  _TIME                                  : "5225"

environment variable(s) :
  I                     : "42"
  S                     : "Hello World"
  MSG                   : "Hello World 42"
```

2 Simulation

Scenic modeling is divided into two phases, a *system specification* phase and *simulation* phase. System specification means describing the hierarchical modules, the *static* module configuration, and the module interconnects. The system specification phase will end when the command SIM is executed, which launches the SystemC simulator and constructs the simulation models (elaboration). After elaboration, simulation modules can not be instantiated, but *dynamic* configuration can be sent to the simulation modules.

2.1 Starting a Simulation

Simulation architectures are described using XML language, and are loaded before the simulation can begin. The architectural description format will be further explained in Section 4. Modular systems can be described using *multiple* XML files, which will be merged into a single simulation. If two modular systems share a common simulation module with the same instance name, Scenic will automatically create a single instance to which both subsystems can connect.

```
[0 s]> xml load -file demo_system.xml
[0 s]> sim

             SystemC 2.2.0 --- Feb  8 2008 16:25:45
        Copyright (c) 1996-2006 by all Contributors
                    ALL RIGHTS RESERVED

Simulation Started [1]

[0 s]>
```

After loading the XML system descriptions, the command SIM launches the SystemC simulator, instantiates and configures the library modules, and binds the module ports. If the optional argument NOSTART is set to true, the simulator stays in specification phase until the simulation is actually run.

2.2 Running a Simulation

A simulation runs in either blocking or non-blocking mode, using the commands RUNB and RUN, respectively. The former is useful when the execution time is known or when running the simulation from a script, while the latter is useful for user-interactive simulations. A non-blocking simulation can be aborted using the STOP command, which will halt the simulation at the start of the next *micro-step*.

```
[1 ms]> step 25 us
[1 ms]> runb 1 ms
[2 ms]> run 10 ms
[2 ms]> stop
  Simulation halted at 3125 us
[3125 us]>
```

A non-blocking simulation is executing in time chunks referred to as micro-steps. The size of a micro-step is configurable using the STEP command with the preferred micro-step size as a parameter. Using the STEP function without parameters will run the simulation for a time duration equal to one micro-step. It can be used to for example single-step a processor simulation module, where the step size is set to the processors clock period. A larger step size results in a higher simulation performance, but may not be able to halt the simulation with clock cycle accuracy.

3 Simulation Modules

Simulation modules are described in SystemC and uses SCENIC macros to register the module in a *module library*, read *configuration data* during instantiation, export internal member variables, register simulation events, and generate *debug* messages and *trace* data.

3.1 Module Library

The module library handles both module *registration* and *instantiation.* When SCENIC is launched, all simulation modules will automatically be registered in the module library (using a SCENIC macro). System specification refers to simulation modules in the module library, which will be instantiated with the dynamic module configuration specified in the XML file.

The simulation modules in the module library can be listed using the command LIST LIB, which will show the module name and the number of currently instantiated components of each type.

```
[0 s]> list lib

library module(s) :

  GenericProcessor_v1_00     [1 instance(s)]
  mpmc_v1_00                 [1 instance(s)]
  spb2mpmc_v1_00             [1 instance(s)]
  spb_v1_00                  [1 instance(s)]
```

3.2 Module Hierarchy

During the simulation phase, the hierarchical view of the constructed system can be shown using the LIST MOD command. The command shows the hierarchical names of each module and the module type.

