FPGA Design Methodologies for High Performance Applications

LEONG Monk Ping, B.Eng.

Supervised by
Prof. LEONG Heng Wai Philip

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy
in
Computer Science and Engineering

To my parents, for their love and care.

FPGA Design Methodologies for High Performance Applications

Submitted by

LEONG Monk Ping

for the degree of Doctor of Philosophy
at The Chinese University of Hong Kong in September 2001

Abstract

Many mainstream electronic systems in applications such as digital signal processing (DSP), networking and wireless communications require performance, size, cost and power consumption which is beyond that achievable with a single microprocessor. In such cases, particularly those in which fine grained parallelism can offer a performance advantage, an application specific integrated circuit (ASIC) based coprocessor is often used. As the non-recurrent engineering costs of ASICs continue to rise and the density of field programmable gate arrays (FPGAs) continues to improve, FPGAs are claiming a larger and larger share of the coprocessor market. Furthermore, FPGAs have advantages of field upgradeability and faster development time over ASICs.

Realizing an FPGA-based coprocessor system poses many challenges and this thesis addressed three issues in designing an FPGA coprocessor. Firstly, as programming and hardware design are predominately treated as different entities, tools for developers not intimately familiar with hardware design to translate a software implementation to hardware can greatly improve productivity. Secondly, resources on an FPGA device are limited so designers should be able to explore the tradeoff between area and performance using differing degrees of parallelism. Thirdly, as the execution of a program is divided into two interconnected portions, the interfacing issue between the two entities need to be addressed.

In this dissertation, a high level FPGA coprocessor design system which can automatically translate a high level floating-point algorithmic description into an optimized FPGA hardware/software co-design system was developed. This system utilizes two commonly used but seldom simultaneously applied design methodologies, namely floating to fixed-point conversion and digit-serial computation. The system takes a floating-point dataflow algorithmic description and translates it into a fixed-point design via a simulation-based optimization. The optimizer assigns a wordlength and digit size to each individual variable while minimizing a cost function which takes into account the tradeoff between performance and area. The optimizer achieves a design which would be too tedious for a designer to perform manually, and which optimally meets the requirements. In order to achieve a high performance FPGA coprocessor system, a further consideration is the speed of the bus which connects the FPGA to the microprocessor. A memory slot based coprocessor was developed which achieves significantly improved performance over the standard peripheral bus.

The above techniques were applied to a number of applications in image processing, cryptography, rendering and auditory signal processing. In each application, the approach was shown to offer a considerable performance improvement over the standard approach.

高性能現場可編程門陣列應用的設計方法學

作者 梁望平

香港中文大學 二零零一年九月

摘要

電子系統應用的幾種主流，包括數碼訊號處理器、網絡及無線通訊等都需要考慮到計算能力、面積、成本及耗電量。基於這些考慮，採用單一微處理器的方法並不是適合的方法。在這些情況下，尤其在一些能以細分並行特性達致增加計算能力的應用中，通常都會採用到特定應用集成電路。不過隨著特定應用集成電路的非循環工程成本上漲，與現場可編程門陣的密度不斷改進，現場可編程門陣逐漸在協處理器的市場佔有重要席位。現場可編程門陣有兩個優點，一是現場可升級性，二是它有比特定應用集成電路短之發展時間。

由於認識到一個現場可編程門陣協處理器的設計過程中尚有很多問題要克服，因此便進行了研究。本篇論文針對了其中三個具挑戰性問題加以研究。首先是，有鑒於軟件及硬件的設計法有所不同，本文嘗試爲不熟悉硬件設計的軟件設計者提供一工具，幫助他把軟件自動轉換成硬件設計以提高生產力。其次是，由於一顆現場可編程門陣中的可用資源有限，設計者要能夠控制並行的程度，使硬件需求及計算能力的平衡恰到好處，令其切合實際應用上的需要。最後是，由於程式分開在微處理器及現場可編程門陣上執行，使連接兩者的介面要有高的效率是不可忽略的。

這研究的其中一個成果是發展了一個自動將高階浮點算法描述，變成一個應用現場可編程門陣的硬件和軟件配合設計系統。這系統使用了兩種技巧，分別是浮點至定點轉換和數位串列運算。這兩種技巧常爲人分開應用，現設計到兩者能同時運用。其中的浮點至定點轉換採用了一個模擬式的優化，優化器爲

字長及數位大小安排一個變數，以此衡量其對總體計算能力與資源需求，並找出一個只需要最低成本就合乎需求的設計。另外，爲使現場可編程門陣協處理器系115有最佳效率，論文中也對連接微處理器與現場可編程門陣的排流速度加以考察，並發展了一個配置了記憶體介面的現場可編程門陣協處理器平台。這平台比標準的週邊排流擁有更高的效能。

上述的技術已應用於好幾方面，比如應用於影像處理、密碼技術、繪制及聲像訊號處理上。每方面的應用都顯示，這些技術提供到的效率，比傳統設計方法所提供的效率更高。

Acknowledgments

I would like to take this opportunity to express my hearty gratitude to some people. This dissertation cannot be done without their help and support.

First, I would like to thank my supervisor, Dr. Philip Leong, for his guidance and encouragement. The ideas, advices and research opportunities he gave me play an important role in completing my thesis. A number of discussions with him lead me towards new and successful research directions. My deepest appreciation for his generosity during these three years of my postgraduate study.

I would like to specially thank Dr. Craig Jin for reviewing my thesis. He gave me a lot of valuable comments in improving my thesis. I appreciate his effort and patience in reviewing this thesis.

I would like to thank my colleagues, Mr. Ray Cheung, Mr. Ivan Leung, Mr. Cliff Sze, Ms. Polly Wan, Mr. Hiu-Yung Wong and Ms. Wing-Seung Yuen for bringing me a comfortable working atmosphere. I appreciate their support and the happiness they brought me.

I would also like to thank my family for their never-ending warmth and support. This thesis is dedicated to my parents.

Last but no means least, I thank Polly. Her love and care are the most important support to me and make my life meaningful.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Aims

The latest high-end microprocessors utilize 0.18 μm complementary metal oxide semiconductor (CMOS) technology with 64-bit data buses, multiple functional units and megabytes of integrated cache packed on a single die with up to a hundred million transistors operating at clock frequencies over 1.5 GHz. Although the computational abilities achieved by these advanced microprocessors are beneficial to a wide range of science, engineering and business applications, they cannot always fulfill the needs of certain applications. Some applications, such as real-time signal processing, high-throughput cryptographic systems and large-scale optimization, require even higher computational power. Furthermore, microprocessor systems usually have large footprint, high power requirements and large heat dissipation. Applications of microprocessors to mobile devices and embedded systems, of which performance, power consumption and compactness are crucial design considerations, are often not feasible.

An important limitation of microprocessor architecture is due to the nature that software programs executing on microprocessor systems are essentially sequences of operations chosen from the set of instructions supported by the microprocessor architecture. In contrast, hardware implementations utilize hardware parallelism and dedicated logic to potentially achieve significant performance improvements over a microprocessor implementation. At the same time, a hardware design can be customized for a specific application, with power efficiency and density better than that of any microprocessor system. However, it is not always feasible in terms of design time and cost to build a custom hardware implementation for every application due to long design time and high manufacturing cost.

Field-programmable custom computing machines (FCCMs) are a potential solution to some of the problems mentioned above. As computational components, FCCMs use field-programmable gate arrays (FPGAs) which are hardware devices with programmable logic cells and routing. The reconfigurable nature of FPGAs allows multiple designs to be programmed on the same hardware device at different times, therefore eliminating the costly design and fabrication process associated with very large scale integration (VLSI). The continuous improvement in silicon technology offers faster and larger FPGA devices over time. When newer FPGA devices become available, the same design obtains performance improvements accordingly. In addition, with the improved density and shorter design time, designers may attempt implementations employing more sophisticated algorithms which lead to a further improvement in performance.

The most straightforward way to deploy an FCCM system is to couple it with a microprocessor system, with the computationally intensive portions extracted into hardware which is executed on an FPGA coprocessor. This approach poses the following challenges:

- The design complexity is increased as this approach splits a design into hardware and software portions. Methods that can partially or entirely automate the process of translating a program into a hardware design can significantly reduce design times.

- One difficulty in designing FPGA applications is that hardware resources are strictly limited due to cost constraints. Although density of FPGA devices has improved markedly, hardware-efficiency is still a primary design issue in many cases. Very often, performance is sacrificed to fit a design into a given FPGA device. Designers should be able to explore the tradeoff between area and performance easily, and a single design description can lead to multiple implementations with different area and performance requirements. In this way, designers can be less concerned with the area and performance requirements when developing the algorithm, and later choose an implementation which offers an acceptable tradeoff.
- A high bandwidth and low latency interface between the FPGA device and the central processing unit (CPU) is desired to reduce communication overheads between the hardware and software portions. An efficient interface between the two entities is essential or else the interface may become a bottleneck.

The objective of this thesis was to investigate and develop efficient methods to address the above challenges with a goal to improve upon existing FPGA-based hardware/software co-design approaches.

1.2 Contributions

Two major contributions were made in this thesis. They respectively addressed the problems of translating a software algorithm to a hardware implementation and the CPU/FPGA bandwidth issue

The first contribution was the development of a design methodology that translates floating-point software algorithms to fixed-point hardware implementations. An important step involved in translating a floating-point design to fixed-point is to determine the minimum required wordlength for each variable such that the algorithm uses the least area while retaining user-specified performance criterion. This criterion can be specified in the form of constraints on the precision, area, latency or throughput of the resultant implementation and is application-dependent. In this work, an optimization approach was used to find an optimal implementation subject to the design constraints. The optimizer tries different wordlengths for every variable, extracts their corresponding dynamic range and precisions via a software simulation, and derives the implementation. This approach, similar to an optimizing compiler, compiles software algorithms to hardware implementations with modest resource requirements. To further explore the tradeoff between area and performance, a variable-radix variable-wordlength architecture was developed. This architecture incorporated the advantages of both multiple wordlength and digit-serial architectures. Using this architecture, different implementations of the same algorithm can be generated, each of which has a different tradeoff among precision of the algorithm, resource requirements and performance. Consequently, designers can more efficiently decide the most suitable hardware implementation for an application.

The second contribution was the development of an improved interface that optimizes data transfers between the FPGA device and the CPU via the memory bus. Most currently available FPGA reconfigurable computing (RC) platforms communicate with the CPU via a peripheral bus, such as the Peripheral Connectivity Interface (PCI) Local Bus. In this work, an FPGA interface communicating with the CPU via the memory bus was developed in an attempt to improve the speed of data transfers between the FPGA device and the CPU. Being an integral part of the CPU datapath in the computer's organization, the memory bus interface has substantial throughput and latency improvements upon the PCI Local Bus. Data transfers on the PCI Local Bus have significant overheads due to the complicated handshaking involved (such as bus mastering, direct memory access (DMA) and interrupts). As compared with the PCI Local Bus, the memory bus interface is simpler because it involves only simple read and write accesses.

These design methodologies were used to produce designs with significant benefits in terms of productivity and performance over software and many conventional FPGA designs. The advantages of applying these techniques were justified in several applications:

- **International Data Encryption Algorithm (IDEA) cipher:** The IDEA cipher is a private key cryptosystem with very high cryptographic strength [Sch96]. This application makes use of the memory bus FPGA interface.

- **Post-rendering 3D warping:** This algorithm renders an image based on an input image (typically obtained from geometrical rendering) and a depth map. Its computational complexity depends only on the output image size and is independent of scene complexity, making it very suitable for real-time rendering systems [MMB97]. This application makes use of the automatic floating-point to fixed-point design approach.

- **An electronic cochlea model:** The Lyon and Mead electronic cochlea model mimics the qualitative behavior of the human cochlea [LM88]. This application makes use of floating-point to fixed-point translation and a memory bus FPGA interface.

- **A systolic structure for the computation of the Discrete Cosine Transform (DCT):** By setting up the mathematical relationships between the DCT and discrete moments (DMs), the DCT can be computed by a systolic structure which involves only additions and multiplications [LLCL98]. This application makes use of the floating-point to fixed-point design approach and the variable-radix variable-wordlength architecture.

1.3 Organization of the Thesis

A background review of FPGAs and their hardware design methodologies as well as interface issues are presented in Chapter 2. It begins with a description of the basic structure of an FPGA, followed by literature reviews of hardware design methodologies including hardware compilation, floating-point to fixed-point translation and digit-serial computation. The chapter ends with a description of a typical microprocessor-based RC system, highlighting the design of the interface between FPGAs and CPUs.

In Chapter 3, a system that automatically translates floating-point algorithms to fixed-point hardware (called *fp*) is presented. This system facilitates the implementation of software algorithms in hardware. During the translation process, the tradeoff among various conflicting design criteria is addressed. In Chapter 4, the extension of *fp* to support digit-serial architectures is presented.

An FPGA RC platform called *Pilchard* is described in Chapter 5. It overcomes the bandwidth limitation of the peripheral bus by using the memory bus. Both hardware and software design aspects regarding the development of *Pilchard* are presented.

Applications are presented in subsequent chapters. These include an implementation of the IDEA cipher (Chapter 6), a post-rendering 3D warping algorithm (Chapter 7), a parameterized electronic cochlea model (Chapter 8) and a systolic array implementation of the DCT (Chapter 9). In the final chapter, a summary of the work in this thesis is given. Some directions for the future work are also discussed.

In the appendixes, the implementation details of *fp* are first described (Appendix A). Next, the technique of directly modifying FPGA design bitstreams are presented (Appendix B). This technique was applied in the implementation of the IDEA cipher. Finally, a description of distributed arithmetic (DA) is given (Appendix C). DA is a computational algorithm used in the implementation of the electronic cochlea model.

Chapter 2

Background

This thesis addressed methods to improve upon existing FPGA-based coprocessor approaches by reducing design effort and increasing the performance of the resultant implementation. In this chapter, background information on FPGAs is first discussed. To show the potential performance advantages of an FPGA-based design, some high-performance FPGA implementations are described.

In this work, two issues in the FPGA-based coprocessor approach were identified and improvements were made on these. The first issue is concerned with the synthesis of an FPGA design from a high-level algorithmic description. An FPGA design needs to satisfy various implementation criteria, including precision, area, latency and throughput. In this thesis, a design methodology which facilitates the translation of an algorithm to an efficient hardware implementation was developed. This methodology reduces the design effort associated with porting a purely software-based algorithm to an FPGA-based coprocessor platform and involves hardware compilation, floating-point to fixed-point conversion and digit-serial computation. A review of the literature in these areas of research is also presented in this chapter.

The second issue addressed in this work is concerned with the data transfers between the FPGA and CPU. Most existing reconfigurable computing (RC) platforms use a peripheral bus for the communication between the FPGA and CPU and this interfacing bus is often a bottleneck in the FPGA-based coprocessor approach. In this work, an RC platform with a memory bus interface was developed which has improved throughput and latency. Background information related to the interfacing issue are given in this chapter. A brief description of the RC platforms (each of which have a peripheral bus interface) used in this work follows.

2.1 Field-Programmable Gate Arrays

FPGAs are digital hardware devices with their functionality programmable after fabrication. A typical FPGA consists of an array of programmable computational cells interconnected by programmable routing resources. Most commercial devices, such as the Xilinx XC4000, Xilinx Virtex and Altera A8000 series FPGAs, use four-input look-up tables (LUTs) to implement the processing elements in a cell [RFLC90]. The architecture of such an FPGA is depicted in Figure 2.1. Each logic cell (LC) consists of a four-input LUT (the computational element) together with a register or a latch (the storage element). The inputs and outputs of the LCs are connected together via programmable interconnections. These are implemented as horizontal and vertical wire segments between adjacent blocks that are interconnected via programmable switch matrices (PSMs). The ratio of storage elements to computational elements is much higher than that of a typical ASIC or VLSI digital design. Also, the locations of the storage elements are very close to the processing elements. These two properties enable very efficient pipelined architectures.

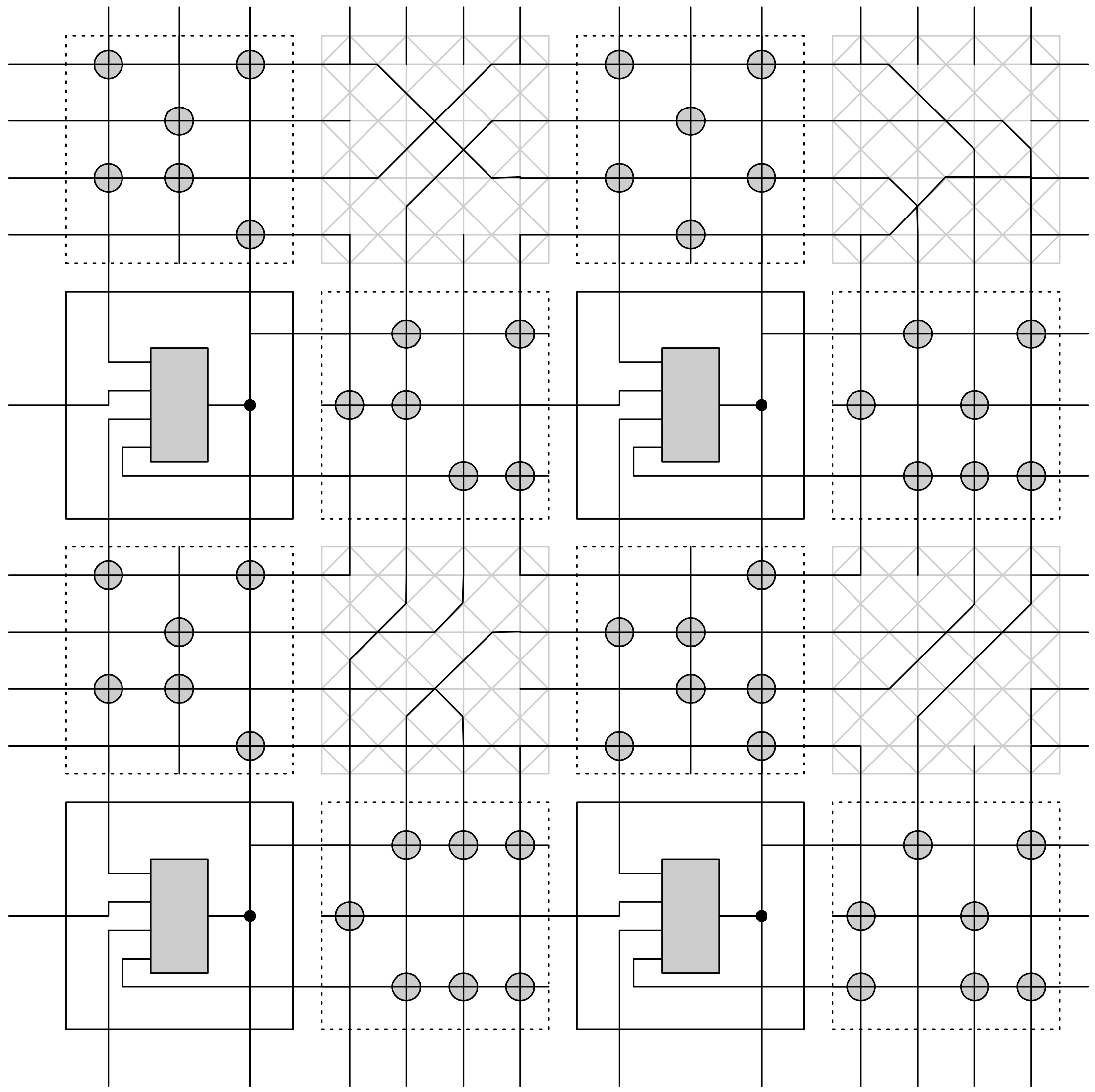


Figure 2.1: Architecture of an FPGA with four-input LUT cells.

The functional blocks and routing in a configured FPGA are essentially hard-wired to solve a specific problem. Compared with a microprocessor, fine-grained parallelism can be achieved and bit-level operations can replace word-level operations. These give FPGA-based computing machines the potential of having higher computational density than general-purpose microprocessor systems.

Modern microprocessors typically have superscalar architectures in which multiple instructions are issued and executed in a single clock cycle. The latest microprocessors also incorporate single instruction multiple data (SIMD) technologies, such as the Intel Pentium III Streaming SIMD Extension (SSE) [Int00a], enabling an instruction execution pattern similar to that of high-end vectorized supercomputers. Furthermore, microprocessors typically have an order of magnitude higher clock rate than an FPGA design. Despite these disadvantages, well-designed implementations of an algorithm in an FPGA can employ fine-grain parallelism which not only compensates for these deficiencies, but also offers higher performance for many applications [DeH00]. Field-programmable custom computing machines (FCCMs) have achieved the fastest or the most economical results for many important problems. Some examples include:

- **Cryptography:** An FPGA-based computing platform called a Programmable Active Memory (PAM) machine [VBR$^+$96] achieved the fastest reported Rivest-Shamir-Adelman (RSA) encryption/decryption rate in any technology (600 Kbits/sec and 165 Kbits/sec for 512-bit and 1024-bit encryption/decryption respectively). In this research work, an implementation of the International Data Encryption Algorithm (IDEA) achieves 1166 Mbits/sec on a Xilinx Virtex XCV300-6 device, as compared with a 147 Mbits/sec parallelized software implementation on a Sun Enterprise E4500 machine equipped with twelve 400 MHz Ultra-IIi processors [CTLL01]. An implementation of the Data Encryption Standard (DES) on a Xilinx

XCV150-6 device achieves an 10.7 Gbits/sec encryption rate, which was 10% faster than the fastest reported ASIC implementation (9.3 Gbits/sec) and 700 times faster than an optimized software implementation on a 200 MHz Intel Pentium processor (15 Mbits/sec) [Pat00].

- **Signal processing:** Optimized implementations of digital filters on FPGAs outperform dedicated digital signal processing (DSP) chips by an order of magnitude. For instance, the performance of an implementation of an 8-bit, 16-tap finite impulse response (FIR) filter on a Xilinx XC4013E-2 FPGA was 22 times faster than a 50 MHz, 50 million instructions per second (MIPS) fixed-point DSP processor (such as the Texas Instruments TMS320C5401 DSP) and 92 times faster than a 133 MHz Intel Pentium processor [Kna98].

- **Pattern matching:** FPGA implementations for DNA or protein sequence matching have achieved the highest performance among all implementation technologies. A systolic array for searching in genetic database implemented on a Splash 2 programmable logic array outperforms contemporary massively parallel processors (MPP) and supercomputers (CM-2 and Cray-2) by two orders of magnitude and typical workstations (Sun SparcStation I) by three orders of magnitude [Hoa93]. An FPGA-based point pattern matching processor has been applied to fingerprint matching. It achieved a matching speed of 2.6×10^5 fingerprints per second, which was 3700 times faster than a software implementation on a Sun SparcStation 10 [RJR95].

- **Stereo vision:** Using a software implementation on a Sun SparcStation II machine, the time taken to compute the correlation between a pair of images so as to produce stereo vision requires 59 seconds. A four-chip (Motorola DSP96002 DSP) DSP-based implementation performs the same task in 9.6 seconds. An FPGA-based implementation completes this task in 0.28 second, which is 34 times faster than the DSP-based implementation and 210 times faster than the software implementation [Mol93].

- **Speech synthesis:** Application of FPGAs to speech synthesis on the DECPeRLe-1 architecture achieves 11 million samples per second, which amounts to 22 million 16-bit multiplications, 100 million arithmetic and logic unit (ALU) operations and 45 million memory operations. This FPGA-based implementation, which is capable of producing 256 independent voices, can be compared with a DSP-based implementation which can only compute 24 voices at the same sampling rate on a 27 MHz Motorola 56001 DSP processor [VBR+96].

Other applications of FPGAs that achieved notable performance include an implementation of the finite difference method for solving the heat and Laplace equation (39×10^9 add and shift operations per second) [HRR93] and a hardware emulator for binary neural networks based on the Boltzmann machine model (500 megasynapses per second, equivalent to 0.5×10^9 additions and 0.5×10^9 multiplications by small coefficients) [Sku90, Sku92]. As a comparison, the latest microprocessors have performances in the order of a billion arithmetic operations per second using a 1 GHz clock.

All of the work described in this thesis was developed using Xilinx FPGAs (XC4000 and Virtex series). The information below was obtained from the databook published by Xilinx [Xil00c]. The two major configurable elements in an Xilinx series FPGA are configurable logic blocks (CLBs) and input/output blocks (IOBs). CLBs provide the functional elements for constructing the user's logic, whereas IOBs provide the interface between the package pins and internal signal lines. The CLBs are organized in a two-dimensional array with IOBs sur-rounding this array. For the XC4000 series FPGA, a CLB consists of two LCs. For the Virtex series FPGA, a CLB consists of four LCs organized in two Virtex slices. Each LC has a four-input function generator and a storage element. The four-input function generators are implemented as four-input LUTs, each of them providing the functions of one four-input LUT or a 16×1-bit synchronous random access memory (RAM) (called distributed RAM). Furthermore, two LUTs of the same slice (Virtex series) or the same CLB (XC4000 series) can be combined to create a 16×2-bit or 32×1-bit synchronous RAM or a 16×1-bit dual-port synchronous RAM. For the Virtex series, one of the LCs in every slice can be configured as a 16-bit shift register (called a shift register LUT (SRL)).

The density of FPGAs has been increasing rapidly. The Xilinx XC4000 series FPGAs have up to 180000 system gates (XC4085XL FPGA). The later Xilinx Virtex series FPGAs have up to 4 million system gates and 832 Kbytes 32-bit memory (called BlockRAM) (Virtex XCV3200E FPGA). The system clock rate has also been improving. The Xilinx XC4000 series FPGAs can operate at a system clock rate of 80 MHz and an internal clock rate over 150 MHz. The Xilinx Virtex series FPGAs can operate at 240 MHz system clock rates including input/output (I/O) [Xil00c]. The latest FPGA devices, the Xilinx Virtex-II series FPGAs, have a density of up to ten million system gates. They use a 0.15 μm CMOS eight-layer metal process which can operate at 1.5 V with a 420 MHz internal clock speed and 840 Mbits/sec I/O bandwidth per pin. In addition Virtex-II FPGAs have dedicated 18-bit multipliers and up to 3.5 Mbytes of dual-port RAM [Xil01a]. As integrated circuit technology continues to improve at an exponential rate, one can be assured that even more powerful FPGA devices will be available in the future.

2.2 Hardware Compilation

In Chapter 1, recent developments in microprocessor technology were described. Compared with FPGA designs, high-performance microprocessors have the advantage of higher clock rates, dedicated functional units (specifically the floating-point unit), and virtually unlimited memory. The programmable nature of FPGA devices, although providing the flexibility for logic customization, cannot usually match the clock rate of microprocessors. This is because the same logic function and routing requires more levels of logic and has higher capacitance in an FPGA than in a microprocessor. An FPGA also has a finite number of LUTs, registers and routing resources. To achieve an overall performance which is competitive to that of a microprocessor, it is essential for an FPGA design to achieve a higher degree of parallelism. On the other hand, resources should be carefully conserved. In general, to increase the degree of parallelism it is important to make as many LUTs compute simultaneously while at the same time ensuring that the design will fit in the given resources.

It is important to have a convenient design environment that elaborates parallelism and generates area-efficient designs. In the FPGA-based coprocessor approach, software programs should be able to be automatically translated to hardware without requiring much extra effort. Ideally, designers should be able to develop an application as a single program, and later this program can be automatically partitioned so that some parts execute in hardware and some parts in software, with both parts individually optimized for efficiency.

Hardware designers and programmers have traditionally considered hardware and software to be distinct entities with very little in common. Programs are predominantly regarded as ordered sets of statements that are to be interpreted sequentially. They encapsulate the execution flow of the algorithmic level behavior. Describing a circuit is still different from expressing an algorithm. Circuit descriptions are viewed as sets of concurrent operations in which the same operations recur every clock cycle [Wir98].

Hardware compilation bridges this gap by automatically translating a software program into a circuit description. This is similar to a software compiler that compiles a program into an executable. Hardware compilation permits designers to primarily focus on the algorithm with most of the circuit specification being automatically generated from the algorithmic description.

Research on hardware compilation can be considered divided into two main streams. The first concentrates on extracting the dataflow of an algorithm from a textual or graphical description [RCHP91, Wei97, WL99, MHD$^+$97, MRHH97, GSB$^+$00, Xil00a]. The other focuses, not only on the dataflow, but also on the generation of control sequences [PL91, PLL93, LW94, GL95, LFP94, Pag96, Wir98, ZL99, SVR$^+$98, JD99, PD00]. The latter approach is more general because most algorithms consists of combinations of data and control transfers. Nevertheless, the former approach yields simplified control circuitry and is suitable for dataflow-intensive, high-throughput applications, such as DSP.

2.2.1 Dataflow Extraction

The idea of dataflow extraction is to isolate the data transfers and operations of an algorithm and convert these into a hardware design. This approach cannot handle arbitrary transfers of control such as loops. When this approach is used in FPGA-based coprocessor design, designers need to explicitly identify the suitable parts of a program to be executed in hardware (usually computationally intensive inner-loops). The hardware generated from the dataflow extraction approach can be efficiently pipelined and its control circuitry is simple, enabling implementations with high throughputs. These implementations often require high-bandwidth data sources and sinks.

In 1991, Rabaey et. al. proposed a system that compiles a Silage language description into a dataflow graph and then maps and schedules the operations to a given number of operators [RCHP91]. This system is targeted for DSP chips so scheduling of the limited amount of operators on the chip is crucial.

In 1997, Weinhardt reported a compilation tool that takes a Pascal-like description and synthesizes a pipelined architecture [Wei97]. During compilation, alias and dependency analysis were performed. This analysis tells whether it is possible to normalize the inner loops. Normalization is the process of identifying whether the statements in the loops can be transformed such that the loop becomes the innermost loop (without nested loops) after transformation. The resultant architecture employs an acyclic dataflow corresponding to the normalized inner loops, and the outer loops become feedback paths around these inner loops. This work was later refined to include loop unrolling, tiling and merging so as to elaborate more parallelism in loop bodies [WL99].

In work by McCanny et. al., a hierarchical VHDL library that implements DSP functions was developed [MHD+97, MRHH97]. To construct a DSP system, designers pick the required components from the libraries and connect them, essentially describing the dataflow of an algorithm. The libraries were targeted for ASIC implementations, but its idea applies to FPGA implementations as well.

PipeRench, reported by Goldstein et. al., is another system that transforms a C-based description into a pipelined architecture [GSB+00]. The PipeRench compiler inlines all functions and unrolls all loops to generate a straight-line, single-assignment program which can be trivially mapped to a fully-pipelined architecture.

In 2000, a commercial tool that translates a Simulink system model (Simulink is bit-true system simulator that runs in the Matlab programming environment) to Xilinx hardware descriptions was introduced [Xil00a]. In this design flow, the system model description specified by a designer is first simulated in the Simulink environment for a given set of inputs. The system model may then be adjusted until the simulation output satisfies performance requirements. The system model is then translated to a netlist by the tool and taken for hardware implementation.

2.2.2 Dataflow and Control Sequence Extraction

By generating both dataflow and control sequences it is possible to convert arbitrary software programs to hardware implementations. This is a more general approach which for some algorithms may require substantial resource overhead. In such an approach, the dependencies of some operations, such as loops with variable number of iterations, are resolved at runtime but not during compilation. These dependencies degrade the throughput of the resultant implementation as compared with the dataflow extraction approach due to two reasons. First, the degree of parallelism is degraded because some operators are forced idle when waiting for an operand (such as the result from a loop); second, the control circuitry is more complicated.

In 1991, Page and Luk reported the compilation of Occam programs for FPGAs [PL91]. The hardware implementation sequentially executes one assignment statement or a set of explicitly specified parallel assignment statements in each clock cycle. Later, this language-based approach for developing hardware was extended to use the Ruby language [PLL93, LW94, GL95]. Ruby is particularly suitable for hardware designs. Instead of using assignment statements, the Ruby language uses functions and relations on the states and I/O of the system to describe a program. Another extension of the hardware compiler was to exploit the syntax of the source program to guide the layout in the implementation [LFP94]. This was achieved by introducing size and performance estimation into the compiler. Using guided layouts, faster and better P&R were obtained. In 1996, Page proposed a language called Handel and a compiler that translates Handel programs to hardware implementations [Pag96]. The Handel language is a small subset of Occam2 programming language. It has both sequential and parallel compositions. In 1998, Wirth reported a hardware compiler for programs written in an Oberon-like language [Wir98]. The above work has the commonality that they require designers' explicit specification of parallelism at the statement level.

In contrast, the hardware compilation system proposed by Zhu and Lin in 1999 [ZL99] employs process level parallelism. This system compiles a C-like programming model based on the communicating sequential processes (CSP) formalism. Each C-like function in the program corresponds to a process which can be either run in hardware or in software. This process-based model enabled an execution pattern similar to a multi-thread software program. This work addressed the partitioning and the interfacing issue between hardware and software.

Schaumont et. al. proposed a framework for the design of high speed ASICs, in which a hardware circuit is derived from two C++ classes, namely the signal class and the signal flow graph (SFG) class [SVR+98]. In this framework, expressions are operations on signal objects, and a set of signals assemble a SFG object. The SFG objects in a program were translated into a finite state machine (FSM), followed by the scheduling of these objects onto hardware.

There is also a commercial tool called A|RT which automatically converts C to VHDL, in which resource sharing can be specified explicitly such that implementations with different throughput and area can be addressed [JD99]. The tool emphasizes one-to-one mappings between VHDL and C statements, but did not seriously consider the performance of the resulting implementations.

Handel-C is another commercial tool, which is itself a hardware design language and a hardware compiler [PD00]. Being derived from the Handel language, Handel-C also requires explicit parallel constructs to control the spatial aspects of the design. Novel features of Handel-C include its support for I/O statements, which makes it suitable for the design of embedded systems.

2.3 Floating-Point to Fixed-Point Translation

Fixed-point implementations are characterized by the assignment of a fixed wordlength and a fixed exponent to each variable. This is in contrast with a floating-point implementation in which wordlengths of variables remain fixed but the exponents can change at runtime. Fixed-point hardware implementations are usually preferred over floating-point implementations since they are more favorable in terms of hardware resources, design cost and complexity, latency and power consumption, especially for those designs having variables with small dynamic range.

The major difficulty with designing fixed-point implementations arises from the effects of quantization. Quantization refers to the fact that a finite wordlength is used to represent a real number. There are two sources of quantization errors in an algorithm: first, the difference between the a real number and the value represented by the finite wordlength; second, the error produced using these finite wordlength representations as operands. To minimize relative error produced by quantization effects, the number of bits for the fractional part of a fixed-point variable should be maximized. At the same time, the remaining bits which are allocated for the integer part should be sufficient to avoid overflow during execution. Fixed-point representations are sensitive to quantization errors, particularly when the dynamic range of a variable is large. The quantization effects of a floating-point variable with the same number of bits is less pronounced if it has a large dynamic range. In other words, the exponent of a floating-point variable can be optimally adjusted according to its dynamic range, even if the exponent uses some bits and reduces the precision.

In software, programmers tend to use floating-point arithmetic for implementing algorithms since floating-point datatypes are integrated in the programming languages and libraries. The IEEE 754 Floating-Point Standard [ANS85] is used in almost every microprocessor system. In practice, a 64-bit double-precision floating-point format is used. A typical fixed-point design begins with a floating-point algorithmic description. After verification, each floating-point variable is translated into a fixed-point variable. The bit width of the fixed-point operators are also decided, usually by the wordlengths of the operands. To ensure the results obtained are correct, designers observe the dynamic ranges and errors of variables, and specify sufficient numbers of integer and fractional bits for each fixed-point variable and operator. Noting that an algorithm consists of many variables, each of which can be independently assigned different wordlengths for fractional and integer parts, it is obvious that searching for a set of optimal values in this enormous space is extremely tedious.

This floating-point to fixed-point design methodology is particularly meaningful for FPGA implementations because FPGAs are capable of bit-level processing and the wordlength of each variable can be chosen based on the precision required. In contrast, DSP and microprocessor implementations have fixed wordlengths and optimal scaling is used in the floating-point to fixed-point translation. For FPGA implementations, a set of optimized wordlengths and optimized variable scaling is applied to each variable.

In the literature, there have been two types of approaches, namely simulation-based [Sun91, KS94, SK94, SK95, KKS98, WBK+97, WBGM97, KKS99, KKS00, CRS+99, JD99] and analytical-based [CC99, CCL00, CCL01, CRS+99].

2.3.1 Simulation-Based Approach

The simulation-based approach compares the performance of the implementation with an error-free reference model (typically a floating-point software implementation). The wordlengths of variables are chosen heuristically while observing certain error criteria. Different configurations of wordlengths are simulated until the error is acceptable. This method effectively models the runtime behavior of the system, so it yields precise optimization results provided that the simulation dataset is representative. The drawback of this approach is its long simulation time (particularly for large complex systems) and the difficulty of choosing a representative dataset.

In 1991, Sung reported an automatic scaling method for fixed-point DSP designs [Sun91]. The method derives the dynamic ranges of every fixed-point variable based on the simulation of a hypothetical floating-point hardware model. This dynamic range information was used to determine the pre-scaling and post-scaling of the operands of every operation. Later, Kim and Sung reported a floating-point to fixed-point assembly program translator using a similar method [KS94]. The above work targeted fixed-point DSP chips, therefore only the optimal scaling of the variables were considered and the wordlengths of the variables were fixed to the word size of the DSP chips. In addition to their work on the scaling of variables, in 1995, Sung and Kim proposed a system which optimizes the wordlengths of variables in DSP systems [SK94, SK95]. The system minimizes a cost function which reflects resource requirements by modifying the wordlengths of the variables. The optimization is constrained so that some error criterion is satisfied. Furthermore, the authors developed an optimization utility for C++ based DSP programs [KKS98]. This utility implements a C++ class for fixed-point representation arithmetic operations. During the initialization of the fixed-point objects, the wordlength and modes of sign, overflow and quantization can be specified. There is a simulator, a range-estimator and an optimizer associated with the class, hence it is possible for users to specify only the algorithm in a C++ function, and the wordlengths of variables are automatically determined.

In 1997, Willems et. al. proposed a system called Fixed-point pRogramming DesiGn Environment (FRIDGE) which employs a similar approach, in which fixed-point variables are also modeled as a C++ class [WBK^{+}97, WBGM97]. FRIDGE initially obtains a worst-case estimation of integer and fractional wordlengths of every variable, and then progressively reduces these wordlengths. At every step of wordlength reduction, the system performs a simulation to ensure the design criteria is fulfilled.

2.3.2 Analytical-Based Approach

In the analytical-based approach, wordlengths are derived from source code structures, local annotations, interpolation and propagation of ranges of variables. This approach has faster runtime than the simulation-based approach, but its result is conservative and often leads to an over-estimation of wordlengths of the variables.

Cilio and Corporaal reported a tool that converts floating-point C codes to fixed-point in 1999 [CC99]. The tool computes the wordlengths of fixed-point variables by propagation of precision formats along the directed acyclic graph (DAG) built from the input C function. Later, Constantinides, Cheung and Luk reported a system called Synoptix [CCL00, CCL01] which takes a Simulink block diagram representing the dataflow of an algorithm. Variables are optimized based on estimating the observable effects from truncation and roundoff errors. To facilitate this estimation, a noise model was proposed. This model injects the errors of all the operators towards the overall system outputs, hence the error of the overall system outputs is a weighted function of every possible truncation error inside the system. The output of Synoptix is synthesizable VHDL code.

The work proposed by Cmar et. al. [CRS$^+$99] is an integration of the two approaches: simulated-based and analytical-based. They presented a system which converts floating-point C++ code to fixed-point C++ code. For the integer part, an analytical approach was used. The integer wordlengths were determined either by a statistical method or by propagation of dynamic ranges. For the fractional part, a simulation-based approach was used. Minimization of fractional wordlength was carried out by comparing the value of the fixed-point representation with that of the original floating-point representation.

The A|RT tool described in Section 2.2.2 also supports fixed-point modeling [JD99]. However, it requires a manual specification of variable wordlengths. Simulation of the algorithm with the specified variable wordlengths can subsequently be carried out.

2.4 Digit-Serial Computation

The most straightforward and common method to implement fixed-point arithmetic in hardware is to employ a two's complement bit-parallel architecture. In a bit-parallel architecture, an n-bit variable requires an n-bit wide datapath for its transmission, and on any clock cycle all the n data bits appear on their associated wires to form a two's complement representation of the value. Bit-parallel arithmetic operators process the entire variable in a single clock cycle.

In a bit-parallel implementation, an n-bit operation would require an n-bit operator. Most two's complement arithmetic operators are effectively repeating the same bitwise operation for every bit of a variable. For instance, an n-bit adder is formed by cascading n identical one-bit full-adders; an n-bit multiplier is formed by the summation of n partial products which needs n n-bit adders. It can be observed that if time-multiplexing is used, the computational logic can be reused so that different bits of the operands are processed in different clock cycles. The result is complete after all the bits have been processed.

This time-multiplexing technique is known as digit-serial computation. Instead of processing the entire word of a variable, digit-serial arithmetic operations are carried out in multiple clock cycles, a digit being processed on each cycle. A digit is a collection of bits from a numerical representation. Suppose a variable has n-bits and is divided into d-bit digits. Then, at least $\lceil n/d \rceil$ digits are required to form a complete representation of the variable. In order to identify which digit is being transmitted over the associated data wires in a clock cycle, each variable in a digit-serial system is associated with a control signal which its logic level is high when the first digit is being transmitted. This mechanism is depicted in Figure 2.2. There are two special cases of digit-serial architectures, namely when the digit size equals one and when the number of digits equals one. These two cases correspond to a bit-serial and a bit-parallel architecture respectively.

Due to the reuse of hardware, digit-serial architectures offer reduced area over bit-parallel implementations. Moreover, since the number of bits to be processed in a clock cycle is reduced, a higher clock rate can be achieved. The drawback of employing a digit-serial architecture are increased latency and perhaps decreased throughput. Increased latency is a direct consequence of processing an operation in multiple clock cycles. Decreased throughput is due to the dependencies and overhead of registers and feedback paths, which are essential to carry partial

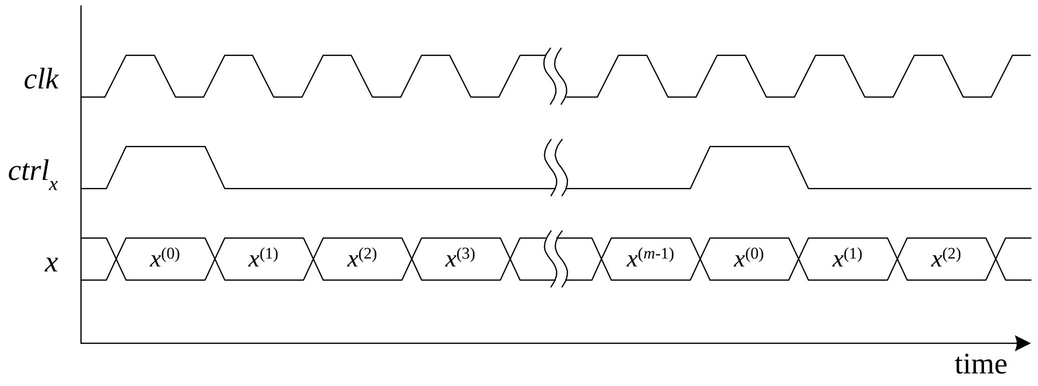


Figure 2.2: The control signal associated with a variable in a digit-serial system. clk is the clock signal, x is an m-digit variable and $ctrl_x$ is its control signal. The bracketed term of x indicates the digit of the variable.

results across clock cycles [PW90, HP95, SCP98].

A special class of bit-serial computation is distributed arithmetic (DA). DA offers an efficient method to implement a sum of products (SOP). SOP is widely used in DSP algorithms such as for infinite impulse response (IIR) and FIR filters. DA is applicable provided that one of the operands of every product term does not change during execution. Instead of requiring multipliers, DA utilizes a pre-computed LUT [Gos95, Xil96]. A detailed description of DA is given in Appendix C.

2.5 FPGA-Based Coprocessors

In high performance FPGA-based coprocessors, the bandwidth and latency of data transfers between the FPGA and CPU is of primary importance. Examples of applications that have high bandwidth requirements include image rendering, DSP and cryptographic systems. A low transaction overhead is also crucial for applications involving numbers of data transfers.

The organization of a typical computer, such as the Intel Pentium III processor-based personal computer (PC), is shown in Figure 2.3. The organization centers around the CPU. Most modern CPUs incorporate cache memory on chip to maximize the transfer bandwidth. The latest CPUs operate at over 1 GHz. The cache memory usually operates at full speed or half speed with respect to the CPU and the cache data bus is quad quadwords (256 bits) wide, yielding an equivalent bandwidth higher than 32 Gbytes/sec [Int01b].

The external connection of the CPU is commonly known as the Front-Side Bus (FSB). As the FSB is an off-chip component, its operating frequency is typically lower than that of the CPU (between 100 and 400 MHz). Mainstream systems typically have a 64-bit FSB operating at 133 MHz, equivalent to a bandwidth of 1064 Mbytes/sec. The primary interface between the FSB and the CPU is controlled by the Graphics and Memory Controller Hub (GMCH) that connects to the main memory (a bandwidth of 1064 Mbytes/sec according to the PC133 standard [Int98d, IBM98]). Other high-bandwidth devices include the video interface with a bandwidth of 1064 Mbytes/sec for the industrial Accelerated Graphics Port (AGP) 4X standard [Int98a].

The GMCH is connected to the I/O Controller Hub (ICH), which handles the connections to comparatively low-bandwidth devices. The two standard interfaces for these low-bandwidth devices are the 132 Mbytes/sec Peripheral Connectivity Interface (PCI) Local Bus [PCI] and the 33 Mbytes/sec Industrial Standard Architecture (ISA) Bus.

In general, bandwidth decreases as one moves from the CPU towards the peripheral bus. Similarly, the latency of data transfers is higher in the peripheral bus because more components are involved for data transfers between the CPU

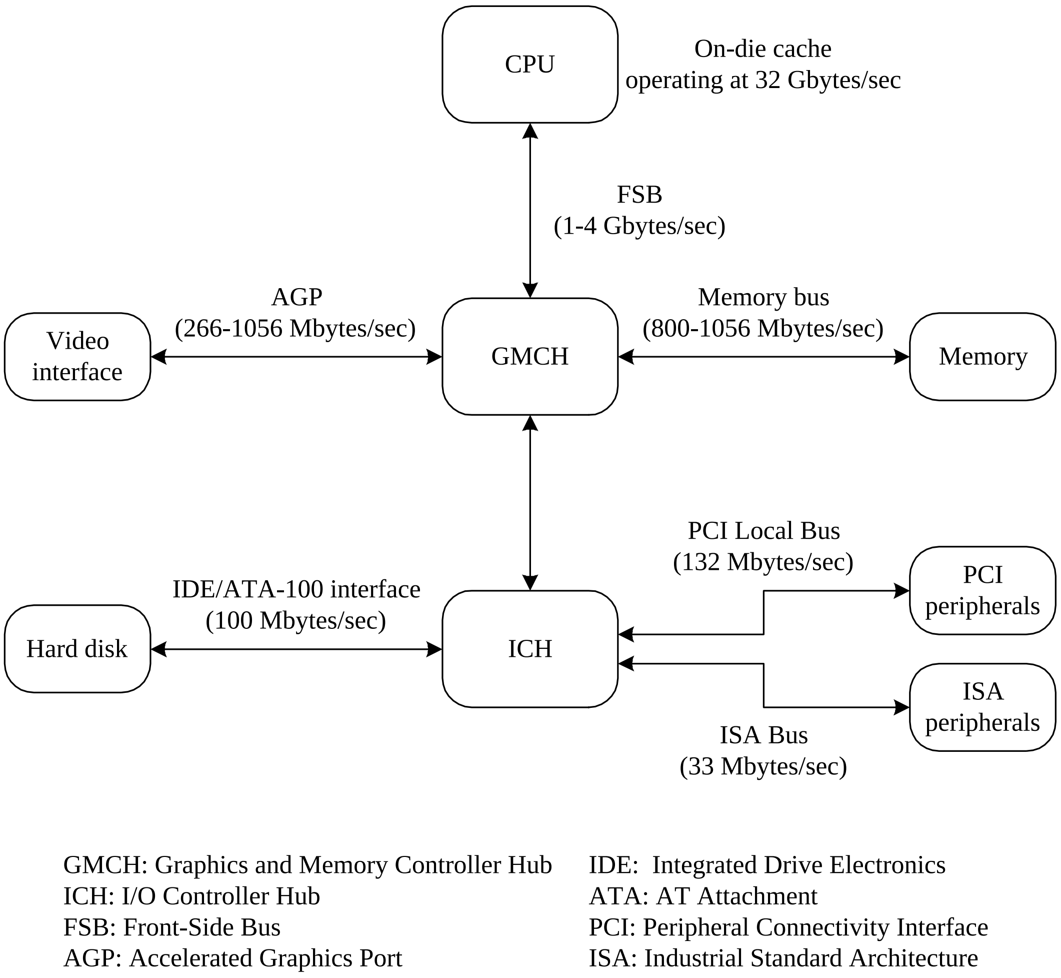


Figure 2.3: Organization of a typical computer (adapted from Intel 815E chipset block diagram).

and the peripheral bus. Also, each intermediate component accumulates small data packets to form larger data packets before a data transfer operation in order to minimize the number of transactions and reduce the cost of transaction overhead.

2.5.1 Reconfigurable Computing Platforms

Existing RC platforms generally connect to the CPU using the fastest available peripheral bus. Currently, most FPGA RC platforms use the PCI or PCI64 Local Bus (66 MHz, 64-bit, 528 Mbytes/sec) interfaces. The FPGA RC platforms used in this research include the Annapolis Wildforce [Ann99b], Wildstar [Ann00] and Wildcard [Ann99a] Reconfigurable Computing Engines.

The Annapolis Wildforce Reconfigurable Computing Platform is PCI Local Bus device with one Xilinx XC4085XL FPGA (Processing Element 0 (PE0)) and four Xilinx XC4062XL FPGAs (PE1 to PE4) interconnected by a crossbar. A Mezzanine card which contains up to 4M $\times$ 32-bit Static RAM (SRAM) can be connected to each PE. This platform offers approximately 700000 equivalent system gates and up to 80 Mbytes SRAM [Ann99b].

The Annapolis Wildcard Reconfigurable Computing Platform is CardBus (equivalent to the PCI Local Bus in a different form factor) device with one Xilinx Virtex XCV300 FPGA as the PE and two 64k $\times$ 32-bit SRAM. This system has 400000 equivalent system gates [Ann99a].

The Annapolis Wildstar Reconfigurable Computing Platform is a PCI64 Local Bus device with three Xilinx Virtex XCV1000 FPGAs (PE0 to PE2) with four 512k $\times$ 32-bit SRAM (two connected to PE1 and two connected to PE2). Mezzanine cards can be attached between PE0 and PE1, or between PE0 and PE2 (the platform can be populated with up to four Mezzanine cards) to further expand the system's memory. Each Mezzanine card contains two SRAM (each of which up to 256k $\times$ 64-bit SRAM) and a crossbar for simultaneously accesses from PE0 (32-bit) and PE1/PE2 (64-bit). This platform offers approximately 4.7 million equivalent system gates and a memory subsystem with up to 24 Mbytes SRAM. The aggregated memory width is $4 \times 32 = 128$ bits for PE0 and $2 \times 64 + 2 \times 32 = 192$ bits for PE1/PE2 [Ann00].

2.6 Summary

This chapter reviewed the literature on FPGA design methodologies. In particular, hardware compilation, floating-point to fixed-point translation, digit-serial computation and FPGA-based coprocessors were introduced.

Chapter 3

fp – From Floating-Point Algorithms to Fixed-Point Hardware

In this chapter, a design environment called *fp* which automatically translates floating-point algorithmic descriptions into hardware-efficient fixed-point implementations is described. First, the motivations and aims for developing this tool as well as its features are presented. Next, a description of the system components and the design flow are given. Each of the four main procedures in the design environment, namely compilation, simulation, optimization and hardware generation are then presented. Finally, the modular and hierarchical design of *fp*'s fixed-point and module libraries are described.

3.1 Introduction

In Section 2.2, hardware design methodologies were reviewed. The methods to translate a floating-point algorithm to a fixed-point implementation using digit-serial computation were discussed. Although these research threads are useful in designing FPGA applications, they have not been linked together in a design and synthesis tool.

In this dissertation, an attempt to incorporate these design methodologies into a single design space so as to improve the productivity of FPGA-based hardware designs was made. In particular, emphasis was placed on a design methodology that could be used efficiently in an FPGA-based coprocessor. It was a goal that a designer be able to identify the most computationally intensive inner-loops and then automatically translates these into FPGA-based hardware designs.

As described in Section 2.3, the translation of floating-point algorithms to fixed-point requires the assignment of a fractional wordlength to every variable. These fractional wordlengths determine the quantization errors of variables, which consequently affects the performance of the algorithm. If multiple wordlengths are allowed, the integer wordlength of every variable has to be decided also. This poses a tradeoff between area and precision. In Section 2.4 it was suggested that digit-serial architectures with low radix offer reduced area and an increased clock rate at the expense of increased latency and reduced throughput. One of the important steps in a digit-serial design is to determine the most appropriate radix so that the required balance in various performance measures is satisfied. Traditionally, these two design methodologies were not applied to the same design. Better designs may result if these design methodologies are applied simultaneously in a way that their advantages are incorporated. As both the floating-point to fixed-point translation and digit-serial computation design methodologies have an independent design space, the result of combining these design methodologies is a large design space which offers great flexibility if the most suitable design can be chosen.

In addition, ease of use is desired for the automated synthesis tools. Ideally, the design tools should translate a program to a circuit, optimize it automatically, and produce a near optimal implementation of an algorithm that satisfies user specifications: constraints on precision, area, latency and throughput of the resultant implementation. A computer-optimized implementation is likely to be more efficient than a human-optimized version because the translation process itself has a large and complex design space. Also, a computer can perform low-level optimizations which human designers may find too tedious.

To address these challenges, a design environment called *fp* was developed. The aims are enabling and simplifying the FPGA-based coprocessor designs while retaining maximum performance results. It has the following features:

- The input is a floating-point algorithmic description written in a subset of the C language which does not support pointers, dynamic memory allocation and I/O operations. Using the C language allows both software and hardware portions of an application to be designed with a single programming language.

- The dataflow of the algorithm is extracted and represented as a directed acyclic graph (DAG) [ASU85, FH95] using a method similar to the work of Weinhardt and Luk [Wei97, WL99] and Goldstein et. al. [GSB+00], ensuring implementations with high throughputs.

- Floating-point to fixed-point translation is performed by applying a simulation method to a user-supplied dataset. This approach follows that of Sung et. al. [Sun91, KS94, SK94, SK95, KKS98, KKS99, KKS00] and

Willems et. al. [WBK+97, WBGM97] but offers more precise estimations of the required wordlengths of variables, and therefore better area-efficiency of the resultant hardware implementations.

- Digit-serial implementations are generated with the option of sacrificing throughput and latency in order to obtain smaller area implementations.
- A cost function is used to explore the tradeoff among various performance measures, including precision, area, throughput and latency.
- The design environment is modular, hierarchical, and multiple module libraries are supported, allowing different hardware architectures (such as pipelined operators or multi-cycle operators) with different performance criteria. More complicated operations are built using the primitive operators in the library.
- Automatic synthesis of the datapath is performed and the output is in standard VHDL, enabling both synthesis and simulation.

This chapter focuses on the translation of floating-point algorithms to fixed-point hardware implementation in *fp*. An extension to support digit-serial implementations is described in Chapter 4.

3.2 System Components and Design Flow

To give a clear illustration of the design flow, the program in Figure 3.1 is used as an example throughout this chapter. This program contains relatively small number of operations and has loops and conditional statements. In this code, `fixed` is a datatype used by *fp* to identify variables that will be converted to fixed-point representations in the resultant hardware implementation.

```
void g(fixed &a, fixed &b, fixed &x, fixed &y)
{
    fixed r;
    int i;
    x = a + b;
    y = a * b;
    r = y - x;
    for (i = 0; i < 2; i++)
    {
        x = ((i == 0) ? y : x) * r;
        y = y + (a + b);
    }
}
```

Figure 3.1: The example algorithmic description used for illustrating the *fp* design flow.

The *fp* design environment consists of the following components:

- **Dataflow-extraction library:** A C++ compiler uses the dataflow-extraction library to compile a C program into a dataflow-extraction executable which generates an assembly-like DAG description of the algorithm in C++.
- **Fixed-point library:** The fixed-point library models fixed-point arithmetic operations, and tracks quantization and computation errors under different wordlength configurations.
- **Optimizer:** The optimizer processes and performs sample input vectors for simulation of an algorithm under different wordlength configurations in order to minimize area or errors at the outputs.
- **VHDL generator:** The VHDL generator uses the wordlengths determined by the optimizer and calls the module library to generate a synthesizable VHDL description of individual modules. It connects the modules together and inserts appropriate control circuitry.
- **Module library:** The module library contains modules for arithmetic operators. These modules serve two purposes: first, they provide area, latency and throughput estimations to the optimizer; second, they generate VHDL descriptions of the arithmetic operator.

Figure 3.2 illustrates the *fp* design flow. The primary input taken by the system is the algorithmic description. It is processed by a C++ compiler with the dataflow-extraction library to generate a dataflow-extraction executable. After executing this program, a DAG describing the dataflow of the algorithm in the form of a C++ program is generated. The DAG description is then compiled with four components, namely the fixed-point library, the optimizer, the module library and a user-defined cost function. In this pass, the module library is

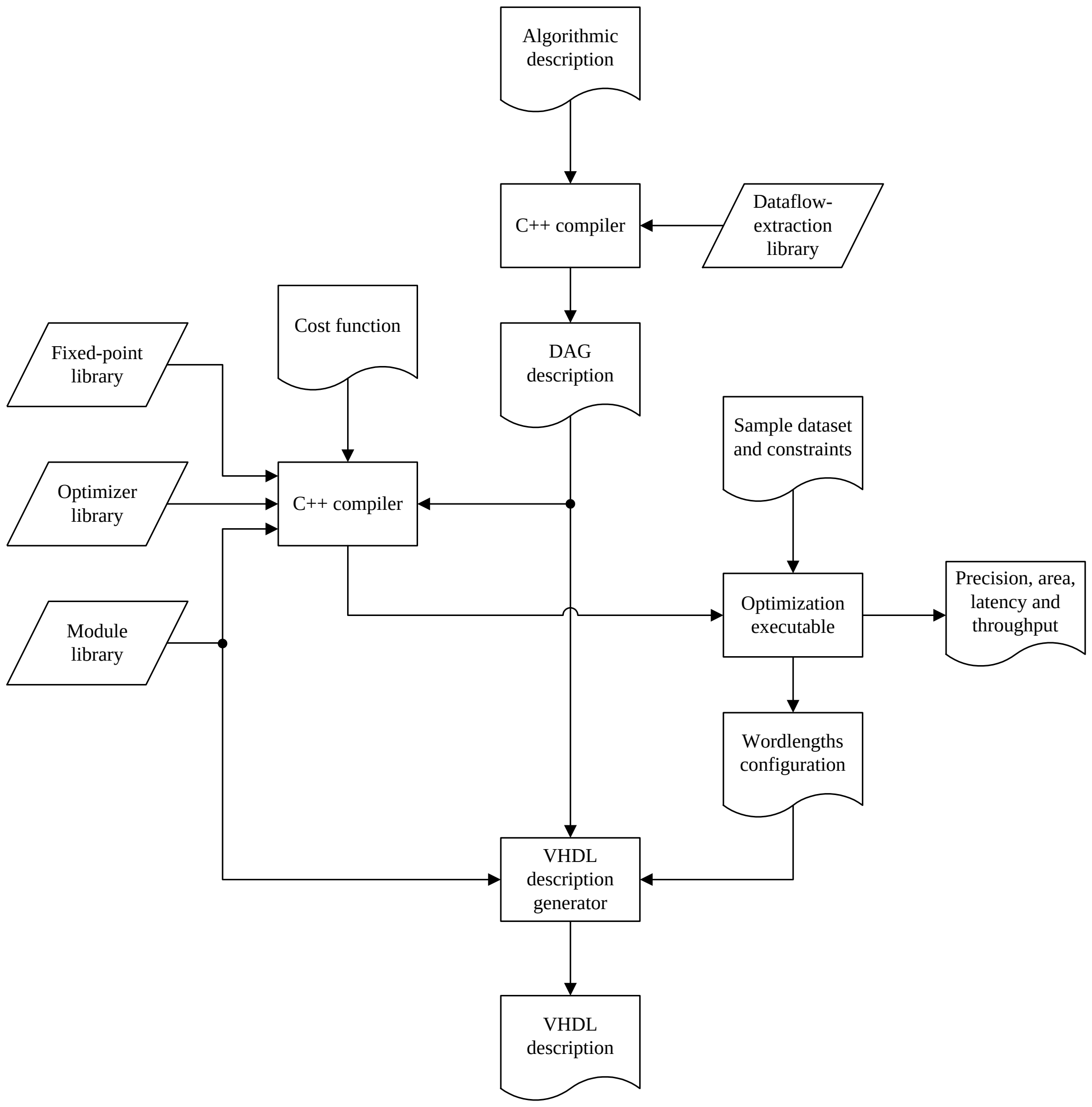


Figure 3.2: Design flow of *fp* design environment.

used for area, latency and throughput estimation only. The output of this pass is an executable (its main function is known as the *fp* kernel) which is used for wordlength optimization. This executable simulates the algorithm with a sample dataset, varying the wordlengths of variables to reduce a cost function (described later in Section 3.6.1) which incorporates quantization error, area, latency and throughput measures. The output of the *fp* kernel specifies a configuration of wordlengths that minimizes the cost function while satisfying user-specified constraints. Estimations of area and errors of every variable and output, as well as the latency and throughput of the resultant implementation are also calculated. The VHDL description generator uses the wordlength configuration and the intermediate DAG description along with the appropriate module library to generate VHDL descriptions of individual operators used in the design. At this step, proper control circuitry is also inserted into the design. The overall output of the design flow is a fixed-point VHDL description that implements the algorithm.

3.3 Compilation

The first step in the *fp* design flow is the translation of an algorithmic description written in C to an equivalent DAG description in C++. The use of a DAG facilitates the choice of employing a straight-line fully-pipelined architecture for throughput maximization.

To carry out this translation, a C++ class called the `fixed` class was developed. Operations on `fixed` objects, for example `a + b`, do not actually perform the arithmetic operations as typical classes do. Instead, this expression builds a sub-graph which represents $a + b$ in the eventual DAG description. In other words, during the execution of the algorithmic description, the DAG that repre-

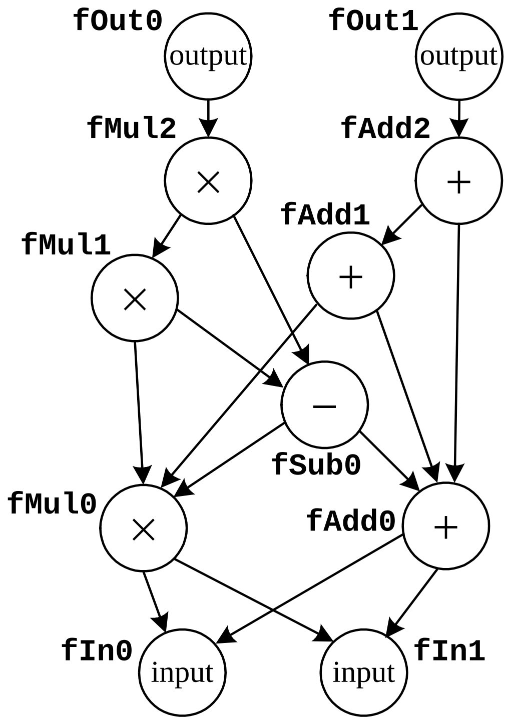


Figure 3.3: The resultant DAG built from the program in Figure 3.1.

sents the algorithm is progressively constructed. In the system architecture of *fp*, the `fixed` class is provided by the dataflow-extraction library.

Since the DAG is built according to the execution pattern of the program, loop-unrolling and resolution of conditional statements are carried out automatically. Due to the choice of using a fully-pipelined, non-feedback implementation, loops with a variable number of iterations cannot be facilitated. The mechanisms and the implementation details of the dataflow-extraction library is presented in Appendix A.1. The resultant DAG built from the code in Figure 3.1 is shown in Figure 3.3. Notice that the loop and the conditional statement at lines 8 to 12 are resolved according to the execution of the program.

The output of the dataflow-extraction process is the DAG description of the

```
Module g;
g = ModuleBegin("g");
FIn("fIn0", "a");
FIn("fIn1", "b");
FAdd("fAdd0", "fIn0", "fIn1");
FMul("fMul0", "fIn0", "fIn1");
FSub("fSub0", "fMul0", "fAdd0");
FMul("fMul1", "fMul0", "fSub0");
FAdd("fAdd1", "fMul0", "fAdd0");
FMul("fMul2", "fMul1", "fSub0");
FAdd("fAdd2", "fAdd1", "fAdd0");
FOut("fOut0", "fMul2", "x");
FOut("fOut1", "fAdd2", "y");
ModuleEnd("g");
```

Figure 3.4: The DAG description of the algorithmic description in Figure 3.1.

algorithmic description in C++ code. The DAG description corresponding to the DAG in Figure 3.3 is shown in Figure 3.4. The names of the operators (the node names in the DAG) are assigned by the dataflow-extraction library.

The dataflow-extraction library uses the compiler to handle loop-unrolling and resolution of conditional statements. With the optimization feature of the compiler enabled, arithmetic optimization can be carried out. The design of *fp* was simplified as the development of a dedicated compiler was unnecessary.

3.4 Conventions

3.4.1 Modules and Operators

The DAG description generated by the compilation step (Figure 3.4) is bounded by a pair of `ModuleBegin()` and `ModuleEnd()` function calls which indicate the beginning and the end of a module respectively. A DAG module can be viewed as an implementation of an algorithmic description. Eventually, a module is translated to a VHDL entity.

In between `ModuleBegin()` and `ModuleEnd()` are function calls that declare operators. Functions beginning with "`F`" declare operators inside the module. Operators can be interpreted as components in the final VHDL output. In *fp*, an operator carries out a certain class of operation (mostly arithmetic operations), with the wordlengths of the operation parameterizable and configurable. As shown in Figure 3.5, an operator contains an arithmetic core, surrounded by N_i inputs and N_o outputs. Inputs and outputs are assigned port names `in`n_i and `out`n_o respectively, where $0 \leq n_i < N_i$ and $0 \leq n_o < N_o$. In addition, an operator itself can instantiate sub-operators thus enabling a hierarchical design. Operators, sub-operators and ports are referenced by dot-separated names. For instance, the name of a port `out0` of an operator `fAdd0` is `fAdd0.out0`; the name of a port `in1` of a sub-operator `fMul1` of an operator `fSop0` is `fSop0.fMul1.in1`.

3.4.2 Simulation Directed Acyclic Graph

The simulation DAG is a data structure that is constructed and maintained during the execution of an optimization executable. The nodes of the simulation DAG are operator objects, each of which is created when an operator call is executed. The order of operator calls in the DAG description are essentially topologically

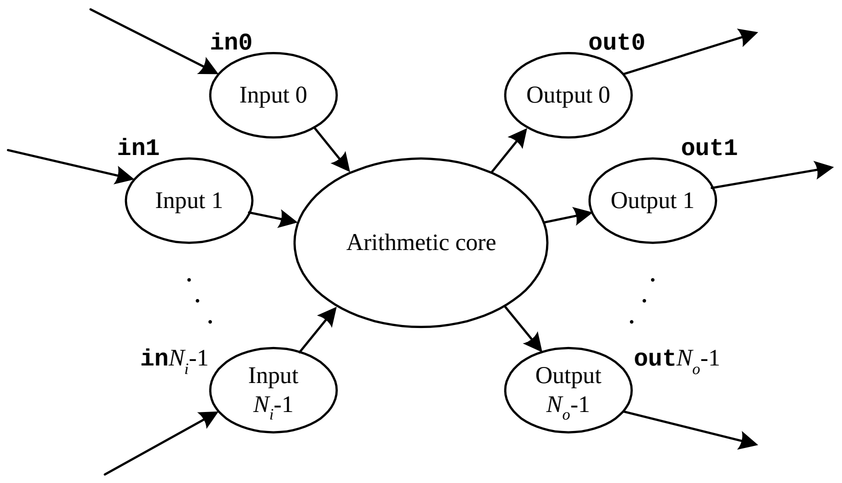


Figure 3.5: Structure of a fixed-point operator.

sorted, so the sequential execution of a code segment corresponding to a DAG description (the output of the compilation step) builds the simulation DAG. The simulation DAG has the same topology as the DAG built in the compilation procedure.

To illustrate the construction of the simulation DAG, the code segment in Figure 3.4 is used. The `ModuleBegin()` function call in line 2 indicates the beginning of the construction of a new module called g. The `Fin()` function calls in lines 3 and 4 indicate that module g has two inputs, namely a and b. In *fp*, inputs (and outputs) are considered as operators. The input operators themselves are named `fIn0` and `fIn1`. Lines 5 to 11 instantiate addition (`FAdd`), subtraction (`FSub`) and multiplication (`FMul`) operations. The first parameter to each function call is the name of the operator. The second and the third parameters are the operands. Note that the names of the operands are all aliases to the output port names of the operators. In lines 12 and 13, two outputs, namely x and y are declared. Their operator names are `fOut0` and `fOut1` respectively.

The `ModuleEnd()` function call in line 14 indicates the end of the construction of module g.

There is a class variable in the `Module` class which is an array of pointers. During the process of constructing the module, this array of pointers is appended with references to its operators. In subsequent steps of the simulation, the operators are accessible via this array.

3.4.3 Precision Format of Variables

The precision format of a fixed-point variable is parameterized by a pair (w, f), where w is the wordlength (including integer and fractional wordlengths) of the variable and f is the fractional wordlength. The exponent of the most significant bit (MSB) of the two's complement representation of the variable is 2^{w-f-1} whereas that of the LSB is 2^{-f}.

As an example, the two's complement representation $b = (b_{w_v-1}b_{w_v-2}\dots b_1b_0)$ of a value v with the precision format (w_v, f_v) is

$$v = -b_{w_v-1} \times 2^{w_v-f_v-1} + b_{w_v-2} \times 2^{w_v-f_v-2} + \dots + b_1 \times 2^{-f_v+1} + b_0 \times 2^{-f_v}$$

where $b_{w_v-1}, b_{w_v-2}, \dots, b_1, b_0 \in \{0, 1\}$.

3.4.4 Parameterization of Operators

In traditional fixed-point architectures [KKS98, WBK+97, WBGM97], only the precision of variables are parameterized. Operators derive their required bit widths from the precision of their inputs. A disadvantage of this method can be understood with consideration of variables that are shared among multiple operators. Consider a variable which is the input to two operators, with the output error of one operator more dependent on the quantization of the input than the other. For the first operator a higher precision input variable may be required, but for the second operator some fractional bits may be truncated to reduce area while still satisfying the precision requirements. When only the precision of variables are parameterized one cannot individually adjust the precision of a variable for different operators.

In contrast to previous approaches which only perform parameterization on the precision of variables, parameterization in *fp* is performed on the operators, allowing a variable to effectively have different precision formats when used as inputs for different operators. FPGAs have rich routing and storage resources, hence it is preferable to conserve computational resources by minimizing the bit widths of operators, rather than to conserve routing (the bus width for the connection between operators) and storage (stage latches inserted between operators for time-alignment, as described in Section 3.5.3) by minimizing the wordlengths of variables.

In the fixed-point library, all operator objects are parameterized. By modifying these parameters (using the array of pointers in the `Module` class described in Section 3.4.2), simulation of the algorithmic description with a given set of operator configurations can be achieved. The tradeoff between precision and area can thus be studied. The parameters of an operator describe the changes of wordlengths of its inputs and outputs. A typical parameterization of an N_i-operand N_o-output operator is depicted in Figure 3.6. In this figure, the terms in parentheses are precision formats of variables.

There are $N_i + 2 \times N_o$ parameters for a typical N_i-operand N_o-output operator. The parameters `inTruncation`n_i, $0 \leq n_i < N_i$, control the number of bits to be truncated from the LSB side of the N_i operands respectively. The arithmetic core derives the precision formats of every output based on the precision formats of

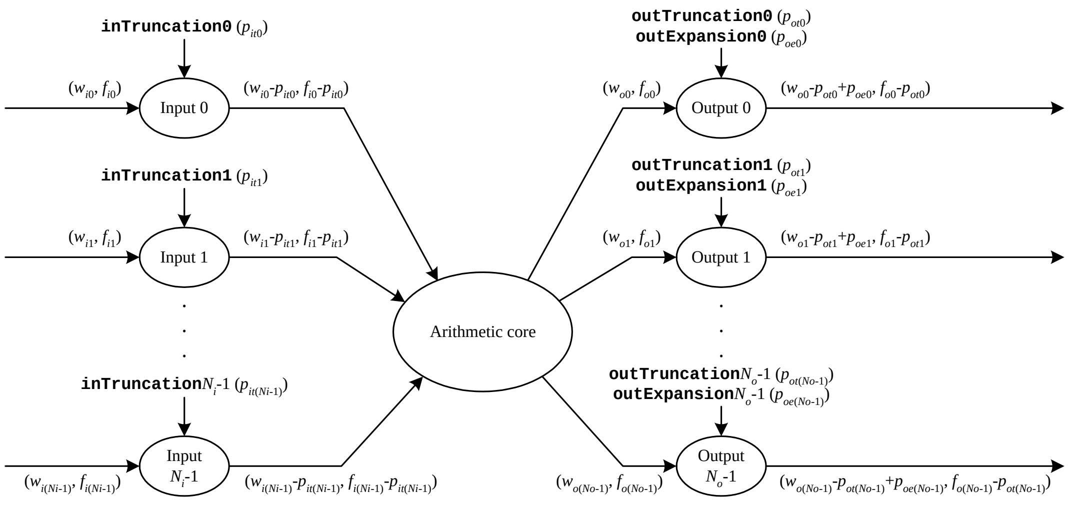

inTruncation0 (p_{it0})
(w_{i0}, f_{i0})
Input 0
(w_{i0}-p_{it0}, f_{i0}-p_{it0})
inTruncation1 (p_{it1})
(w_{i1}, f_{i1})
Input 1
(w_{i1}-p_{it1}, f_{i1}-p_{it1})
inTruncationN_i-1 ($p_{it(Ni-1)}$)
($w_{i(Ni-1)}$, $f_{i(Ni-1)}$)
Input N_i-1
($w_{i(Ni-1)}$-$p_{it(Ni-1)}$, $f_{i(Ni-1)}$-$p_{it(Ni-1)}$)
Arithmetic core
outTruncation0 (p_{ot0})
outExpansion0 (p_{oe0})
(w_{o0}, f_{o0})
Output 0
(w_{o0}-p_{ot0}+p_{oe0}, f_{o0}-p_{ot0})
outTruncation1 (p_{ot1})
outExpansion1 (p_{oe1})
(w_{o1}, f_{o1})
Output 1
(w_{o1}-p_{ot1}+p_{oe1}, f_{o1}-p_{ot1})
outTruncationN_o-1 ($p_{ot(No-1)}$)
outExpansionN_o-1 ($p_{oe(No-1)}$)
($w_{o(No-1)}$, $f_{o(No-1)}$)
Output N_o-1
($w_{o(No-1)}$-$p_{ot(No-1)}$+$p_{oe(No-1)}$, $f_{o(No-1)}$-$p_{ot(No-1)}$)

Figure 3.6: Parameterization of an N_i-operand N_o-output operator.

the inputs. The parameters `outTruncation`n_o, $0 \leq n_o < N_o$, control the number of bits to be truncated from the LSB side of the N_o outputs respectively. The parameters `outExpansion`n_o, $0 \leq n_o < N_o$ indicate the number of bits to be added to the MSB-side.

During an optimization, *fp* tries different values of the “truncation” parameters. By adjusting these parameters, the fractional wordlength and hence the precision of variables are reduced or increased and the optimizer investigates these effects on the overall algorithm.

In addition, because the arithmetic core computes the integer wordlength based on a worst-case analysis but a runtime analysis may indicate that a reduced integer wordlength does not lead to overflow (as will be presented in Section 3.5.1), the optimizer may adjust the values of the “expansion” parameters to override the integer wordlength determined by the arithmetic core. More specifically, negative “expansion” parameters reduce the integer wordlengths of variables and their dynamic ranges are reduced.

Nevertheless, all these parameters can be positive and negative, with negative "truncation" parameters indicating an increase in the fractional wordlength and positive "expansion" parameter indicating an expansion in the integer wordlength. It will be shown in Chapter 4 that these parameter settings are useful in a variable-radix variable-wordlength implementation. The standard technique where only the precision of variables is parameterized is the case where all of the parameters related to the inputs are zero.

For the sample program in Figure 3.1 which contains seven two-operand one-output operators, two zero-operand one-output operators (the inputs) and two one-operand zero-output operators (the outputs), there are $7 \times (2 + 2 \times 1) + 2 \times (0 + 2 \times 1) + 2 \times (1 + 2 \times 0) = 34$ parameters in total.

3.4.5 Quantization and Overflow Models

The fixed-point library in *fp* uses the truncation quantization model and the wrap-around overflow model. In Appendix A.2, a description of the commonly used quantization models (truncation and rounding) and overflow models (wrap-around and saturation) is given and their hardware implementations are highlighted. It is shown that the truncation quantization model and the wrap-around overflow model are the simplest form for hardware implementations so these were chosen for implementations with the highest area-efficiency. Concerning quantization, it is always possible to use one more fractional bit with the truncation model to obtain the same precision as the rounding model. Concerning overflow, range estimation is based on worst-case analysis so that overflow does not occur. Nevertheless, provided the sample dataset is representative of the runtime data, runtime analysis of dynamic ranges can be used. The optimization procedure may apply the runtime dynamic ranges via setting negative "expansion" parameters.

Using truncation quantization and wrap-around overflow models, with a precision format (w, f), the representable range is $[-2^{w-f-1}+2^{-f}, 2^{w-f-1}-2^{-f})$ and the maximum absolute quantization error is 2^{-f}. Assuming random quantization error, the mean absolute quantization error is statistically $2^{-f}/2 = 2^{-f-1}$.

3.5 Simulation

The second step in the *fp* design flow is to derive an implementation of the algorithmic description which has appropriate wordlengths satisfying the users' requirements. To achieve this, an optimization approach was used. In every iteration of the optimization, different configurations of variables' wordlengths are tried. Each trial is essentially a simulation of the algorithm under a specific configuration, from which the corresponding precision and area is estimated. Additionally, latency and throughput are computed.

A complete simulation of an implementation of the algorithmic description using a specific configuration of parameters involves the following steps:

1. Performing error extraction and range estimation;
2. Trimming operators whenever applicable;
3. Calculating latency and throughput of the module;
4. Estimating the area requirement of the implementation.

The outputs of simulation include the precision, area, latency and throughput measures of the implementation using the specific set of parameters.

3.5.1 Error Measurement and Range Estimation

Error measurement refers to the calculation of the difference between a fixed-point implementation with a specific configuration of precision formats and an error-free reference model. In *fp*, the reference model is a software double-precision floating-point implementation. Range estimation refers to the calculation of the maximum dynamic ranges of variables throughout the execution of the fixed-point implementation of the algorithmic description.

The terms used in the description of error measurement and range estimation procedures are explained below:

- **Worst-case analysis:** The analysis of the maximum magnitudes of dynamic range and quantization errors of an operation for arbitrary inputs.
- **Runtime analysis:** The estimation of dynamic range and quantization errors of an operation via a simulation using a representative dataset as input.
- **Fixed-point value of a variable:** The numerical value of a variable obtained from a bit-true fixed-point simulation of the algorithmic description (with a certain set of fixed-point precision formats of variables).
- **Floating-point value of a variable:** The numerical value of a variable obtained by executing the algorithmic description with double-precision floating-point operations.
- **Quantization error:** The absolute difference between the fixed-point and floating-point values of a variable.

The quantization error arises from fixed-point truncation and arithmetic operations. Consider the truncation of the fractional wordlength of a variable x from

m bits to n bits ($m > n$) represented by y. In this truncation, the maximum error is $2^{-n} - 2^{-m}$. If x has a quantization error of ϵ_x before truncation, then the accumulated quantization error of y, ϵ_y, is $\epsilon_x + (2^{-n} - 2^{-m})$. These two components, $2^{-n} - 2^{-m}$ and ϵ_x, are the sources of generated (quantization) error and propagated (quantization) error respectively. As an example, suppose x_1 and x_2 have quantization errors ϵ_{x_1} and ϵ_{x_2} respectively, $y' = x_1 + x_2$ and the fractional wordlength of y' is then truncated from six bits to four bits. In this case, the accumulated quantization error of y', $\epsilon_{y'}$ is the sum of $\epsilon_{x_1} + \epsilon_{x_2}$ (propagated error) and $2^{-4} - 2^{-6}$ (generated error).

The error measurement and range estimation procedure is described below. It begins by calculating, either automatically from the sample dataset or from the user input, the range of every input and the quantization error (if the sample dataset was in floating-point and was quantized to fixed-point). These initial ranges and quantization errors are processed using the simulation DAG to obtain a worst-case error analysis of the algorithm based on the given set of parameters. After the worst-case analysis, the wordlengths of variables are determined. Next, a runtime analysis of the algorithm based on the wordlengths of variables determined is performed. The input vectors in the sample data are retrieved and processed according to the algorithm. These sample data are computed both in fixed-point and in double-precision floating-point, and the quantization error is determined as their difference. During processing of the sample data, the maximum runtime dynamic ranges are also recorded. After all the entries in the sample data are processed, the worst-case and runtime error analysis of the inputs and outputs of the operators are calculated. Specifically, mean error, maximum error and signal-to-noise ratio (SNR) are calculated in the runtime error analysis. Denote the SNR of an input or an output z be SNR_z, measured in decibels (dB).

For the sample dataset **S**,

$$SNR_z = 20 \times \log \sqrt{\frac{\sum\limits_{s \in \mathbf{S}} ref_z(s)^2}{\sum\limits_{s \in \mathbf{S}} err_z(s)^2}} \tag{3.1}$$

where $ref_z(s)$ is the result at the input or output z using s as input to the fixed-point implementation and $err_z(s)$ is the corresponding quantization error at z. A more detailed description of this procedure can be found in Appendix A.3.

To illustrate the error measurement and range estimation procedure, the program in Figure 3.1 is used as an example. The input dataset contains 1000 entries of random values in the range $[-0.75, 0.25)$ for each of the two inputs. The input dataset is quantized to eight binary digits, so its maximum quantization error is $2^{-8} = 3.9063 \times 10^{-3}$. The parameters of all the operators are set to zero, corresponding to an implementation that neither truncates the fractional wordlength nor overflow. Table 3.1 presents the worst-case analysis of dynamic ranges, quantization errors and the set of precision formats of the inputs and outputs of operators. Table 3.2 presents the runtime analysis of dynamic ranges and quantization errors of the inputs and outputs, in which mean error, maximum error and SNR are listed.

By comparing the results in Tables 3.1 and 3.2, it can be observed that runtime analysis offer a less pessimistic estimate of quantization error, leading to smaller wordlengths. For instance, worst-case analysis suggests that the maximum quantization errors at the outputs are 4.292×10^{-2} and 2.142×10^{-2} for x and y respectively, whereas runtime analysis suggests these error magnitudes are 5.795×10^{-3} and 1.882×10^{-2} for the given inputs. Results of runtime analysis can be as little as one-eighth of that obtained by worst-case analysis. Furthermore, runtime analysis gives narrower bounds of dynamic ranges of variables than worst-case analysis. The dynamic ranges of the outputs extracted by runtime

Input or output	Worst-case analysis		
	Dynamic range	Maximum error	Precision format
`fIn0.out0`	$[-0.2500, 0.7500)$	3.891×10^{-3}	$(9, 8)$
`fIn1.out0`	$[-0.2500, 0.7500)$	3.891×10^{-3}	$(9, 8)$
`fAdd0.out0`	$[-0.5000, 1.5000)$	7.782×10^{-3}	$(10, 8)$
`fMul0.out0`	$[-0.1875, 0.5625)$	5.852×10^{-3}	$(17, 16)$
`fSub0.out0`	$[-1.6875, 1.0625)$	1.363×10^{-2}	$(18, 16)$
`fMul1.out0`	$[-0.9492, 0.5977)$	1.762×10^{-2}	$(33, 32)$
`fAdd1.out0`	$[-0.6875, 2.0625)$	4.292×10^{-2}	$(19, 16)$
`fMul2.out0`	$[-1.0085, 1.6018)$	4.292×10^{-2}	$(49, 48)$
`fAdd2.out0`	$[-1.1875, 3.5625)$	2.142×10^{-2}	$(19, 16)$
`fOut0.in0`	$[-1.0085, 1.6018)$	4.292×10^{-2}	$(49, 48)$
`fOut1.in0`	$[-1.1875, 3.5625)$	2.142×10^{-2}	$(19, 16)$

Table 3.1: Worst-case analysis of dynamic ranges, quantization errors and precision formats.

Input or output	Runtime analysis			
	Dynamic range	Mean error	Maximum error	SNR
`fIn0.out0`	$[-0.2500, 0.7461]$	1.988×10^{-3}	3.895×10^{-3}	44.54 dB
`fIn1.out0`	$[-0.2500, 0.7461]$	1.993×10^{-3}	3.900×10^{-3}	44.52 dB
`fAdd0.out0`	$[-0.4805, 1.4922]$	3.981×10^{-3}	7.586×10^{-3}	43.73 dB
`fMul0.out0`	$[-0.1769, 0.5567]$	1.132×10^{-3}	4.780×10^{-3}	40.38 dB
`fSub0.out0`	$[-0.9355, 0.5382]$	2.967×10^{-3}	8.740×10^{-3}	44.31 dB
`fMul1.out0`	$[-0.5208, 0.1201]$	8.710×10^{-4}	5.366×10^{-3}	39.43 dB
`fAdd1.out0`	$[-0.4228, 2.0488]$	4.994×10^{-3}	1.180×10^{-2}	43.12 dB
`fMul2.out0`	$[-0.0824, 0.4872]$	7.268×10^{-4}	5.795×10^{-3}	38.38 dB
`fAdd2.out0`	$[-0.9032, 3.5410]$	8.975×10^{-3}	1.882×10^{-2}	43.41 dB
`fOut0.in0`	$[-0.0824, 0.4872]$	7.268×10^{-4}	5.795×10^{-3}	38.38 dB
`fOut1.in0`	$[-0.9032, 3.5410]$	8.975×10^{-3}	1.882×10^{-2}	43.41 dB

Table 3.2: Runtime analysis of dynamic ranges, mean quantization errors, maximum quantization errors and SNR.

analysis are $[-0.0824, 0.4872]$ and $[-0.9032, 3.5410]$ for x and y respectively. As compared with those derived by worst-case analysis, namely $[-1.0085, 1.6018)$ for x and $[-1.1875, 3.5625)$ for y, runtime analysis may give an order of magnitude narrower bounds and hence leads to a saving of two bits.

3.5.2 Trimming of Operators

After the worst-case analysis is performed in the error extraction and range estimation procedure, the precision formats of the inputs and outputs of all the operators are decided. There are two special cases of interest, namely when there exist operators with all the outputs having zero wordlength, or when there exist operators with all the inputs having zero wordlength.

In either case, the operators concerned can be removed because their results are considered to be zero regardless of the values of the operands and the class of the operation. There is another case of interest, in which only one among the inputs of an operator has non-zero wordlength. For certain classes of arithmetic operations, such operators can be transformed to simpler logic or even removed. For example, additions and subtractions become identity operations and multiplications can have their outputs set to zero. Furthermore, the removal of operators may consequently provide the opportunity to trim other operators which are connected to the removed operators.

Operator trimming is the process of removing or transforming unnecessary operators. Trimming of operators may improve the performance of an implementation, not only in terms of area, but also in terms of latency and throughput. The removal of an operator shortens the critical path or improve the latency of a module.

The operator trimming procedure uses an iterative approach. In a single execution of the procedure, all operators that can be trimmed are removed or transformed. The trimming procedure repeats until no further removal or transformation is possible. A detailed description of this procedure is given in Section A.4.

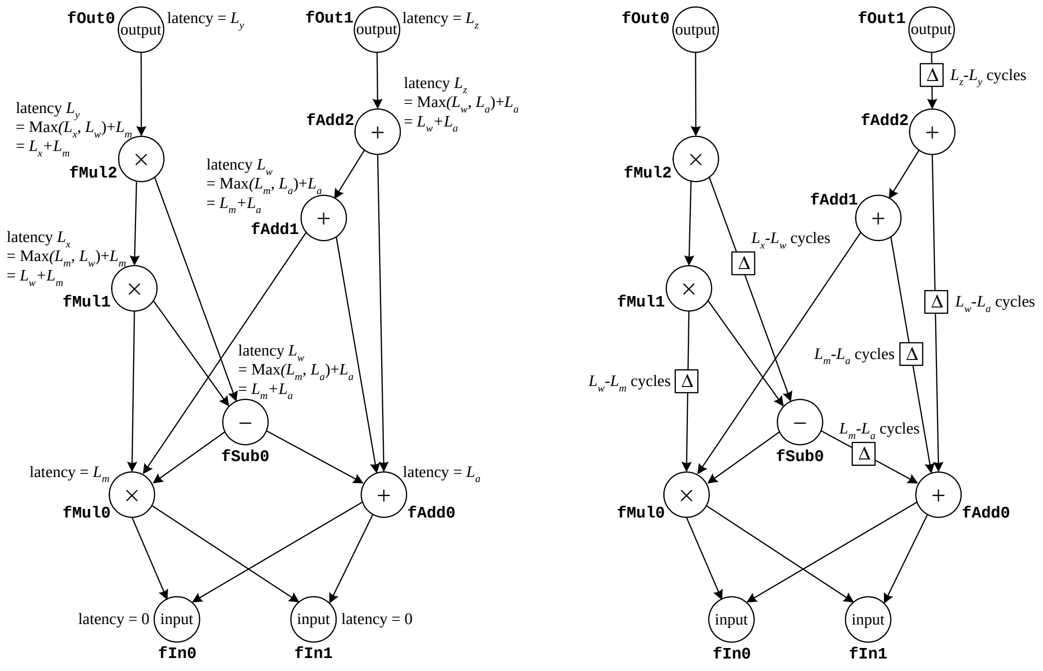
3.5.3 Latency and Throughput Calculation

The latency of an implementation is the time taken for an input to propagate to the output. The throughput of an implementation is the data processing rate. They are important design considerations in a hardware implementation. In *fp*, the latency and throughput are measured in the number of clock cycles only, and it is assumed that the clock frequency will not change significantly.

The latency or throughput of a module is derived from those of its operators. The *fp* module library enables one to compute these for every operator. The program queries the module library with first, the class of the operator, and second, the parameters of the operator. The module library responds to this query and sends the results back to the program.

Some operators, such as multiplication and division, can be implemented in multiple architectures (for example, restoring and non-restoring dividers, pipelined and multi-cycle multipliers). Each of these architectures offers a different performance tradeoff in terms of area, latency and throughput. As described in Section 3.8, the module library contains multiple architectures for the same operator. Users may choose among these architectures for the most suitable one for a given algorithmic description.

Most arithmetic operators require all of the operands to enter the operator in the same clock cycle. Time-alignment is applied when the latencies of the operators are different. It is the process of inserting stage latches (implemented using


fOut0
output
latency = L_y
fOut1
output
latency = L_z
latency L_z = Max(L_w, L_a)+L_a = L_w+L_a
fAdd2
latency L_y = Max(L_x, L_w)+L_m = L_x+L_m
fMul2
latency L_w = Max(L_m, L_a)+L_a = L_m+L_a
fAdd1
latency L_x = Max(L_m, L_w)+L_m = L_w+L_m
fMul1
latency L_w = Max(L_m, L_a)+L_a = L_m+L_a
fSub0
latency = L_m
fMul0
latency = L_a
fAdd0
latency = 0
input
fIn0
input
latency = 0
fIn1
L_z-L_y cycles
L_x-L_w cycles
L_w-L_a cycles
L_m-L_a cycles
L_w-L_m cycles
L_m-L_a cycles

(a) latencies of every operator (b) stage latches inserted

Figure 3.7: Time-alignment of the DAG in Figure 3.3.

shift registers) between operators so that operands have the same latency after passing through these shift registers [HP95]. Time-alignment is carried out simultaneously with the latency and throughput calculation procedure. Figure 3.7 illustrates the process of time-alignment using the DAG in Figure 3.3 as an example. In this example, it is assumed that adders and subtracters have latencies of L_a cycles, multipliers have L_m cycles, inputs and outputs have zero cycles, $L_m > L_a$. The time-alignment process first identifies the input with the maximum latency for every operator. A stage latch, with a latency equal to the latency difference between the identified input and every other input of the operator, is then inserted between the operator and the corresponding preceding output.

In the *fp* design flow, insertion of stage latches is done before area estimation because, as described in Section 3.6, the optimization process requires an estimation of area and a more accurate estimate can be obtained if the stage latches are taken into account.

Detailed descriptions of the latency and throughput calculation procedure and the stage latch insertion procedure are given in Appendixes A.5 and A.6.

3.5.4 Area Estimation

Area estimation is the process of evaluating the area of an implementation of the algorithmic description under a specific configuration of parameters. At this stage it is only possible to estimate the area of an implementation because firstly, optimizations could be performed in the synthesis of the VHDL descriptions and secondly, technology mapping in the FPGA design flow may further optimize the logic.

The *fp* kernel obtains the area of individual operators in a query to the module library. The kernel sums the areas of individual operators, and adds the area occupied by the stage latches to yield the approximate area of an implementation. *fp* is mainly targeted for the Xilinx XC4000 and Virtex series FPGAs, hence one unit of area corresponds to one logic cell (LC) during area estimation.

3.6 Optimization

In the optimization stage, different configurations of parameters are tried. The optimization objective is to derive a set of parameters which minimizes a cost function (in which the tradeoff among various performance measures is specified mathematically) as well as satisfying a set of performance constraints. Hence, the procedure is a constrained optimization.

The optimization procedure is outlined as follows:

1. Propose a set of parameters;
2. Simulate the algorithmic description according to the set of parameters;
3. Use the simulation results to compute a cost function;
4. Check for constraint satisfiability, and record the set of parameters that yield a lower cost;
5. Repeat until the optimization converges or user interrupts.