```
[0 s]> list mod

instantiated module(s) :

  Adapter                   [spb2mpmc_v1_00]
  BUS                       [spb_v1_00]
  CPU                       [GenericProcessor_v1_00]
  CPU.program               [sci_memory]
  MEM                       [mpmc_v1_00]
  _memory                   [access object]
  _scheduler                [sci_module]
```

An addition to the user simulation modules, SCENIC creates a number of special simulation objects. The special simulation objects use an underscore in the beginning of the name to separate them from user simulation modules. The "_memory" module handles global memory and implements function to list, read, and write memories that are registered as global. The "_scheduler" module handles logging of registered variables to improve simulation performance.

3.3 Module Commands

Each simulation module support basic functionality, inherited from SCI_ACCESS, to list parameters and variables, read and write variables, periodical logging of variables, creating conditional simulation events, and tracing data to file. In addition, a module based on SCI_MODULE also supports port binding. The HELP command lists the supported functions and shows which class that implements the functionality.

```
[0 s] > CPU.program help
  ---------------- access object ----------------
  .                                              : Show sub-modules
  list <-display>                                : Show internal variables
  param <name>                                   : Show / get parameter
  get [-var] <-vmin> <-vmax>                     : Get variable value
  set [-var] [-value] <-vmin> <-recursive>       : Set variable value
  size [-var]                                    : Get the size of a variable
  log [-var] <-depth> <-every>                   : Log to history buffer
  push [-var]                                    : Push variable to history
  read [-var] <-vmin/vmax/tmin/tmax> <-timed>    : Read history buffer
  event [name] [-when] <-message>                : Create conditional event
  trace <on,off> <-filename>                     : Trace to file
  ----------------- sci_memory ------------------
  rd <-addr> <-size> <-bin> <-file>              : Read memory contents
  wr <-addr> <-size> <-bin> <-file> <-data>      : Write memory contents
```

Scenic contains base classes for frequently used objects, such as sci_memory and sci_processor, which provide additional module specific functionality. For example, the memory class implements functions to read and write memory contents as shown below. In the same way, user modules can implement custom functions to respond to requests from the Scenic shell.

```
[0 s]> CPU.program rd -addr 0x0 -size 64

00000000h: 64 00 00 0c 00 00 20 0c 00 00 40 0c 00 10 01 08  d..... ...@.....
00000010h: 00 10 01 04 04 00 21 10 02 00 00 10 00 10 01 08  ......!.........
00000020h: fc ff 00 18 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

3.4 Module Parameters

The parameters used when instantiating a module are listed using the param command. The command shows the parameter name, the type index, and the parameter value.

```
[10 ns]> MEM param
  C_BASE_ADDR          [typeID=6]      { 0 }
  C_BYTE_SIZE          [typeID=6]      { 1048576 }
  C_CAS_LATENCY        [typeID=6]      { 3 }
  C_DATA_WIDTH         [typeID=6]      { 32 }
  C_ENABLE_DDR         [typeID=0]      { FALSE }
  C_PERIOD             [typeID=13]     { 10 ns }
```

Parameters are read-only and can be accessed individually by specifying the parameter name as a second argument.

3.5 **Module Variables**

Each module can export local member variables to be accessible from the Scenic shell. This reflects the variable type, size and value during simulation time. Accessible variables enable the possibility of dynamic configuration, and extraction of performance data, during simulation. All registered variables in a module can be viewed using the modules LIST command. It shows the name, vector size, data type size, history buffer size, and the variable value.

```
[1 ms]> CPU list
  _debug_level           1x4 [0]         { MESSAGE }
  _trace_file            1x32 [0]        {  }
  _trace_flag            1x1 [0]         { 0 }
  %PC                    1x1 [0]         { 19 }
  stat_exec_instructions 1x4 [0]         { 98662 }
  batch_size             1x4 [0]         { 1 }
  use_data_bus           1x1 [0]         { TRUE }
  use_instr_bus          1x1 [0]         { TRUE }
```

Each registered variable can be accessed and modified using the GET and SET commands. The example below illustrated how the program counter (PC) in a process or set to instruction 4. After simulating two clock cycles, the PC reaches the value 6. For vector variables, the value is specified as a string of values.

```
[10 ns]> CPU set -var "%PC" -value 4
[10 ns]> runb 20 ns
[30 ns]> CPU get -var "%PC"
  [ 6 ]
```