3.6.1 Cost Function

As inputs, an *fp* cost function takes the set of errors of inputs and outputs of every operator $\mathbf{E}$, estimated area A, latency LAT and throughput TP of the implementation. Note that the set of parameters proposed by the optimizer is not directly used to compute the cost function. Instead, these parameters are passed to the simulation procedure. Simulation can be considered as the mapping between the set of parameters and the performance measures. Denote $\mathbf{V} = \{\mathbf{E}, A, LAT, TP\}$. The cost function $f_{cost}(\mathbf{V})$ computes the sum of individual weighted functions of these variables,

$$f_{cost}(\mathbf{V}) = f_E(\mathbf{E}) + f_A(A) + f_{LAT}(LAT) + f_{TP}(TP). \tag{3.2}$$

A well-designed cost function should have the following properties:

- The function yields a global minimum cost $f_{cost}(\overrightarrow{v_{min}}) = cost_{min}$ if the corresponding implementation has the best compliance with the specified performance requirements.

- At any point $\overrightarrow{v}$ in the multi-dimensional space, $\overrightarrow{v} \neq \overrightarrow{v_{min}}$, the cost along the direction $\overrightarrow{vv_{min}}$ is monotonically decreasing.
- The function encourages faster movements (has a greater slope) when the current point is far away from $\overrightarrow{v_{min}}$, and slower movements (has a smaller slope) when the current point is relatively closer to $\overrightarrow{v_{min}}$. This improves the stability of the system by preventing jumps across the global minimum.

In practice, it is often easier to set latency and throughput via constraints and set f_{LAT} and F_{TP} to the zero function. Moreover, the elements in the set of errors **E**, except those corresponding to the overall module output (the inputs of output operators), are neglected by setting their effects on f_E to zero. This is due to the fact that only the errors at the outputs are considered in a typical hardware design. In most cases, setting f_E to be a weighted sum of the reciprocals of SNRs at the module outputs (Equation 3.1) is preferred, because SNR (sum square signal divided by sum square quantization error) is invariant to signal amplitude.

Hence, instead of using the general form of *fp* cost functions in Equation 3.2, the following form is often used.

$$f'_{cost}(\mathbf{V}') = f'_E(\sum_{z\in\mathbf{Z}} \frac{k_z}{SNR_z}) + f'_A(A) \tag{3.3}$$

where **Z** is the set of outputs of the module, k_z are weighting factors of the errors at output, $k_z \in \mathbb{R}^+$. An example of a cost function in this form is given in Appendix A.7.

3.6.2 Constraints

Throughout the optimization procedure, the *fp* kernel monitors whether the implementations satisfy certain user-specified design constraints. A set of parameters is only accepted if its corresponding implementation satisfies the performance

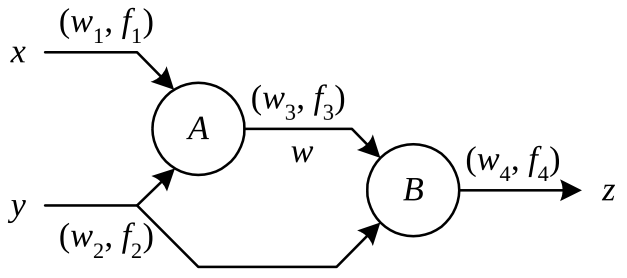


Figure 3.8: The two cascaded adders, A and B, used to illustrate the effect of adjusting parameters on precision formats.

constraints. These constraints include error, area, latency and throughput requirements. More specifically, error constraints can be specified as the mean error, the maximum error, the worst-case analysis error and the SNR on every input and output of the operators or the module; area constraints can be specified for the whole module or on a subset of the operators in the module.

3.6.3 Relationship between Parameters and Precision Formats

The effect of adjusting the set of parameters to the precision formats of inputs and outputs of operators, and consequently the precision, area, latency and throughput of the resultant implementation, is complex. To illustrate the effects of operator parameters on the precision formats, consider two cascaded adders, A and B, in Figure 3.8.

Let the precision formats of the inputs x and y, (w_1, f_1) and (w_2, f_2), be $(10, 8)$ and $(8, 7)$, their ranges be $[-2.0, 2.0)$ and $[-1.0, 1.0)$ respectively. If all the parameters (`inTruncation0`, `inTruncation1`, `outTruncation` and `outExapnsion0`, Section 3.4.4) of operators A and B are zero, the operators do not generate error but propagate errors (Section 3.5.1). The ranges of w and z are $[-3.0, 3.0)$ and $[-4.0, 4.0)$ respectively, and their precision formats, (w_3, f_3) and (w_4, f_4), are both $(11, 8)$. In this configuration, the errors at w and z are 45.99 dB and 43.50 dB (these errors are propagated from the inputs). The estimated area is 22 (two 11-bit adders).

Suppose the parameter `inTruncation0` of operator A becomes one. The precision formats of w and z would consequently be changed to $(10, 7)$. In this configuration, the errors at w and z are 43.50 dB and 42.03 dB. The estimated area is reduced to 20 (two 10-bit adders).

As another example, suppose the parameter `inTruncation0` is two, and the parameter `inTruncation1` of operator A and the parameter `inTruncation1` of operator B are both one. The precision formats of w and z would both become $(9, 6)$ and the errors at w and z are 37.49 dB and 36.05 dB respectively. The estimated area is further reduced to 18 (two 9-bit adders).

In previous approaches [SK94, SK95, KKS98], emphasis was made on searching the set of wordlengths of the fixed-point variables that minimizes resource requirements and the wordlength of each variable is treated independently. A disadvantage of such an approach is that a change in the wordlength of one variable does not affect the wordlengths of other variables accordingly. For example, in Figure 3.8, if the wordlength of w is reduced by one, the wordlength of z can also be reduced by one without further degrading the output precision.

Recall that the quantization error of a variable contains two components, propagated error and generated error (Section 3.5.1). In *fp*, the above mentioned problem is addressed by isolating the propagated and generated errors. More specifically, the propagated error of an operator is determined by its input wordlengths, whereas the generated error is determined by the operator parameters.

Modifying the parameters of an operator effectively adjusts the generated error of an operator and sets aside the effects of the propagated error. Using this approach in the above example, if adjusting a parameter causes the wordlength of w to reduce by one, the wordlength of z is also reduced by one automatically. The reduction of the wordlength of z does not affect the overall precision yet it reduces area requirements. Using this approach, the optimizer may omit the sets of variable wordlengths which lead to unnecessary area overhead and improves the optimization efficiency.

Note that with a zero optimization vector (that is, when all the operator parameters are zero), the operators produce bit-exact results without generating any error. In a wordlength minimization process, the number of bits being removed is small compared with the original variable wordlength [CCL99]. Based on this observation, the optimized parameters should be close to the zero. Settings all the parameters to zero is therefore used as an initial guess for optimization and it usually leads to fast convergence.

3.6.4 Example of Optimization

To demonstrate the optimization procedure, the program in Figure 3.1 is used as an example with the same dataset as that in Section 3.5.1, which contains 1000 entries of random values quantized to eight binary digits in the range $[-0.75, 0.25)$ for each of the two inputs. The aim of the optimization procedure is to minimize the area of the implementation with a minimum output SNR of 35 dB. The cost function in Equation A.1 (constructed using a Bessel function, details given in Appendix A.7) with $k_z = 1.0$, $k_E = 0.5$ (k_E was reduced by half because there are two outputs in the sample program), $k_A = 1.0$, $k_e = 6.7\times10^{-2}$ and $k_a = 1.0\times10^{-3}$ was used. Optimization constraints are the SNR requirements at the outputs being greater than or equal to 35 dB. This search space is multi-dimensional and discontinuous. The optimizer uses the downhill simplex method of Nelder and Mead [NM65] because it does not require the computation of derivatives and is suitable for such a search space.

The resultant implementation has SNRs of 35.51 dB and 38.23 dB at outputs x and y respectively with an estimated area of 187. The operator parameters and the precision formats of their inputs and outputs are listed in Table 3.3. In this table, the operator parameters are ordered as {`inTruncation0`, `inTruncation1`, `outTruncation0`, `outExpansion0`}. The runtime error analysis is summarized in Table 3.4. These results can be compared with the unoptimized implementation obtained by setting all parameters to zero (results in Tables 3.1 and 3.2). The unoptimized implementation has output SNRs 38.38 dB and 43.41 dB and an estimated area of 760.

3.7 Generation of VHDL Description

The output of the *fp* design flow is a VHDL description that implements the algorithmic description and uses the set of parameters derived from the optimization procedure. The generation of the VHDL description involves four steps, namely the declaration of the entity, the instantiation of components inside the entity, the insertion of stage latches, and the insertion of control circuitry.

The VHDL description begins with the declaration of the entity. As described in Section 3.4.1, a module corresponds to an entity in the VHDL description. It follows that the inputs and outputs of the module are respectively mapped as input and output ports of the entity.

Next, the VHDL descriptions that implement the operators with the specific

Operator	Operator parameter	Precision format		
		Input 0	Input 1	Output 0
`fIn0`	{N/A, N/A, 0, 0}	N/A	N/A	$(9, 8)$
`fIn1`	{N/A, N/A, 0, 0}	N/A	N/A	$(9, 8)$
`fAdd0`	{0, 0, 0, 0}	$(9, 8)$	$(9, 8)$	$(10, 8)$
`fMul0`	{1, 2, 2, 0}	$(8, 7)$	$(7, 6)$	$(12, 11)$
`fSub0`	{4, 1, 0, -1}	$(8, 7)$	$(9, 7)$	$(8, 7)$
`fMul1`	{3, 1, 4, 0}	$(9, 8)$	$(7, 6)$	$(11, 10)$
`fAdd1`	{2, 0, 0, 0}	$(10, 9)$	$(10, 8)$	$(12, 9)$
`fMul2`	{2, 0, 5, -1}	$(9, 8)$	$(8, 7)$	$(10, 10)$
`fAdd2`	{2, 1, 0, 0}	$(10, 7)$	$(9, 7)$	$(10, 7)$
`fOut0`	{0, N/A, N/A, N/A}	$(10, 10)$	N/A	N/A
`fOut1`	{0, N/A, N/A, N/A}	$(10, 7)$	N/A	N/A

Table 3.3: The parameters and precision formats of the operators after optimization.

Input or output	Runtime analysis			
	Precision format	Mean error	Maximum error	SNR
`fIn0.out0`	$(9, 8)$	1.988×10^{-3}	3.895×10^{-3}	44.54 dB
`fIn1.out0`	$(9, 8)$	1.993×10^{-3}	3.900×10^{-3}	44.52 dB
`fAdd0.out0`	$(10, 8)$	3.981×10^{-3}	7.586×10^{-3}	43.73 dB
`fMul0.out0`	$(12, 11)$	3.471×10^{-3}	1.563×10^{-2}	30.50 dB
`fSub0.out0`	$(8, 7)$	3.660×10^{-3}	1.686×10^{-2}	41.33 dB
`fMul1.out0`	$(11, 10)$	2.144×10^{-4}	9.831×10^{-3}	32.37 dB
`fAdd1.out0`	$(12, 9)$	7.792×10^{-3}	2.085×10^{-2}	39.06 dB
`fMul2.out0`	$(10, 10)$	1.182×10^{-3}	8.354×10^{-3}	35.51 dB
`fAdd2.out0`	$(10, 7)$	1.662×10^{-2}	3.212×10^{-2}	38.23 dB
`fOut0.in0`	$(10, 10)$	1.182×10^{-3}	8.354×10^{-3}	35.51 dB
`fOut1.in0`	$(10, 7)$	1.662×10^{-2}	3.212×10^{-2}	38.23 dB

Table 3.4: Runtime mean quantization errors, maximum quantization errors and SNR of the inputs and outputs after optimization.

configurations derived from the optimization procedure are individually generated. The VHDL generator carries out this task by passing these configurations to the module library. It is described in Section 3.8 that the module library is responsible for the generation of VHDL descriptions in additional to the calculation of latency, throughput and estimated area.

Operands in the implementation need to be time-aligned (Section 3.5.3). After the instantiation of operators, the VHDL description generator inserts stage latches at appropriate locations. Depending on the target device, the module library can be configured to use a shift register LUT (SRL) (for Virtex series) or a read-only memory (ROM) primitives with linear feedback shift registers (LFSR) (for XC4000 series) [Xil00g, Alf96] to implement the stage latches.

As the final step, proper control circuitry for the sequencing of operators is inserted. The control circuitry is implemented as a one-hot encoded finite state machine (FSM). The input control signal, *ctrl*, is set high when there is an input to the module. This signal passes through a shift register with a size equal to the latency of the module. All of the operators are connected to this shift register at the offset identical to the accumulated latency (Appendix A.5) of the operator. Equivalently, each of the operators receives a high control signal in the clock cycle when their operands are valid, and it may make use of this control signal to initialize an operation. The output of the shift register, *done*, is a signal that indicates valid outputs. Figure 3.9 demonstrates the insertion of the control circuitry for the implementation in Figure 3.7(b).

3.8 Module Library

The module library serves two purposes. During the optimization procedure, it is used by the optimizer to determine area, throughput and latency of individual

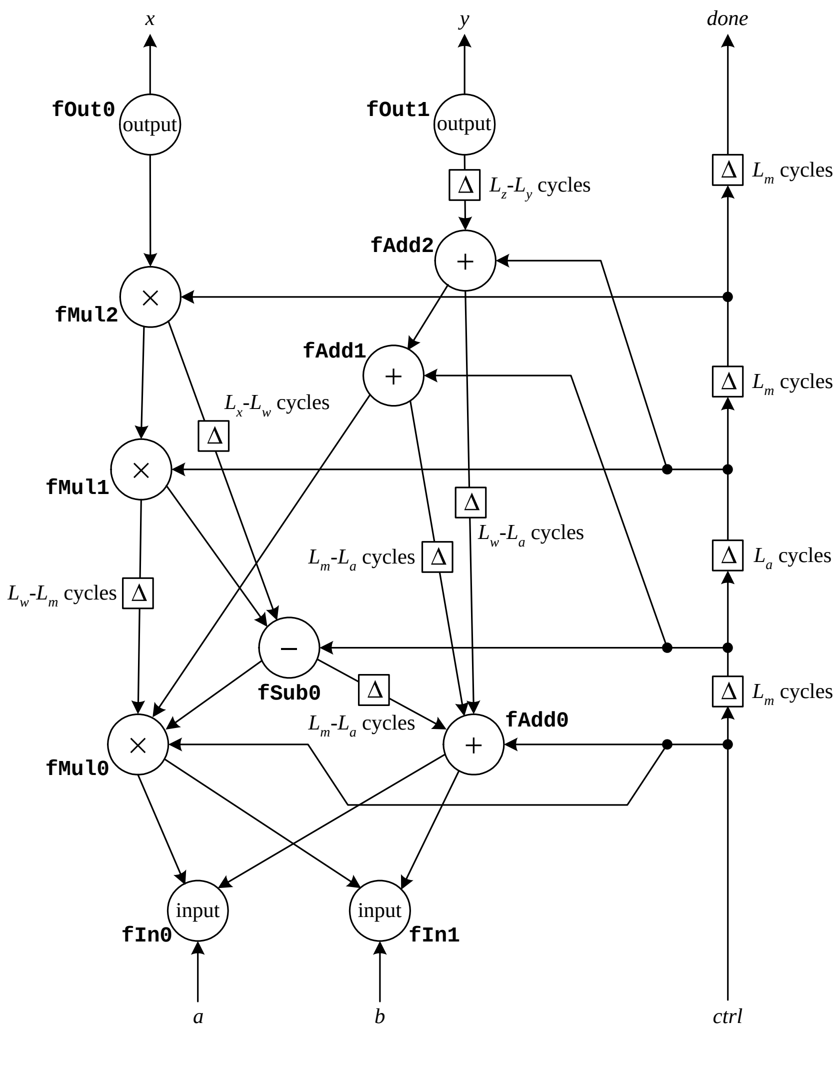


Figure 3.9: Control circuitry inserted for the implementation in Figure 3.7(b).

modules with a given set of parameters. The library is also used to produce VHDL descriptions of operators.

The fixed-point library (described in Section 3.5) is used to model the fixed-point arithmetic operations and is independent of the resultant architecture (for example, a pipelined or a multi-cycle implementation). In contrast, the module library is architecture-dependent. More specifically, the module library may contain multiple implementations of the same arithmetic operation, each of which has different area, latency and throughput.

The module library is essentially a collection of different implementations of operators. Depending on the performance requirements, users may pick an architecture for each class of operator used in the algorithmic description from the module library and pack them into an archive, known as the customized module library, for use in the design flow. This modular design allows users to quickly explore different implementations and their tradeoff in area, latency and throughput.

Due to the modular design, the module library is also hierarchical. An implementation of an operator can make use of other operators as sub-operators. For instance, an FIR filter operator may use addition and constant multiplication sub-operators. This hierarchical design enables the efficient development of complex operators.

3.9 Summary

A tool for the automatic translation of floating-point algorithms into fixed-point hardware called *fp* was presented. *fp* translates all the floating-point variables in an algorithm to fixed-point and optimizes their wordlengths so that the resultant implementation satisfies performance requirements in terms of precision, area, latency and throughput.

During optimization, the fixed-point operators are parameterized and the precision of every inputs and outputs can be individually adjusted. As compared with the standard technique where only the wordlengths of variables are parameterized (equivalently, only the wordlengths of operator outputs can be adjusted), the approach used in the *fp* explored a larger space for optimization. Also, in the standard technique the parameters are directly applied as wordlengths of variables. In contrast, the parameterization of operators in *fp* specifies the reduction or expansion of wordlengths. This method splits the effects of propagated and generated quantization error, and both of them can hence be independently adjusted.

In Chapter 4, the extension of *fp* to support digit-serial architectures will be described. In Chapters 7, 8 and 9, three applications of this tool, namely post-rendering 3D warping, an electronic cochlea and a systolic array for the DCT are presented. As will be shown in these applications, large and complex design spaces can be efficiently explored with *fp* and higher productivity can be achieved.

Chapter 4

Variable-Radix and Variable-Wordlength Architectures

Traditional digit-serial implementations use variables and operators with a common fixed radix or digit size [PW90, HP95]. This chapter describes the conflict between the traditional digit-serial and the multiple wordlength approaches. In order to combine these two design methodologies, an architecture in which the variables and operators can have different wordlengths and radices was developed. This architecture is known as the variable-radix, variable-wordlength architecture. A description of the extension of *fp* to support the variable-radix, variable-wordlength architecture is presented.

4.1 Conflicting Design Methodologies

Traditional digit-serial implementations use a common radix or digit size for all variables and operators (an implementation with a radix r is equivalent to having a digit size of $\log_2 r$). As described in Section 2.4, digit-serial architectures make use of a time-multiplexing technique so that the same computational logic is reused and different bits of the operands are processed in different clock cycles. Suppose the two's complement representation of a variable has w bits and is divided into d-bit digits (radix-2^d). Then, at least $n = \lceil w/d \rceil$ digits are required to form a complete representation of the variable. In digit-serial architectures, an operator often needs to identify when the first digit of an operand is entering the operator. To achieve this, every variable has an associated control signal. The logic level of this control signal is high only when the first digit is being transmitted. This mechanism is depicted in Figure 4.1. clk is the clock signal, x is a radix-2^d, w-bit, n-digit variable and $ctrl_x$ is its associated control signal. $x_{(a)}$ indicates the a-th bit in the datapath of x. $x_b^{(p,q)}$ is the b-th bit of the two's complement representation of the p-th value of x in the q-th digit. Note that $ctrl_x$ is high only when the first digit is being transferred on the datapath ($q = 0$).

Digit-serial architectures have the following properties:

- The size of a two's complement digit-serial operator is strongly dependent on its radix. For example, the size of an adder is proportional to the radix, and that of a multiplier has a square dependence on the radix. The larger the radix r, the higher the area requirements of the operator.

- The throughput of an operator (the number of clock cycles taken between consecutive inputs are taken) depends on the number of digits n. A variable requires at least n clock cycles for its two's complement representation to be transmitted over the datapath, and n clock cycles are required to complete

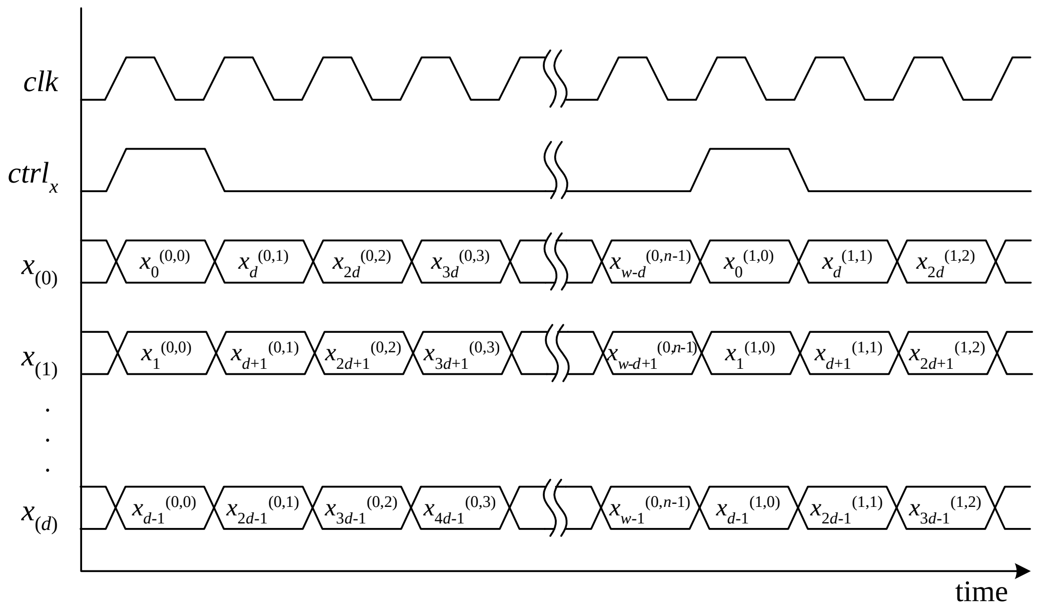

clk
ctrl_x
x_(0)
x_(1)
x_(d)
x_0^(0,0) x_d^(0,1) x_2d^(0,2) x_3d^(0,3) x_w-d^(0,n-1) x_0^(1,0) x_d^(1,1) x_2d^(1,2)
x_1^(0,0) x_d+1^(0,1) x_2d+1^(0,2) x_3d+1^(0,3) x_w-d+1^(0,n-1) x_1^(1,0) x_d+1^(1,1) x_2d+1^(1,2)
x_d-1^(0,0) x_2d-1^(0,1) x_3d-1^(0,2) x_4d-1^(0,3) x_w-1^(0,n-1) x_d-1^(1,0) x_2d-1^(1,1) x_3d-1^(1,2)
time

Figure 4.1: The transmission of a radix-2^d, w-bit, n-digit variable x and its associated control signal $ctrl_x$.

an operation.

- An operator for a w-bit variable, where $w \mod d \neq 0$, $w > d$, requires the same area and clock cycles as that of an operator for a $(\lceil w/d \rceil \times d)$-bit variable.

For a fixed wordlength implementation (w being fixed for all variables), the tradeoff between area and throughput can be addressed by adjusting the overall digit size. This is because the wordlength of a variable is the product of its number of digits and digit size ($n = \lceil w/d \rceil$).

Unfortunately, when applying digit-serial computation to a multiple wordlength implementation of a fixed radix r, some problems arise:

- **Resources are not fully utilized:** Suppose that two operators A and B have a common radix but their bit widths are w_A and w_B respectively. If $w_A \gg w_B$, operator A takes significantly more clock cycles (n_A) to complete its calculation than operator B (n_B) ($n_A \gg n_B$). Hence, unless all operators have a common bit width (as in a fixed wordlength implementation), there will be idle cycles for some operators. This effect is in more pronounced if the deviation of bit widths among operators is large (a common occurrence in multiple wordlength implementations, such as in the implementation of the DCT presented in Chapter 9).

- **Tradeoffs between area and throughput:** If the radix is fixed, the operator (C) with the largest bit width, w_C, has the largest number of digits, n_C, and the throughput of a system is limited to n_C. In order to increase throughput while preserving the precision of the implementation, the radix may be increased so that $n_C = \lceil w_C/r \rceil$ is reduced. However, this increases the area requirements of all operators because the radix is a common parameter. Therefore, the area overhead to increase the throughput is large.

4.2 Combining the Two Design Methodologies

To overcome the above mentioned problems, an architecture where each variable can have a different wordlength as well as a different radix is proposed. In this approach, the number of digits n remains fixed for all the variables and operators in the implementation.

The precision format of a variable is hence parameterized by a triplet (w, f, d). As before, w and f are the total wordlength and the fractional wordlength respectively and d is the radix of the variable. In general, $w \geq 0$, f is an integer and $d \geq \lceil w/n \rceil$. When $d > w/n$, sign-extension is carried out on the variable concerned.

As an example, the representation of a variable with precision format $(22, 5, 8)$ and $n = 3$ is

$$\overbrace{\underbrace{b_{16}b_{16}}_{\text{sign extension}}b_{16}b_{15}b_{14}b_{13}b_{12}b_{11}}^{\text{digit 2}}\ \overbrace{b_{10}b_9b_8b_7b_6b_5b_4b_3}^{\text{digit 1}}\ \overbrace{b_2b_1b_0.\underbrace{b_{-1}b_{-2}b_{-3}b_{-4}b_{-5}}_{\text{fractional part}}}^{\text{digit 0}}$$

The subscripts of b refer to their exponents in base 2. For instance, if the most significant bit of digit 1 is one, its partial value is $2^{10} = 1024$. Since the wordlength of the variable is 22 bits but 24 bits (3 digits $\times$ 8 bits) are required in the digit-serial implementation, two sign-extension bits are inserted in the most significant digit.

This variable-radix, variable-wordlength architecture has the following features:

- **Higher utilization of resources:** The number of digits n is common to all variables and operators throughout the system. This ensures all arithmetic operators have the same throughput and maximal resource utilization is guaranteed.

- **Operators have the same throughput:** Since all the operators have the same number of digits, they take the same number of cycles to complete and hence no bottleneck can occur.
- **Tradeoff between area and precision can be smoothly explored:** In contrast to the traditional digit-serial architecture, radices of individual variables can be increased or decreased independently and does not change those of other variables. Area overhead of increasing the wordlength of a variable is hence localized. This is in contrast with the traditional digit-serial architecture, in which a global area overhead is imposed if the throughput of the implementation must be retained. A more cost-effective tradeoff between performance and resource requirements may possibly be achieved.

The drawbacks of this approach are:

- **Operators expect operands with the same radix:** Most two's complement arithmetic operators require that their operands have the same radix. This requires extra hardware to convert variables from one radix to another. Fortunately, as discussed in Section 4.3, the property that FPGAs are register-rich and the ratio of storage elements to computational elements is high (Section 2.1) enable an efficient implementation of the converters.
- **Complicated design space:** Since $d \geq \lceil w/n \rceil$ (but not necessarily $d = \lceil w/n \rceil$, as will be described below), the digit size of every variable needs to be determined. The digit size, being an additional parameter to be considered for every variable, means that finding a good solution requires searching a large design space.

In the traditional digit-serial architecture approach the radices are the same

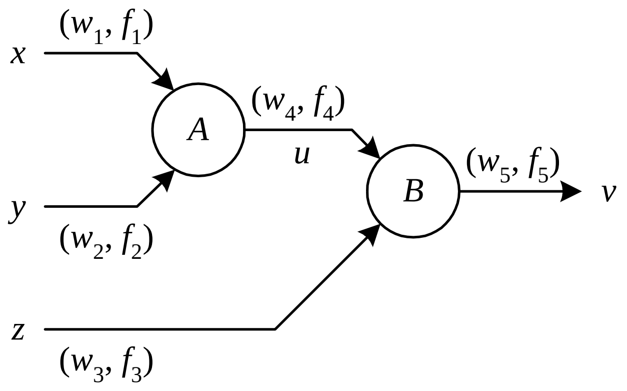


Figure 4.2: The two cascaded adders, A and B, used to illustrate the variable-radix, variable-wordlength architecture being a generalization of the traditional digit-serial architecture.

for all variables in the system. This variable-radix, variable-wordlength architecture is a generalized form of the traditional digit-serial architecture. The two cascaded adders, A and B, in Figure 4.2 are used to illustrate this generalization. Let the precision formats of x, y and z be $(9, 4)$, $(11, 8)$ and $(14, 8)$ respectively. To ensure an implementation produces the same result as a floating-point implementation, the precision formats of u and v should be respectively $(12, 8)$ and $(15, 8)$. Let $n = 3$, the triplet parameterization of x, y, z, u and v can be $(9, 4, 3)$, $(11, 8, 4)$, $(14, 8, 5)$, $(12, 8, 4)$ and $(15, 8, 5)$ respectively, in which the total width of the datapath (but not necessarily the area requirement) is minimal. In this case, the bit width of operator A is four and that of B is five.

In this example, the deviation in dynamic range of the variables is not significant, and the variable-radix, variable-wordlength architecture may not offer improvements over the traditional serial architecture. Hence, the *fp* kernel may apply the following set of parameters during optimization, so that the resultant implementation becomes a traditional digit-serial architecture: x.`outTruncation0` =

-1 (the fractional wordlength of x increases by one bit), x.`outExpansion0` $= 3$ (the integer wordlength of x increase by three bits), y.`outExpansion0` $= 4$ (the integer wordlength of y increase by four bits) and A.`outExpansion0` $= 2$ (the integer wordlength of u increase by two bits). The resultant wordlengths of all variables would become 15, therefore they have a common digit size of $15/3 = 5$ (radix-32). This example also illustrates the usefulness of negative "truncation" and positive "expansion" parameters (Section 3.4.4).

To identify whether the variable-radix, variable-wordlength architecture may offer a better area-efficiency during the optimization procedure, the area occupied by the conversion modules (Section 4.3) is also taken into account during area estimation. The optimization procedure applies different combinations of the following strategies to every operator, evaluates the area-efficiency of the corresponding implementation and determines how a better implementation can be achieved:

- Apply conversion modules at inputs and outputs of the operator, so that the bit width of the arithmetic core and hence the core area is optimized. The cost of this strategy is the area and perhaps latency overhead imposed by the conversion modules.
- Utilize an arithmetic core with a larger bit width, such that no conversion modules are required (at the expense of an increase in the area of the arithmetic core).
- Choose a compromise between the above two strategies so that some inputs and outputs undergo radix conversions, while the others remain unchanged. In this case, the arithmetic core is not of the minimal area, but the area overhead imposed by the conversion modules is reduced.

Usually, in a variable-radix, variable-wordlength implementation, all three strategies mentioned are applied to different operators.

Similar to the traditional digit-serial architecture, the variables in the variable-radix, variable-wordlength architecture have associated control signals. The control signal is set high when the first digit of the variable is being transmitted.

4.3 Conversion Modules

Conversion modules are used to translate the precision format of a variable (specified by the triplet (f, w, d)) from one to another. A conversion module is composed of registers and multiplexers. The inputs are latched into the registers which connect to multiplexers. The multiplexers perform the conversion of the precision format.

The latency of a conversion module is critical to its design. Suppose the latency l is known and assume the input datapath propagates through a shift register (with a width equal to the input digit size). The synthesis tool simulates this shift register for every clock cycle since the l-th cycle after the input enters the module, and derives the mapping between the contents of the shift register and the input's two's complement representation. The conversion module is constructed using this information by multiplexing the appropriate bits in the shift register and input bits to determine each output bit. In the final step, unused registers (those with zero fanout) are removed and the multiplexers are simplified. Details of these algorithms are given in Appendix A.8.

As an example, Figure 4.3 shows the conversion module for a three-digit variable x from a $(12, 3, 4)$ format into $(15, 4, 5)$ format. The input datapath is $a = \{a_3, a_1, a_1, a_0\}$ and *ctrl* is the control signal associated with a which resets the counter Δ_c when it is high. Δ_3, Δ_2, Δ_1 and Δ_0 are latches and the output datapath is $b = \{b_4, b_3, b_2, b_1, b_0\}$. When the first digit of x is available at a (that is, $a_3 = x_0$, $a_2 = x_{-1}$, $a_1 = x_{-2}$ and $a_0 = x_{-3}$), *ctrl* is high. In the next clock cycle, Δ_3, Δ_2, Δ_1 and Δ_0 are latched with the values of x_0, x_{-1}, x_{-2} and x_{-3} respectively, and Δ_c is reset to zero since *ctrl* was high in the previous clock cycle. Consequently, the outputs of the multiplexers and hence $b = \{\Delta_3, \Delta_2, \Delta_1, \Delta_0, 0\} = \{x_0, x_{-1}, x_{-2}, x_{-3}, 0\}$ (zero-padding is carried out). Similarly, $b = \{a_0, \Delta_3, \Delta_2, \Delta_1, \Delta_0\} = \{x_5, x_4, x_3, x_2, x_1\}$ when $\Delta_c = 1$ and $b = \{\Delta_3, \Delta_3, \Delta_3, \Delta_2, \Delta_1\} = \{x_8, x_8, x_8, x_7, x_6\}$ when $\Delta_c = 2$ (sign-extension is carried out).

As can be observed, the ratio of combinational logic and registers is suitable for FPGA implementations. Also, conversion modules can be placed at inputs and outputs of operators which reduces the spatial separation between them. Generally, this architecture produces little impact on the overall system clock rate.

4.4 Extending *fp* to Support the Variable-Radix Variable-Wordlength Architecture

In order to support the variable-radix, variable-wordlength architecture, the simulation, optimization and hardware generation procedures in *fp* were updated in the following manner:

- The optimization procedure was updated so that it is able to control the precision formats (the three elements of the triplet) of variables by adjusting the operator parameters. The optimizer is also able to control and optimize the number of digits n (discussed in Section 4.2).

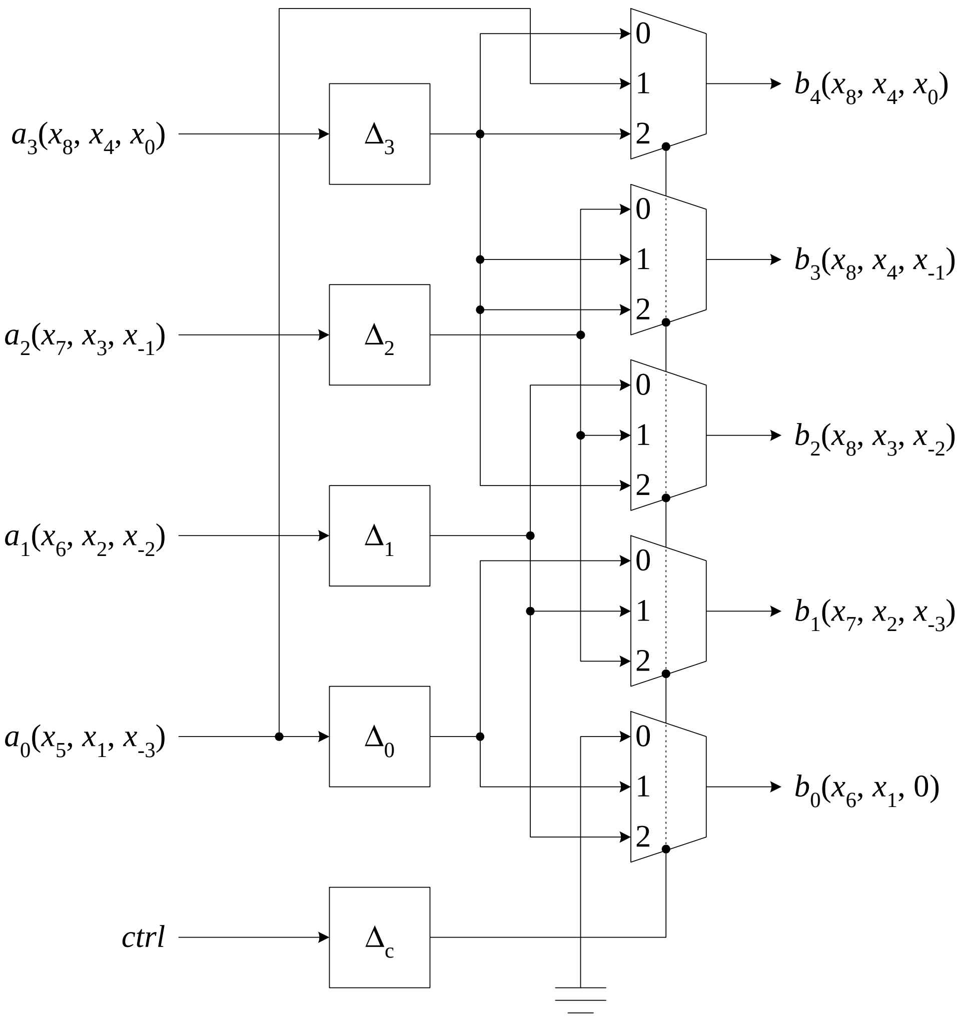


Figure 4.3: The module for converting a three-digit variable from a precision format of $(12, 3, 4)$ to $(15, 4, 5)$.

- The simulation procedure was updated so that the area estimation procedure includes the conversion modules and the latency calculation is includes the conversion modules (discussed in Section 4.3).
- The module library was updated to include a module for the conversion modules.

4.5 Summary

This chapter described the integration of multiple wordlength and digit-serial design methodologies into a single design environment. The efficient utilization of hardware resources in the traditional digit-serial architecture was addressed using a variable-radix, variable-wordlength architecture in which the number of digits is fixed, but variables can have different wordlengths. An empirical study of the utility of the variable-radix, variable-wordlength capabilities of *fp* was explored in an implementation of DCT presented in Chapter 9.

Chapter 5

Pilchard – Improving the Interface between PC and FPGA

Improvements in VLSI technology have led to microprocessors operating at over 1 GHz and FPGA devices with capacities of several million equivalent transistor gates. Unfortunately, current bus technology has not kept pace with these improvements. Nowadays the performance of an FPGA-based coprocessor system is often not limited by the speed of the FPGA circuit, but by the interconnecting bus between the CPU and the FPGA board.

This chapter presents an FPGA reconfigurable computing (RC) platform called *Pilchard*. For an FPGA-based coprocessor system in which the FPGA and the CPU should be tightly-coupled, having a high data rate is particularly important and *Pilchard* was designed to improve this so as to eliminate the bottleneck of data transfer between the two entities. Design considerations of *Pilchard* is first discussed. Its hardware and software components followed by an extensive performance evaluation are then presented.

In this work, the design of the PCB and device driver is the work of Dr. Philip

Leong, while the design of the PC-FPGA interface and the application programming interface (API) is the work of the author.

5.1 *Pilchard* Design Considerations

Modern CPUs operate at over 1 GHz internal clock frequencies, with 100 to 400 MHz 64-bit I/O. These clock rates produce I/O requirements that saturates the bus which connects the memory to the CPU. The sustained I/O bandwidth of a CPU is usually over 500 Mbytes/sec.

With the latest FPGA devices, FPGA design can now operate with external clock frequencies over 100 MHz with 64-bit I/O and up to 200 MHz internal clock rates. With low-voltage differential signaling (LVDS), up to 311 MHz external I/O can be achieved [HB01]. The sustained I/O bandwidth of an FPGA design is usually around 300 Mbytes/sec, but can be as high as 2 Gbytes/sec.

However, when an FPGA is used as a coprocessor to a typical PC, the throughput is on the order of 10 to 100 Mbytes/sec. The limited bandwidth arises from the interconnecting bus between the two devices. The majority of PCs use the 33 MHz, 32-bit PCI Local Bus and the CardBus interface, both having a maximum bandwidth of 132 Mbytes/sec and a typical bandwidth of 60 Mbytes/sec. Server class machines are equipped with the higher speed, higher bandwidth 66 MHz 64-bit PCI64 Local Bus (maximum bandwidth 528 Mbytes/sec), but the sustained transfer rate depends on the chipset and the peripherals connected to the bus (the bandwidth is shared among all the peripherals attached).

The PCI Local Bus is not considered an ideal interconnecting bus for FPGA-based coprocessors. The disadvantages of the PCI Local Bus for this application are:

- The PCI Local Bus is distant from the CPU in the PC architecture as compared with the Front-Side Bus (FSB) and the memory bus. The processor and the FPGA coprocessor are not tightly-coupled.
- The 33 MHz clock rate and 32-bit datapath has a maximum throughput of only 132 Mbytes/sec.
- The bandwidth provided by the PCI Local Bus is shared among many devices, such as disks, network interfaces and audio devices.
- The typical bandwidth that a single device can obtain is considerably lower than the 132 Mbytes/sec maximum throughput.

In any balanced PC, the memory bandwidth is higher and of a lower latency than that of any peripheral bus (the PCI64 and PCI Local Buses, the Accelerated Graphics Port (AGP) and the Industrial Standard Architecture (ISA) bus). This is because memory accesses are made much more frequently than I/O requests. As an example, the standard dual inline memory modules (DIMM) used even in low-end PCs operate at either 100 MHz or 133 MHz with 64-bit data, providing a maximum bandwidth of 1064 MB/sec. In the next generation of PC, more advanced memory interfaces will be deployed. For instance, the double data rate (DDR) synchronous dynamic random access memory (SDRAM) interface, which achieves double the date rate of SDRAM by supporting data transfer on both rising and falling edges of a 133 MHz clock, has a maximum transfer rate of 2128 Mbytes/sec [Tra00a].

To address the PC to FPGA bandwidth issue, a DIMM-based RC platform, *Pilchard*, was developed with the following objectives:

- achieve higher bandwidth and lower latency than the traditional PCI Local Bus or CardBus based RC systems;

- unlike the PCI Local Bus in which complicated handshakings are involved in transactions (bus mastering, direct memory access (DMA) and interrupts), *Pilchard* uses a simple register-based interface which has minimum interface hardware requirements;
- use the Linux operating system, allowing user mode applications to access the FPGA via a character device driver or memory mapping;
- able to take the advantage of Intel Pentium write-combining (WC) feature via programming the Memory Type Range Registers (MTRR) [Int00b].

It is envisaged that the *Pilchard* system can provide a better platform to facilitate the FPGA-based coprocessor approach, as it will be shown that the latency and throughput between the CPU and the FPGA was improved comparing with PCI Local Bus based RC platforms.

The development of *Pilchard* was split into hardware and software parts. In the hardware part, a printed circuit board (PCB) was designed which can be populated with a Xilinx Virtex or Virtex-E device and can be attached to a memory slot. In the software part, a driver (written as a Linux kernel module) which supports both ordinary 32-bit and Intel Multimedia Extension (MMX) [Int00c] 64-bit data transfers was developed.

5.2 *Pilchard* Hardware

The system level design of *Pilchard* is depicted in Figure 5.1. The main components are the FPGA device, a configuration programmable read-only memory (PROM) interface, a design bitstream download and debug interface, and an expansion header which is used for connection to a logic analyzer or interfacing

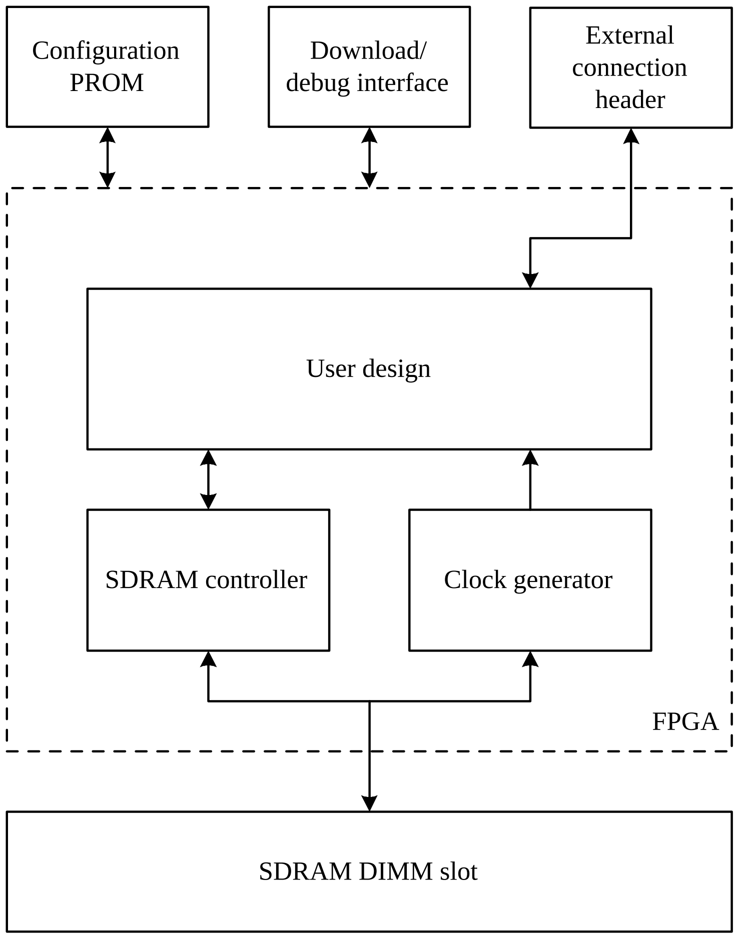


Figure 5.1: System level design of *Pilchard*.

to other peripheral or memory devices. The logic for the DIMM interface and clock generation is implemented in the FPGA. The board was designed for Xilinx Virtex and Virtex-E series FPGAs. A photograph of the populated *Pilchard* board is shown in Figure 5.2.

Pilchard was designed to be compatible with the 168-pin 3.3 volt, 133 MHz, 72-bit, registered SDRAM DIMM PC133 standard. The PC100 (100 MHz) and PC66 (66 MHz) standards can also be supported because the pinouts are the

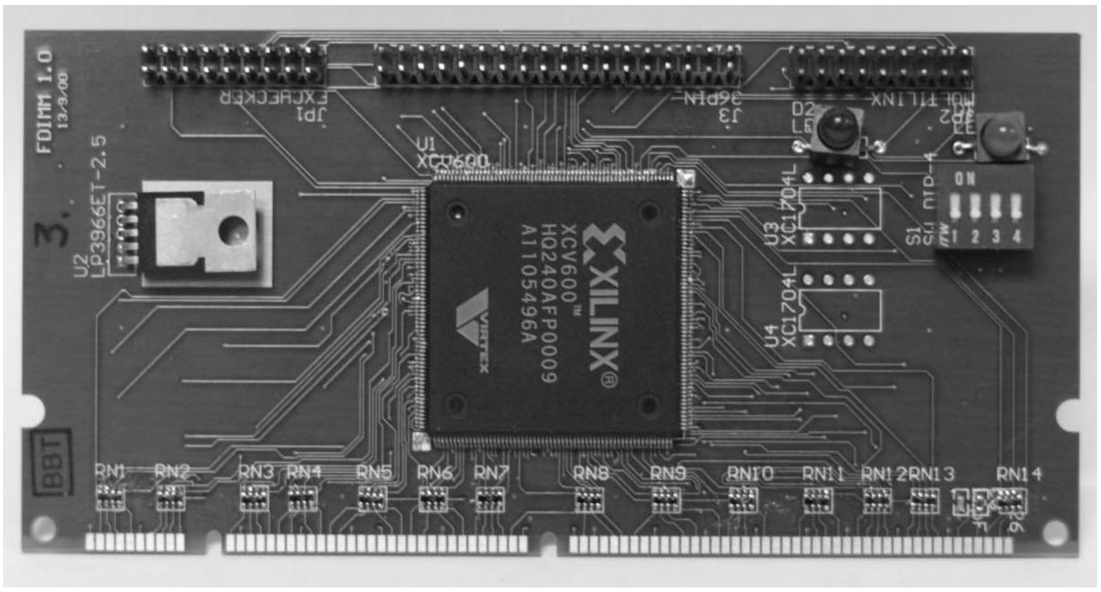


Figure 5.2: A photograph of the *Pilchard* board.

same. The PC133 standard supports up to 1 Gbyte capacity which offers an ample address space for memory mapped I/O [Int98d, IBM98].