Each variable can be periodically logged to study trends over time. The command LOG configures the *history buffer* depth and the logging time interval. To disable logging, the time interval is set to 0. The READ command is used to access data in the history buffer, where time and vector range are optional parameters. Time/value pairs can also be acquired by setting the *timed* flag to true.

```
[0 us]> CPU log -var "%PC" -depth 10 -every "10 ns"
[0 us]> runb 1 us
[1 us]> CPU read -var "%PC"
  [ 4 ; 5 ; 6 ; 7 ; 8 ; 4 ; 5 ; 6 ; 7 ; 8 ]
```

3.6 **Module Debug**

Modules use a set of macros to write debug information to the screen, shown in Table 1. In this way, the level of detail can be configured for each individual module. Each module contain a system variable named "_debug_level" that can be changed by either the global command DEBUG, which configures all simulation modules, or individually by changing the variable with the SET command. The following command will set the debug level for all modules to *warning*, and then set the debug level for the processor to *message*. During simulation, the processor reports the executed instructions using a message macro, which prints the information on the screen.

```
[0 s]> debug -level WARNING
[0 s]> CPU set -var _debug_level -value MESSAGE
[0 s]> runb 50 ns
  [0  ns] <CPU> In reset
  [10 ns] <CPU> processing instruction @0
  [20 ns] <CPU> processing instruction @1
  [30 ns] <CPU> processing instruction @2
  [40 ns] <CPU> processing instruction @3
[50 ns]>
```

Tracing is used to extract information from a simulation models to a file, and the macros can be found in Table 1. If the simulation model generates trace data, it can be configured using the TRACE command to start streaming to file. Tracing can be interactively turned on and off during simulation and the hierarchical name of the module will be used if a filename is not given.

```
[0 us]> MEM trace on -file "memory_trace.txt"
[0 us]> runb 1 us
[1 us]> MEM trace off
```

4 System Specification

System description is based on eXtensible Markup Language (XML), which is divided into module *instantiation*, *binding*, and *configuration*. Instantiation and binding is handled during the specification phase using the command XML LOAD, while configurations can be loaded during the simulation phase using XML CONFIG. Multiple XML files can be loaded and will be merged into a single system during elaboration.

```
xml load -file system.xml                 % load system decription
xml load -file config.xml                 % load configurations
sim                                       % build the simulation
runb 1 ns                                 % start simulation phase
xml config -name "config_processor"       % configure module
```

4.1 Module Instantiation

Simulation models are instantiated from the module library based on a system specification. The specification contains the models to instantiate (type), the hierarchical name of the module (name), and parameters to statically configure each module.

```
<INSTANTIATE>
  <MODULE type="Processor_v1_00" name="CPU">
    <PARAMETER name="C_PERIOD"     type="TIME"  value="5 ns"/>
    <PARAMETER name="C_USE_CACHE"  type="BOOL"  value="TRUE"/>
    <PARAMETER name="C_REGISTERS"  type="UINT"  cmd="$I"/>
  </MODULE>
  ...
</INSTANTIATE>
```

Parameter are given for each module to be instantiated, and consists of a parameter *name*, *type*, and *value*. The parameter name is unique and used by the simulation model (in the constructor) to read out the configuration. The type can be one of the following: STRING, BOOL, CHAR, UCHAR, SHORT, USHORT, INT, UINT, LONG, ULONG, FLOAT, DOUBLE, TIME. STRING is assumed if no type name is given, and the value must match the type.

4.2 Module Binding

If a simulation module supports binding to another simulation module, then the modules can be bound from the system specification using bind tags. Each module implements functionality to bind to compatible modules, which can be called during the specification phase.

```
<CONNECT>
  <BIND src="CPU" dst="BUS" if="ICACHE"/>
  <BIND src="CPU" dst="BUS" if="DCACHE"/>
</CONNECT>
```

All parameters specified in the bind tag are sent to the modules *bind* function, for example parameter *if*, but *src* and *dst* represents the modules to be bound.