The PC133 and PC100 DIMM standards include a mandatory serial presence detect (SPD) interface which allows a DIMM card to describe its configuration to the PC [JED]. The basic input/output system (BIOS) of a PC interrogates the SPD from each DIMM to determine the presence or absence of a card, its timing parameters and its memory size. It then performs a memory test on all available memory before proceeding with the rest of the boot process. This memory test provides an obstacle for a non-memory card, particularly if the card does not have sufficient memory on board to completely mimic a normal DIMM. A possible solution would be to modify the BIOS so that it does not perform a memory test, but this is very difficult in practice when the source code of the BIOS is not available. Furthermore, this method is motherboard and BIOS dependent, hence different machines would need to be patched differently.

The *Pilchard* board does not have a SPD and all the pins are in a high-impedance state during boot up. Therefore, the BIOS identifies the slot as an empty slot and does not perform a memory test on the corresponding DIMM slot. Once the operating system has booted, the chipset registers of the PC are modified to enable operation of the DIMM slot. Although this technique is also dependent on the PC's chipset, it can be easily done for any chipset which has adequate documentation of its memory controller registers.

In the context of a memory mapped device, only read and write commands are needed and the remaining commands sent from the motherboard to the DIMM [Mic99] can be ignored. The interface to a user's core design consists of two parts, namely the PC/*Pilchard* interface and the *Pilchard*/PC interface. The architecture of the interface is illustrated in Figure 5.3. A SDRAM multiplexes its address inputs to save pins and hence addresses are decomposed into banks, rows and columns. Although a typical SDRAM has megabytes of address space, memory mapped peripherals normally use few registers. Hence in the present design, the interface interprets only the 8-bit column addresses and ignores the row and bank select commands. The control uses four signals from the DIMM interface, namely S (Select), RAS (Row Address Strobe), CAS (Column Address Strobe) and WE (Write Enable), to generate the appropriate board $read$ and $write$ signals. Both the interfaces can be independently configured as registers, a BlockRAM or a direct connection. The register configuration supports up to $2^8 = 256$ 64-bit registers. The BlockRAM configuration uses two 256×32-bit dual-port BlockRAMs organized in 256×64-bit. With dual-port BlockRAMs, the internal core can operate at a different clock rate than the 133 MHz DIMM interface. The direct connection configuration is a simple bypass between the input and the output of the interface. The advantage of the register configuration is that the core has simultaneous access to all memory locations. The advantage

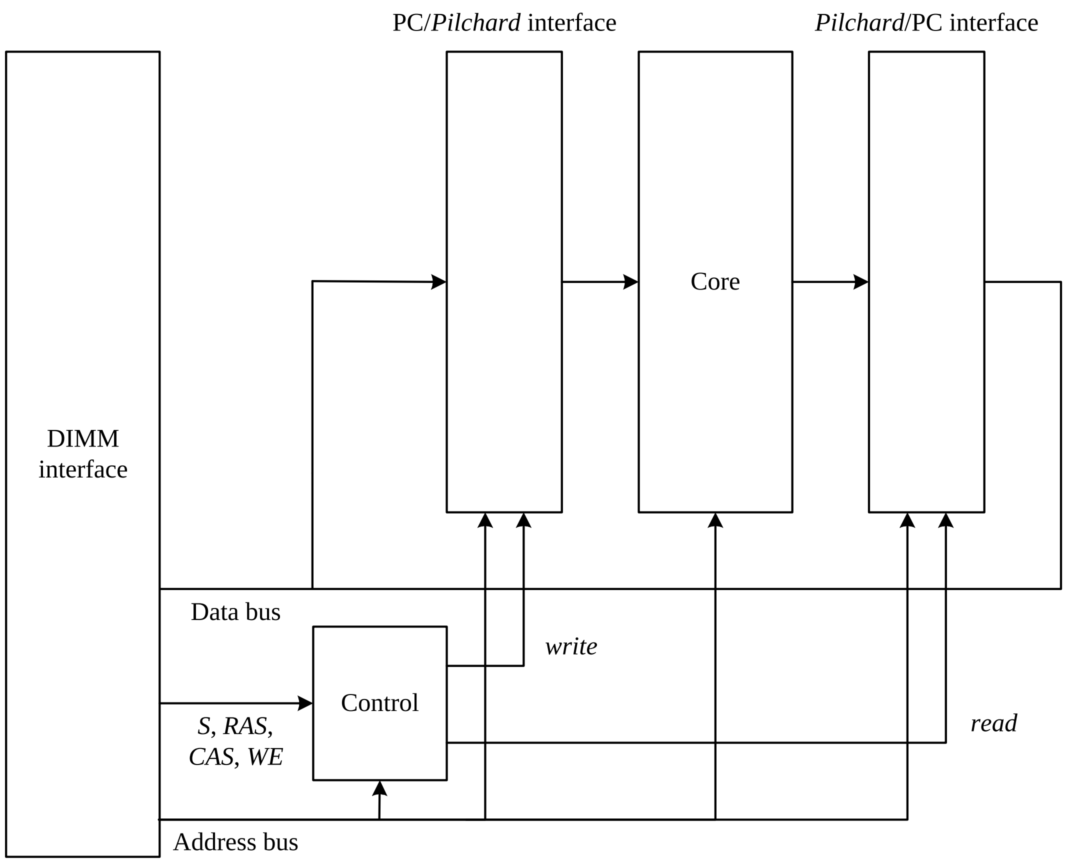


Figure 5.3: Architecture of the PC-FPGA interface.

of the BlockRAM configuration is its reduced area overhead. The direct connection configuration requires minimal area (for example, only two Virtex slices are required if both interfaces are configured as direct connections), but it requires the core to decode the address bus by itself.

The only clock input to the FPGA is that supplied by the SDRAM interface. This 133 MHz clock is de-skewed using a high frequency delay-locked loop (DLL) (Xilinx Virtex CLKDLLHF primitive) within the Virtex chip [Xil00d, Xil00c]. It can also be divided down inside the FPGA and multiplied by another DLL to generate different frequencies. For designs which cannot meet a 133 MHz interface timing constraint, it is possible to use the *Pilchard* board at 100 MHz via setting dual-inline package (DIP) switches or BIOS of the motherboard. Unfortunately, this also limits the speed of all memory accesses to 100 MHz.

The *Pilchard* system currently requires connection from another machine via a Xilinx Xchecker or Multilynx cable [Xil00c] for bitstream download. Readback and single stepping via the Multilynx cable are also supported. Optional PROMs are also supported so that it is possible that the FPGA is configured automatically upon power up. It is currently not possible to download a bitstream to the *Pilchard* system via the DIMM interface, but intuitively this is achievable by decoding signals on the DIMM interface to the SelectMAP configuration data pins with external logic [Xil00c].

For debugging and expansion purposes, an expansion header connected to some general purpose I/O pins of the FPGA is provided for the connections to a logic analyzer or daughter boards.

5.3 *Pilchard* Software

The software components of *Pilchard* consists of a device driver and an application programming interface (API). The device driver configures chipset registers to enable the DIMM slot where *Pilchard* resides to be accessible, and modify the cache control to that specific physical memory range. The API performs a memory mapping upon initialization, and offers read and write function calls for data transfers between the PC and *Pilchard*.

The *Pilchard* device driver was developed on Linux kernel 2.2.17 and has been tested with kernel versions up to 2.2.19. Porting of this driver to Linux kernel 2.4

is in progress. Although the memory management subsystem of the kernel 2.4 was almost rewritten, the code relevant to the *Pilchard* driver remains unchanged. Ports of this driver to other operating systems, although not yet tested, should be possible.

The device driver was written as a loadable kernel module and does not require any modifications to the kernel source. However, the kernel must be compiled with support for loadable kernel modules, a feature in all Linux version 2.2 and succeeding kernel versions. All accesses to the driver and hence the *Pilchard* hardware are via a character device with major and minor numbers 240 and 0 respectively.

During initialization, the device driver is responsible for programming the PC chipset's memory controller registers to enable the DIMM slot where *Pilchard* is populated. This is achieved using the PCI Utilities package from Martin Mares [Mar00] (the host bridge device (bus 0, slot 0, function 0) contains the configuration of memory slot population). Re-programming the chipset registers fools the motherboard into thinking that the slot is populated with a DIMM memory card and access cycles directed to this portion of the memory space will generate appropriate signals in the DIMM slot. This operation is chipset dependent, because the address of the registers which specifies how the DIMM slots are populated and how these registers are encoded varies from chipset to chipset. The chipsets that have been tested include the Intel 440LX [Int98c], Intel 440BX [Int98b] and Intel 815EP [Int01a] chipsets.

For instance, suppose a PC has 128 MB RAM organized as two 64 MB double-sided DIMM card. Upon booting of the operating system, the chipset registers would encode this physical memory configuration in specific registers. The device driver modifies the chipset register, specifying that a third DIMM card, such as a single-sided 64 MB DIMM card, is installed. Afterwards, the driver initializes the base address of the *Pilchard* memory mapped space, *pilbase*. This address is exactly the top physical memory address, which is 0x8000000 (128 MB) in this example. Subsequent memory accesses to addresses 0x8000000 to 0xBFFFFFF (a 64 MB memory range) select the *Pilchard* board.

CPU caching of reads and writes to *Pilchard* registers could lead to incorrect results. The Intel Pentium Pro, Pentium II and Pentium III has a Memory Type Range Register (MTRR) which allows different memory regions to have different caching behavior [Int00b]. The MTRR registers are directly accessible from the Linux kernel via the `/proc` filesystem [Goo99]. The four modes of MTRR are uncachable (UC), write-through (WT), write-back (WB) and write-combining (WC). Among these four modes, UC and WC are used. WT and WB modes offer potentially higher transfer rates but are more difficult to control because the CPU manipulates the caching internally.

The UC memory type guarantees that all reads and writes will appear on the system bus in the same order as the program. Furthermore, no speculative memory accesses, page-table talks or prefetches of speculated branch targets will occur [Int00b]. Although the most conservative, it also leads to the lowest performance.

The WC memory type allows 32-bit writes to be delayed and later merged together in write-combining buffers. It is typically used to improve the performance of frame buffers for graphics. Upon reaching a serializing event such as a read from an UC location, the write-combining buffer is flushed in an efficient manner. For example, when a WC buffer becomes full, the processor will evict the buffer to system memory in a single burst transaction of 64-bit writes [Int00b]. Careful use of the WC memory type can lead to greatly improved performance (discussed in Section 5.4). WC was not used on *Pilchard* for reads since this mode allows speculative reads which could interfere with memory mapped read cycles that have side effects.

Therefore, in normal operations reads are made on UC locations whereas writes are made on UC or WC locations. This was achieved by setting different MTRRs mode for different address regions. To support WC writes, a design on *Pilchard* must be either insensitive to out-of-order data transfers or has a re-ordering buffer. For the latter case, the size of the buffer should be at least that of the WC buffer on the processor.

The use of Pentium III Streaming SIMD Extension (SSE) may further improve the transfer throughput. The execution pattern of SIMD instructions is similar to vector machines, in which the same operation is applied on a relatively large range of consecutive memory locations (SIMD instructions are typically applied to several hundreds kilobytes of data) [Int00a]. To enable this execution pattern, memory accesses must be made in burst mode such that data retrieved from the memory bus is directly fed to the SSE pipeline. Use of these instructions may provide high bandwidth bursting transfers without corrupting the cache. However, this has not yet been tested.

The user interface of *Pilchard* uses the UNIX `mmap()` system call. User programs can access the registers on the *Pilchard* board by performing a `mmap()` call which maps virtual addresses in the user space to the bus address of the *Pilchard* board. Following this process, the user can manipulate the registers of the *Pilchard* board directly without incurring the overhead of a system call. For the user program to perform data transfer between the CPU and *Pilchard*, four function calls are provided:

- `write32(d, a)`: Write a 32-bit word referenced by the pointer d to the physical memory address a. This function call fails if a is not in the memory range mapped by the *Pilchard* driver.

- `read32(d, a)`: Read a 32-bit word from the physical memory address a and put the word into the memory location referenced by the pointer d. This function call fails if a is not in the memory range mapped by the *Pilchard* driver.

- `write64(d, a)`: Write a 64-bit double-word pointed to by the pointer d to the physical memory address a. This function call fails if a is not in the memory range mapped by the *Pilchard* driver. To achieve this, MMX instructions are used.

- `read64(d, a)`: Read a 64-bit double-word from the physical memory address a and put the word into the memory location referenced by the pointer d. This function call fails if a is not in the memory range mapped by the *Pilchard* driver. Similar to the 64-bit write function call, MMX instructions are used.

Note that the mode of memory caching used in the function calls depends on the MTRR mode of the address concerned. If the MTRR modes of 1 MB memory ranges beginning from addresses *pilbase* and *pilbase*+0x100000 are UC and WC respectively, then a memory access to address b, $\mathit{pilbase} \leq b < \mathit{pilbase}+0\text{x}100000$ would be made UC, whereas that to address c, $\mathit{pilbase}+0\text{x}100000 \leq c < \mathit{pilbase}+0\text{x}200000$ would be made WC.

The MMX instruction used to perform 64-bit data transfer is the `movq` instruction. To perform a 64-bit write, a `movl` instruction is first issued to copy the 32-bit address to a 32-bit general purpose register (GPR), followed by a `movq` instruction that copies the double-word to a 64-bit MMX register. Finally, another `movq` instruction is issued to copy the double-word from the MMX register to the memory location. Similarly, to perform a 64-bit read, a `movl` instruction is first issued. Then, two `movq` instructions are issued to copy the 64-bit data from the memory location to an MMX register, and then to the user program memory.

5.4 Performance Evaluation

The performance of the *Pilchard* board using a Xilinx XCV300-6 FPGA was compared with a PCI Local Bus (33 MHz, 32-bit) card (a Nallatech Ballyinx card [Nal]) also using an XCV300-6 device. The PCI card used the Xilinx LogiCORE PCI 64 Interface [Xil01b] for its PCI Local Bus controller. All the experiments presented in this section were measured on the same machine, an Asus CUSL2 motherboard (Intel 815EP chipset) with 800 MHz Pentium III processor and 32-bit PCI Local Bus slots (the PCI64 card was used in a 32-bit slot in backwards compatible mode). All tests were conducted with the DIMM slot operating at 133 MHz.

Unfortunately, since the Linux driver for the PCI card does not support direct memory access (DMA), its performance in this mode could not be tested. DMA would certainly offer better performance for large blocks, but there are large overheads associated with setting up DMA transactions and leads to very long latencies.

The size of the interface of *Pilchard* is much smaller than that of the Xilinx LogiCORE PCI 64 Interface. A minimal 64-bit *Pilchard* interface requires only two Virtex slices to generate the board's *read* and *write* signals. Latching of the address and data buses are performed in the input-output blocks (IOBs). Registers and BlockRAMs used for interfacing purposes require additional resources. In contrast, the LogiCORE interface uses 300 to 350 slices [Xil01b].

As for *Pilchard*, the PCI card also uses a Linux loadable kernel module device driver to ensure that the best performance was achieved. All the measurements were performed using the Linux kernel function `do_gettimeofday()` and the results reflect associated software overheads.

A simple design was used to measure the I/O performance of the *Pilchard* board. In this design, read and write cycles cause the lower and upper 32 bits of a 64-bit register on the *Pilchard* board to be incremented. Upon completion of the benchmark, the number of reads and writes are read back to the PC to verify that all the data were transferred. The results, measured in Mbytes/sec, are presented in Table 5.1.

5.4.1 Write Benchmark

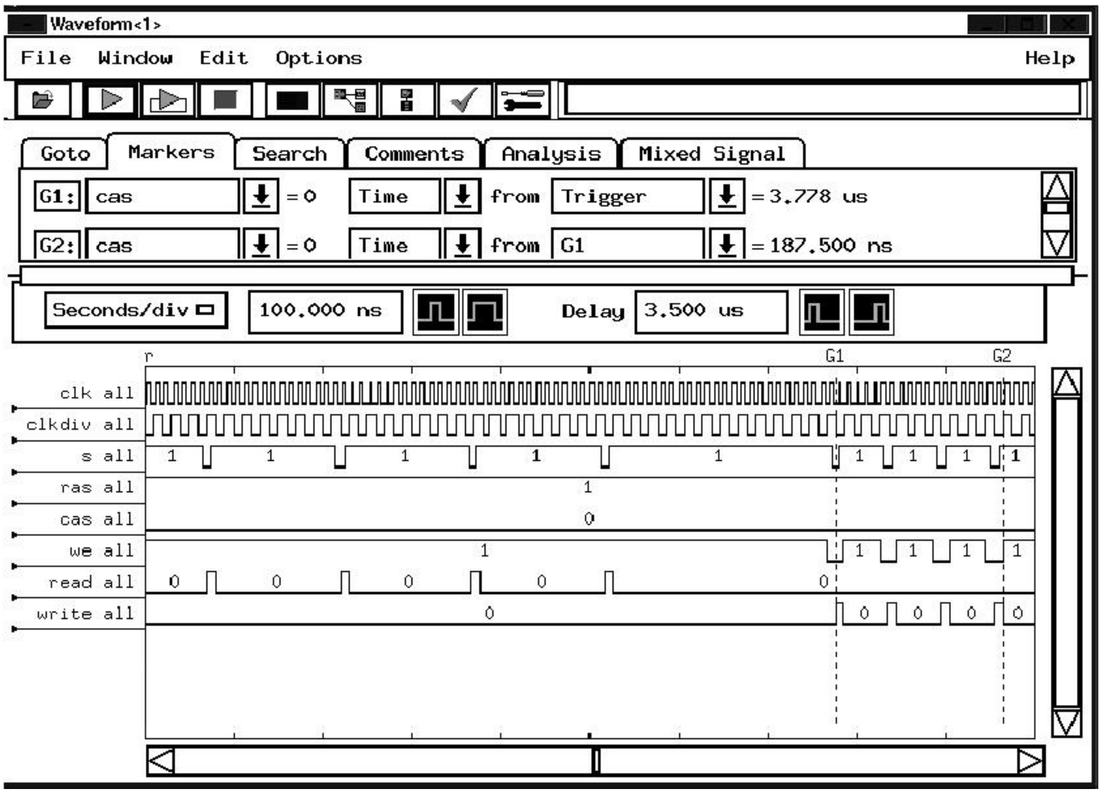
The write benchmark was conducted by performing $2^{20} = 1048576$ 32-bit writes to blocks of consecutive memory locations on the respective cards. This test was conducted with MTRRs set to WC and UC on the *Pilchard* board, and UC for the PCI Local Bus.

Measurements of write throughput for different 32-bit block sizes are presented in the first two rows in Table 5.1 and are plotted in Figure 5.6. The PCI Local Bus interface (the sixth row) is always slower than the *Pilchard* interface with UC MTRR, particularly for small block sizes. WC on *Pilchard* gives a further three to four-fold performance gain over UC since it is able to combine software 32-bit cycles and write them using 64-bit transfers. As can be seen from the rightmost entries of the two rows and Figure 5.6, write performance can be further improved using the 64-bit write transactions by the MMX `movq` instruction.

An Agilent Technologies 16700A logic analyzer was used to capture the waveforms for UC and WC writes to the *Pilchard* board. The captures are shown in Figures 5.4 and 5.5 respectively. In the UC case shown in Figure 5.4, all writes must occur immediately and no write bursts will occur. The highest performance

Benchmark[†]	Block size (words)							
	1	2	4	8	16	32	64	64-bit[‡]
Pil/UC-WR	72.80	78.22	81.51	81.47	81.46	81.49	81.49	132.88
Pil/WC-WR	69.83	150.02	297.53	298.28	297.93	298.08	297.80	409.64
Pil/RD	25.29	28.93	31.67	32.65	32.98	33.28	32.76	52.80
Pil/UC-RW	29.47	35.95	42.43	42.42	42.41	42.42	42.42	74.79
Pil/WC-RW	24.34	36.18	46.11	46.11	46.11	46.11	46.11	120.78
PCI/WR	25.47	42.45	61.74	62.68	63.17	63.42	63.17	N/A
PCI/RD	6.37	6.53	6.61	6.66	6.67	6.67	6.65	N/A
PCI/RW	9.62	10.07	10.84	11.71	11.90	11.99	12.01	N/A

Table 5.1: Measured performance (Mbytes/sec) of *Pilchard* and a comparison with the PCI Local Bus interface. ([†]The benchmarks include Pil/UC-WR: *Pilchard* write performance with MTRR set to UC, Pil/WC-WR: *Pilchard* write performance with MTRR set to WC, Pil/RD: *Pilchard* read performance, Pil/UC-RW: *Pilchard* read/write performance with MTRR set to UC, Pil/WC-RW: *Pilchard* read/write performance with MTRR set to WC, PCI/WR: PCI Local Bus write performance, PCI/RD: PCI Local Bus read performance, PCI/RW: PCI Local Bus read/write performance. [‡]The performance of 64-bit data transfer for any block size is the same.)

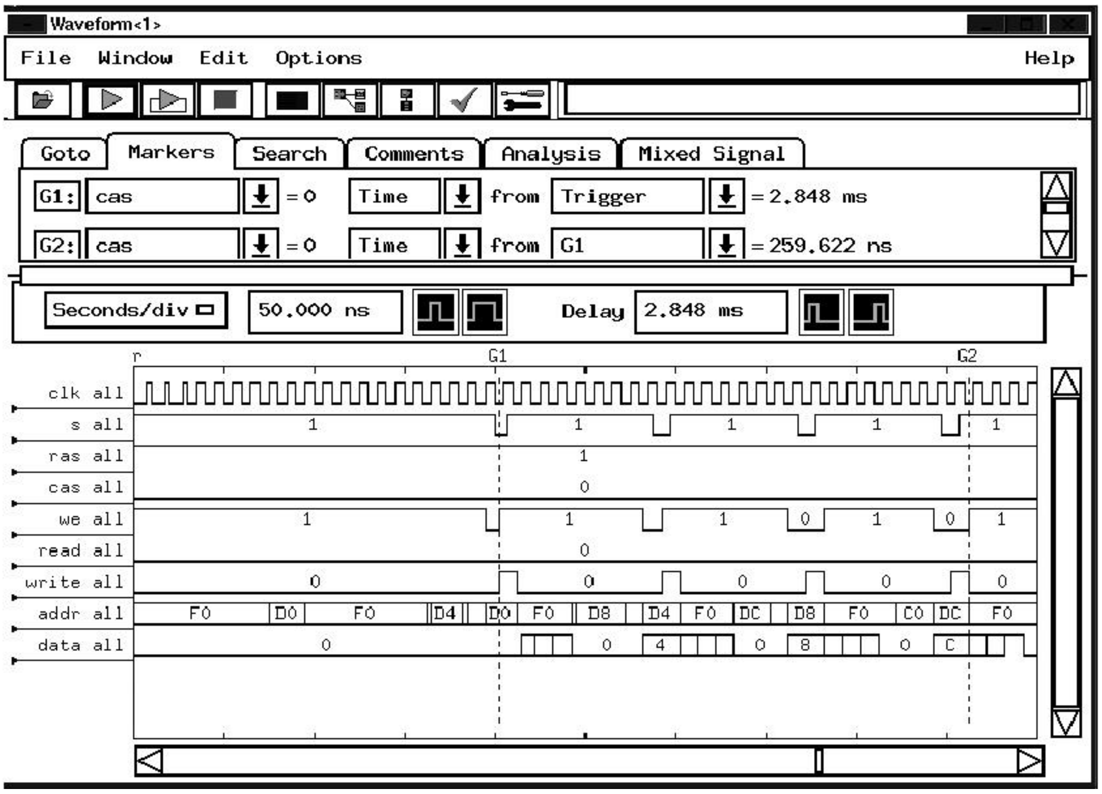


G1: cas = 0 Time from Trigger = 3.778 us

G2: cas = 0 Time from G1 = 187.500 ns

Seconds/div 100.000 ns Delay 3.500 us

Figure 5.4: Logic analyzer trace showing 64-bit *Pilchard* read and write cycles to UC memory regions using the `movq` instruction. *clk* is a 133 MHz clock, *s*, *ras*, *cas* and *we* are the DIMM interface signals, and *read* and *write* are the *Pilchard* decoded read and write signals. This trace shows four 64-bit reads followed by four 64-bit writes.


Waveform<1>

G1: cas = 0 Time from Trigger = 2.848 ms

G2: cas = 0 Time from G1 = 259.622 ns

Seconds/div 50.000 ns

Delay 2.848 ms

File Window Edit Options Help
Goto Markers Search Comments Analysis Mixed Signal
clk all
s all
ras all
cas all
we all
read all
write all
addr all
data all
G1
G2

Figure 5.5: Logic analyzer trace showing bursting behavior for 64-bit write cycles to consecutive addresses in a WC memory region. *clk* is a 133 MHz clock, *s*, *ras*, *cas* and *we* are the DIMM interface signals, *read* and *write* are the *Pilchard* decoded read and write signals; and *addr* and *data* show the *Pilchard* address and data buses. This trace shows sixteen 64-bit writes which have been merged in the write-combining buffer into four burst transactions.

is achieved in the WC case of Figure 5.5 where the start of a burst transfer can be identified by the *write* signal being high. In this particular example, it can be seen that 16×64-bit writes are performed in approximately 300 ns which equates to 426 Mbytes/sec, consistent with the measured results in Table 5.1.

The measured WC performance is six times that of the measured PCI Local Bus transfer rate. The theoretical maximum bandwidth of a DIMM interface is 1064 Mbytes/sec and the best measured performance using WC and including software overheads was 400 Mbytes/sec (for `movq` transfers). This performance is more than three times the maximum transfer rate of the PCI Local Bus.

5.4.2 Read Benchmark

Similar to the write benchmark, the time taken to perform $2^{20} = 1048576$ reads of UC memory locations in differently sized blocks of consecutive locations was measured. The read performances for different block sizes are shown in the third row in Table 5.1 and are plotted in Figure 5.7.

Four 64-bit read cycles can be seen in Figure 5.4 when the *read* signal is high, and the throughput can be seen to be approximately 64 Mbytes/sec which is consistent with a measured value of 52 Mbytes/sec in Table 5.1. It is still unable to produce burst 64-bit read transactions, but the use of SSE instructions may enable this feature. The read performance is approximately five times higher than that of the PCI Local Bus (the seventh row).

Using 64-bit data transfers via `movq` instructions may further improves the read performance to about eight times that of the PCI Local Bus. This can be seen by comparing the rightmost entry of the third row with the seventh row in Table 5.1.

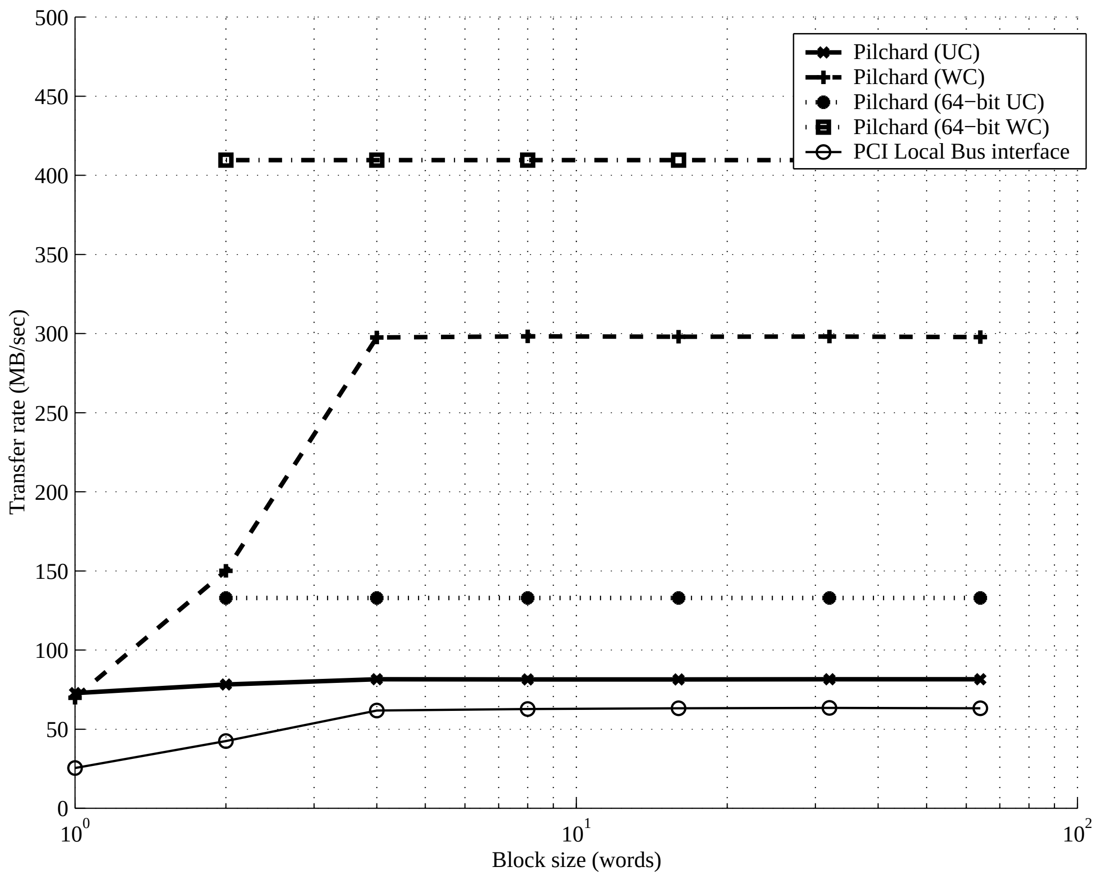


Figure 5.6: Write performance of *Pilchard* and the PCI Local Bus interface for different block sizes.

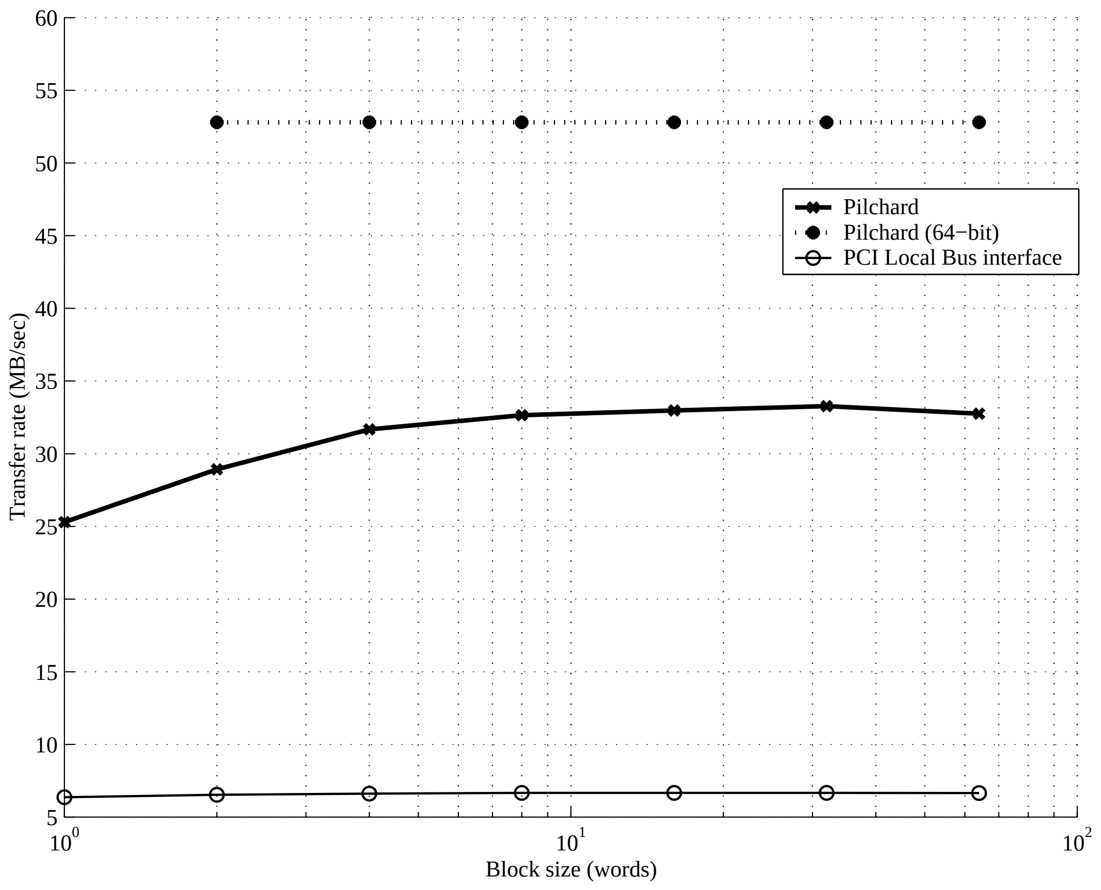


Figure 5.7: Read performance of *Pilchard* and the PCI Local Bus interface for different block sizes.

5.4.3 Read/Write Benchmark

The read/write benchmark involves alternating writes and reads of data. Two separate memory regions were used for this test, reads being made on an UC region and writes to a WC or UC region. Results are shown in the fourth and the fifth rows of Table 5.1 and the corresponding plots are plotted in Figure 5.8. This mode is even faster than pure read cycles since writes are faster than reads, thus improving the overall transfer rate.

As for read and write cycles, using the `movq` instruction to achieve 64-bit transfers significantly improves the performance and the results are presented in the rightmost entries of the two rows. The 64-bit *Pilchard* read/write transfer rates was approximately six to ten times faster than the PCI Local Bus (the eighth row).

5.5 Summary

The PC/FPGA interface is a major bottleneck for current RC platforms. A solution to this problem was developed in the form of the *Pilchard* system which demonstrates the feasibility of utilizing the DIMM slots of a standard PC as a bus for attaching a RC platform. This was shown to offer greatly improved bandwidth and latency over the ubiquitous PCI Local Bus. The *Pilchard* system is simpler, uses less resources than conventional systems and may enable the development of reconfigurable systems with lower cost and significantly improved performance. It may be possible to achieve even higher performance via the Pentium SSE instructions. These two features are crucial in an FPGA-based coprocessor system.

As will be presented in Chapters 6 and 8, an implementation of the Interna-

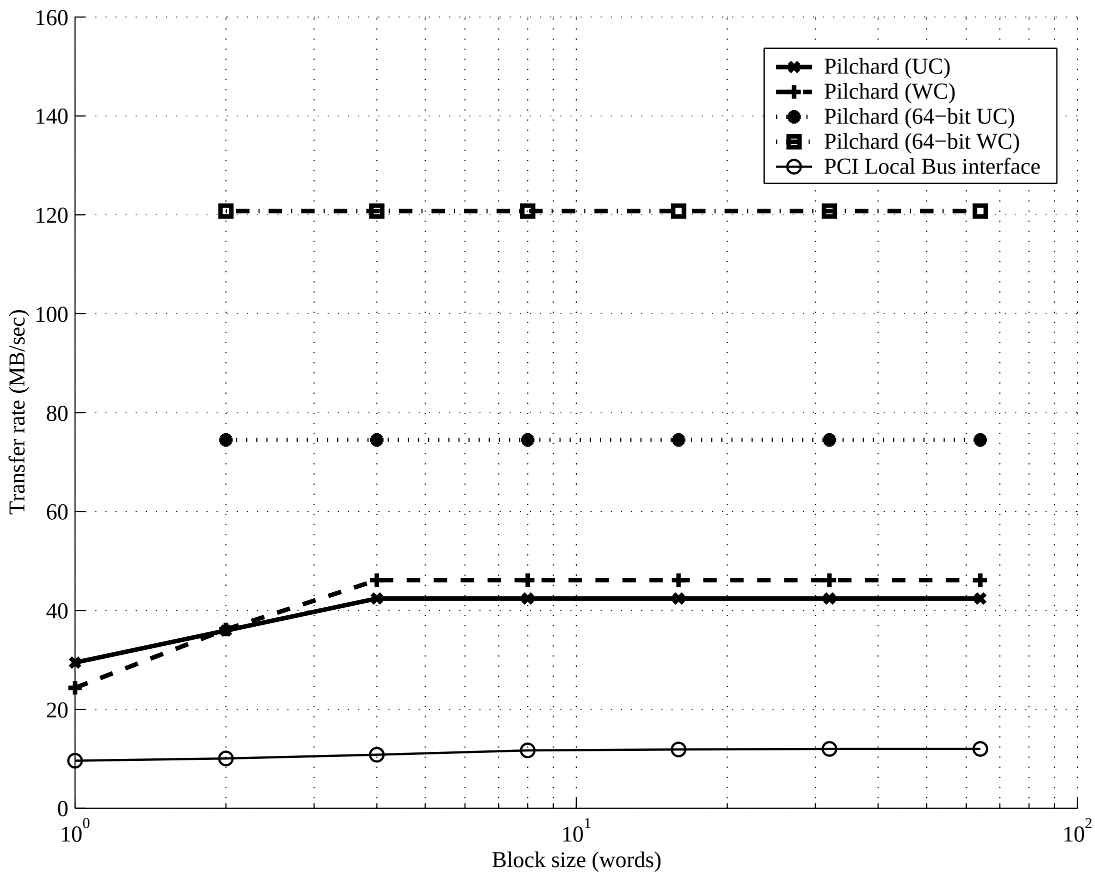


Figure 5.8: Read/write performance of *Pilchard* and the PCI Local Bus interface for different block sizes.

tional Data Encryption Algorithm (IDEA) cipher and an electronic cochlea on *Pilchard* achieve significantly faster throughput than on the PCI or PCI64 Local Bus. These applications demonstrated the use of the memory bus interface as a solution to the bottleneck problem which exists in current RC platforms.

Chapter 6

Application I – The IDEA Cipher

A high-performance implementation of the International Data Encryption Algorithm (IDEA) is presented in this chapter. Using a bit-serial architecture to perform modular multiplication, this implementation occupies a minimal amount of hardware and has a high clock rate. The key schedule of this implementation can be modified through bitstream modification (Appendix B) and does not require the time-consuming place and route (P&R) procedure. The implementation was developed on the *Pilchard* platform (Chapter 5) and results showed significant improvement in encryption/decryption throughput over the PCI Local Bus.

This chapter gives an overview of cryptography followed by a description of the IDEA. Implementations and results are then given. Specifically, the method of modifying the key schedule by bitstream modification and the performance on *Pilchard* are highlighted.

6.1 Background

Cryptography is concerned with the transfer of information between parties so that only the intended parties can read the data. Despite an assumption that an adversary may have full knowledge of the algorithms used and has access to the media where data is transmitted, it is desired that the retrieval of data without knowledge of a secret piece of information called a key is intractable.

The Data Encryption Standard (DES) algorithm has been a popular secret key encryption algorithm and is used in many commercial and financial applications. Although introduced in 1976, it has proved resistant to all forms of cryptanalysis. However, its key size is too small by current standards and its entire 56-bit key space can be searched in approximately twenty-two hours [Ele99].

In 1990, Lai and Massay introduced an iterated block cipher (the cryptographic key and algorithm are applied to a block of data at once as a group of bits in a block cipher) known as Proposed Encryption Standard (PES) [LM90]. Later, the same authors, joined by Murphy, proposed a modification of PES called Improved PES (IPES) [LMM91], which improves the security of the original algorithm against differential analysis and truncated differentials [HL94, Knu95, Bor97]. In 1992, IPES was commercialized and was renamed the International Data Encryption Algorithm. Some believe that, to date, the algorithm is the best and the most secure block cipher available to the public [Sch96].

Although IDEA involves only simple 16-bit operations, software implementations of this algorithm still cannot offer the encryption rate required for on-line encryption in high-speed networks. Ascom's implementation of IDEA [Asc99a] (Ascom hold the patent on the IDEA) achieves 0.37×10^6 encryptions per seconds, or an equivalent encryption rate of 23.53 Mbits/sec, on an Intel Pentium II 450 MHz machine. Implementation of IDEA using the Intel Multimedia Extension (MMX) instructions was proposed by Lipmaa [Lip98] achieves 0.51×10^6 encryptions per second or a equivalent encryption rate of 32.64 Mbits/sec on an Intel Pentium II 233 MHz machine. A parallelized software implementation derived from Ascom's implementation running on a Sun Enterprise E4500 machine with twelve 400 MHz Ultra-IIi processor, performs 2.30×10^6 encryptions per second or a equivalent encryption rate of 147.13 Mbits/sec, still cannot be applied to applications such as encryption for 155 Mbits/sec asynchronous transfer Mode (ATM) networks.

Hardware implementations offer significant speed improvements over software implementations by exploiting parallelism among operators. In addition, they are likely to be cheaper, have lower power consumption and smaller footprint than a high speed software implementation. A paper design of an IDEA processor which achieves 528 Mbits/sec on four XC4020XL devices was proposed by Mencer et. al. [MMF98]. The first VLSI implementation of IDEA was developed and verified by Bonnenberg et. al. in 1992 using a 1.5 μm CMOS technology [BCF+91]. This implementation had an encryption rate of 44 Mbits/sec. In 1994, VINCI, a 177 Mbits/sec VLSI implementation of IDEA in 1.2 μm CMOS technology, was reported by Curiger et. al. [CBZ+93, ZCB+94]. A 355 Mbits/sec implementation in 0.8 μm technology of IDEA was reported in 1995 by Wolter et. al. [WMSL95], and later a 424 Mbits/sec implementation in 0.7 μm technology was proposed by Salomao et. al. [SAF98]. Goldstein et. al. reported an implementation on the PipeRench FPGA which achieves 1013 Mbits/sec [GSB+00]. A commercial implementation of IDEA called the IDEACrypt Kernel developed by Ascom achieves 720 Mbits/sec at 0.25 μm technology [Asc99c]. The single chip device derived from the IDEACrypt Kernel, called the IDEACrypt Coprocessor, has a throughput of 300 Mbits/sec [Asc99b].

6.2 Algorithm

IDEA belongs to a class of cryptosystems called secret-key cryptosystems which is characterized by the symmetry of encryption and decryption processes, and the possibility of implying the decryption and encryption keys from each other. IDEA takes 64-bit plaintext inputs and produces 64-bit ciphertext outputs using a 128-bit key.

The IDEA block cipher, as depicted in Figure 6.1, consists of eight identical blocks known as rounds, followed by a half-round or output transformation. In each round, exclusive-or (XOR), addition modulo 2^{16} and multiplication modulo $2^{16} + 1$ (a Fermat prime) operations are applied on 16-bit sub-blocks. The IDEA algorithm is believed to be of strong cryptographic strength because its primitive operations are of three distinct algebraic groups of 2^{16} elements, multiplication modulo a Fermat prime provides desirable statistical independence between plaintext and ciphertext, and its property of having iterative rounds made differential attacks difficult.

The encryption process is as follows. The 64-bit plaintext is divided into four 16-bit plaintext sub-blocks, X_1 to X_4. The algorithm converts the plaintext blocks into ciphertext blocks of the same bit-length, similarly divided into four 16-bit sub-blocks, Y_1 to Y_4. 52 16-bit subkeys, $Z_i^{(r)}$, where i and r are the subkey number and round number respectively, are computed from the 128-bit secret key. Each round uses six subkeys and the remaining four subkeys are used in the output transformation. The decryption process is essentially the same as the encryption process except that the subkeys are derived using a different algorithm [Sch96].

The algorithm for computing the encryption subkeys (called the key schedule) involves only logical rotations. Order the 52 subkeys as $Z_1^{(1)}$, …, $Z_6^{(1)}$, $Z_1^{(2)}$, …, $Z_6^{(2)}$, …, $Z_1^{(8)}$, …, $Z_6^{(8)}$, $Z_1^{(9)}$, …, $Z_4^{(9)}$. The procedure begins with partitioning

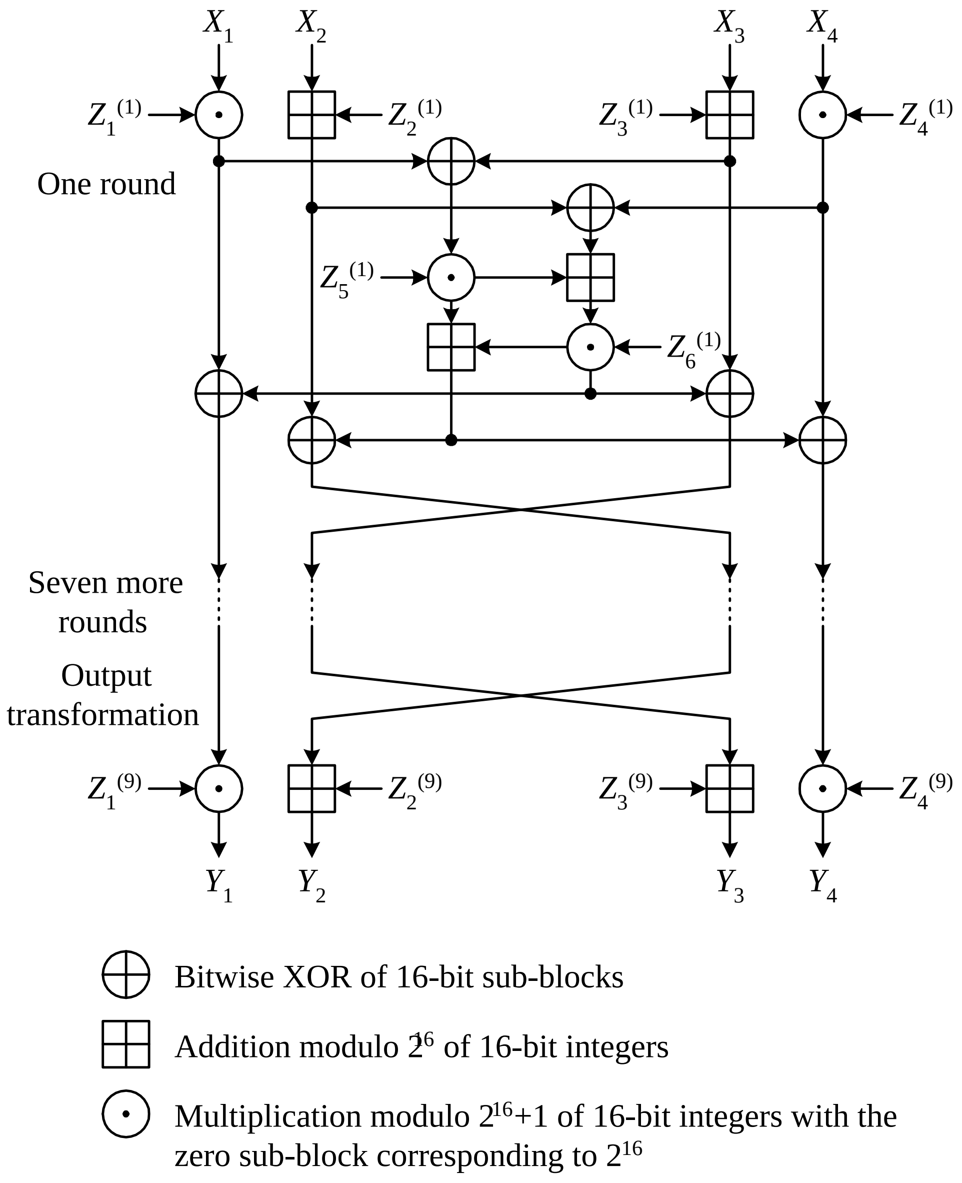


Figure 6.1: Block diagram of the IDEA algorithm.

	$r = 1$	$2 \leq r \leq 81$	$r = 9$
$Z'^{(r)}_1$	$(Z_1^{(10-r)})^{-1}$	$(Z_1^{(10-r)})^{-1}$	$(Z_1^{(10-r)})^{-1}$
$Z'^{(r)}_2$	$-Z_2^{(10-r)}$	$-Z_3^{(10-r)}$	$-Z_2^{(10-r)}$
$Z'^{(r)}_3$	$-Z_3^{(10-r)}$	$-Z_2^{(10-r)}$	$-Z_3^{(10-r)}$
$Z'^{(r)}_4$	$(Z_4^{(10-r)})^{-1}$	$(Z_4^{(10-r)})^{-1}$	$(Z_4^{(10-r)})^{-1}$
$Z'^{(r)}_5$	$Z_5^{(9-r)}$	$Z_5^{(9-r)}$	N/A
$Z'^{(r)}_6$	$Z_6^{(9-r)}$	$Z_6^{(9-r)}$	N/A

Table 6.1: IDEA decryption subkeys $Z'^{(i)}_r$ derived from encryption subkeys $Z_r^{(i)}$, where $-Z_i$ and Z_i^{-1} denote additive inverse modulo 2^{16} and multiplicative inverse $2^{16}+1$ of Z_i respectively.

the 128-key secret key Z into eight 16-bit blocks and assigning them directly to the first eight subkeys. Z is then rotated left by 25 bits, partitioned into eight 16-bit blocks and again assigned to the next eight subkeys. The process continues until all 52 subkeys are assigned. The decryption subkeys $Z'^{(r)}_i$ can be computed from the encryption subkeys with reference to Table 6.1.

In electronic codebook (ECB) mode [Sch96], the data dependencies of the IDEA have no feedback paths. Additionally, latencies of order of microseconds are acceptable in practice. These features make the algorithm suitable to be implemented as a deep bit-serial pipeline.

Of the basic operations used in the IDEA, multiplication modulo $2^{16}+1$ is the most complicated and occupies most of the hardware. Curiger et. al. [CBK91] described and compared several VLSI architectures for multiplication modulo 2^n+1 and found that an architecture proposed by Meier and Zimmerman [MZ91], using modulo 2^n adders with bit-pair recoding offers the best performance. The pseudocode for the modular multiplication operation by module 2^n adders using

```
uint16 mulmod(uint16 x, uint16 y)
{
    uint32 t;
    x = (x - 1) & 0xFFFF;
    y = (y - 1) & 0xFFFF;
    t = (uint32) x * y + x + y + 1;
    x = t & 0xFFFF;
    y = t >> 16;
    x = (x - y) + (x <= y);
    return x;
}
```

Figure 6.2: The pseudocode for the multiplication modulo $2^{16}+1$ using 2^{16} adders bit-pair recoding.

bit-pair recoding is shown in Figure 6.2.

This algorithm requires a total of six additions and subtractions, one 16-bit multiplication and one comparison. Nevertheless, in IDEA one of the operands of a modular multiplication operation is always a subkey, so the second subtraction can be eliminated if the associated subkeys are pre-decremented.

6.3 Implementation

6.3.1 The Bit-Serial IDEA Pipeline

The implementation of IDEA presented in this dissertation was targeted for the Xilinx Virtex FPGA architecture. This design employs a bit-serial architecture which offers a high degree of fine-grain parallelism and high clock rate in a compact implementation. Applications of this design include Virtual Private Networks (VPNs) and embedded encryption/decryption devices.

The bit-serial pipeline that implements a round in IDEA is depicted in Figure 6.3. The half-round can be constructed in a similar manner. A fully-pipelined IDEA implementation can be constructed by cascading eight rounds followed by a half-round.

The implementation of two primitive operators used in IDEA, namely XOR and addition modulo 2^{16} are straightforward. These two operators have latencies of one clock cycle and are capable of taking consecutive bit-serial operands. The multiplication modulo $2^{16}+1$ operator, which is described in the subsequent paragraphs, has a latency of 35 clock cycles. For the best area-efficiency, stage latches and constants are implemented using Virtex shift register LUT (SRL) primitives [Xil00g]. More specifically, a constant is implemented as a SRL16E primitive, with its output connected to its input to form a cyclic shift register.

Multiplication modulo $2^{16}+1$ is the most critical operation in IDEA. Choosing a suitable multiplier is therefore a crucial design issue. An $N \times N$-bit multiplier generates a $2N$-bit result, and requires $2N$ cycles to complete. Thus, throughput of bit-serial multipliers are restricted because the minimum interval between consecutive multiplications must be at least $2 \times N$ cycles. In IDEA, one of the operands of every modular multiplication is a subkey and treated as a constant.

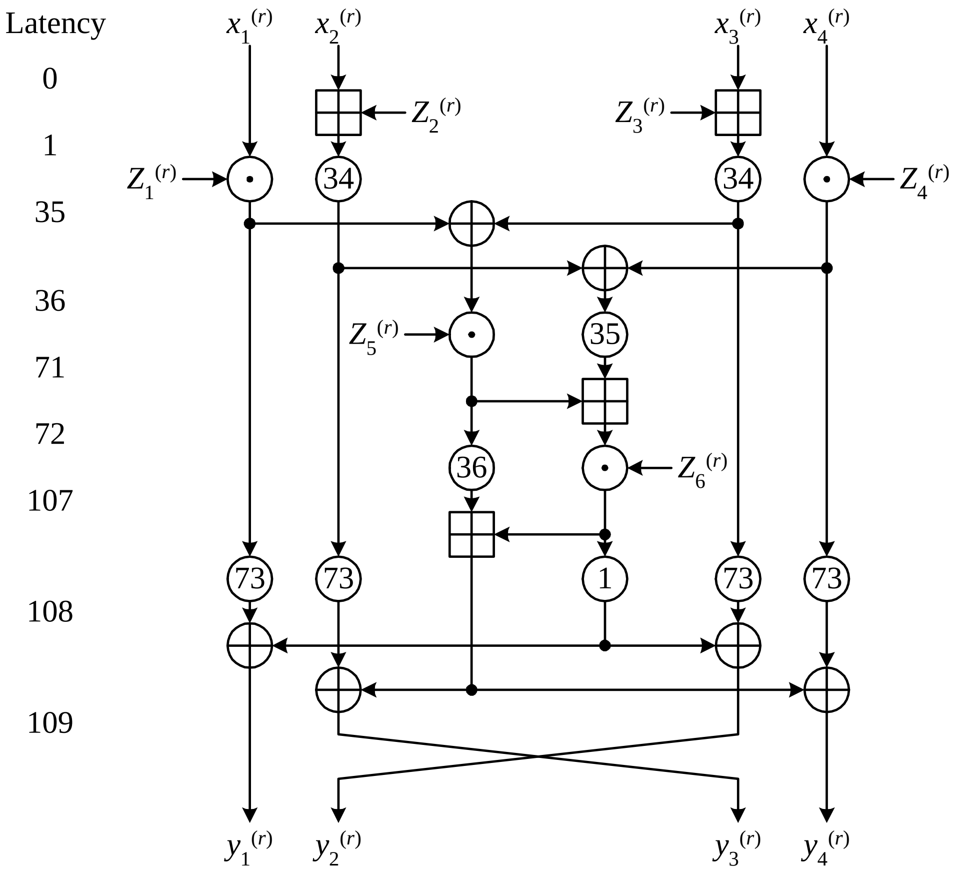


Figure 6.3: Pipelined datapath for one round of IDEA. Circled numbers indicates latches of that number of stages.

Recall in the modular multiplication algorithm that the intermediate result t is divided into two portions (Figure 6.2, lines 6 to 8). The two portions are respectively the upper and lower 16 bits of the double-word, which are operands to subsequent operations. A design that computes the upper and lower words of t independently is desirable, allowing all the inputs, outputs and intermediate variables of the operator to be 16-bit long. Using this scheme and duplicating hardware, the throughput of a modular multiplication operation can be doubled.

To overcome the above mentioned throughput problem, a modified version of Lyon's serial-parallel multiplier [Lyo76] was developed. To generate two 16-bit results in sixteen cycles, the throughput of the multiplier must be doubled. This is achieved by duplicating the hardware for multiplication, as illustrated in Figure 6.4. Registers storing the constant are shared among the two multiplication pipelines. The outputs p and q correspond to the results of two consecutive multiplications, where the two 32-bit long variables have a time-difference of 16 cycles. The *ctrl* signal, which is high one clock cycle before the least significant bit enters the module, toggles the control register. The vector of input variables $a_{n-1} \ldots a_1 a_0$ is consequently redirected into the two multiplication pipelines alternately. While the vector is being redirected to one pipeline, logic zero enters the other pipeline carrying out zero-padding. A timing diagram of the modified multiplier is shown in Figure 6.5.

To obtain the time-aligned upper and lower words of t, a sixteen-stage shift register is required. The input and output of the shift register are the upper and lower words of t respectively, sixteen cycles after t is valid. In the implementation the shift register is implemented as a SRL16E primitive [Xil00g]. The complete architecture for the modular multiplication operation is shown in Figure 6.6. Upon initialization, the subkey associated with the operator is passed into the operator bit-serially. The pre-decremented subkey is shifted into the registers of

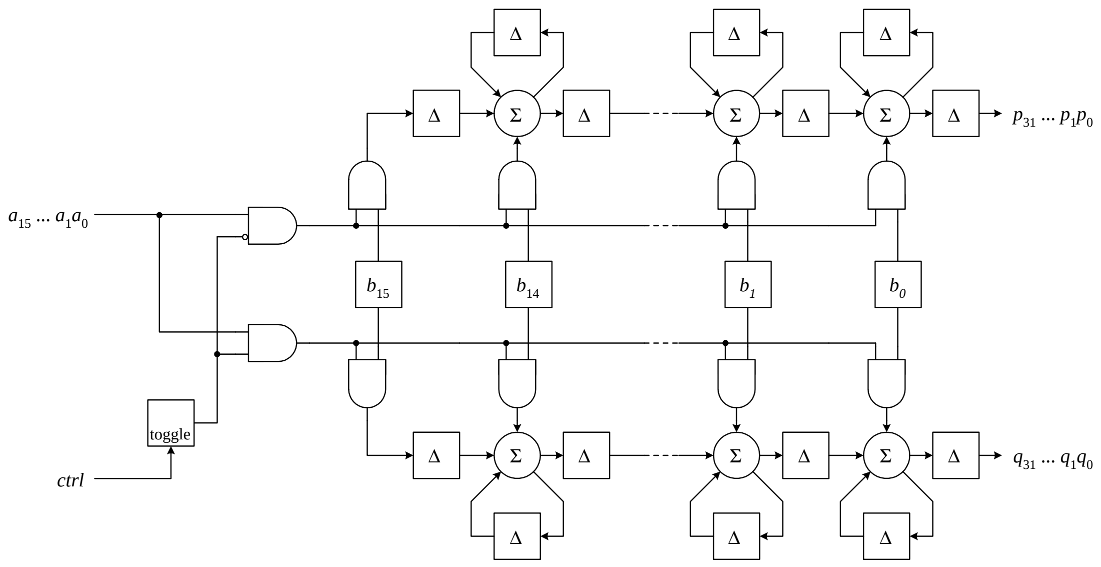


Figure 6.4: Parallel-serial multiplier modified for increased throughput.

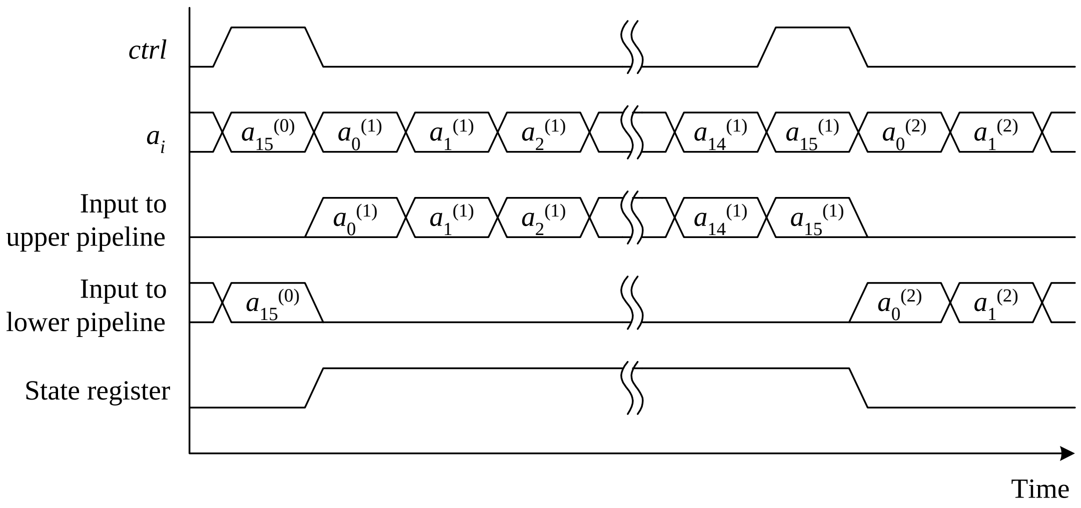


Figure 6.5: Timing diagram of the modified multiplier in Figure 6.4.

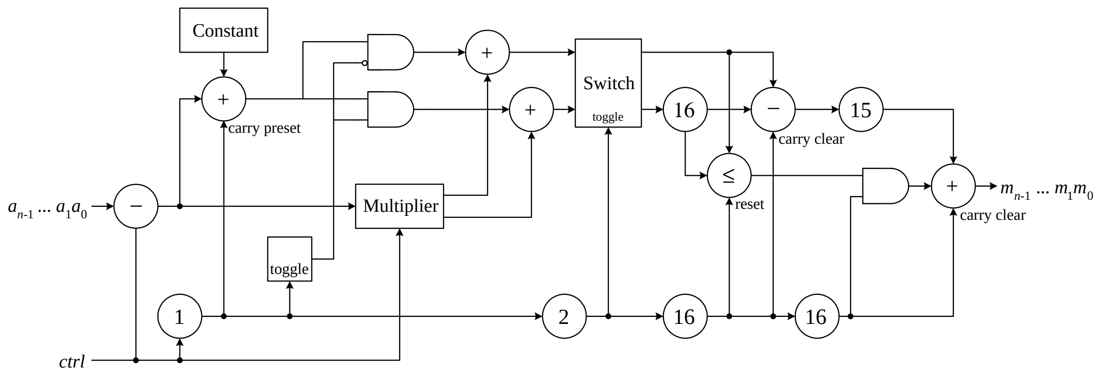


Figure 6.6: The bit-serial architecture for multiplication modulo $2^{16} + 1$ operations. Circled numbers indicates latches of that number of stages.

the multiplier, and at the same time stored into the SRL16E primitive responsible for key storage.

Utilizing the idea of multiple pipelines, the modular multiplication operation offers a throughput of sixteen cycles, even though a 32-bit intermediate result is computed. This scheme doubles the throughput but since sharing of the b registers can occur, the hardware cost is less than double.

The core implementation of IDEA is obtained by cascading eight identical rounds of operations shown in Figure 6.3, followed by a output transformation. For convenient interfacing, four parallel-to-serial converters are inserted before the first round and four serial-to-parallel converters are appended after the output transformation. The core takes one 64-bit plaintext once every 16 cycles, yielding an effective encryption rate of $f \times 64 \div 16$ Mbits/sec at a system clock rate of f MHz. As each round has a latency of 109 cycles, the output transformation has a latency of 35 cycles and each serial-to-parallel converter at the outputs has a latency of 16 cycles, the IDEA core has an overall latency of $109 \times 8 + 35 + 16 = 923$ cycles.

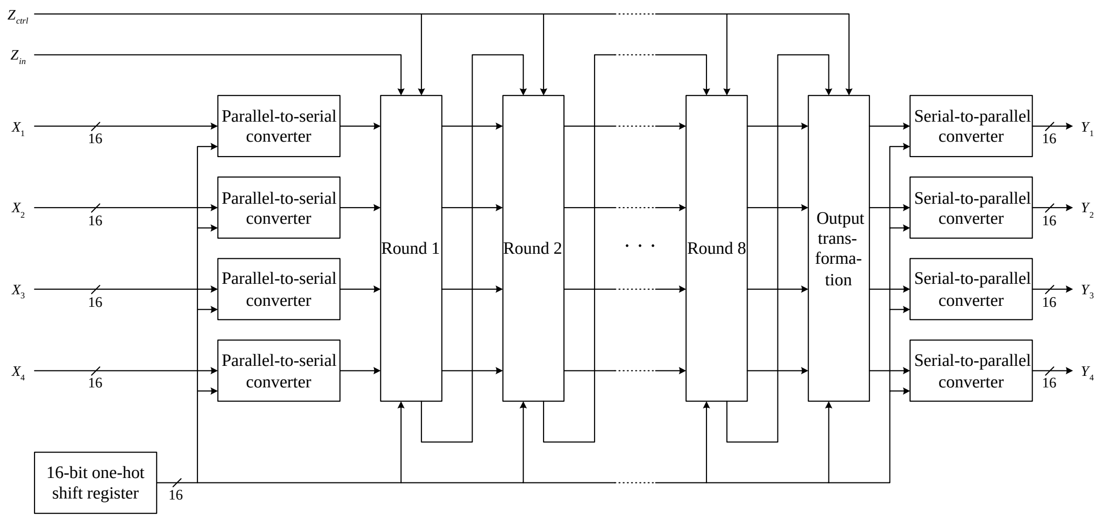


Figure 6.7: Shifting the key schedule into the IDEA core.

6.3.2 Modifying the Key Schedule

In this IDEA implementation, key schedule can be modified by two methods. The first method is to make all the shift registers (all implemented as SRL16E primitives) for key storage linked during initialization cycles. Upon initialization, the pre-computed 52 16-bit subkeys (a total of 832 bits) are passed bit-serially into the core via the shift registers. This design requires little overhead in routing and logic requirements. Figure 6.7 illustrates this design. During initialization Z_{ctrl} is set and the subkeys are passed bit-serially from Z_{in} to the rounds and the output transformation.

The second method is to utilize the bitstream modification technique (Appendix B). In this implementation, the subkeys are stored in SRL16E primitives. Hence, for each design bitstream containing this IDEA core, it is possible to extract the locations of the corresponding LUTs for these subkeys. By modifying the initial contents of these LUTs, an IDEA implementation with a new key schedule is obtained. There are totally 52 LUTs needed to be modify using the bitstream modification technique which takes about 0.12 second.

6.4 Results

With the Xilinx Virtex XCV300-6 FPGA as the target device, the fully-pipelined IDEA implementation requires 2801 slices, accounting for 91.18% of the total 3072 slices on an XCV300 device. Enabled by the deeply pipelined bit-serial architecture, a high clock rate is achieved. The clock rate reported by the implementation tool was 150.16 MHz, equivalent to an encryption rate of $150.16 \times 64 \div 16 = 600.64$ Mbits/sec at a latency of $923 \div 150.16 = 6.146$ microseconds.

This IDEA implementation were implemented using both the Annapolis Wildcard Reconfigurable Computing Engine [Ann99a] (Chapter 2.5) and *Pilchard*. The Wildcard platform has a Xilinx Virtex XCV3000-6 FPGA and a CardBus interface, while the *Pilchard* board used a Xilinx Virtex XCV300-6 FPGA.

On the Wildcard platform, the time taken to complete a FPGA to PC transaction was dominated by operating system overheads. When designing the interface between the IDEA core and the PC, it is crucial that the number of discrete CardBus read and write transactions be minimized and the amount of data transfered per transaction be maximized. Data is written directly to the core using a burst mode transfer of 512 64-bit plaintext blocks. The ciphertext is written to consecutive locations in the BlockRAM. The results are read by the PC from the IDEA processor by doing a burst mode transfer of the contents of the BlockRAM. This interface between the PC and the IDEA core on Wildcard requires an additional 238 slices, resulting in a total of 3039 slices, or 98.93% utilization of the FPGA.

Although the CardBus has a 1056 Mbits/sec (33 MHz, 32-bit) maximum transfer rate, its actual data transfer rate using programmed I/O is degraded due to very large operating system overheads in setting up a CardBus transaction. The implementation achieved a measured performance of 0.61×10^6 encryptions per second (39 Mbits/sec). The situation could be improved by using direct memory access (DMA), but the DMA interface requires approximately 400 slices and cannot be fit on a Virtex XCV300 with the IDEA core. The DMA interface was tested in a stand-alone configuration and the measured performance was 142 Mbits/sec. A larger device which can accommodate both the IDEA core and the DMA interface could achieve this performance.

On the *Pilchard* platform, the interface between the FPGA and the PC is comparatively simpler because it is not necessary to minimize the number of transactions. Plaintext from the PC are stored in a 64-bit first-in-first-out (FIFO). The IDEA core obtains these plaintext from the FIFO and put the ciphertext into another 64-bit FIFO for the PC to retrieve. To ensure data are written to the board in the correct sequence, the UC MTRR mode was used. This interface requires an additional 66 Virtex slices, resulting in a total of 2867 slices, or 93.33% utilization of the FPGA.

The measured performance of the IDEA implementation on *Pilchard* was 2.28×10^6 encryptions per second (146 Mbits/sec), a 3.74 times higher throughput than the same implementation on the Wildcard platform. This performance can be further improved if burst transfers to the *Pilchard* card could be effected, either using the WC MTRR (Section 5.3) or via Pentium SSE instructions.

6.5 Summary

A high-performance implementation of the IDEA is presented in this chapter. Using a dedicated bit-serial design of the multiplication modulo $2^{16}+1$ operator, the implementation occupies a minimal amount of hardware. The bit-serial architecture enabled the algorithm to be deeply pipelined to achieve a system clock rate of 150 MHz. An implementation on a Xilinx Virtex XCV300-6 delivers a throughput of 600 Mbits/sec.

For this implementation, the modification of the key schedule can be done by bitstream modification which offers improved security. Experiments on *Pilchard* showed significant performance improvements over the traditional interfacing technique. The measured throughput of the IDEA implementation on *Pilchard* was 146 Mbits/sec which is 3.74 times faster than the 39 Mbits/sec implementation on the Wildcard platform.

Chapter 7

Application II – Post-Rendering 3D Warping

The application of *fp* (Chapter 3) to a post-rendering 3D warping algorithm is presented in this chapter. An FPGA-based coprocessor approach was used, in which the most computationally intensive inner-loop (pixel re-projection) is executed on an FPGA while the remaining in software.

This application demonstrates the use of a variable wordlength design methodology to achieve better area efficiencies. It also demonstrates the ability of *fp* to obtain multiple implementations with a tradeoff between different precision and area from a single high-level description.

A literature review on related research in post-rendering 3D warping is first given. The algorithm chosen to be implemented on FPGA is then described, followed by its implementation and performance evaluation.

7.1 Background

Traditional rendering systems use linearly interpolated triangles to render a 2D graphical image from a complex 3D internal representation so as to obtain realistic rendering result. The computational requirement is strongly dependent on the scene complexity. To address this problem, an image-based rendering system was introduced by McMillan and Bishop [MB95] and after that, post-rendering 3D warping was proposed [MMB97]. Post-rendering 3D warping provides an order of magnitude improvement in apparent frame rates over conventional rendering, with computational requirements not dependent on scene complexity but rather on the image size.

The idea behind post-rendering 3D warping is illustrated in Figure 7.1. The inputs to the algorithm are a reference image and its corresponding depth map. Each pixel is projected to its actual 3D space and then reprojected to the new view plane. Given the reference image and depth map obtained from geometrical rendering, post-rendering 3D warping produces scenes with quality comparable to geometrical-based rendering for small changes in view angle and its performance is significantly faster. Hence, this technique is typically used in conjunction with traditional rendering for the smooth transition (small view angle changes) between key frames [MMB97]. This rendering technique is very effective and has been an active field of computer graphics research [MB97, SGHS98, RB98, RAPL98].

Figure 7.2 shows the inputs and outputs of a simple post-rendering 3D warping. The algorithm takes an input image (Figure 7.2(a)) and its corresponding depth map (Figure 7.2(b)). Two different views of the same scene obtained using simple post-rendering 3D warping are shown (Figures 7.2(c) and 7.2(d)). The resultant image have dark lines which are caused by under-sampling. This prob-

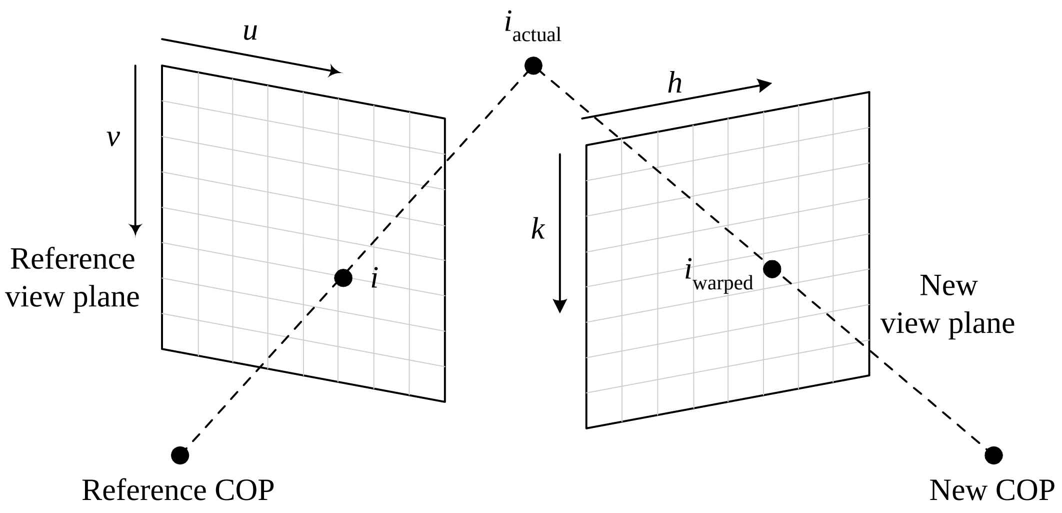


Figure 7.1: Post-rendering 3D warping of pixel i to i_{warped}. i_{actual} is the pixel projected to its actual 3D space and COP is the center of projection. u, v, h and k are the horizontal and the vertical offsets of the pixel in the reference and new viewplanes respectively.

lem can be overcame by splat reconstruction or mesh reconstruction [MMB97]. One other difficulty which must be addressed by the 3D warping problem is that there is no information about objects occluded in the reference image but which might appear in the new view. This problem, called the occlusion problem, can be handled by having multiple reference images [RAPL98] or layered depth images [SGHS98].

The post-rendering 3D warping algorithm is simple so its hardware and memory requirements are modest and high bandwidth memory is not necessary. These two features make it feasible to employ an FPGA-based coprocessor approach, in which the FPGA renders the images, the host PC displays the warped images and obtains new view parameters from an user interface. The under-sampling and occlusion problems were not addressed in this dissertation.

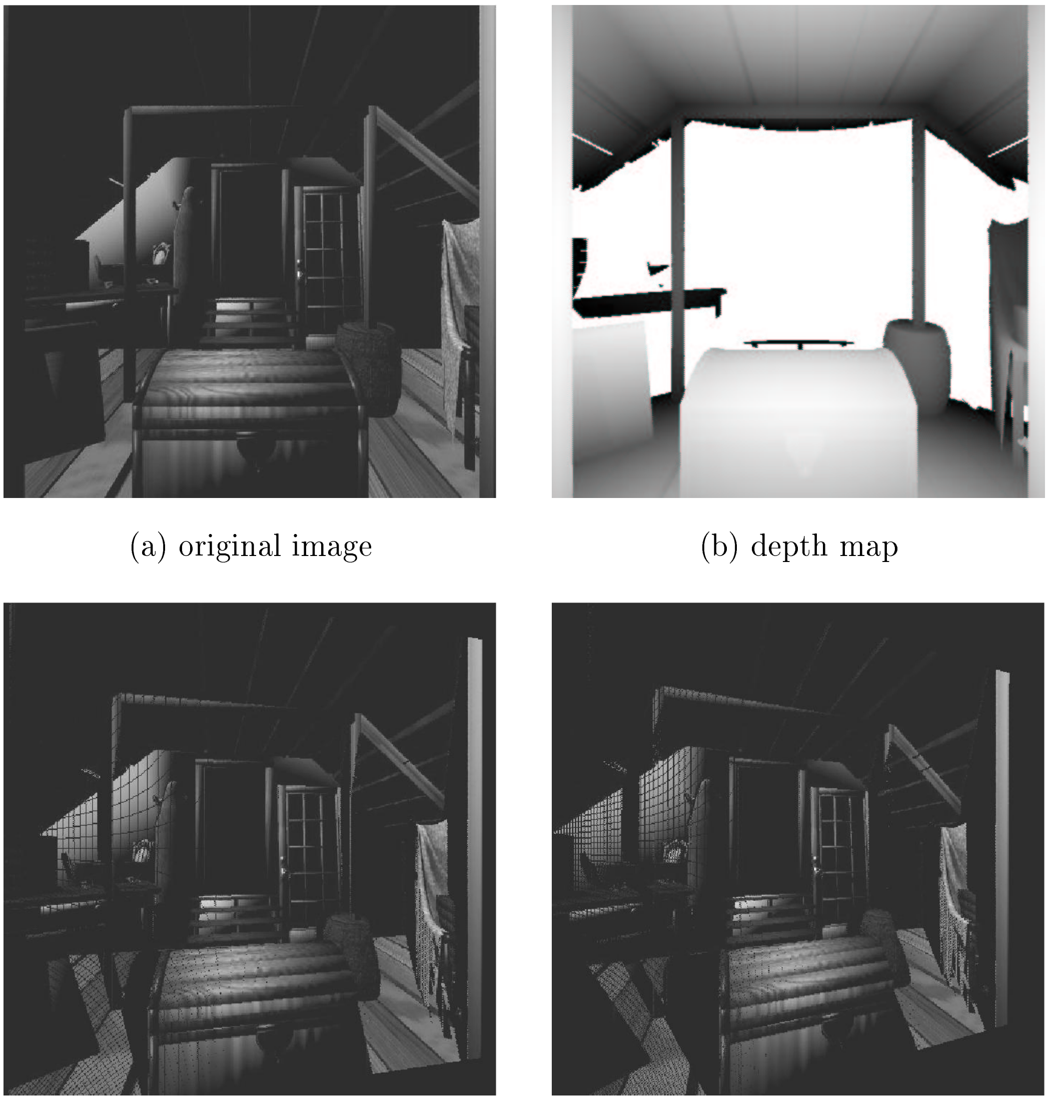
7.2 Algorithm

Typically the 3D representation of the image is generated by a software geometrical renderer. A perspective projection model with image plane is used for the reference view.

The framework of re-projecting a pixel with respect to a new viewpoint or a center of projection (COP) is shown in Figure 7.3. Denote the original and the new viewpoints be $\overrightarrow{u}$ and $\overrightarrow{v}$, the new viewplane's top-left, top-right, bottom-left and bottom-right corners be $\overrightarrow{A_{00}}$, $\overrightarrow{A_{10}}$, $\overrightarrow{A_{01}}$ and $\overrightarrow{A_{11}}$ respectively. The vectors from $\overrightarrow{v}$ to $\overrightarrow{A_{00}}$, $\overrightarrow{A_{10}}$, $\overrightarrow{A_{01}}$ and $\overrightarrow{A_{11}}$ are named as $\overrightarrow{v_{00}}$, $\overrightarrow{v_{01}}$, $\overrightarrow{v_{10}}$ and $\overrightarrow{v_{11}}$ respectively.

The post-rendering 3D warping algorithm is as follows:

1. Project the pixel from the reference image to its actual 3D position, $\overrightarrow{p}$ with respect to the coordinate system centered by the new viewpoint and aligned


(a) original image (b) depth map

(c) warped image (d) warped image

Figure 7.2: The inputs and outputs of post-rendering 3D warping.

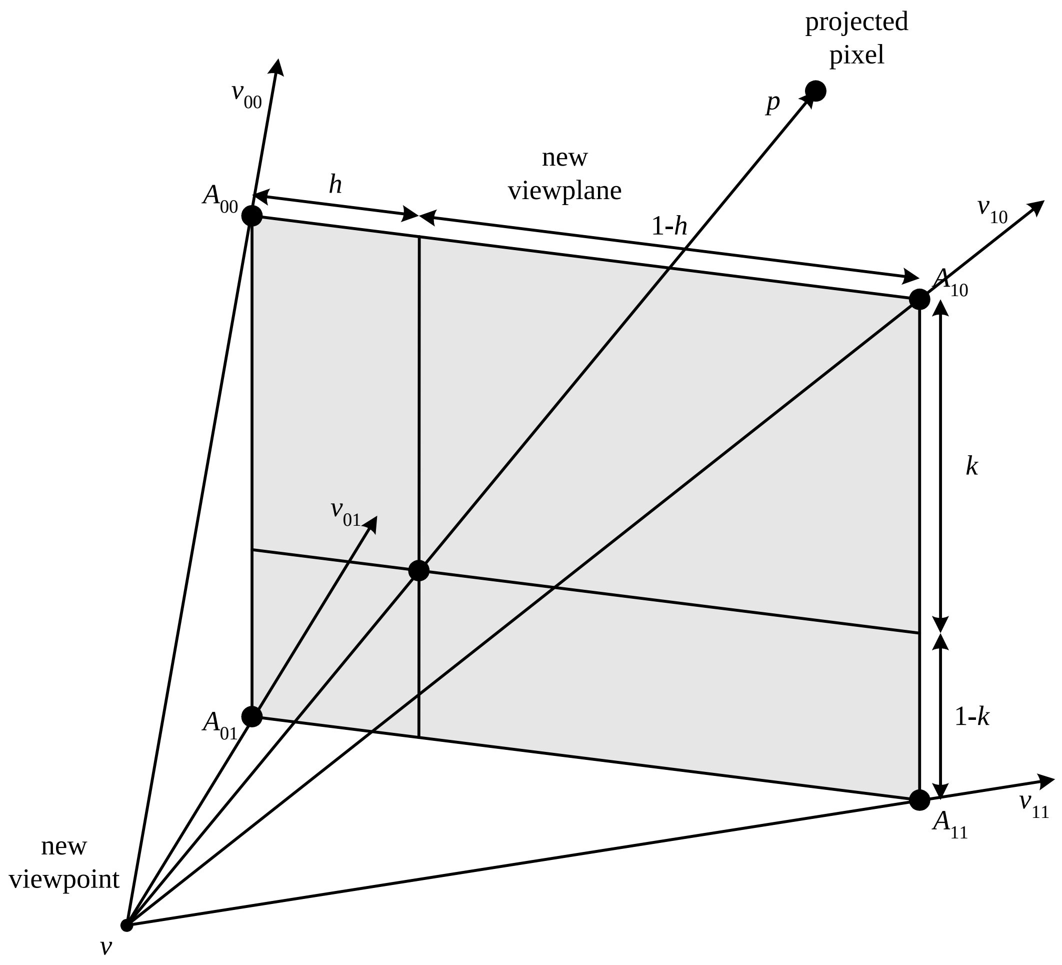


Figure 7.3: Framework of re-projecting a pixel with respect to a new viewpoint.

by the new viewplane.

2. Find the projected position of $\overrightarrow{p}$, $\lambda\overrightarrow{p}$ on the new viewplane $\overrightarrow{I}(h,k) = (1-k)((1-h)\overrightarrow{v_{00}} + h\overrightarrow{v_{10}}) + k((1-h)\overrightarrow{v_{01}} + h\overrightarrow{v_{11}})$.

3. Equating $\lambda\overrightarrow{p} = \overrightarrow{I}(h,k)$ gives three independent equations along the x, y and z dimensions, hence real numbers h and k can be solved.

4. If $h, k \in [0.0, 1.0)$, then the pixel will be splattered onto the new image plane at the coordinate $(res_x \times h, res_y \times k)$, where res_x and res_y is the resolution.

5. Steps 1 to 4 are repeated for every pixel of the input image.

7.3 Implementation

In the hardware fixed-point implementation, it is assumed that the original viewpoint is the origin of the 3D space ($\overrightarrow{u} = \{0,0,0\}$). The original viewplane is parallel to the XY-plane, centers at $\{0,0,1\}$, with up vector along the Y-axis and the size being four unit squares (two units by two units). ($\overrightarrow{u_{view}} = \{0,0,1\}$ $\overrightarrow{u_{up}} = \{0,1,0\}$, the four corners of the original viewplane are $\{-1,1,1\}$, $\{1,1,1\}$, $\{-1,-1,1\}$ and $\{1,-1,1\}$).

The post-rendering 3D warping algorithm was split into two parts. The projection part projects every pixel to its actual 3D position $\overrightarrow{p}$ from the depth map. The 3D projection of the pixel at the i-th column and j-th row of the input image can be obtained by solving $\overrightarrow{p} = \overrightarrow{u} + d\overrightarrow{up}$, where $\overrightarrow{up}$ is the normalized vector from the origin $\overrightarrow{p} = \{0,0,0\}$ to the coordinate of the pixel on the original viewplane $\{i/res_x - 1, j/res_y - 1, 1\}$, d is the depth of the pixel obtained from the depth map. The results of preprocessing is the 3D coordinate of every pixel in the original image. Only one execution of this part is needed for an input image and it is done in software.

The re-projection part of the algorithm is required for every change in the new viewpoint or the new view direction. This part is executed on hardware. The re-projection of the post-rendering 3D warping algorithm was described by the algorithmic description as shown in Figure 7.4. In order to simplify the problem, it is assumed that the up vector of the view direction is always parallel to the Y-axis. ($\overrightarrow{v_{up}} = \{0, 1, 0\}$). As inputs, this function takes the projected coordinates of a pixel $\overrightarrow{q}$, the new viewpoint $\overrightarrow{v}$ and the new view direction $\overrightarrow{v_{view}}$. As outputs, the offsets of the pixel after projected on the new viewplane along the two directions, h and k, are computed. From the 25-line algorithmic description in Figure 7.4, about 20000 lines of VHDL descriptions are generated.

In a software implementation, all of the operations are performed using double-precision floating-point arithmetic. Although the algorithm is very simple, it is not straightforward to implement it using fixed-point arithmetic since it is difficult to determine the number of bits required for λ, h and k due to the use of fixed-point division, fractional multiplication and square root functions. The challenge of implementing this re-projection algorithm in hardware is that the tradeoff between area requirements and the quality of output image needs to be considered.

The VHDL description generated from the algorithmic description is used for the synthesis of a component, namely the algorithm function, in the implementation, as illustrated in Figure 7.5. The new viewpoint and direction are obtained from an user interface. The 3D coordinates of every pixel was retrieved from an external memory and enter the algorithm function to compute the projected coordinate on the new viewplane. At the same time, their corresponding red-green-blue (RGB) color values were read from the memory and enter a shift

```
#define EQ(w, x, y, z) {w[0] = x; w[1] = y; w[2] = z;}
void normalize(fixed *n)
{
    fixed m;
    m = sqrt(n[0] * n[0] + n[1] * n[1] + n[2] * n[2]);
    EQ(n, n[0] / m, n[1] / m, n[2] / m);
}
void warp(fixed *v, fixed *vd, fixed *p,            // inputs
    fixed &h, fixed &k)                              // outputs
{
    fixed vp[3], v00[3], v01[3], v10[3], v11[3], nvp, l;
    EQ(vp, vd[2], 0, -vd[0]);
    normalize(vp);                   // normalize the view vector
    EQ(v00, vd[0] + vp[0], vd[1] + 1.0, vd[2] + vp[2]);
    EQ(v01, vd[0] + vp[0], vd[1] - 1.0, vd[2] + vp[2]);
    EQ(v10, vd[0] - vp[0], vd[1] + 1.0, vd[2] - vp[2]);
    EQ(v11, vd[0] - vp[0], vd[1] - 1.0, vd[2] - vp[2]);
              // the vector to the four corners of the new viewplane
    EQ(nvp, p[0] - v[0], p[1] - v[1], p[2] - v[2]);
    normalize(nvp);
                  // the vector from the new viewpoint to the pixel
```

Figure 7.4: The *fp* algorithmic description of the re-projection of the post-rendering 3D warping algorithm.

```
    l = (v00[0] * (v10[2] - v00[2]) - v00[2] *
        (v10[0] - v00[0])) /
        (nvp[0] * (v10[2] - v00[2]) - nvp[2] *
        (v10[0] - v00[0]));                         // lambda
    h = (l * nvp[0] - v00[0]) / (v10[0] - v00[0]);
    k = (l * nvp[1] - v00[1]) / (v01[1] - v00[1]);
}
```

Figure 7.4 (continued): The algorithmic description of the re-projection of the post-rendering 3D warping algorithm.

register which has exactly the same latency as the algorithm function. Eventually, time-aligned pairs of new viewplane coordinates and color values are produced and are written back to the memory. The color values are written to a memory location indexed by the new viewplane coordinate, hence a resultant image is formed at the output buffer which can be directly displayed by the host PC. In order to save external memory requirements, the wordlengths of the 3D representation of the image was reduced to 16 bits. This constraint made it impossible to achieve a pixel-exact correspondence between the floating-point and fixed-point implementations.

In this implementation, bit-parallel multi-cycle arithmetic operators were used. The choice of bit-parallel operators is due to that division and square root cannot be implemented efficiently with a digit-serial architecture. Multi-cycle but not pipelined operators were employed in order to conserve area. Specifically, the fixed-point division is implemented using restoring-division and the square root is implemented by completing the square method [Kor95]. These were all implemented as modules inside *fp*.

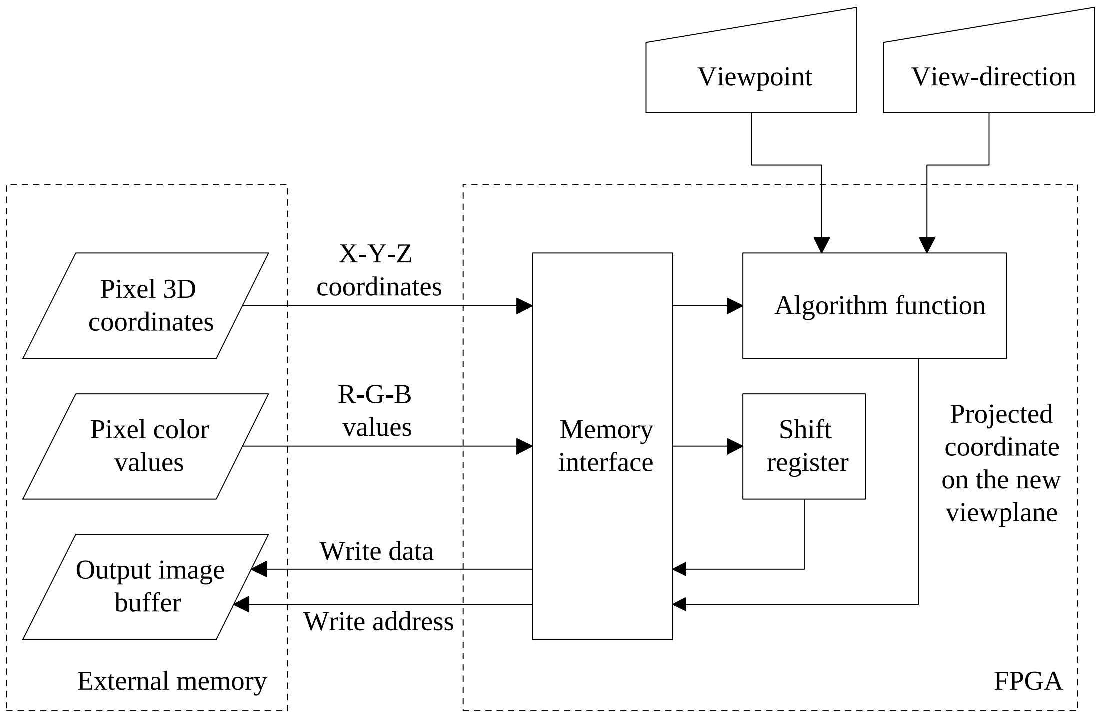


Figure 7.5: Hardware implementation of the re-projection of the post-rendering 3D warping algorithm.

7.4 Results

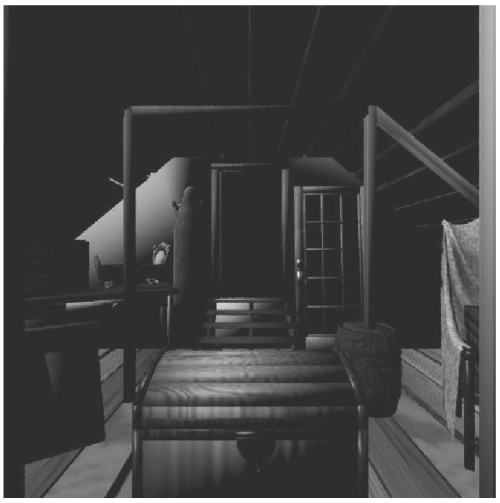
The post-rendering 3D warping routine consists of 39 operations. Note that the Y-axis component of the view vector $\overrightarrow{vp} = \{vp_x, vp_y, vp_z\}$ is zero ($vp_y = 0$, line 11 in Figure 7.4). When normalizing $\overrightarrow{vp}$ (that is, computing vp_x/m, vp_y/m and vp_z/m, where $m = \sqrt{vp_x^2 + vp_y^2 + vp_z^2}$), one multiplication, one addition and one division operators are unnecessary and thus removed ($vp_y^2 = 0$ and $vp_y/m = 0$).

The cost functions are weighted sums of two components, one of which is the estimated area while the other is the reciprocal of sums of output errors. No constraint was set. Two sets of cost function weights were experimented. The first set has a larger coefficient for the error component and produced an implementation with a higher precision (an average of 21 bits per fixed-point variables). The second set has a larger coefficient for the area component and produced an implementation with smaller area (an average of 14 bits per variable).

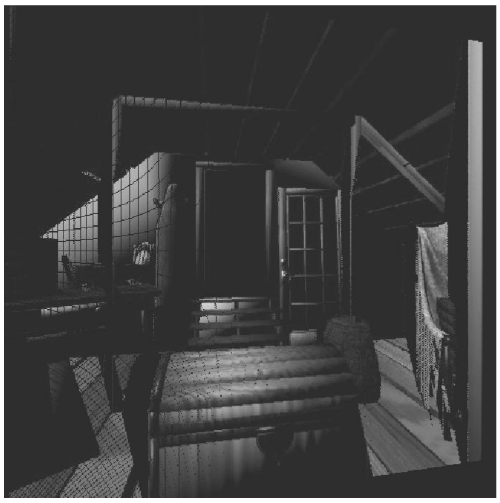
The average errors of outputs h and k (Section 7.2) are respectively 4.191×10^{-4} and 5.248×10^{-4} for the 21-bit average wordlength implementation, and 9.798×10^{-4} and 1.016×10^{-3} for the 14-bit average wordlength implementation. Due to the choice of multi-cycle operators, the throughput of implementations are limited by the operator with the lowest throughput (the longest latency). The execution time for the 21-bit and 14-bit average wordlength implementations are 37 and 33 clock cycles per pixel respectively.

The resultant images from the floating-point software and the fixed-point hardware implementations are shown in Figure 7.6. It was observed that the quality of the resultant images were dependent on the wordlengths of variables, as predicted by the magnitude of the output error.