4.3 Module Configuration

In some cases it is useful to dynamically reconfigure the simulation modules during the simulation phase. Therefore, it is possible to create different *named* configuration that can be loaded during simulation. A configuration is loaded using the *xml config* command and by specifying the name of the configuration.

```
<CONFIG name="config_processor">
  <MODULE name="CPU">
    <PARAMETER name="C_PROGRAM"  type="STRING" value="demo.asm"/>
    <PARAMETER name="C_ADDR"     type="UINT"   value="0"/>
  </MODULE>
</CONFIG>
```

The configuration contains information about how to reconfigure one or more simulation modules. It uses the modules hierarchical name, and parameters are given in the same way as for module instantiation.

Table 1: Description of SCENIC macros and functions.

Scenic macros and functions	**Description**
SCI_REGISTER_LIBRARY_MODULE(module)	Register a simulation module in the module library (global context)
SCI_REGISTER_ARCHITECTURE(module)	Register an architecture. An alternative to using XML
SCI_REGISTER_GLOBAL_MEMORY(mem,base,name)	Register a memory in the global memory space
SCI_REGISTER_COMMAND(name,func_ptr,desc)	Register a user defined command in the SCENIC shell
SCI_REGISTER_SYSTEM_VARIABLE(name,func_ptr)	Register a user defined system variable in the SCENIC shell
SCI_REGISTER_CODE_GENERATOR(gen,format,desc)	Register a code generator for a specific file type
SCI_VERBOSE(message)	Print verbose message on debug level *verbose*
SCI_MESSAGE(message)	Print message on debug level *message*
SCI_WARNING(message)	Print warning on debug level *warning*
SCI_ERROR(message)	Print error on debug level *error*
SCI_TRACE(label,data)	Generate trace data if tracing is enabled
SCI_TRACE_V(var)	Trace a variable if tracing is enabled
SCI_TRACE_I(var,index)	Trace a variable[index] if tracing is enabled
SCI_SIM_EVENT(event,name,message)	Register a simulation *event* to execute variable *name*
SCI_SIM_EVENT_NOTIFY(event)	Trigger the simulation event to execute the command specified in its variable
scenic_callback(var,name,size)	Export callback variable *var* as *name*
scenic_variable(var,name)	Export variable *var* as *name*
scenic_variable(var,name, size)	Export variable *var* with *size* elements as *name*
scenic_vector(var,name)	Export vector *var* as *name*
scenic_vector(var,name,size)	Export vector *var* with *size* elements as *name*

Table 2: Description of SCENIC user commands and global simulation modules.

Command	**Description**
codegen [-format] [-top] [-proj] [-dir]	run the code generator specified by *format* on a hierarchical simulation module *top*.
debug [-level]	set the global debug level.
echo [text]	print text message.
eval [A] [op] [B]	evaluate mathematical or logical operation.
exec [-file] [-repeat] [-fork]	execute a script file with scenic commands *repeat* times. *fork* executes the script in a new context.
exit	exit program.
foreach [-var] [-iterator] [-command]	execute a command *cmd* for every value in a set *var*.
function(parameters)	function declaration with parameter list.
list [mod,gen,lib,env,arch]	list *mod*ule, *gen*erator, *lib*rary, *env*ironment, *arch*itecture.
random [-min] [-max] [-size] [-round] [-seed]	generate a sequence with *size* random numbers in the range *min* to *max*.
return [value]	return *value* from function call.
run [time] [unit]	run simulation for *time* time *units*. valid time units are [fs,ps,ns,us,ms,s].
runb [time] [unit]	blocking version of run.
set [var] [value]	assign or list environment variables.
sim [-arch] [-nostart]	launch SystemC simulator and create system from *arch*itectural description or from loaded XML.
step [time] [unit]	run a single micro-step / set micro-step value.
stop	halts a running simulation at next micro-step.
system [command]	execute system shell *command*.
unset [var]	remove environment variable *var*.
xml [clear,load,config,view]	load XML system description or configure simulation modules from XML.
Global modules	**Description**
_memory [map,rd,wr]	Scenic module that manages the global memory map
_scheduler [active]	Scenic module that manages access variable logging

Appendix B

DSP/MAC Processor Architecture

The DSP and MAC processors, presented in Part IV, are based on the same architectural description, but with minor differences in the instruction set. The DSP processor uses a 32-bit ALU and supports real and complex valued radix-2 butterfly. The MAC unit is based on a 16-bit ALU with multiplication support, and implements instructions to join and split data transactions between the external ports and the internal registers.

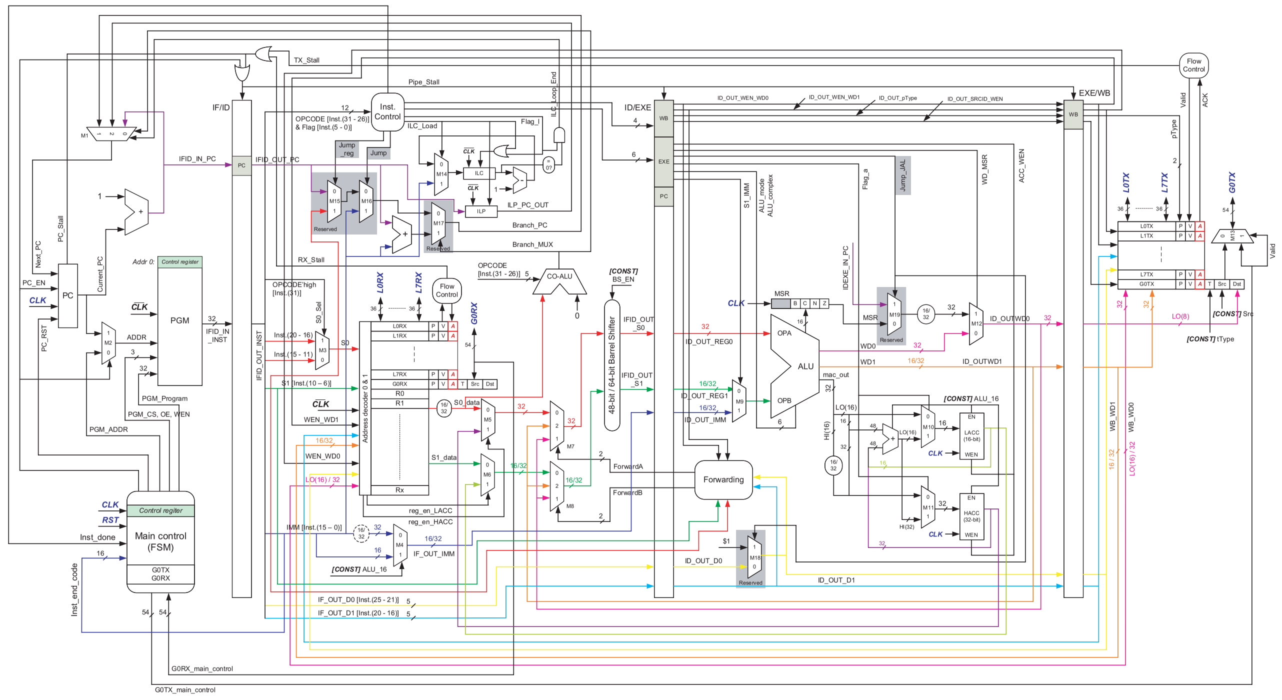
Table 2 presents the instruction set for the VHDL implementation, while the SCENIC models support additional and configurable functionality. Figure 1 presents a detailed description of the processor architecture. It is based on a three-stage pipeline for instruction decoding, execution, and write-back. The program is stored internally in the processing cell (PGM), and the *main controller* handles configuration management and the control/status register.