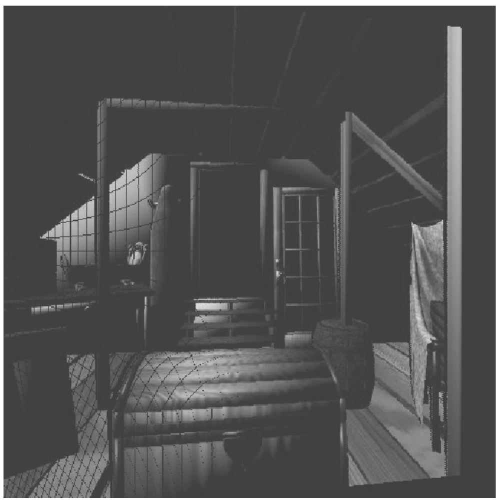
The generated implementations were used with a manually designed memory interface to produce the complete designs. The designs were synthesized with a



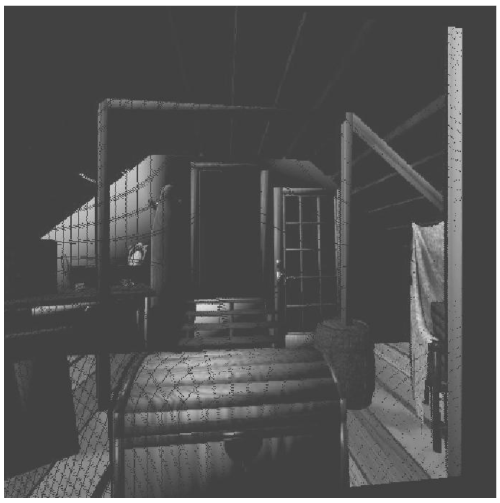
(a) original image



(b) result of floating-point implementation



(c) result of 21-bit average wordlength fixed-point implementation



(d) result of 14-bit average wordlength fixed-point implementation

Figure 7.6: Post-rendering 3D warped images obtained from floating-point software and fixed-point hardware implementations.

Average wordlength	21 bits	14 bits
Output error at h	4.191×10^{-4}	9.798×10^{-4}
Output error at k	5.248×10^{-4}	1.016×10^{-3}
Area	3099 CLBs	2334 CLBs
Verified clock rate	40.0 MHz	41.0 MHz
Latency	37 clock cycles	33 clock cycles
Maximum frame rate	4.12 fps	4.62 fps

Table 7.1: Summary of post-rendering 3D warping fixed-point implementations.

Xilinx XC4085XL-1 FPGA (contains 3136 CLBs) as the target device. The design with 21-bit average wordlength algorithm function requires 3099 CLBs, and the one with 14-bit average wordlength algorithm function requires 2334 CLBs (a XC4000 CLB has roughly of the same amount of logic as a Virtex slice).

The maximum clock rate reported by the implementation tool was about 20 MHz, but both designs have been successfully verified at approximately 40 MHz. With a 40 MHz clock rate and an image resolution of 512×512 pixels, the 21-bit average wordlength implementation achieved a frame rate (measured in frames per seconds (fps)) of $40 \div (37 \times 512 \times 512) = 4.12$ fps, while the 14-bit average wordlength implementation achieves $40 \div (33 \times 512 \times 512) = 4.62$ fps. These implementations were verified on the Annapolis Wildforce Reconfigurable Computing Platform (Chapter 2.5). These results are summarized in Table 7.1.

As a comparison, the performance of an optimized software implementation running on a Sun Ultra-5 270 MHz workstation was 3.70 fps. The speed improvement obtained by the hardware implementations was limited because limited by the choice of multi-cycle operators for multiply, divide and square root fixed-point operators so that the design could fit on the Xilinx XC4085XL-1 device.

The computational bottleneck in the current implementation can be eliminated by using pipelined operators instead of multi-cycle operators. The pipelined implementation, with an estimated size of 6000 CLBs, should fit in larger FPGA devices which are already available.

7.5 Summary

The application of *fp* in post-rendering 3D warping has demonstrated a floating-point algorithm written in a general programming language to be directly converted into fixed-point hardware description. The algorithmic description is written in the C programming language and can be used in a hardware implementation with *fp* as well as in a software implementation with a C compiler. This property facilitated the FPGA-based coprocessor approach, in which the program can be used directly for a hardware implementation to gain immediate speedup. Hardware speedup was limited (1.11 and 1.25 times faster than a software implementation) due to the choice of multi-cycle operators.

The process of implementing the algorithm in hardware involves floating-point to fixed-point conversion and the generation of hardware description. With the multiple wordlength design methodology, the tradeoff between area and performance was addressed. Expressing the required tradeoff as a cost function enabled designer to concentrate more on the higher level algorithmic issues and less on the implementation details.

Chapter 8

Application III – Electronic Cochlea Filter Model

The application of *fp* (Chapter 3) to Lyon and Mead's electronic cochlea filter model is presented in this chapter. This model can be used as an accelerator for a PC or as the front end for embedded auditory signal processing systems.

To efficiently design this application, an infinite impulse response (IIR) filter operator was developed and added to the *fp* fixed-point and module libraries. It demonstrated that users can build their own set of operators on top of the primitives to meet application requirements. Implementations of the cochlea filter model on *Pilchard* (Chapter 5) and on a typical RC platform (with a PCI Local Bus interface) demonstrated improved system performance using a memory slot interface. This application also shows the benefits of parameterized module generation which significantly improves productivity and allows designers to explore the tradeoff between area and precision associated with a fixed-point implementation.

This chapter begins with a literature review and a description of the Lyon and Mead's cochlea model. The implementation and performance analysis are given, followed by the application of the electronic cochlea to speech processing is presented.

8.1 Background

It is clear that biological-based systems perform feats of signal processing that human being cannot approach using even the most sophisticated computers and digital signal processing techniques. Generally, biological-based auditory systems operate with greater functionality, lower power consumption and increased robustness than their man-made electrical counterparts. This is particularly true in tasks such as speech recognition where humans are able to process signals far better than the most sophisticated computer-based systems. A lot can be learnt from the elegant designs of nature.

The field of neuromorphic engineering has the long term objective of taking architectures from the understanding of biological systems to develop novel signal processing systems. This field of research, pioneered by Carver Mead [Mea89] has concentrated on using analogue VLSI to model biological systems. Research in this field has led to biologically inspired signal processing systems which have improved performance compared to traditional systems.

The human cochlea is a transducer which converts mechanical vibrations from the middle ear into neural electrical discharges, and additionally provides spatial separation of frequency information in a manner similar to that of a spectrum analyzer [LM88]. It serves as the front end signal processing for all functions of the auditory nervous system such as auditory localization, pitch detection and speech recognition.

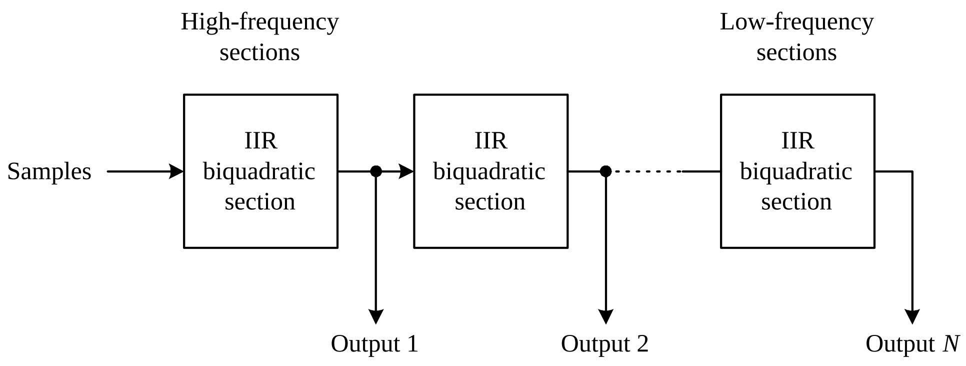


Figure 8.1: Cascaded IIR biquadratic section used in the Lyon and Meads cochlea model.

Although it is possible to simulate cochlea models in software, hardware implementations may have orders of magnitude of improvement in performance. Hardware implementations are also attractive when the target applications are on embedded devices in which power-efficiency and small-footprint are design considerations.

The electronic cochlea, first proposed by Lyon and Mead [LM88] is a cascade of biquadratic filter sections (as shown in Figure 8.1) which mimics the qualitative behavior of the human cochlea. Electronic cochleae have been successfully used in many auditory signal processing systems such as spatial localization [LM89a], pitch detection [LM89b], a computer peripheral [LWK94], amplitude modulation detection [vSM99], correlation [MAL91] and speech recognition [LWL97].

There have been several implementations of electronic cochleae in analogue VLSI technology. The original implementation by Lyon and Mead was published in 1988 and used continuous time subthreshold transconductance circuits to implement a cascade of 480 stages [LM88, Lyo91]. In 1992, Watts et. al. reported a 50-stage version with improved dynamic range, stability, matching and compactness [WKLM92]. A problem with analogue implementations is that transistor matching issues affect the stability, accuracy and size of the filters. This issue was addressed by van Schaik et. al. in 1997 using compatible lateral bipolar transistors (CLBTs) instead of metal oxide semiconductor field effect transistors (MOSFETs) in parts of the circuit [vSFV97]. Their 104-stage test chip showed greatly improved characteristics. In addition, a switched capacitor cochlea filter was proposed by Bor et. al. in 1996 [BW96].

There have also been several previously reported digital VLSI cochlea implementations. In 1992, Summerfield and Lyon reported an ASIC implementation which employed bit-serial second-order filters [SL92]. In 1997, Lim et. al. reported a VHDL-based pitch detection system which used first-order Butterworth band-pass filters for cochlea filtering [LTJM97]. Later in 1998, Brucke et. al. designed a VLSI implementation of a speech preprocessor which used gammatone filter banks to mimic the cochlea [BNS+98]. The implementation by Brucke et. al. used fixed-point arithmetic and they also explored the tradeoff between word-length and precision.

Recent advances in FPGA technology have resulted in devices with a density where it is possible to develop neuromorphic systems on a single device. Many interesting neuromorphic signal processing systems can be implemented using FPGA technology, enjoying the following advantages over analogue VLSI:

- shorter design time;
- faster fabrication time;
- more robust to power supply, temperature and transistor mismatch variations;
- wider dynamic range and higher signal to noise ratios;
- better stability;

- the chips can be reused for different application;
- simpler computer interface.

In this dissertation, an FPGA implementation of an electronic cochlea which can serve as an accelerator in its own right, or as a front end preprocessing stage for embedded auditory applications is presented. A module generator which can generate synthesizable VHDL descriptions of arbitrary wordlength fixed-point cochlea filters was developed. The module generator was developed with the *fp* tool to determine the minimum and maximum ranges as well as the error statistics of all variables. The range information is used to determine the number of fractional bits used in the variable's two's complement fraction representation. The error statistics provide information to the designer concerning the accuracy of the resultant implementation.

8.2 Algorithm

Lyon and Mead proposed the first electronic cochlea in 1988 [LM88, LM89c]. This model captured the qualitative behavior of the human cochlea using a simple cascade of second order filter stages which they implemented in analogue VLSI. In this section a very superficial summary of the Lyon and Mead cochlea model is given. More detailed descriptions of the cochlea can be found in the work of Lyon and Mead [LM88] and Pickles [Pic88].

The human cochlea, or inner ear, is a three dimensional fluid-dynamic system which converts mechanical vibrations from the middle ear into neural electrical discharges [LM88]. It is composed of the basilar membrane, inner hair cells and outer hair cells. The cochlea connects to higher levels in the auditory pathway for further processing.

The basilar membrane is a longitudinal membrane within the cochlea. The oval window is the input to the cochlea. Vibrations of the eardrum are coupled via bones in the middle ear to the oval window causing a traveling wave from base to apex along the basilar membrane. The basilar membrane has a filtering action and can be thought of as a cascade of lowpass filter with exponentially decreasing cutoff frequency from base to apex.

The result of the filtering of the basilar membrane at any point along its length is a bandpass filtered version of the input signal, with center frequency decreasing along its length. Different distances along the basilar membrane are tuned to specific frequencies in a manner similar to that of a spectrum analyzer. A simplified model showing a sinusoidal wave traveling along an uncoiled cochlea is shown in Figure 8.2.

Several thousand inner hair cells are distributed along the basilar membrane and convert the displacement of the basilar membrane to a neural signal. The hair cells also perform a half-wave rectifying function since only displacements in one direction will cause neurons to fire.

The outer hair cells perform automatic gain control by changing the damping of the basilar membrane. It is interesting to note that there are approximately three times more outer hair cells than inner hair cells.

In order to simulate the properties of the basilar membrane, Lyon and Mead's cochlea model used a cascade of scaled second-order low-pass filters with the transfer function

$$H(s) = \frac{1}{\tau^2 s^2 + \frac{1}{Q}\tau s + 1} \tag{8.1}$$

where Q represents the damping characteristic (or quality) of the filter and τ the time constant. In the cochlea filter, the τ of each filter is varied exponentially along the cascade, causing filters to have exponentially decreasing cutoff

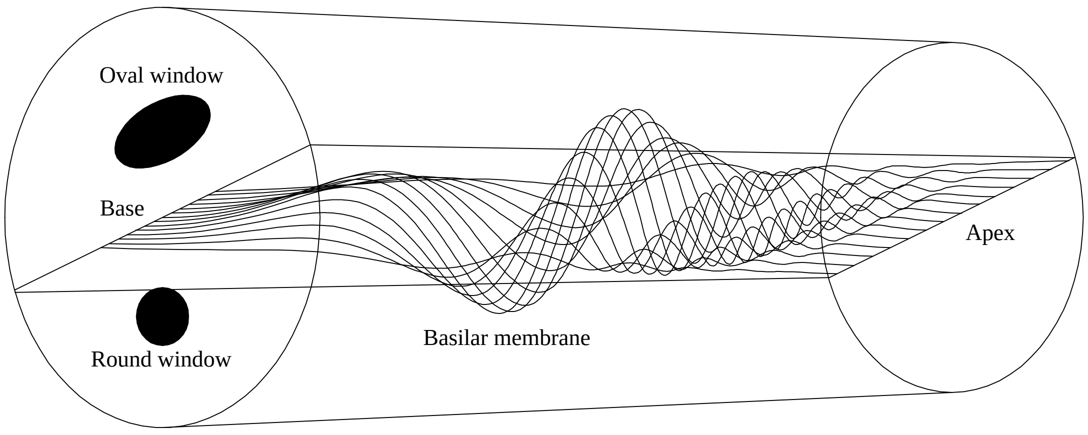


Figure 8.2: Illustration of a sine wave traveling through a simplified model of an uncoiled cochlea (adapted from [Jin01]).

frequencies. The Q of all the filters is held constant. The outputs of each filter corresponds to the displacement of different positions along the basilar membrane.

8.3 Implementation

As depicted in Figure 8.1, the Lyon and Mead's cochlea model is a cascade of second-order infinite impulse response (IIR) filters. The structure of the IIR filter plays an important role in determining the area-efficiency of the implementation. To achieve a highly area-efficient implementation on FPGAs, the IIR filters were implemented with distributed arithmetic (DA). The implementation of IIR filters using DA is presented in Section 8.3.1.

In Section 8.3.2, the implementation of the module generator for the electronic cochlea is presented. With the set of filter coefficients and a sample dataset, the module generator can produce implementations of electronic cochlea in any wordlength so that the tradeoff between area and precision can be addressed.

8.3.1 IIR Filters Using Distributed Arithmetic

DA offers an efficient method to implement a sum of products (SOP) provided that one of the variables does not change during execution. DA is applicable to bit-serial, LSB-first architectures. Instead of requiring a multiplier, DA utilizes a pre-computed LUT, namely the DA ROM. The corresponding bits from the bit-serial inputs form an address to the DA ROM to yield a partial sum. Accumulating the partial sums for every inputs bits forms the SOP. Calculation of SOP by DA offers not only compactness, but also low quantization error comparing with ordinary two's complement arithmetic operations [Gos95, Xil96]. A detailed description of DA is given in Appendix C.

Equation 8.1 can be converted from the s-domain to the z-domain via a bilinear transform. The resulting transfer function has the form

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

The corresponding time domain IIR filter can be implemented by the function

$$y(n) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) + a_0 y(n-1) + a_1 y(n-2)$$

where $x(n-k)$ is the k'th previous input, $y(n-k)$ is the k'th previous output and $y(n)$ is the output. The operation is essentially the SOP of five terms, and can be directly map to a biquadratic section as shown in 8.3.

Figure 8.4 illustrates the actual implementation using distributed arithmetic on a Xilinx Virtex FPGA. The previous values $x(n-1)$, $x(n-2)$, and $y(n-2)$ are implemented using shift registers with the number of stages equal to the wordlength of the variables used. The shift registers are implemented by cascades of Virtex SRL16E primitives for minimum area. The DA ROM takes $x(n)$, $x(n-1)$, $x(n-2)$, $y(n-1)$ and $y(n-2)$ as inputs to generate partial sums. As there are 5 inputs, the required number of entries in the ROM is $2^5 = 32$, leading

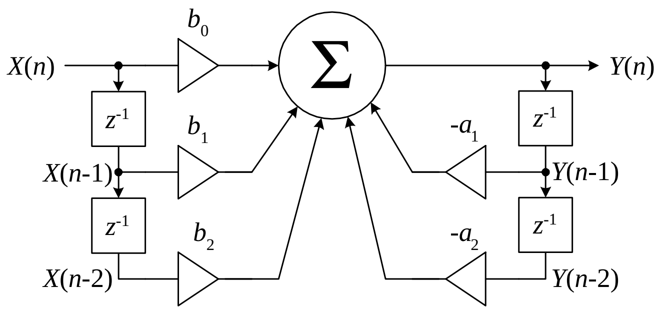


Figure 8.3: The architecture of an IIR biquadratic section.

to an efficient implementation using Xilinx ROM32X1 primitives. The scaling accumulator shifts and adds the output from the ROM (unscaled partial sum in bit-parallel organization) at every cycle to produce $y(n)$. In the last cycle of scaling and accumulation, the parallel to serial converter latches the value at the scaling accumulator. Since the scaling accumulator has a latency equal to the wordlength of the variables, the value latched by the converter is $y(n-1)$.

8.3.2 Module Generator

Given the filter coefficients, the designer may select values of filter wordlength and the number of bits of the DA ROM's output. The module generator restricts the wordlengths of the filters to be the same. In a bit-serial architecture (the use of DA implies a bit-serial architecture) the throughput cycles is limited by the variable with the maximum wordlength. Using a common filter wordlength ensures all the filters have the same throughput and hence the highest overall resource utilization. Note that all filter sections have the same wordlength but the allocation of integer and fractional parts used within each filter section can vary.

To develop this module generator, an *fp* IIR biquadratic section operator was

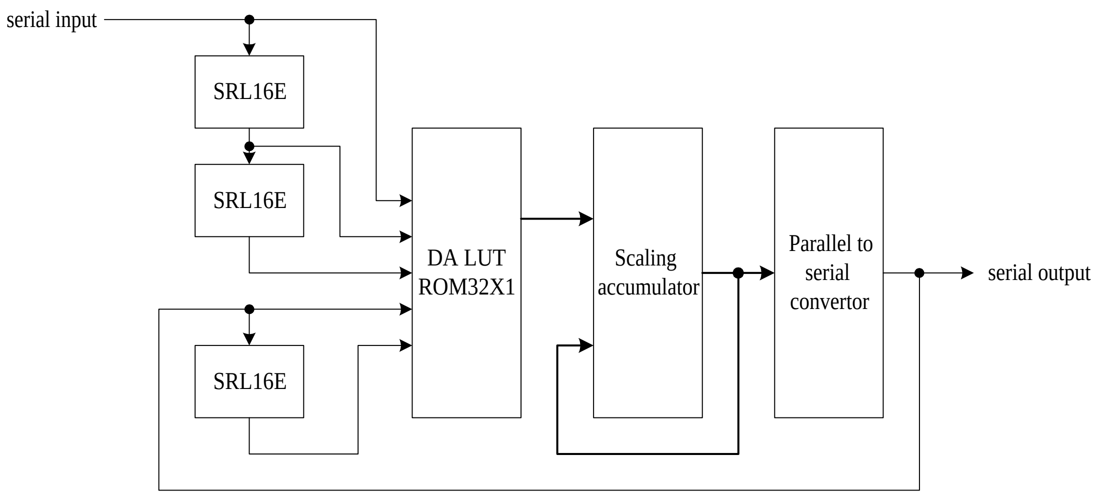


Figure 8.4: Implementation of an IIR biquadratic section on a Xilinx Virtex FPGA.

designed. Indeed, the sole class of operator used in this cochlea model is the IIR biquadratic section. Similar to other *fp* operators, the IIR biquadratic section operator has the three basic parameters, `inTruncation0`, `outTruncation0` and `outExpansion0` (Section 3.4.4) plus a special parameter `daRomWidth` which controls the width of the DA ROM's output. This module did not utilize the wordlength optimization feature of *fp* (Section 3.6) because the wordlengths of the inputs and outputs of the biquadratic sections are fixed and can be easily derived. Nevertheless, it uses the fixed-point modeling feature of *fp* to extract the dynamic range of each variable. Knowing the dynamic ranges of variables, the minimally required integer wordlength is determined. As the total wordlength is fixed, maximal fractional bits are therefore assigned to every variable to minimize the quantization error. Put in another way, optimal scaling of variables are obtained. If a sample dataset is supplied, runtime analysis is performed. In normal circumstances, using broadband noise of the maximum amplitude as the sample input dataset results in a set of filter scalings that will not overflow for any reasonable inputs. If the module generator is not supplied with a dataset, range extraction can be carried out based on worst-case analysis. However, as explained in Section 3.5.1, worst-case analysis is more conservative and usually leads to unnecessarily long integer wordlengths.

The core component of the module generator is an algorithmic description of the cochlea filter model. This algorithmic description is shown in Figure 8.5. It takes the floating-point coefficients for the biquadratic filters from an external data file (`coeff.dat`), which is obtained using Malcom Slaney's Auditory Toolbox [Sla98]. This Matlab toolbox has several different cochlea models, test inputs and visualization tools. The same toolbox was used to verify the designs and produce cochleagram plots.

In additional to the filter coefficients, the cochlea model generator takes user supplied wordlengths for the variables and the width of the DA ROM as inputs. With this information, the error statistics at every filter output are calculated. Also, the area of the cochlea model is estimated. If the error or the area does not satisfy requirements, the user may supply another set of parameters.

After deducing the best representation for each variable, the generator outputs synthesizable VHDL code that describes an implementation of the corresponding cochlea model. The fractional wordlengths of the scaling accumulator and the output variable can be different, so the operator must also include a mechanism to convert the former to the latter. Since the output of the scaling accumulator is bit-parallel while the output variable is bit-serial, the parallel to serial converter can perform format scaling by selecting the appropriate bits to serialize. The resulting VHDL description can then be used as a core in other designs.

From the 17-line algorithmic description, this cochlea model generator produces approximately 40000 lines of VHDL code for the case of a cochlea filter

```
void cochlea(fixed &sample, fixed *results)
                              // sample is the input to the first filter
                              // results is the array of filter outputs
{
    double coeff[5];
    int i;
    FILE *fp = fopen("coeff.dat", "rb");
    for (i = 0;; i++)
    {
        fread(coeff, sizeof(double), 5, fp);
        if (feof(fp))
            break;
        if (i == 0)                                    // the first filter
            results[i] = iir(sample, coeff);
        else                                     // the remaining filters
            results[i] = iir(results[i - 1], coeff);
    }
    fclose(fp);
}
```

Figure 8.5: The *fp* algorithmic description of the cochlea filter model.

with 88 biquadratic sections.

8.4 Results

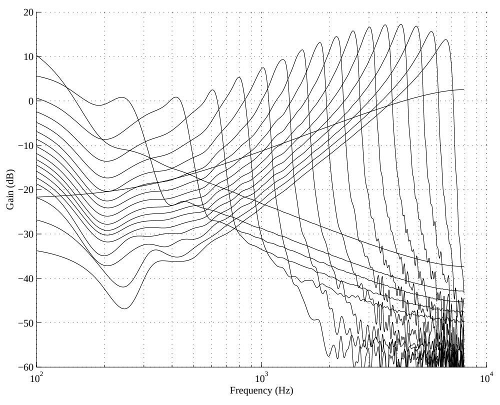
8.4.1 Implementations of the Cochlea Model

The cochlea implementations were tested on an Annapolis Wildstar Reconfigurable Computing Engine [Ann00] (Chapter 2.5) and *Pilchard*. The Wildstar platform contains three Xilinx Virtex XCV1000-6 FPGAs and has a PCI64 Local Bus interface, whereas the *Pilchard* board was populated with a Xilinx Virtex XCV1000-6 FPGA.

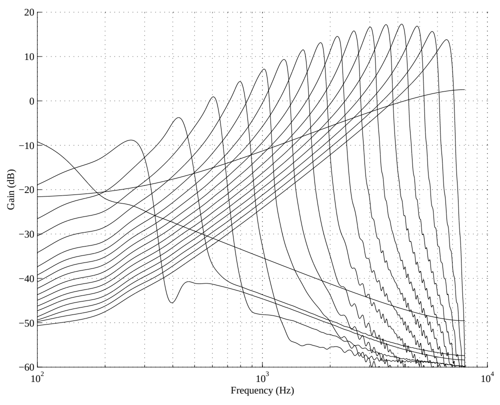
A series of cochlea implementations, with wordlengths from 10 to 32 bits and DA ROM width from 10 to 24 bits, were generated in order to present the tradeoff among wordlengths, widths of DA ROMs and precisions. The coefficients of these implementations were obtained from the Auditory Toolbox using the Matlab command `DesignLyonFilters(16000, 8, 0.25)`, which specifies a 16 kHz sampling rate, $Q = 8$ and a spacing which gives 88 biquadratic filters.

In order to present the improvement in precision with increasing wordlength and ROM width, the frequency responses of several different fixed-point implementations are plotted in Figure 8.6. A full set of results is presented at the end of this chapter. Figure 8.7 shows impulse and frequency responses obtained from a software floating-point implementation and a hardware 16-bit wordlength and 16-bit DA ROM width implementation.

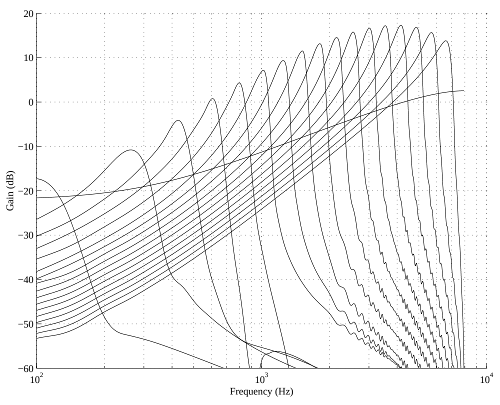
It can be observed that the filter accuracy gradually improves with increasing wordlength or DA ROM width. When wordlengths or DA ROM widths are too small, there are significant quantization effects that may result in oscillation or



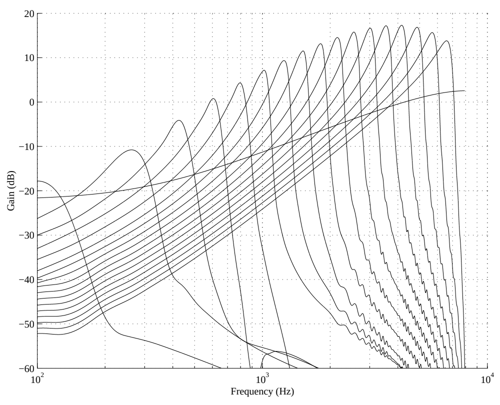
(a) 12-bit wordlength, 12-bit DA ROM implementation



(b) 16-bit wordlength, 16-bit DA ROM implementation

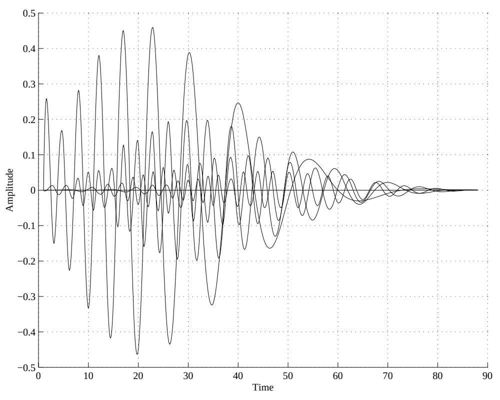


(c) 24-bit wordlength, 20-bit DA ROM implementation

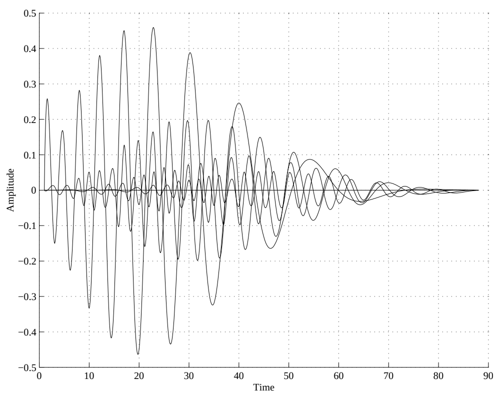


(d) 32-bit wordlength, 24-bit DA ROM implementation

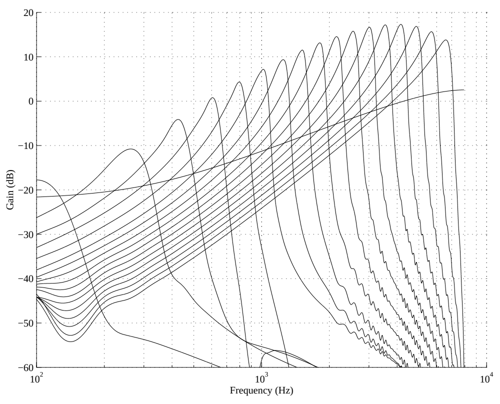
Figure 8.6: Frequency responses of cochlea implementations with different wordlength and width of DA ROMs.



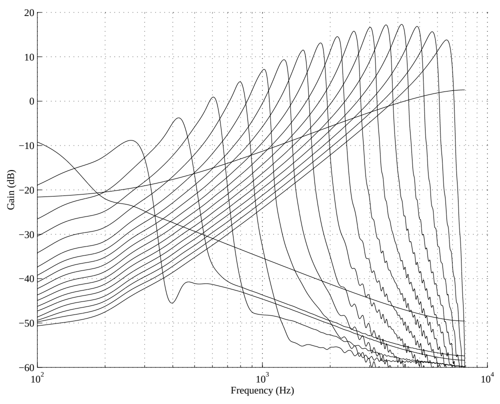
(a) impulse response of the software implementation



(b) impulse response of the hardware implementation



(c) frequency response of the software implementation



(d) frequency response of the hardware implementation

Figure 8.7: Impulse and frequency responses of software floating-point and hardware fixed-point (16-bit wordlength, 16-bit DA ROM width) implementations.

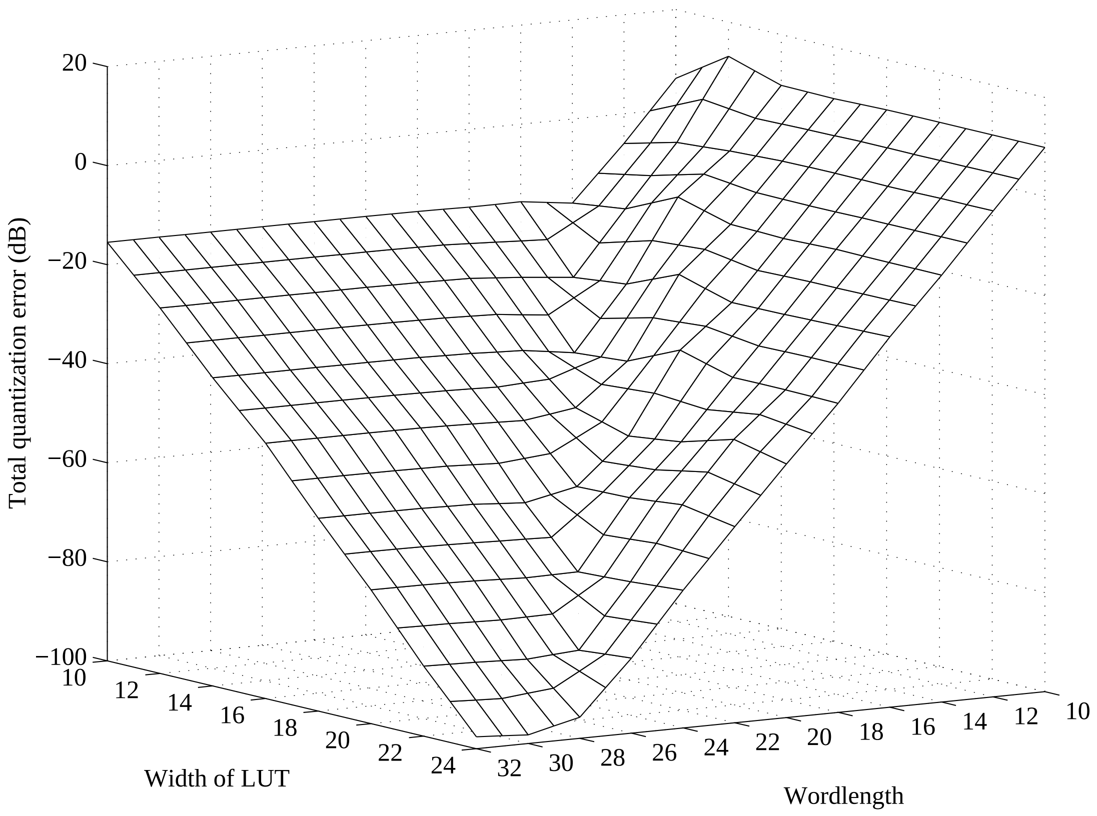


Figure 8.8: Mesh plot showing the quantization errors of implementations with varying wordlengths and DA ROM widths.

improper frequency responses at certain frequency intervals (as in Figure 8.6(a)).

Figure 8.8 shows the trend of improved quantization error with increasing wordlength and DA ROM width. With 24-bit wordlength and 16-bit DA ROM width, for example, the total quantization error is -48.73 dB. This degree of accuracy is sufficient for most speech applications. The total quantization error is the sum of the mean errors at every outputs (the filter sections).

Area requirements, maximum clock rates and maximum sampling rates of these implementations on a Xilinx Virtex XCV1000-6 FPGA, as reported by the Xilinx implementation tools, are shown in Tables 8.1 and 8.2. For each imple-

Wordlength	ROM Width			
	12 bits	16 bits	20 bits	24 bits
12 bits	5770	6582	7440	8340
16 bits	6160	6800	7589	8515
20 bits	6914	7343	7874	8602
24 bits	7620	8048	8578	9106
28 bits	8288	8748	9278	9805
32 bits	9297	9716	10245	10771

Table 8.1: Area requirements (measured in number of Virtex slices) of an 88-section cochlea implementation of different wordlengths and DA ROM width.

mentation, a timing constraint, determined by the corresponding wordlength and DA ROM width, was supplied to the P&R tools. As a bit-serial architecture was employed, the effective sampling rate of the implementations are their maximum clock rates divided by their wordlengths. With increasing wordlength or ROM width, an increase in area requirement and a general trend of decreasing maximum clock rate and sampling rate were observed.

8.4.2 Application to Speech Processing

A 24-bit wordlength, 16-bit DA ROM implementation was used to construct a cochleagram display application. This implementation was chosen because it is the smallest implementation that does not exhibit oscillation effects (refer to Figure 8.6 and Table 8.1).

The block diagram of the cochleagram display on the Wildstar platform is shown in Figure 8.9. On the Wildstar platform, the host PC accesses the FPGA via the LAD bus [Ann00]. The host PC writes input data into a buffer via the

Wordlength	ROM Width			
	12 bits	16 bits	20 bits	24 bits
12 bits	70.89, 5.91	68.03, 5.67	64.94, 5.41	63.91, 5.33
16 bits	67.74, 4.23	67.38, 4.21	61.60, 3.85	60.24, 3.77
20 bits	66.87, 3.34	65.60, 3.28	61.02, 3.05	59.79, 2.99
24 bits	66.15, 2.76	65.58, 2.73	60.53, 2.52	57.08, 2.38
28 bits	65.00, 2.32	63.13, 2.25	59.41, 2.12	57.01, 2.04
32 bits	64.96, 2.03	63.63, 1.99	58.00, 1.81	56.55, 1.77

Table 8.2: Maximum clock rates and corresponding sampling rates of an 88-section cochlea implementations (measured in MHz) for different wordlengths and ROM width (maximum clock rate, maximum sampling rate).

LAD bus. In this design, the buffer is implemented by dual-port BlockRAM (256 $\times$ 32-bit synchronous RAM) so as to provide a larger buffer to reduce the number of transactions. Minimizing the number of transactions helps to improve the system performance as the PCI Local Bus initialization and de-initialization overhead is reduced. The input data stored at the BlockRAM passes through a parallel to serial converter and enters the cochlea core. Each of the outputs of the cochlea core undergoes serial to parallel conversion followed by half-wave rectification to model the functionality of the inner hair cells. The outputs are accumulated to integrate its value over 256 samples. The accumulated output is read by the PC and displayed to obtain a cochleagram.

The design of the cochleagram display on the *Pilchard* platform is shown in Figure 8.10. Its core design is similar to that of the Wildstar platform, but its interface is simplified. Since the *Pilchard* interface does not have the overhead in setting up a transaction between the PC and the FPGA, the buffer (implemented

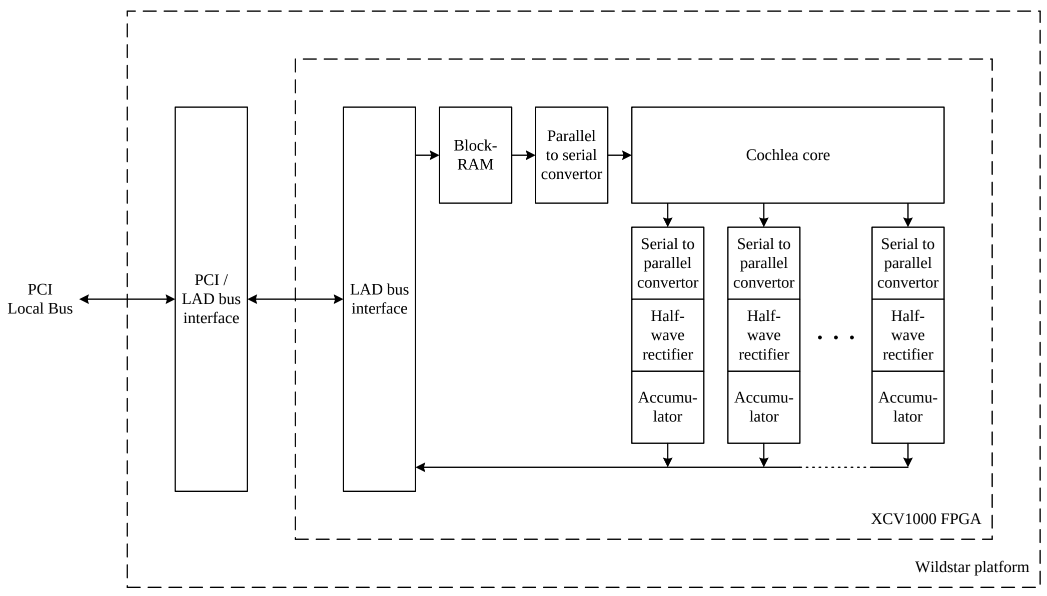


Figure 8.9: System architecture of the cochleagram display application on the Wildstar platform.

as BlockRAM) became unnecessary and was removed. On the *Pilchard* platform, the FPGA is directly connected to the memory bus, hence the SDRAM controller is implemented inside the FPGA. To ensure the samples are written to the board in the correct sequence, the UC MTRR mode (Section 5.3) was used.

The cochleagram display was tested with several different inputs. Figure 8.11 shows the cochleagrams produced from swept-sine wave and the Auditory Toolbox's "tapestry" inputs, the former being a 25 second linear chirp (400000 samples) and the latter the speech file of a woman saying "a huge tapestry hung in her hallway" (50380 samples).

In addition to the cochlea model, the cochleagram display consists of half-wave rectifiers, accumulators and interface. Due to limited hardware resources on a Xilinx XCV1000-6 FPGA, only the first 60 out of the 88 cochlea sections

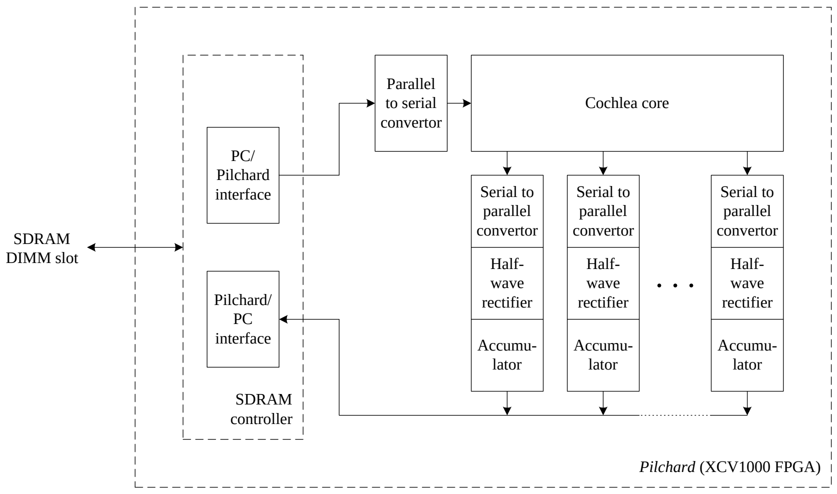
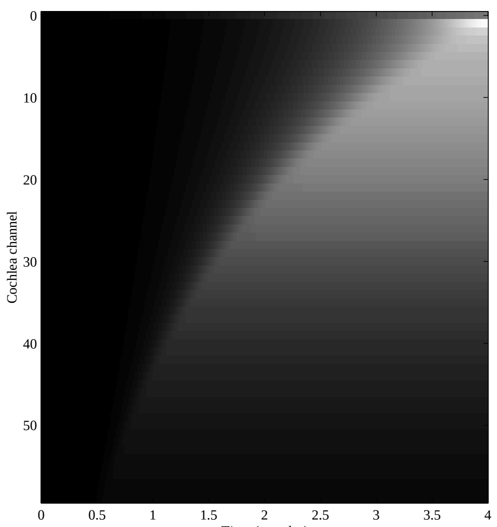
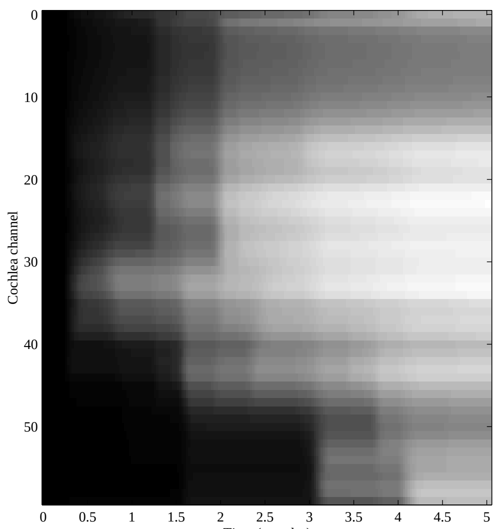


Figure 8.10: System architecture of the cochleagram display application on the *Pilchard* platform.



(a) swept-sine wave input



(b) “tapestry” input

Figure 8.11: Cochleagrams of swept-sine wave and “tapestry” inputs.

were used in order to reduce area requirements. The resultant cochleagram display application on the Wildstar platform requires 10344 slices and can be clocked at 52.51 MHz, yielding a sampling rate of 2.19 MHz (or 137 times faster than real time performance). The same implementation on the *Pilchard* platform requires 9912 slices and can be clocked at 60.36 MHz, yielding a sampling rate of 2.52 MHz (158 times real time performance). The reduced area and increased system clock rate of *Pilchard* is the result of a simplified interface. Including software and interfacing overheads, the measured throughput on the Wildstar and *Pilchard* platforms were 238 kHz and 398 kHz respectively (the *Pilchard* implementation is 1.67 times faster than the Wildstar implementation). As a comparison, the auditory toolbox achieves a 64 kHz throughput on a Sun Ultra-5 360 MHz machine. Hardware speedups on the Wildstar and *Pilchard* platforms are 3.72 and 6.22 times respectively.

It is interesting to compare the FPGA-based cochleagram system with a similar system developed in analogue VLSI by Lazzaro et. al. in 1994 [LWK94]. Using a 2 μm CMOS process, they integrated an 119-stage silicon cochlea (with a slightly more sophisticated hair cell model), non-volatile analogue storage and a sophisticated event-based communications protocol on a single 3.6 mm $\times$ 6.8 mm chip with a power consumption of 5 mW. The analogue VLSI version has improved density and power consumption compared with the FPGA approach. However, the FPGA version is vastly simpler; easier to modify; has a shorter design time; and is much more tolerant of supply voltage, temperature and transistor matching variations. Although qualitative results are not available, it is expected the FPGA version also has better filter accuracy; can operate at higher Q without instability; and has a wider dynamic range.

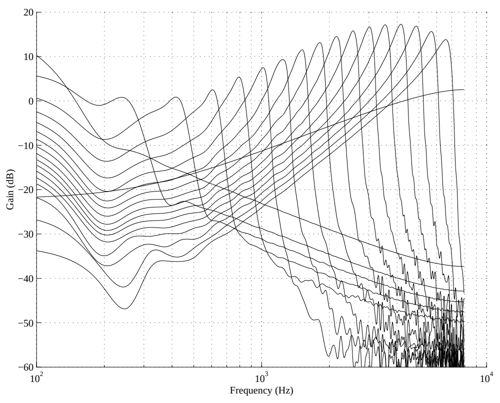
8.5 Summary

An FPGA-based implementation of Lyon and Mead's electronic cochlea filter and its application to a real-time cochleagram display were presented. Compared with analogue VLSI implementations, an FPGA implementation offers shorter design time, improved dynamic range, higher accuracy and a simpler computer interface.

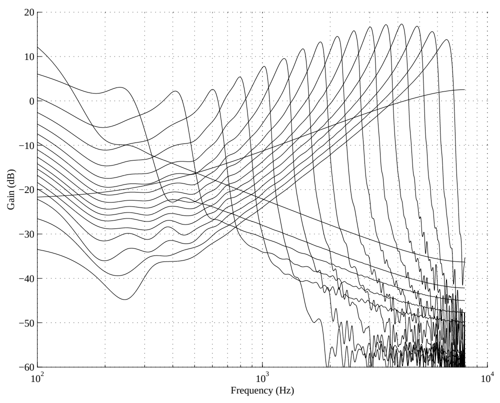
The FPGA cochlea filter is machine generated. Using the fixed-point modeling feature of *fp*, the module generator may take filter coefficients to compile an application-optimized design with arbitrary precision. Furthermore, the designer can choose the tradeoff between area and precision which is the most suitable for their implementation. The module generator can use simulation test vectors to determine the appropriate scaling for each filter.

The resulting model can be used as an accelerator for cochlea model research (the implementation on *Pilchard* is 6.22 times faster than the software implementation of the Auditory Toolbox on a Sun Ultra-5 360 MHz machine) or as the front end for embedded auditory signal processing systems. It is believe that there are many applications of the FPGA cochlea, some including for audio compression, speech recognition, audio and speech visualization, models of human auditory localization and models of bat localization.