Table 1: Flags for instructions of type A.

Flag	Bit	Processor	Description
l	0	DSP/MAC	End of inner loop
c	1	DSP	Separated ALU for complex data (2×16 bits)
a	1	MAC	Accumulate value to xACC
r	2	DSP/MAC	Local I/O port read request
w	3	DSP/MAC	Local I/O port write request
s	5	MAC	Barrel shifter enabled (uses bit 0-4)

Table 2: Instruction set for the 32-bit DSP processor and the 16-bit MAC processor.

Type A Instructions	31-26	25-21	20-16	15-11	10-6	5-0	**Type A Semantics**
Type B Instructions	31-26	25-21	20-16	15-0			**Type B Semantics**
ADD D_0,S_0,S_1	000001	D_0		S_0	S_1	{srwcl}	$D_0 := S_0 + S_1$
SUB D_0,S_0,S_1	000010	D_0		S_0	S_1	{srwcl}	$D_0 := S_0 - S_1$
DMOV D_0,D_1,S_0,S_1	000111	D_0	D_1	S_0	S_1	{srwl}	$D_0 := S_0 - S_1$
ADDI D_0,S_0,Imm	100001	D_0	S_0	Imm			$D_0 := S_0 + \mathrm{sxt}(\mathrm{Imm})$
SUBI D_0,S_0,Imm	100010	D_0	S_0	Imm			$D_0 := S_0 - \mathrm{sxt}(\mathrm{Imm})$
BEQI S_0,Imm	100011		S_0	Imm			PC := PC + sxt(Imm) if $S_0 = 0$
BNEI S_0,Imm	100100		S_0	Imm			PC := PC + sxt(Imm) if $S_0 \neq 0$
BLTI S_0,Imm	100101		S_0	Imm			PC := PC + sxt(Imm) if $S_0 < 0$
BLEI S_0,Imm	100110		S_0	Imm			PC := PC + sxt(Imm) if $S_0 \leq 0$
BGTI S_0,Imm	100111		S_0	Imm			PC := PC + sxt(Imm) if $S_0 > 0$
BGEI S_0,Imm	101000		S_0	Imm			PC := PC + sxt(Imm) if $S_0 \geq 0$
Special instruction							
NOP	000000						No operation
BRI Imm	101001			Imm			PC := PC + sxt(Imm)
END Imm	101010			Imm			End execution with code=Imm
ILC Imm	101011			Imm			ILP := PC + 1 ILC := Imm
GID Imm	101100			Imm			Global port destination ID=Imm
32-bit DSP specific							
BTF D_0,D_1,S_0,S_1	000011	D_0	D_1	S_0	S_1	{srwcl}	$D_0 := S_0 + S_1$ $D_1 := S_0 - S_1$
16-bit MAC specific							
SMOV D_0,D_1,S_0	000101	D_0	D_1	S_0		{srwl}	$D_0 := \mathrm{HI}(S_0)$ $D_1 := \mathrm{LO}(S_0)$
JMOV D_0,S_0,S_1	000110	D_0		S_0	S_1	{srwl}	$\mathrm{HI}(D_0) := S_0$ $\mathrm{LO}(D_0) := S_1$
MUL S_0,S_1	000100			S_0	S_1	{srwal}	HACC := $\mathrm{HI}(S_0 * S_1)$ LACC := $\mathrm{LO}(S_0 * S_1)$



Notes:

$0: Zero register
$1: Jump and Link register (Reserved)
$2 ~ $x: General Purpose registers
▨ Reserved functions or signals

Figure 1: A detailed processor architecture for the 32-bit DSP processor and the 16-bit MAC processor. The processor instruction-set, register setup, port configuration, and special functional units are configurable during design-time.