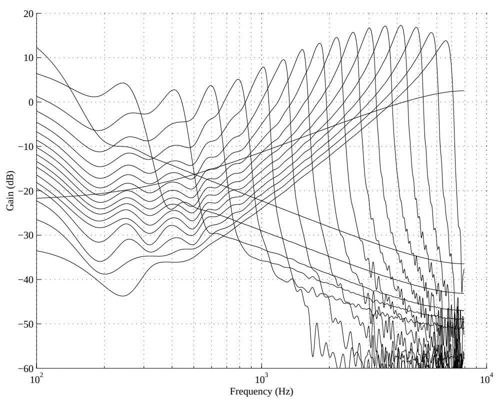
The cochlea filter model was applied to a cochleagram display application. This application has been tested on both the Annapolis Wildstar and *Pilchard* platforms. Results demonstrated increased system performance of *Pilchard* over conventional RC platforms with a PCI Local Bus interface, the *Pilchard* implementation being 1.67 times faster than the Wildstar implementation.



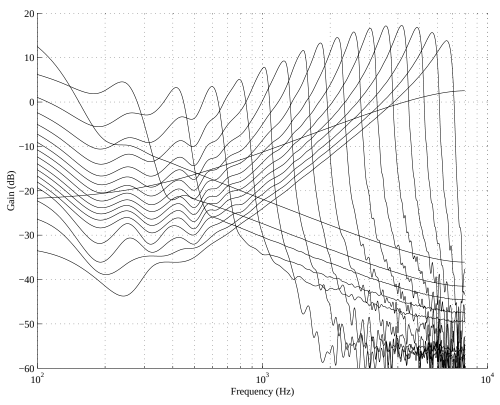
(a) 12-bit wordlength, 12-bit DA ROM implementation



(b) 12-bit wordlength, 16-bit DA ROM implementation

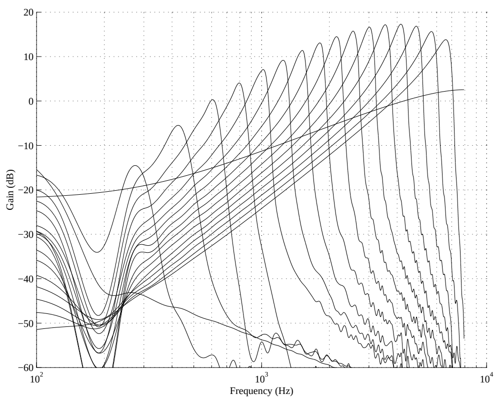


(c) 12-bit wordlength, 20-bit DA ROM implementation

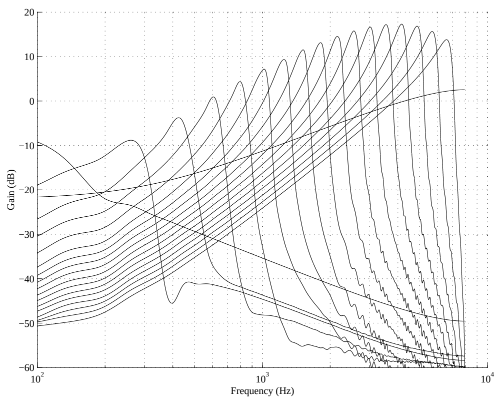


(d) 12-bit wordlength, 24-bit DA ROM implementation

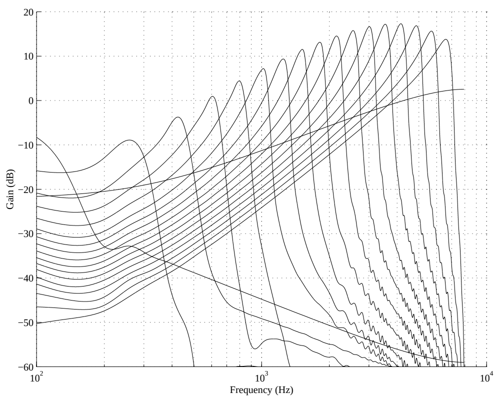
Figure 8.12: Frequency responses of cochlea implementations with different wordlengths and width of DA ROMs.



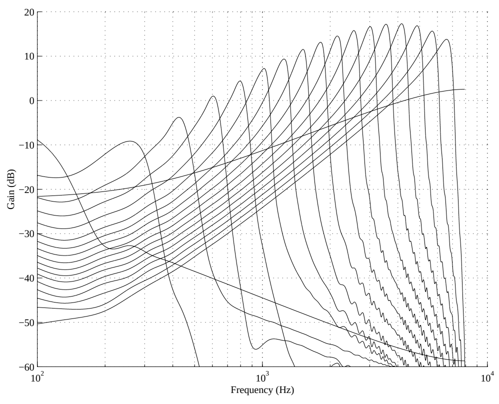
(e) 16-bit wordlength, 12-bit DA ROM implementation



(f) 16-bit wordlength, 16-bit DA ROM implementation

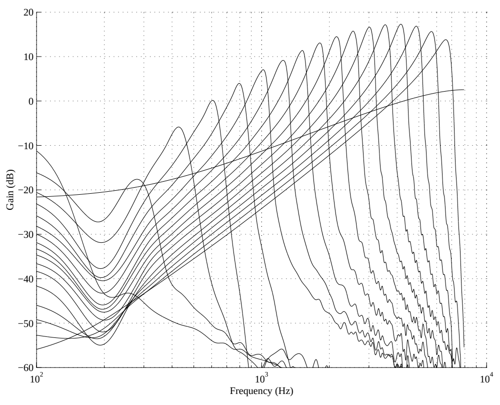


(g) 16-bit wordlength, 20-bit DA ROM implementation

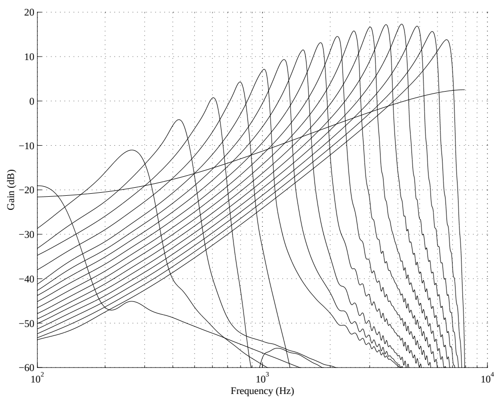


(h) 16-bit wordlength, 24-bit DA ROM implementation

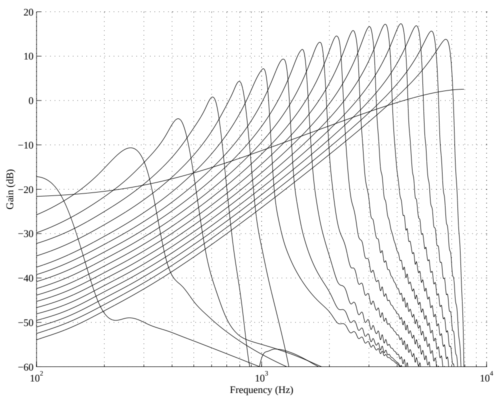
Figure 8.12 (continued): Frequency responses of cochlea implementations with different wordlengths and width of DA ROMs.



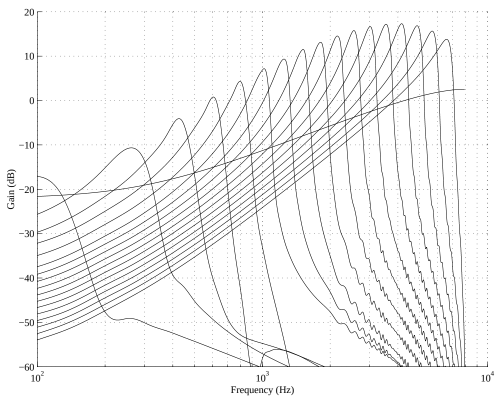
(i) 20-bit wordlength, 12-bit DA ROM implementation



(j) 20-bit wordlength, 16-bit DA ROM implementation

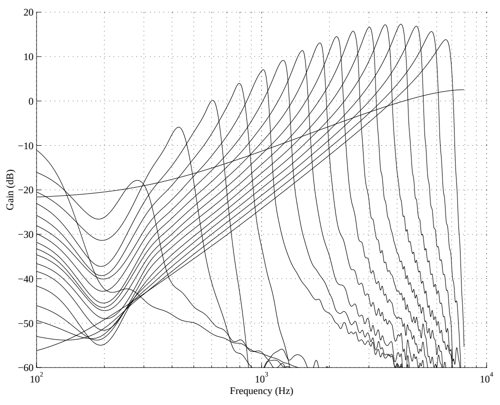


(k) 20-bit wordlength, 20-bit DA ROM implementation

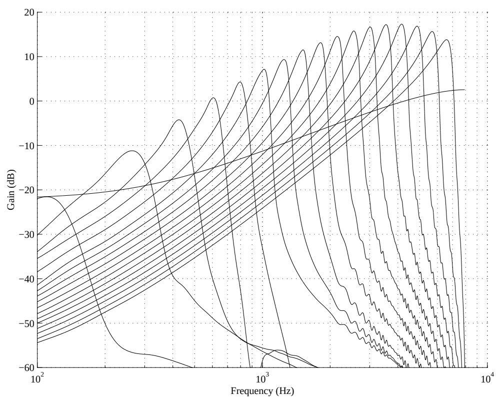


(l) 20-bit wordlength, 24-bit DA ROM implementation

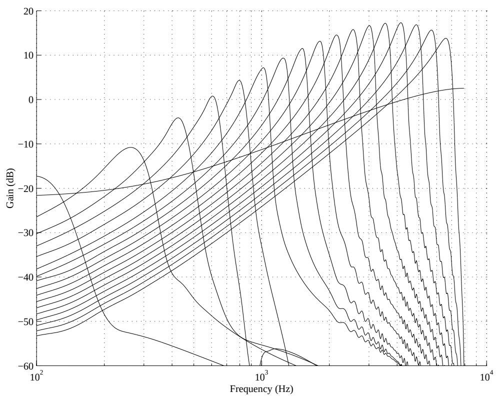
Figure 8.12 (continued): Frequency responses of cochlea implementations with different wordlengths and width of DA ROMs.



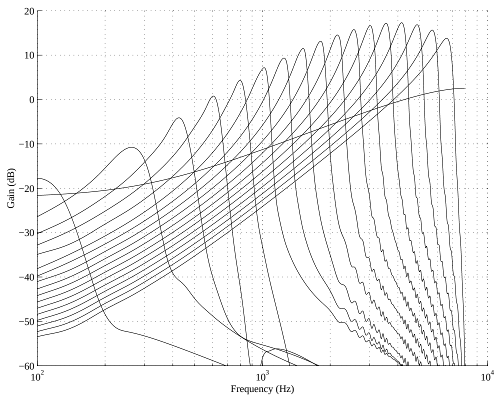
(m) 24-bit wordlength, 12-bit DA ROM implementation



(n) 24-bit wordlength, 16-bit DA ROM implementation

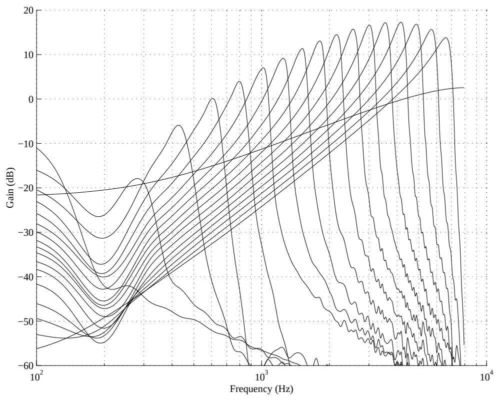


(o) 24-bit wordlength, 20-bit DA ROM implementation

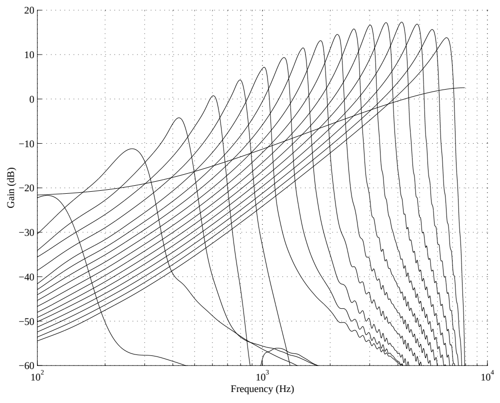


(p) 24-bit wordlength, 24-bit DA ROM implementation

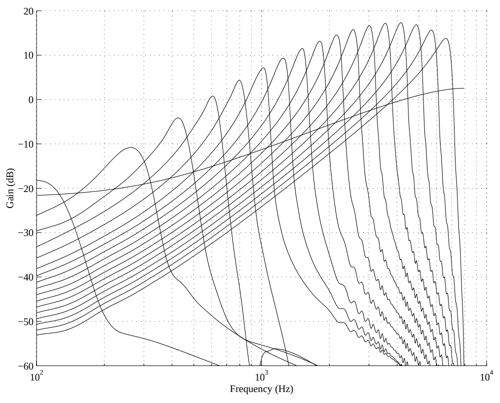
Figure 8.12 (continued): Frequency responses of cochlea implementations with different wordlengths and width of DA ROMs.



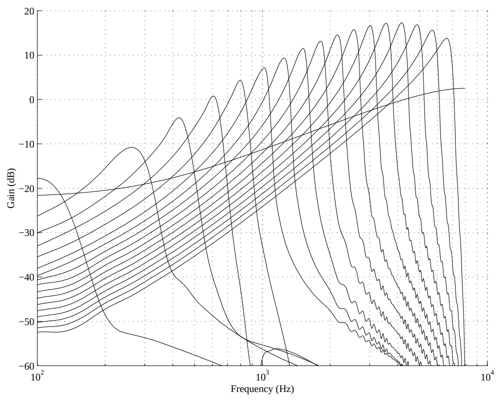
(q) 28-bit wordlength, 12-bit DA ROM implementation



(r) 28-bit wordlength, 16-bit DA ROM implementation

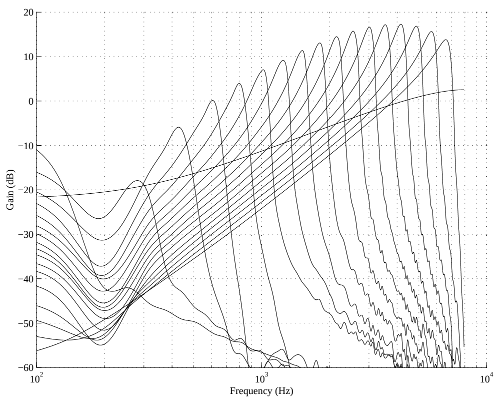


(s) 28-bit wordlength, 20-bit DA ROM implementation

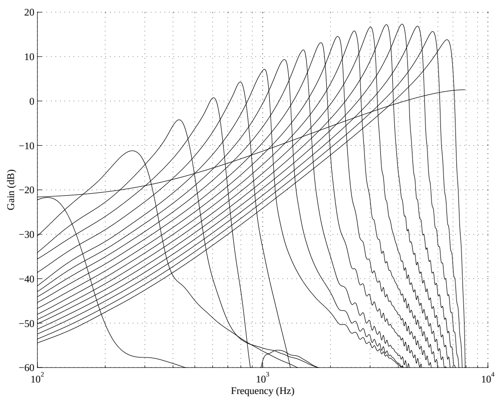


(t) 28-bit wordlength, 24-bit DA ROM implementation

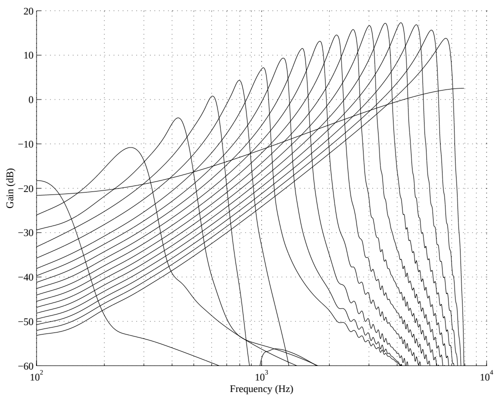
Figure 8.12 (continued): Frequency responses of cochlea implementations with different wordlengths and width of DA ROMs.



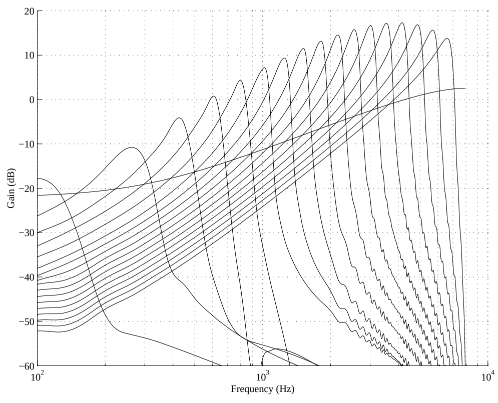
(u) 32-bit wordlength, 12-bit DA ROM implementation



(v) 32-bit wordlength, 16-bit DA ROM implementation



(w) 32-bit wordlength, 20-bit DA ROM implementation



(x) 32-bit wordlength, 24-bit DA ROM implementation

Figure 8.12 (continued): Frequency responses of cochlea implementations with different wordlengths and width of DA ROMs.

Chapter 9

Application IV – The Discrete Cosine Transform

In this chapter, the application of *fp* (Chapter 3) to a systolic structure for computing the Discrete Cosine Transform (DCT) is presented. This application utilizes the ability of *fp* to generate multiple wordlength and the variable-radix variable-wordlength architecture (Chapter 4). A series of implementations of the DCT were obtained from a single description. An almost continuous tradeoff among precision, area, latency and throughput were obtained through the use of the variable-radix variable-wordlength architecture, and designers can choose the most appropriate design via an optimization process.

This chapters begins with an introduction and background concerning the DCT. The algorithm of the systolic structure for computing the DCT which has been implemented is then described. The implementations and the results are presented afterwards. A discussion is also presented in this chapter.

9.1 Background

The DCT, proposed by Ahmed et. al. in 1974 [ANR74], has become an increasingly important task for image and video signal processing applications due to its utility and its adoption in standards such as JPEG (Joint Photographic Experts Group) [PM93], MPEG (Moving Picture Experts Group) [Gal91], CCITT H.261 [Lio91] and H.263 [ITU95]. Compared with other orthogonal transforms, the performance of the DCT is very similar to the optimal Karhunen-Loeve Transform (KLT) for highly correlated data [Har76, Lee84]. Its prominence makes hardware implementations important, particularly in high performance and low power applications.

Previously reported hardware implementations of the DCT in VLSI technology can be divided into two main streams: high-throughput and low-power. High-throughput designs include the 100 million samples per second, 100 MHz VLSI implementation in 0.8 μm CMOS technology reported by Uramoto et. al. [UIT+92], and the 150 million samples per second, 150 MHz VLSI implementation in 0.3 μm CMOS technology reported by Kuroda et. al. [KFM+96]. An example of a low-power design is the VLSI implementation in 0.6 μm CMOS technology by Xanthopoulos and Chandrakasan [XC00] which dissipates 4.38 mW at 14 MHz and 1.56 V, and has a maximum performance of taking 43 million samples per second at a clock rate of 43 MHz. Hunter and McCanny proposed a system which takes parameters such as point size and coefficient wordlengths to generate efficient designs for VLSI synthesis [HM98].

In recent years, FPGA technology has improved in density so that the DCT algorithm can be implemented on a single device. A Xilinx XC6200 series-based implementation was reported by Trainor, Heron and Woods [THW97] which achieves a performance of 15.36×10^6 pixels per second for 2D DCT image coding. Bergmann and Chung reported an implementation on a Xilinx XC4010 FPGA [BC97] which achieves 6.21×10^6 pixels/sec for 2D DCT image coding using the Fast DCT algorithm and 12.44×10^6 pixels/sec using Distributed Arithmetic. Kropp et. al. proposed a generator for pipelined multipliers on FPGAs and applied their work on the DCT [KRDP98]. A 2D DCT processor on an Altera FLEX10K100 FPGA was reported by Mohd-Yusof, Suleiman and Aspar [MYSA00], which achieves a throughput of 5.53×10^6 pixels per second at a clock rate of 11 MHz. Naviner et. al. reported a high accuracy implementation of the DCT on an Altera FLEX10K50 FPGA [NJLDGG00].

There are three main strategies which are employed in high performance implementations of the DCT. The number of arithmetic operations, particularly multiplications, can be reduced by converting the DCT to skew-circular convolutions [Lee91] or a direct sum of matrices [FW92]; parallelism is maximized by systolic structures [CJ90, CW91, HW95a] or distributed arithmetic (DA) [CS92, FCF93, BC97, PSB99]; and finally, the hardware resources required for arithmetic operations can be reduced by applying appropriate approximations, such as replacing multiplications by sequences of shift-and-adds [LO98, Tra00b].

9.2 Algorithm

Liu et. al. [LLCL98] proposed an approach which utilizes all three of the above strategies by transforming the DCT computation into a systolic computation involving discrete moments (DM). The bulk of computation is performed using addition. Specifically, the algorithm permits minimal area hardware implementations with reasonable accuracy and the systolic structure maps well to the adder and register-rich architecture of an FPGA.

The algorithm for computing the DCT proposed by Liu et. al. [LLCL98] centers around setting up the relationship between the DCT and DM. The N-point DCT is defined by the equation

$$X(k) = c_k \sum_{n=0}^{N-1} x(n) \cos \frac{\pi(2n+1)k}{2N}, \quad 0 \le k \le N-1, \tag{9.1}$$

where c_k equals $1/\sqrt{N}$ if $k = 0$, $\sqrt{2/N}$ otherwise.

For each pair of k and n, $0 \le k, n \le N-1$, there exists an integer i, $0 \le i \le N$, which satisfies

$$\cos \frac{\pi(2n+1)k}{2N} = \left(\cos \frac{i\pi}{2N}\right) \text{ or } \left(-\cos \frac{i\pi}{2N}\right).$$

Define $S(k,i)$ and $s(k,i)$ $(i, k = 0, 1, 2, \ldots, N-1)$ by

$$\begin{aligned} S(k,i) &= \{j \mid \cos \frac{\pi(2j+1)k}{2N} = \cos \frac{i\pi}{2N}, \quad 0 \le j \le N-1\}, \\ s(k,i) &= \{j \mid \cos \frac{\pi(2j+1)k}{2N} = -\cos \frac{i\pi}{2N}, \quad 0 \le j \le N-1\}, \end{aligned}$$

and $x_{k,i}$ $(i, k = 0, 1, 2, \ldots, N-1)$ by

$$x_{k,i} = \begin{cases} \sum_{j \in S(k,i)} x(j) - \sum_{j \in s(k,i)} x(j) & \text{if } S(k,i) \cup s(k,i) \ne \emptyset, \\ 0 & \text{otherwise.} \end{cases} \tag{9.2}$$

Substitute them into Equation 9.1 yields

$$\begin{aligned} X(k) &= c_k \sum_{n=0}^{N-1} x(n) \cos \frac{\pi(2n+1)k}{2N} \\ &= c_k \sum_{n=0}^{N-1} x_{k,i} \cos \frac{\pi i}{2N}, \quad 0 \le k \le N-1. \end{aligned} \tag{9.3}$$

By applying the theorem of extended law of the mean to $\cos(\pi i/N)$, for $0 \le i \le N-1$,

$$\cos \frac{\pi i}{2N} = 1 - \frac{(\pi i/2N)^2}{2!} + \ldots + (-1)^p \frac{(\pi i/2N)^{2p}}{(2p)!} + R_i, \tag{9.4}$$

where R_i is the Taylor remainder term,

$$R_i = \cos\frac{\xi_i + (2p+1)\pi}{2}\frac{(\pi i/2N)^{2p+1}}{(2p+1)!}, \quad 0 \le \xi_i \le \frac{\pi i}{2N}. \tag{9.5}$$

Recalling that N-point DM is defined by the Equation

$$m_p = \sum_{i=1}^{n} f_i i^p. \tag{9.6}$$

Substituting Equation 9.4 into Equation 9.3 and simplifying yields

$$X(k) = c_k\left(x_{k,0} + \sum_{r=0}^{p} a_r m_{k,2r}\right) + R_p, \quad 0 \le k \le N-1, \tag{9.7}$$

where

$$\begin{aligned}
a_r &= \frac{(-1)^r \pi^{2r}}{(2N)^{2r}(2r)!},\\
m_{k,2r} &= \sum_{i=1}^{N-1} x_{k,i} i^{2r},\\
R_p &= c_k \sum_{i=1}^{N-1} x_{k,i}\cos\frac{\xi_i + (2p+1)\pi}{2}\frac{(\pi i/2N)^{2p+1}}{(2p+1)!}, \quad 0 \le \xi_i \le \frac{\pi i}{2N}.
\end{aligned}$$

Ignoring R_p, Equation 9.7 establishes the relationship between the DCT and DM by referring to Equation 9.6. If $m_{k,r}$ $(k = 0, 1, \ldots, N-1; r = 0, 1, \ldots, p)$ are available, then each $X(k)$ $(k = 0, 1, \ldots, N-1)$ can be obtained by computing the dot product of these moments and the constant vector a_r, followed by an addition with $x_{k,0}$ and a multiplication with c_k.

To compute $m_{k,r}$, a transformation known as F_p is used. The transformation is conducted by an architecture called the p-network which resembles a Pascal triangle. Figure 9.1 illustrates a p-network for the transformation

$$F_p(1, x, x^2, \ldots, x^{p-1}, x^p) = (1, (1+x), (1+x)^2, \ldots, (1+x)^{p-1}, (1+x)^p).$$

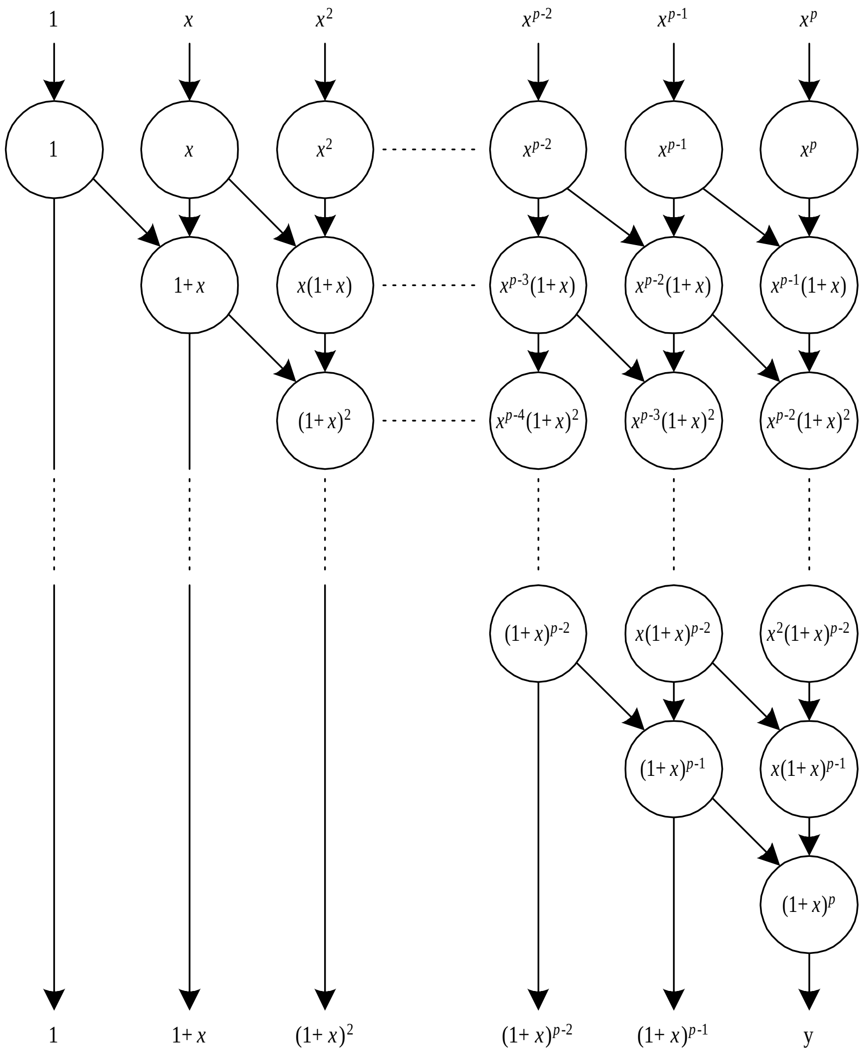


Figure 9.1: The p-network with input vector $(1, x, x^2, \ldots, x^{p-1}, x^p)$, nodes represent additions.

$F_p(a, ax, ax^2, \ldots, ax^{p-1}, ax^p)$ and $F_p(a+b, a+b, a+b, \ldots, a+b, a+b)$ are two special cases of transformation F_p,

$$\begin{aligned} & F_p(a, ax, ax^2, \ldots, ax^{p-1}, ax^p) \\ & \quad = (a, a(1+x), a(1+x)^2, \ldots, a(1+x)^{p-1}, a(1+x)^p), \\ & F_p(a+b, a+b, a+b, \ldots, a+b, a+b) \\ & \quad = F_p(a, a, a, \ldots, a, a) + F_p(b, b, b, \ldots, b, b), \end{aligned} \tag{9.8}$$

and in general,

$$\begin{aligned} & F_p^{n-1}(1, x, x^2, \ldots, x^{p-1}, x^p) \\ & \quad = F_p(F_p^{n-2}(1, x, x^2, \ldots, x^{p-1}, x^p)) \\ & \quad = F_p(\ldots F_p(1, x, x^2, \ldots, x^{p-1}, x^p) \ldots) \\ & \quad = (1, (n-1+x), (n-1+x)^2, \ldots, (n-1+x)^{p-1}, (n-1+x)^p). \end{aligned} \tag{9.9}$$

Referring to Equation 9.9 and by substitution,

$$\begin{aligned} F_p^{n-1}(1, 1, 1, \ldots, 1, 1) &= (1, n, n^2, \ldots, n^{p-1}, n^p), \\ F_p^{n-1}(a, a, a, \ldots, a, a) &= (a, na, n^2a, \ldots, n^{p-1}a, n^pa). \end{aligned}$$

Let $\mathbf{a}_i$ $(i = 1, 2, 3, \ldots, n)$ be a $(p+1)$-dimensional vector with all its values being a_i $(\mathbf{a}_i(a_i, a_i, a_i, \ldots, a_i, a_i))$, from Equation 9.8,

$$\begin{aligned} F_p(F_p(\mathbf{a}_n) + \mathbf{a}_{n-1}) &= F_p(F_p(\mathbf{a}_n)) + F_p(\mathbf{a}_{n-1}) \\ &= F_p^2(\mathbf{a}_n) + F_p(\mathbf{a}_{n-1}). \end{aligned} \tag{9.10}$$

By recursive application of Equation 9.10, the Equation for the p-order moment, $m_p(a_n, a_{n-1}, a_{n-2}, \ldots, a_2, a_1)$, is computed.

$$\begin{aligned}
m_p &= F_p(F_p \ldots (F_p(F_p(F_p(\mathbf{a}_n) + \mathbf{a}_{n-1}) + \mathbf{a}_{n-2}) + \ldots + \mathbf{a}_2) + \mathbf{a}_1) \\
&= F_p^{n-1}(\mathbf{a}_n) + F_p^{n-2}(\mathbf{a}_{n-1}) + F_p^{n-3}(\mathbf{a}_{n-2}) + \ldots + F_p^2(\mathbf{a}_3) + F_p(\mathbf{a}_2) + \mathbf{a}_1 \\
&= \left(\sum_{i=1}^{n} a_i, \sum_{i=1}^{n} a_i i, \sum_{i=1}^{n} a_i i^2, \ldots, \sum_{i=1}^{n} a_i i^{p-1}, \sum_{i=1}^{n} a_i i^p\right).
\end{aligned}$$

Therefore, the computation of DM consists of n-1 F_p transformations and a vector addition. The relationship between the DCT and DM decomposes the computation of the DCT into three steps:

1. **Pre-computation:** $x_{k,i}$ $(i, k = 0, 1, 2, \ldots, N-1)$,
2. **Computation of DM:** $m_{k,2r}$ $(r = 0, 1, 2, \ldots, p;\ k = 0, 1, 2, \ldots, N-1)$, and
3. **Post-computation:** $X(k)$ $(k = 0, 1, 2, \ldots, N-1)$.

Computation of $x_{k,i}$ from x_k is done using Equation 9.2. For instance, when $N = 8$,

$$\begin{aligned}
x_{0,0} &= x(0) + x(1) + x(2) + x(3) + x(4) + x(5) + x(6) + x(7), \\
x_{0,1} &= 0, \\
x_{0,2} &= 0, \\
&\vdots \\
x_{1,0} &= 0, \\
x_{1,1} &= x(0) - x(7), \\
&\vdots \\
x_{7,7} &= x(2) - x(5),
\end{aligned}$$

which can be implemented by registers, adders and subtracters. Computation of DM is implemented using N-1 p-networks each with $2p+1$ adders at the top level, and $2p+1$ adders at the output of the cascaded p-network. Finally, $X(k)$ is computed from Equation 9.7 where a_r and c_k are pre-computed constants and hence its implementation can use a constant multiplier. To deliver a systolic structure, pipelined multipliers should be used. Figure 9.2 shows the overall systolic structure for computing the 1D DCT.

Referring to Equation 9.5, it can be observed that the error term converges to zero rapidly as p increases. The structure offers a smaller area implementation which can be traded off with R_p. The choice of p should be around two to four for a practical hardware implementation. Hardware requirements and the associated error term R_p for some typical values of p and N are listed in Table 9.1. The numbers of adders and multipliers refer to the operators in the systolic array and the post-computation unit only and does not account for those in the pre-computation unit. R_p was calculated by assuming input values in the range of $[-8.0, 8.0)$.

9.3 Implementation

The implementation of the systolic structure for computing the DCT described in Section 9.2 was divided into two parts. The first part is the pre-computation unit which was developed in VHDL. As suggested by Liu et. al. [LLCL98], the pre-computation unit can be implemented with a special linear array. This linear array can be more efficiently implemented with a manual design in which the RTL behavior is specified explicitly.

The second part of the implementation is the cascade of p-network and the post-processing unit. This part is described in an *fp* algorithmic description, as

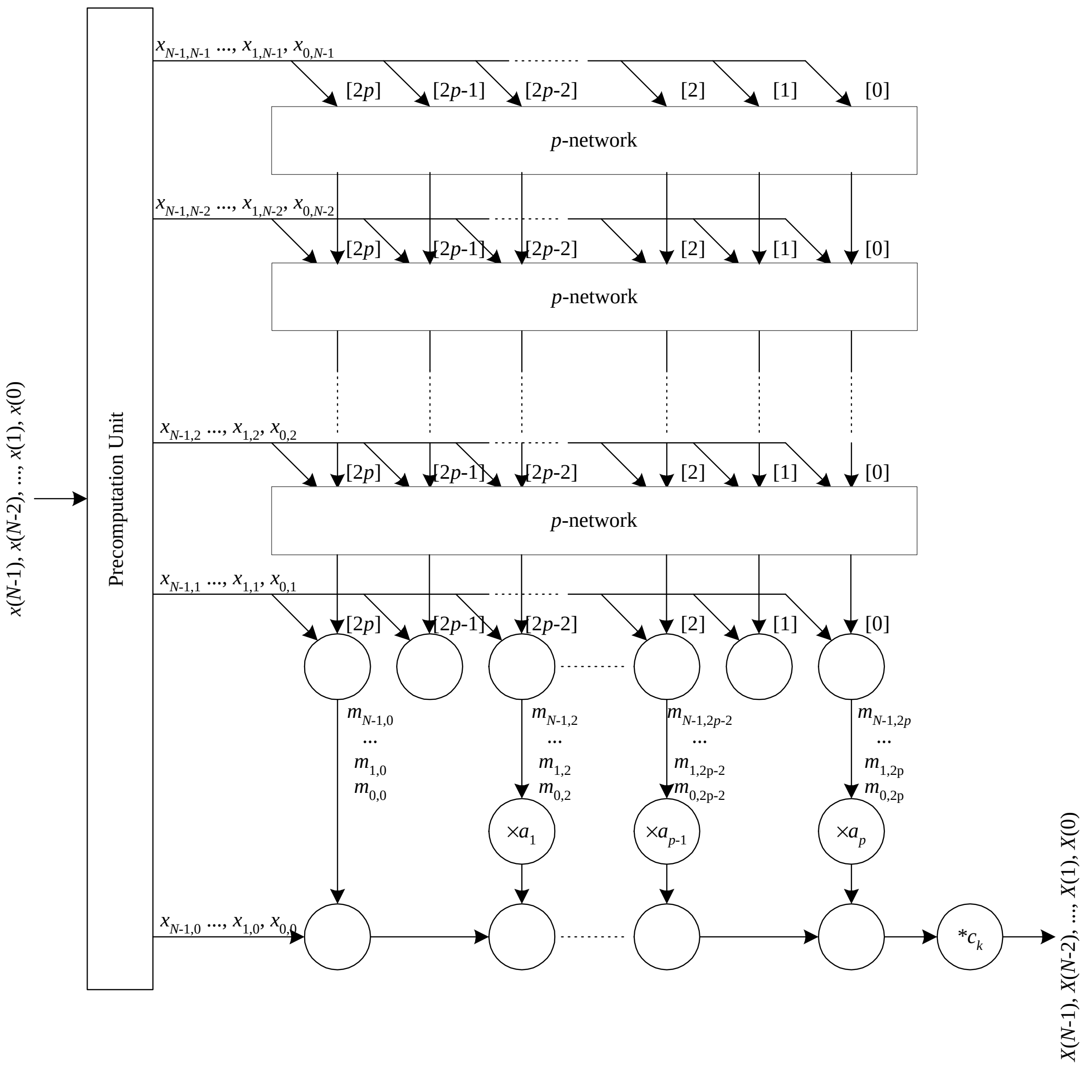

x(N-1), x(N-2), ..., x(1), x(0)
Precomputation Unit
x_{N-1,N-1} ..., x_{1,N-1}, x_{0,N-1}
[2p] [2p-1] [2p-2] [2] [1] [0]
p-network
x_{N-1,N-2} ..., x_{1,N-2}, x_{0,N-2}
x_{N-1,2} ..., x_{1,2}, x_{0,2}
x_{N-1,1} ..., x_{1,1}, x_{0,1}
m_{N-1,0} ... m_{1,0} m_{0,0}
m_{N-1,2} ... m_{1,2} m_{0,2}
m_{N-1,2p-2} ... m_{1,2p-2} m_{0,2p-2}
m_{N-1,2p} ... m_{1,2p} m_{0,2p}
×a_1
×a_{p-1}
×a_p
x_{N-1,0} ..., x_{1,0}, x_{0,0}
*c_k
X(N-1), X(N-2), ..., X(1), X(0)

Figure 9.2: The systolic array for 1D N-point DCT, with p-network blocks as shown in Figure 9.1 and blank nodes represent additions.

p	N	Number of adders	Number of multipliers	R_p	$R_p/\max(x)$
2	4	27	4	1.80324	0.22541
2	8	87	4	2.55016	0.31877
2	16	207	4	3.60648	0.45081
2	32	447	4	5.10033	0.63754
3	4	45	5	0.10594	0.01324
3	8	157	5	0.14982	0.01873
3	16	381	5	0.21187	0.02648
3	32	829	5	0.29963	0.03745
4	4	67	6	0.00363	0.00045
4	8	247	6	0.00513	0.00064
4	16	607	6	0.00726	0.00091
4	32	1327	6	0.01027	0.00128

Table 9.1: The number of adders and multipliers and the associated error bound of the systolic array for computing the DCT for different values of p and N.

shown in Figure 9.3. The DCT algorithm was originally targeted for VLSI where regularity is of the utmost importance. However, for VHDL-synthesized FPGA implementations, regularity is slightly important. Therefore, a simplification of the p-network at the top-level was applied. Since the inputs to the p-network at the top-level are the same variable, the outputs of the first-level adders and subsequently the following levels of adders are of the same value as their neighboring adders. Therefore, the removal of all adders except those along the diagonal can be applied. This simplification is depicted in Figure 9.4. The algorithmic description takes the outputs from the pre-processing unit which contain N values. It outputs two values, namely

$$X_1(k) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x(n) \cos \frac{\pi(2n+1)k}{2N}$$

and

$$X_2(k) = \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} x(n) \cos \frac{\pi(2n+1)k}{2N}.$$

Since $X(k) = X_1(k)$ if $k = 0$, $X(k) = X_2(k)$ if $k = 1, 2, \ldots, N-1$, these two outputs are connected to a multiplexer which is controlled by a simple counter to output the correct $X(k)$. *fp* typically generates 50000 lines of VHDL code from the 33-line algorithmic description of the DCT algorithm.

The designs were tested on the Wildstar platform [Ann00] (Chapter 2.5). The results presented in Section 9.4 were obtained using only one of the three Xilinx Virtex XCV1000-6 FPGAs on the Wildstar platform.

9.4 Results

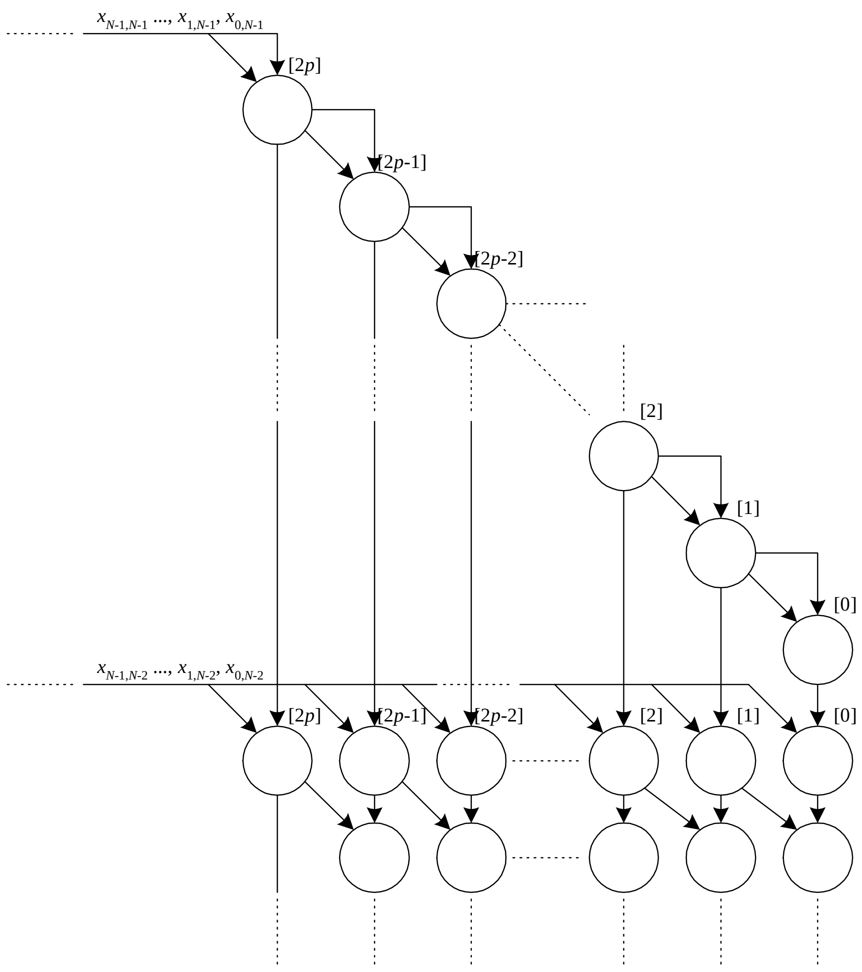
For all of the experiments, $p = 3$ and $N = 8$. The systolic structure thus generated consisted of 157 adders and 5 constant multipliers. The sample dataset consists of

```
#define N 8
#define p 3
void dct(fixed &xp, fixed *X)
                    // xp is the pre-processed inputs, X is the outputs
{
    fixed x[2 * p + 1];
    double a;
    int i, j, k;
    x[2 * p] = xp[N - 1] + xp[N - 1];
    for (i = 2 * p - 1; i >= 0; i--)
        x[i] = x[i + 1] + x[i + 1];
                          // the simplified p-network at the top-level
    for (i = N - 2; i > 1; i--)
    {
        for (j = 0; j <= 2 * p; j++)
            x[j] += xp[i];
        for (j = 2 * p - 1; j >= 0; j--)
            for (k = 0; k <= j; k++)
                x[k] += x[k + 1];
    }                                  // the intermediate p-networks
    for (i = 0; i <= 2 * p; i += 2)
        x[k] += xp[1];
                       // the adders immediately after the p-networks
```

Figure 9.3: The *fp* algorithmic description of the systolic array for computing the DCT.

```
    X[0] = xp[0] + x[2 * p];
    k = 1;
    for (i = 2 * p - 2; i >= 0; i -= 2)
    {
        a = pow(M_PI, (double) (2 * k)) / pow(2 * N);
        for (j = 2; j < 2 * k; j++)
            a /= (double) j;
        if (k++ % 2)
            a = -a;         // compute multiplier coefficients a_r
        X[0] += a * x[i];
    }                    // the bottom-most adders and multipliers
    X[1] = X[0] * sqrt(2 / N);                       // X_1(k)
    X[0] /= sqrt(N);                                 // X_0(k)
}
```

Figure 9.3 (continued): The *fp* algorithmic description of the systolic array for computing the DCT.



$x_{N-1,N-1}$ ..., $x_{1,N-1}$, $x_{0,N-1}$
[2p]
[2p-1]
[2p-2]
[2]
[1]
[0]
$x_{N-1,N-2}$ ..., $x_{1,N-2}$, $x_{0,N-2}$
[2p]
[2p-1]
[2p-2]
[2]
[1]
[0]

Figure 9.4: The simplification made to the p-network at the top-level in the cascade of p-networks.

500 entries, 200 of which were extracted from image data (normalized in the range of $[0.0, 1.0)$), another 200 were the DCT of image data (in the range of $[-8.0, 8.0)$), and the remaining were randomly generated (in the range of $[-8.0, 8.0)$). The 200 entries of transformed image data were included because for 2D image coding experiments (Section 9.4.2), the DCT implementation is used twice to encode the input images.

The objective of the experiments was to produce designs which minimize area subjected to certain error constraints. Nevertheless, it is also possible to minimize the output errors under area constraints with *fp*. In the experiments, the sample dataset are assumed free of external quantization error. The error constituted by R_p described in Equation 9.5 was neglected during area minimization (Section 9.4.1). However, R_p was taken into account when the image coding experiment was conducted (Section 9.4.2).

9.4.1 Area Minimization

The *fp* optimization feature was applied on the algorithmic description shown in Figure 9.3. Error constraints (mean error) at the output were specified and the optimization objective was to minimize the area of the implementation. Note that the operators being adders and multipliers only enabled the algorithm to be efficiently implemented with a variable-radix variable-wordlength architecture.

There are two sets of results presented in this section. Results in Tables 9.2 and 9.3 were obtained by fixing the error constraints and varying the number of digits n (recall from Section 4.2 that n is a global parameter for each variable-radix variable-wordlength implementation). Results in Tables 9.4 and 9.5 were obtained by fixing n and varying the error constraints. More precisely, the error constraints were set to $1/256 = 3.9063 \times 10^{-3}$ and $1/64 = 1.5625 \times 10^{-2}$ for the results in Table 9.2 and 9.3 respectively; n was set to one (bit-parallel) and eight (digit-serial, with variable-radix variable-wordlength architecture) to obtain the results in Tables 9.4 and 9.5 respectively. In the tables, the performance is defined as the number of DCT operations per second, and the performance to area ratio indicates the area efficiency which is measured by dividing the performance by area.

To better illustrate the tradeoff among various conflicting performance measures, the number of slices (Figure 9.5), frequencies (Figure 9.6), performance (Figure 9.7) and performance to area ratio (Figure 9.8) against the number of digits under different error constraints are plotted. It is observable from the plots that the relationship between the number of digits n and area is not straightforward. These graphs can be used to determine the minimal area implementation satisfying certain error and throughput requirements. To more accurately locate the optimal point, both error and throughput constraints should be specified. For instance, Table 9.6 shows a series of implementations with mean error constraints $1/256 = 3.9063 \times 10^{-3}$ and varying throughput constraints.

9.4.2 Image Coding

To further evaluate the variable-radix variable-wordlength DCT implementation, the DCT implementation was applied to the coding of several benchmark images. The experimental framework is shown in Figure 9.9. The 1D DCT was first applied on the input images using the DCT core. Due to the separability of the DCT core, the 2D DCT can be computed using the row-column method. Therefore, the 1D DCT results were transposed and then another 1D DCT using the same hardware was applied, followed by another transposition. In this framework, only the fixed-point 1D DCTs were performed in hardware and transposition was carried

Number of digits n	1	2	4	6	8
Maximum error ($\times 10^{-3}$)	20.277	18.271	40.161	40.161	16.725
Mean error ($\times 10^{-3}$)	3.726	2.303	3.621	3.057	2.190
SNR ($\times 10^{2}$)	1.301	1.955	1.189	1.278	2.069
Average digit size	12.29	7.59	4.02	3.44	2.64
Maximum digit size†	39	15	9	7	5
Minimum digit size†	6	4	3	2	2
Average wordlength	21.86	20.33	19.66	19.76	18.65
Maximum wordlength	39	53	60	48	42
Minimum wordlength	6	9	9	9	9
Trimmed adders	0	0	0	0	0
Trimmed multipliers	0	0	0	0	0
Amount of resources (slices)	1769	2146	1553	1373	1167
Clock rate (MHz)	66.18	64.30	55.85	74.11	95.56
Throughput (clock cycles)	1	2	4	6	8
Latency (clock cycles)	46	58	69	83	85
Performance ($\times 10^{6}$)	66.18	32.15	13.96	12.35	11.95
Performance/area ($\times 10^{3}$)	37.41	14.98	8.99	8.90	10.24

Table 9.2: Digit-serial DCT implementations with different digit sizes obtained by area minimization with output mean error constraint set to 3.9063×10^{-3}. (†rounded up values)

Number of digits n	12	16	24	32	48
Maximum error ($\times 10^{-3}$)	40.161	40.161	36.255	8.558	16.724
Mean error ($\times 10^{-3}$)	3.324	3.074	3.344	2.241	2.243
SNR ($\times 10^{2}$)	1.250	1.359	1.294	2.235	2.037
Average digit size	2.14	1.76	1.29	1.08	1.00
Maximum digit size†	3	3	2	2	1
Minimum digit size†	1	1	1	1	1
Average wordlength	20.76	22.17	22.67	20.58	21.61
Maximum wordlength	39	42	48	49	48
Minimum wordlength	9	10	9	9	10
Trimmed adders	0	0	0	0	0
Trimmed multipliers	0	0	0	0	0
Amount of resources (slices)	1188	1369	1049	618	771
Clock rate (MHz)	89.60	92.40	96.76	113.11	107.95
Throughput (clock cycles)	12	16	24	32	48
Latency (clock cycles)	101	104	125	83	94
Performance ($\times 10^{6}$)	7.47	5.78	4.03	3.54	2.25
Performance/area ($\times 10^{3}$)	6.29	4.22	3.84	5.72	2.92

Table 9.2 (continued): Digit-serial DCT implementations with different digit sizes obtained by area minimization with output mean error constraint set to 3.9063×10^{-3}. (†rounded up values)

Number of digits n	1	2	4	6	8
Maximum error ($\times 10^{-3}$)	67.505	77.127	59.855	91.105	67.505
Mean error ($\times 10^{-3}$)	8.302	13.249	13.988	13.512	8.321
SNR ($\times 10^{2}$)	0.579	0.450	0.424	0.423	0.579
Average digit size	10.63	5.09	2.88	2.16	1.99
Maximum digit size†	27	14	8	6	4
Minimum digit size†	6	4	2	2	1
Average wordlength	14.34	13.83	14.68	15.02	14.49
Maximum wordlength	27	32	32	32	36
Minimum wordlength	6	7	7	8	8
Trimmed adders	117	117	117	117	117
Trimmed multipliers	2	2	2	2	2
Amount of resources (slices)	780	842	845	657	691
Clock rate (MHz)	83.04	70.86	57.14	91.80	96.19
Throughput (clock cycles)	1	2	4	6	8
Latency (clock cycles)	21	29	41	50	68
Performance ($\times 10^{6}$)	83.04	35.43	14.29	15.30	12.02
Performance/area ($\times 10^{3}$)	106.47	42.08	16.91	23.29	17.40

Table 9.3: Digit-serial DCT implementations with different digit sizes obtained by area minimization with output mean error constraint set to 1.5625×10^{-2}. (†rounded up values)

Number of digits n	1	2	4	6	8
Maximum error ($\times 10^{-3}$)	73.474	73.685	67.505	83.130	67.505
Mean error ($\times 10^{-3}$)	11.863	13.559	6.148	12.467	8.662
SNR ($\times 10^{2}$)	0.508	0.414	0.733	0.494	0.604
Average digit size	1.37	1.07	1.05	1.00	1.00
Maximum digit size†	3	2	2	1	1
Minimum digit size†	1	1	1	1	1
Average wordlength	16.00	15.76	15.75	15.70	16.06
Maximum wordlength	36	32	35	32	40
Minimum wordlength	7	9	9	6	9
Trimmed adders	117	117	117	117	117
Trimmed multipliers	2	2	2	2	2
Amount of resources (slices)	633	540	465	385	394
Clock rate (MHz)	104.40	115.38	120.67	126.25	111.60
Throughput (clock cycles)	12	16	24	32	48
Latency (clock cycles)	64	73	45	55	49
Performance ($\times 10^{6}$)	8.70	7.21	5.03	3.95	2.33
Performance/area ($\times 10^{3}$)	13.74	13.35	10.81	10.25	5.90

Table 9.3 (continued): Digit-serial DCT implementations with different digit sizes obtained by area minimization with output mean error constraint set to 1.5625×10^{-2}. (†rounded up values)

Mean error constraint ($\times 10^{-2}$)	0.1953	0.3906	1.5625	6.2500
Maximum error ($\times 10^{-2}$)	0.856	2.028	6.751	22.224
Mean error ($\times 10^{-2}$)	0.195	0.373	0.832	3.930
SNR ($\times 10^{2}$)	2.501	1.301	0.579	0.151
Average digit size	26.71	12.29	9.59	8.94
Maximum digit size†	72	39	27	19
Minimum digit size†	12	6	6	5
Average wordlength	26.84	16.33	14.34	12.57
Maximum wordlength	72	39	27	19
Minimum wordlength	12	6	6	5
Trimmed adders	0	0	117	117
Trimmed multipliers	0	0	2	2
Amount of resources (slices)	4013	1769	780	651
Clock rate (MHz)	58.29	66.18	83.04	120.50
Throughput (clock cycles)	1	1	1	1
Latency (clock cycles)	47	46	21	21
Performance ($\times 10^{6}$)	58.29	66.18	83.04	120.50
Performance/area ($\times 10^{3}$)	14.53	37.41	104.47	185.09

Table 9.4: Bit-parallel DCT implementations obtained by area minimization with different output mean error constraints. (†rounded up values)

Mean error constraint ($\times 10^{-2}$)	0.1953	0.3906	1.5625	6.2500
Maximum error ($\times 10^{-2}$)	0.856	1.672	6.751	22.224
Mean error ($\times 10^{-2}$)	0.195	0.219	0.832	2.883
SNR ($\times 10^{2}$)	2.502	1.343	0.579	0.180
Average digit size	2.94	2.64	1.70	1.61
Maximum digit size†	7	5	4	4
Minimum digit size†	4	2	1	1
Average wordlength	21.24	18.65	14.49	14.28
Maximum wordlength	83	42	36	28
Minimum wordlength	11	9	8	7
Trimmed adders	0	0	117	117
Trimmed multipliers	0	0	2	2
Amount of resources (slices)	1595	1167	691	540
Clock rate (MHz)	69.15	95.56	96.20	122.01
Throughput (clock cycles)	8	8	8	8
Latency (clock cycles)	111	85	68	51
Performance ($\times 10^{6}$)	8.64	11.95	12.02	15.25
Performance/area ($\times 10^{3}$)	5.42	10.24	17.40	28.24

Table 9.5: Eight-digit DCT implementations obtained by area minimization with different output mean error constraints. (†rounded up values)

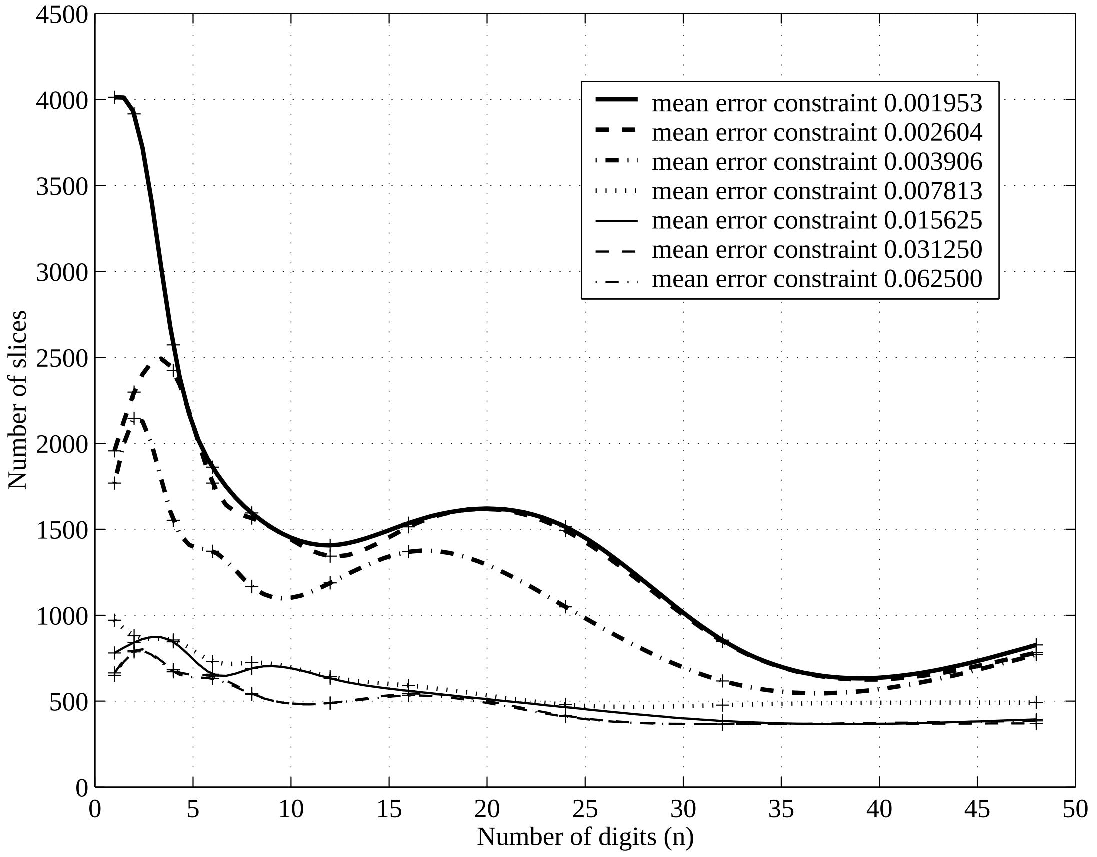


Figure 9.5: Plot of number of slices against the number of digits under different error constraints.

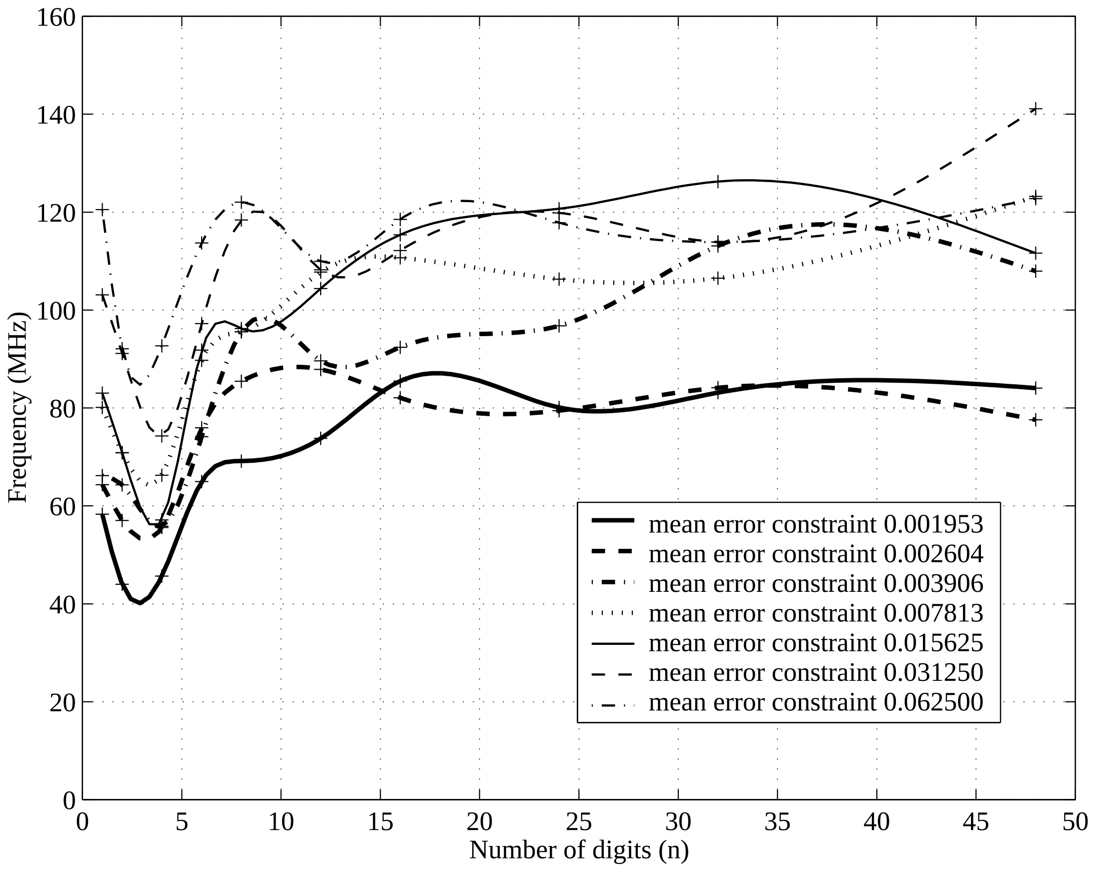


Figure 9.6: Plot of frequencies against the number of digits under different error constraints.

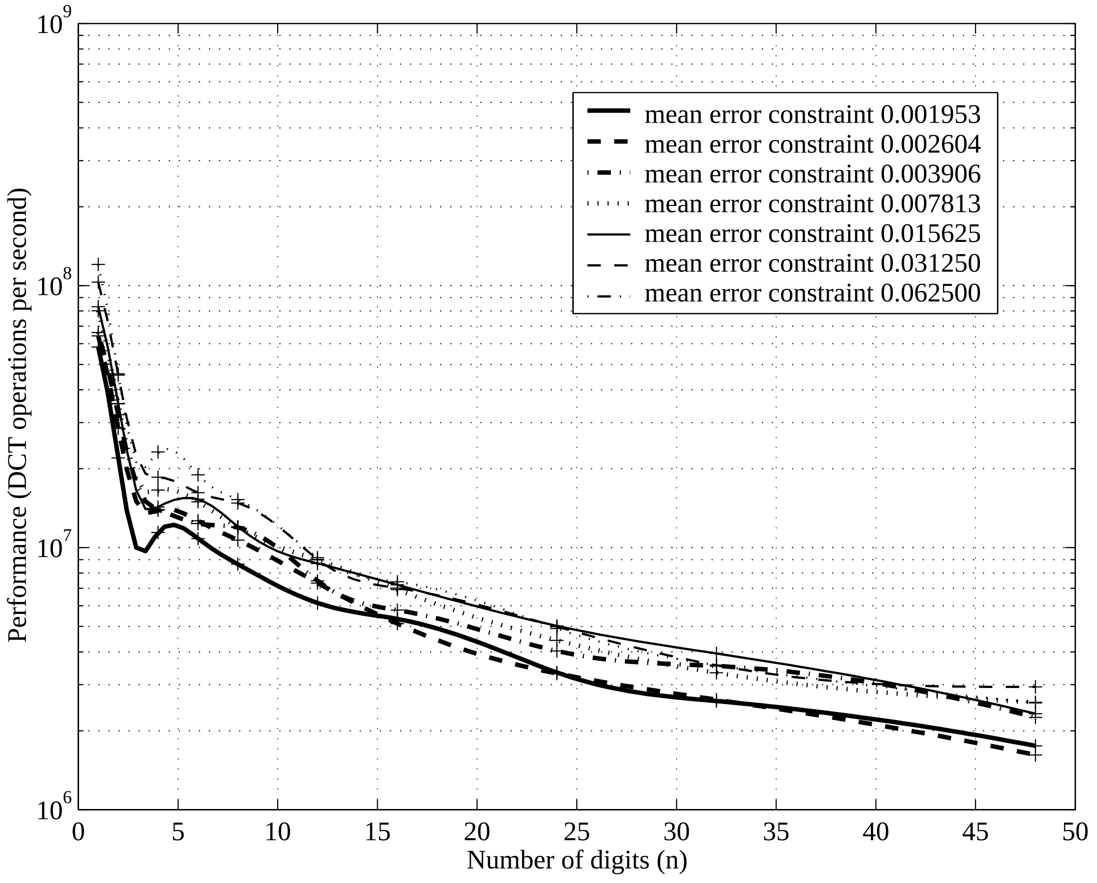


Figure 9.7: Plot of performance, measured in DCT operations per second, against the number of digits under different error constraints.

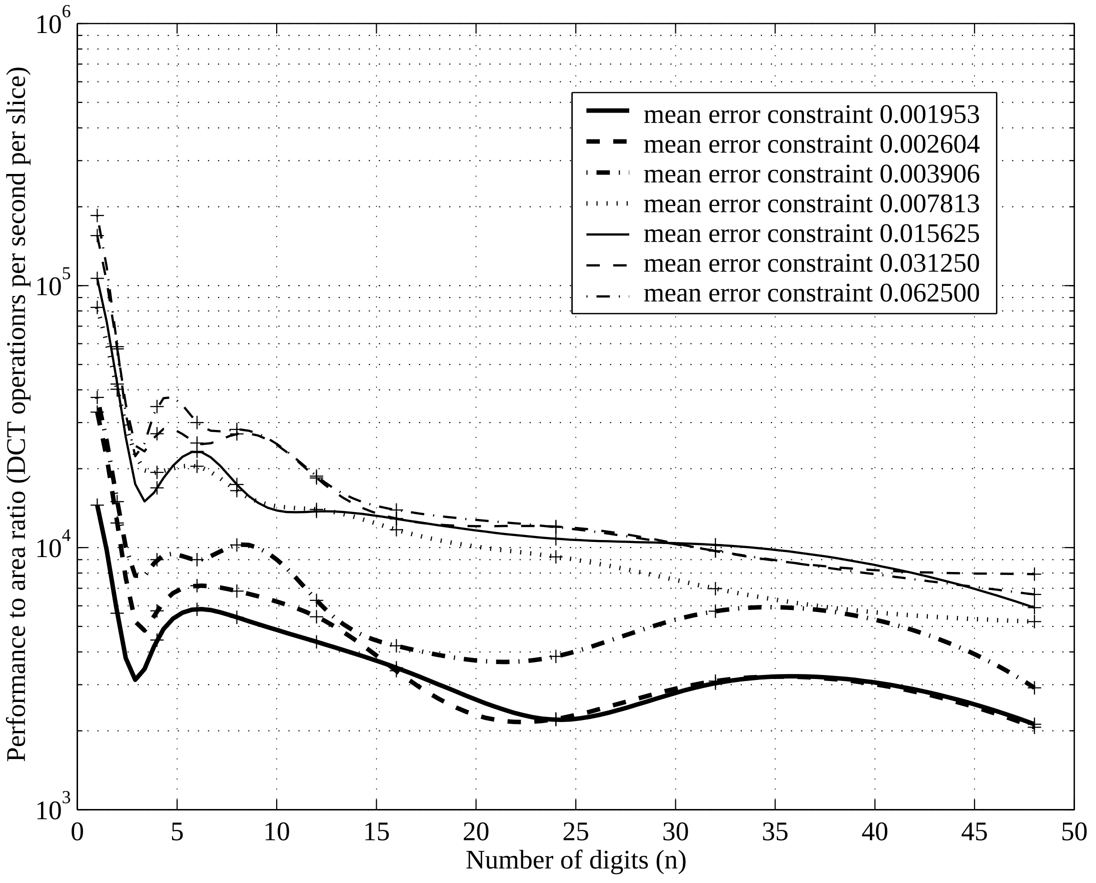


Figure 9.8: Plot of performance to area ratio, measured in DCT operations per second per slice, against the number of digits under different error constraints.

Throughput constraint (clock cycles)	8	16	32	64
Maximum error ($\times 10^{-3}$)	16.724	16.370	22.349	16.370
Mean error ($\times 10^{-3}$)	2.190	2.552	3.087	3.046
SNR ($\times 10^{2}$)	2.069	1.918	1.433	1.554
Average digit size	2.64	2.64	1.08	1.00
Maximum digit size†	5	3	2	1
Minimum digit size†	2	1	1	1
Average wordlength	18.65	22.52	21.12	19.57
Maximum wordlength	42	42	44	39
Minimum wordlength	9	8	8	9
Trimmed adders	0	0	0	0
Trimmed multipliers	0	0	0	0
Amount of resources (slices)	1167	1158	606	599
Clock rate (MHz)	95.56	88.09	111.98	120.53
Throughput (clock cycles)	8	14	30	39
Latency (clock cycles)	85	99	84	84
Performance ($\times 10^{6}$)	11.95	6.29	3.73	3.09
Performance/area ($\times 10^{3}$)	10.24	5.43	6.16	5.16

Table 9.6: DCT implementations obtained by area minimization under different throughput constraints and output mean error constraint 3.0963×10^{-3}. (†rounded up values)

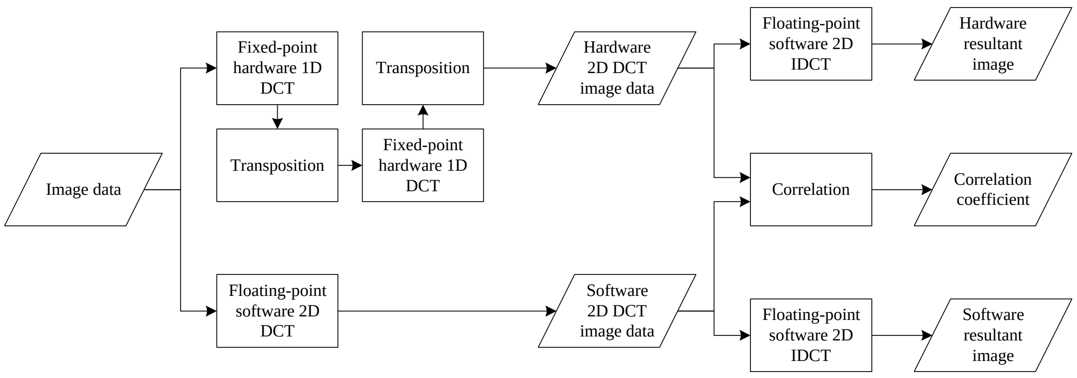


Image data
Fixed-point hardware 1D DCT
Transposition
Transposition
Fixed-point hardware 1D DCT
Floating-point software 2D DCT
Hardware 2D DCT image data
Software 2D DCT image data
Floating-point software 2D IDCT
Correlation
Floating-point software 2D IDCT
Hardware resultant image
Correlation coefficient
Software resultant image

Figure 9.9: Framework for image coding experiments.

out in software. The 2D DCT results from the hardware implementation were then correlated with a software floating-point implementation, and the results are shown in Table 9.7. The benchmark image *Airfield* was chosen and applied the inverse DCT on both the hardware and software DCT results. Resultant images are shown in Figure 9.10.

In Figure 9.10, simulation SNR refers to the SNR based on simulation of sample dataset, actual SNR refers to the SNR based on comparison of images obtained from floating-point and fixed-point implementations and is measured in dB. Simulation SNR and actual SNR were measured separately because during simulation (described in Section 9.4.1) the errors contributed by R_p were neglected, and the image coding experiments require two 1D DCTs, and therefore contain twice the 1D DCT fixed-point errors.

9.5 Discussion

In general, implementations employing variable-radix variable-wordlength architecture have consistent properties with traditional digit-serial implementations.

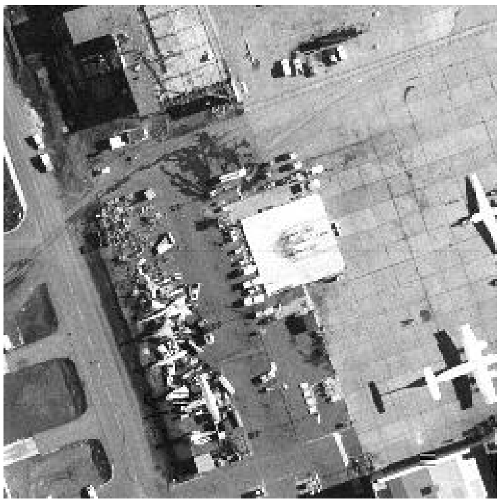
Constraint[†]	SNR[‡]	*Airfield*	*Airplane*	*Bridge*	*Couple*	*Crowd*
0.001953	251.811	0.9998392	0.9997353	0.9998078	0.9997098	0.9997772
0.002604	212.835	0.9995402	0.9988759	0.9990135	0.9994490	0.9995638
0.003906	118.936	0.9995167	0.9992529	0.9992773	0.9993545	0.9994785
0.007813	66.271	0.9989709	0.9988271	0.9987446	0.9987056	0.9989653
0.015625	42.413	0.9948467	0.9906164	0.9939893	0.9912205	0.9931685
0.031250	24.577	0.9837934	0.9822585	0.9814566	0.9765656	0.9824283
0.062500	20.155	0.9831877	0.9714933	0.9807353	0.9704832	0.9773544

Constraint[†]	SNR[‡]	*Harbour*	*Lax*	*Lena*	*Man*	*Peppers*
0.001953	251.811	0.9996340	0.9995413	0.9997498	0.9997479	0.9998022
0.002604	212.835	0.9998601	0.9991282	0.9997892	0.9992854	0.9998324
0.003906	118.936	0.9991240	0.9989422	0.9995340	0.9994413	0.9995960
0.007813	66.271	0.9982936	0.9976151	0.9989699	0.9988836	0.9991613
0.015625	42.413	0.9879892	0.9858949	0.9927063	0.9923737	0.9944384
0.031250	24.577	0.9716344	0.9560052	0.9822122	0.9794762	0.9857037
0.062500	20.155	0.9615852	0.9555060	0.9737961	0.9739921	0.9792801

Table 9.7: Correlations between DCT results obtained from hardware fixed-point and software floating-point implementations under various algorithm performances. ([†]The constraint set was mean error constraints at outputs. [‡]The SNR at the output based on simulation of the sample dataset.)

(a) floating-point DCT

(b) fixed-point DCT
(251.811, 44.520, 4013)


(c) fixed-point DCT
(212.835, 39.489, 1956)

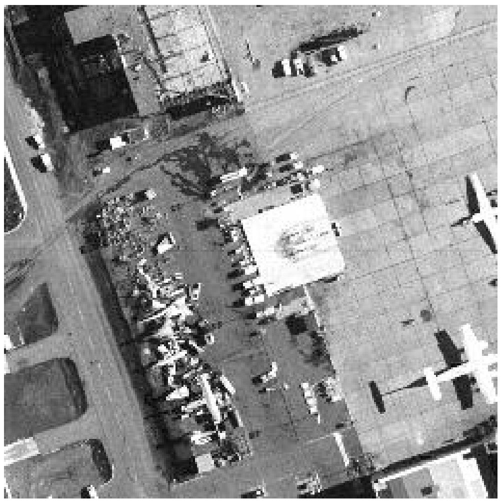
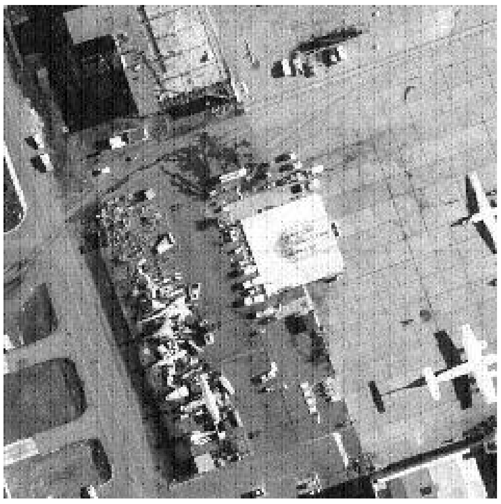

(d) fixed-point DCT
(118.936, 38.408, 1769)

Figure 9.10: Resultant images obtained from software floating-point and hardware fixed-point DCT image coding, followed by floating-point inverse DCT (simulation SNR, actual SNR measured in dB, slices in a bit-parallel implementation).

(e) fixed-point DCT
(66.271, 32.520, 751)

(f) fixed-point DCT
(42.413, 27.660, 734)


(g) fixed-point DCT
(24.577, 24.091, 664)

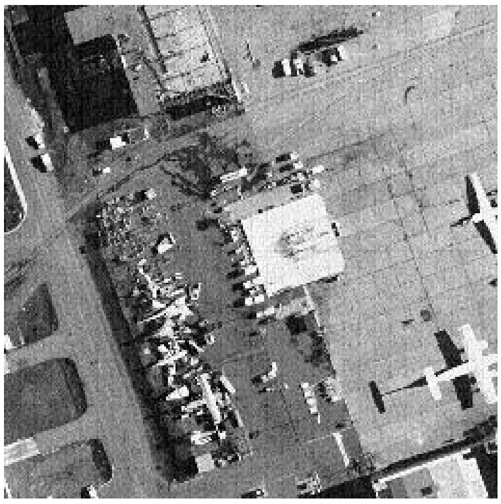

(h) fixed-point DCT
(20.155, 19.664, 651)

Figure 9.10 (continued): Resultant images obtained from software floating-point and hardware fixed-point DCT image coding, followed by floating-point inverse DCT (simulation SNR, actual SNR measured in dB, slices in a bit-parallel implementation).

For instance, the effects of increasing number of digits are reduced area, increased clock rate and increased latency. However, the plots in Section 9.4.1 show that relationships among area, frequency, performance and area efficiency is complex. The data was analyzed in an attempt to understand the cause of such behavior.

9.5.1 Tradeoff between Area and Number of Digits

In Figure 9.5, a general trend of decreasing area with increasing n can be observed. The insertion of conversion modules for variable format conversions resulted in area overhead when the number of digits, n, was increased from one (bit-parallel). Therefore for $n = 2$, resource requirements are usually greater than $n = 1$. To achieve a net decrease in resource requirements, n must be further increased so that the area reduction by folding operators into a lower radix can compensate for the conversion module overhead.

When the implementation was bit-serial ($n = 48$ for a mean error constraint of 3.9063×10^{-3} as in Table 9.2, $n = 32$ for a mean error constraint 1.5625×10^{-2}), the decrease in area is significant since conversion modules can now be implemented solely with registers and do not require multiplexers. As can be seen from Figure 9.5, when the specified n is larger than that required for a bit-serial implementation, area increases because unnecessarily long stage latches are inserted into the circuit.

The above analysis can be justified by a best-fitting polynomial curve for the data of slices required against number of digits. The plot is shown in Figure 9.11. Intuitively, area decreases with increasing n (more parallel implementation) until n becomes unnecessarily large. Hence, a concave function should result which decided the choice of a best-fitting polynomial curve of degree two. Using the data shown in Table 9.2, the best-fitting function obtained is $A = 0.7n^2 - 58.8n +$

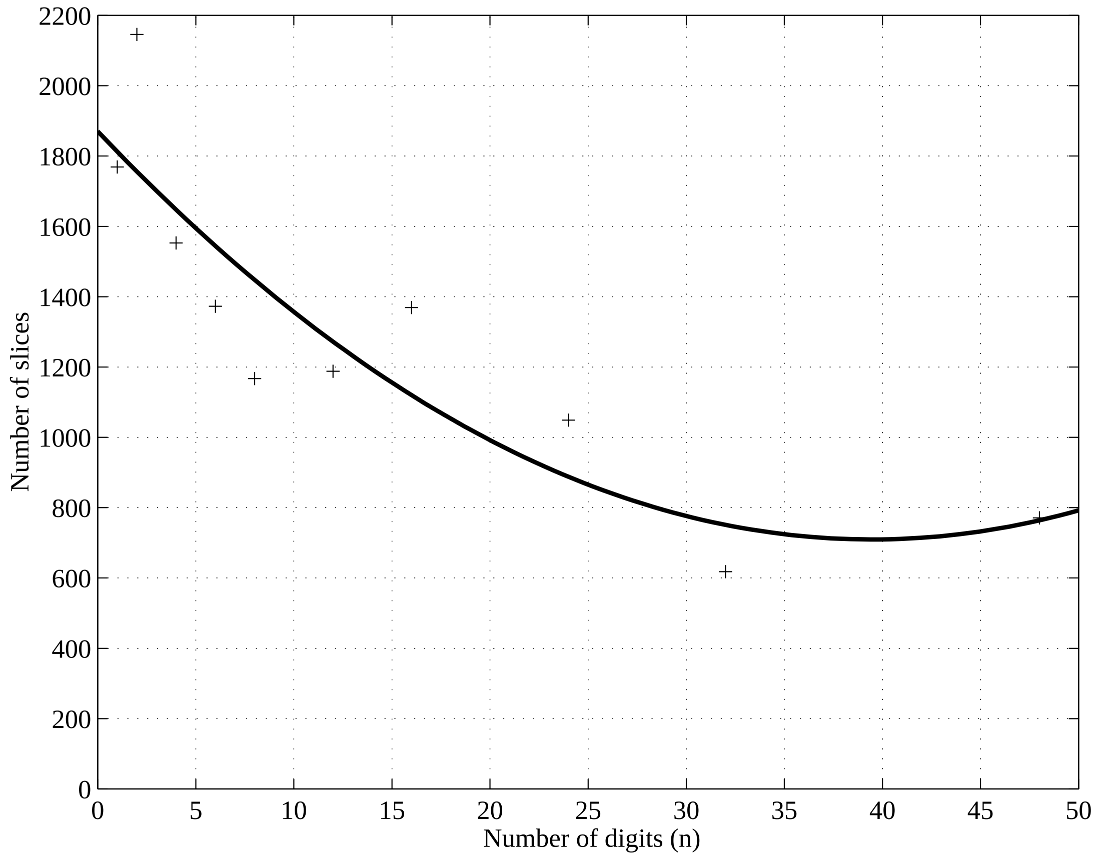


Figure 9.11: Best-fitting polynomial curve of degree two for the plot of number of slices against number of digits, with mean error constraint set to 3.9063×10^{-3}.

1087.6, which reaches its local minimum when $n = 39.5$. This can be compared with the results in Table 9.6, where a minimal area implementation with 599 slices was obtained when $n = 39$. The corresponding implementation is a bit-serial one.

9.5.2 Tradeoff between Clock Rate and Number of Digits

It can be observed from Figure 9.6 that the maximum clock frequency of the resulting implementations generally increases with increasing number of digits n. The exception is when n was increased from one to an integer less than four, a drop in maximum frequency was observed.

Digit-serial architectures with smaller radix often offer higher clock rates due to shorter ripple-carry chains. The variable-radix variable-wordlength architecture has a similar property but it can be seen in Figure 9.6 to be a weak function of n. This is because conversion modules become more complicated with increased n and require more control circuitry, becoming a bottleneck. Several optimization techniques such as pipeline stage insertion have been implemented, but it is still difficult to establish a relationship between these two parameters as in a traditional digit-serial implementation. The drop in frequency when n is between two and four is mainly caused by the increased area of conversion modules. Normally for FPGA implementations, routing produces most of the delay in the critical path. Area has a large effect on consuming routing resources, so it directly affects the maximum frequency.

9.5.3 Tradeoff between Performance and Number of Digits

The variable-radix variable-wordlength architecture has a property that the throughput is controlled by the number of digits n. A variable of n digits takes at least n cycles for its data bits to pass through the associated data wires since it is multiplexed in time.

The overall performance (number of DCT operations per second) is determined by two factors, the number of clock cycles to complete a DCT operation (or equivalently, the maximum number of cycles between adjacent inputs or outputs) and the maximum clock rate of the implementation. As can be seen from Figure 9.7, the first factor has the dominant effect on performance, hence it is observed that the performance drops with increased number of digits. The dip in the performance curves in Figure 9.7 around n from two to four can be explained by the corresponding drop in the frequency seen in Figure 9.6 discussed earlier.

9.5.4 Tradeoff between Area Efficiency and Number of Digits

Figure 9.8 shows the plot of area efficiency (number of DCT operations per second per slice) versus the number of digits. The curves, compared with the previous ones, contain more local minima and maxima. This is because area efficiency is determined by two conflicting factors, namely area requirements and performance.

Certainly better area efficiencies implies better implementations, but implementations that achieve high area efficiency may not satisfy certain design constraints in practice. For instance, implementations on the left side of the curves may not satisfy area constraints, whereas those on the right side may not satisfy throughput constraints. Therefore, one possible design approach is to analyze the curves and pick an implementation that offers the minimal area while satisfying all the design constraints.

9.5.5 Effects of Output Error Constraints

By allowing larger errors at the outputs, the control of fixed-point quantization and computation errors can be relaxed, hence providing a larger space for area minimization. The curves in Figure 9.5 shows the area requirements under different error constraints. When the mean error constraint is less than or equal to $1/128 = 0.007813$, there is a significant reduction in area due to trimming of operators. More precisely, when the error constraint was relaxed say to $1/64 = 1.5625 \times 10^{-2}$, the number of trimmed operators becomes 117 adders and

2 multipliers. The removal of multipliers with small coefficients (for the above case, the multipliers with coefficients a_p and a_{p-1}) does not affect the precision of output $X(k)$ with respect to the error constraint as the partial sums they produce are relatively small. Trimming of these multipliers consequently allowed the removal of their preceding adders hence contributed in significant area reductions.

Trimming of operators not only results in area reduction but also in latency reduction. Since adders in the Pascal's triangles are trimmed, latency in the p-network is shortened, resulting in a significant reduction in the overall latency.

Moreover, the relaxed error constraints allow operators to have reduced precision and hence reduced wordlength leading to an area reduction. In addition, higher maximum clock rates are obtained because operators are simplified. With smaller area and higher clock rate, implementations with relaxed error constraints thus have higher performance and better area efficiency. This can be observed in Figures 9.6 and 9.7.

9.5.6 Optimization by Simulation

The proposed approach to optimize an implementation is based on the simulation of a sample dataset. Although the optimization procedure considers only runtime error analysis and does not take worst-case analysis into account, it is observed from the image coding experiments that the performance of the algorithm is practical for many applications.

In the experiments the runtime maximum and mean error are in the order of 10^{-2}, but analysis suggests that worst-case error could be up to the 10^{5}, seven orders of magnitude more than runtime error analysis. A worst-case analysis is too pessimistic in practice. Area requirements would be drastically decreased if optimization is based on runtime error analysis.

9.5.7 Comparison with Traditional Digit-Serial Architectures

To justify the advantages of employing a variable-radix digit-serial architecture, the variable-radix DCT implementations were compared with those implemented using a fixed-radix digit-serial architecture. These fixed-radix implementations were obtained by setting all variables to be the same wordlength and consequently of the same radix. The integer wordlengths were adjusted so that overflow does not occur and the fractional wordlengths were maximized to minimize quantization errors.

Fixed-radix implementations with 16-bit, 20-bit and 24-bit wordlengths were tried. The corresponding runtime mean error at the outputs are 5.705×10^{-3}, 1.960×10^{-3} and 1.956×10^{-3} respectively. Area requirements and maximum clock rates of these implementations are shown in Table 9.8. As can be seen from the table, there is general trend of decreasing area and increased clock rate with increased digit size.

To compare with these fixed-radix implementations, three variable-radix implementations were chosen and their performance measurements are listed in Table 9.9. These implementations were chosen because their runtime output mean error and performance are close to the some of the fixed-radix implementations in Tables 9.8. The comparison of performance and area between the chosen implementations and the fixed-radix implementations are plotted in Figure 9.12. In this plot, three pairs of implementations were compared (indicated by dotted lines) and three equi-performance to area ratio lines (the larger performance to area ratio, the better the area efficiency) were also plotted. The three pairs of comparing implementation were:

- Variable-radix implementation 1 (mean output error 4.964×10^{-3}, $14.95 \times$

Number of digits[‡]	16-bit wordlength	20-bit wordlength	24-bit wordlength
1	2194, 78.01	4738, 62.13	5765, 57.34
2	1910, 63.92	4070, 51.95	4922, 48.03
4	1230, 58.36	2330, 50.84	2957, 47.67
6	1014, 78.04	2064, 63.10	2285, 60.90
8	913, 84.25	1671, 74.42	1874, 65.05
12	916, 96.62	1568, 75.55	1712, 77.19
16	793, 92.56	1547, 79.20	1681, 78.57
20	N/A[†]	1323, 99.13	1727, 75.97
24	N/A[†]	N/A[†]	1474, 87.41

Table 9.8: Area requirements and maximum clock rates of fixed-radix DCT implementations (slices, maximum clock rate measured in MHz). ([†]Non-applicable because digit size is greater than wordlength. [‡]Digit size is wordlength divided by number of digits.)

10^6 DCT operations per second) was compared with the 16-bit wordlength, 6-digit (radix-8) fixed-radix implementation (mean output error 5.705×10^{-3}, 13.06×10^6 DCT operations per second). The variable-radix implementation has a performance to area ratio of 24.04×10^3 while that of the fixed-radix implementation is 12.83×10^3 (the variable-radix implementation has 87% improvement).

- Variable-radix implementation 2 (mean output error 1.938×10^{-3}, 10.83×10^6 DCT operations per second) was compared with the 24-bit wordlength, 6-digit (radix-16) fixed-radix implementation (mean output error 1.956×10^{-3}, 10.15×10^6 DCT operations per second). The variable-radix implementation has a performance to area ratio of 5.82×10^3 while that of the fixed-radix implementation is 4.44×10^3 (the variable-radix implementation has 31% improvement).

- Variable-radix implementation 3 (mean output error 1.938×10^{-3}, 6.15×10^6 DCT operations per second) was compared with the 20-bit wordlength, 12-digit (radix-4) fixed-radix implementation (mean output error 1.960×10^{-3}, 6.29×10^6 DCT operations per second). The variable-radix implementation has a performance to area ratio of 5.50×10^3 while that of the fixed-radix implementation is 4.01×10^3 (the variable-radix implementation has 37% improvement).

9.6 Summary

An implementation of a systolic structure for the computation of the DCT using *fp* tool, in which an optimization approach which automatically translates floating-point algorithmic descriptions into hardware-efficient fixed-point imple-

Implementation	1	2	3
Maximum error ($\times 10^{-3}$)	75.317	8.558	8.558
Mean error ($\times 10^{-3}$)	4.964	1.938	1.929
SNR ($\times 10^{2}$)	78.081	250.93	251.81
Average digit size	2.12	3.78	2.06
Maximum digit size[†]	4	10	6
Minimum digit size[†]	1	1	1
Average wordlength	15.13	20.17	19.86
Maximum wordlength	36	58	71
Minimum wordlength	8	12	13
Trimmed adders	117	0	0
Trimmed multipliers	2	0	0
Amount of resources (slices)	622	1861	1118
Clock rate (MHz)	89.72	64.98	73.77
Throughput (clock cycles)	6	6	12
Latency (clock cycles)	44	78	72
Performance ($\times 10^{6}$)	14.95	10.83	6.15
Performance/area ($\times 10^{3}$)	24.04	5.82	5.50

Table 9.9: Variable-radix implementations chosen for comparisons with fixed-radix implementations. ([†]rounded up values)

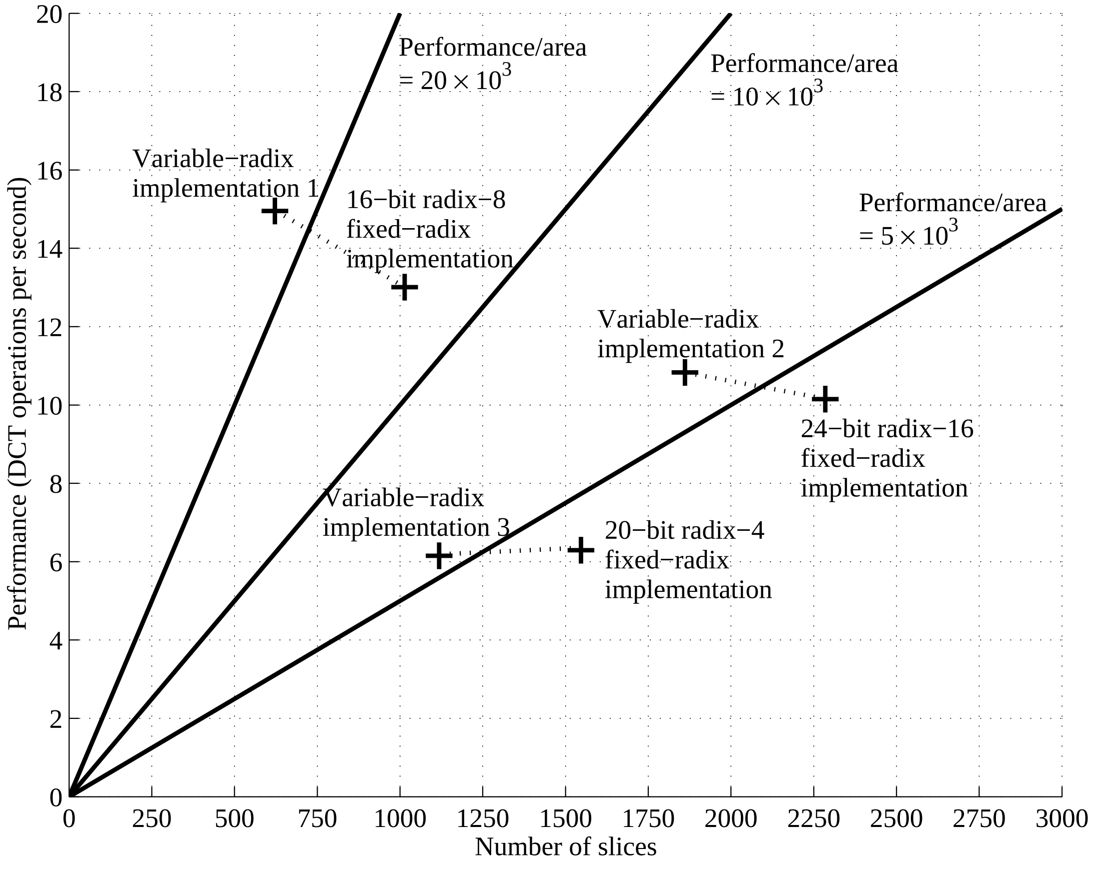


Figure 9.12: Comparison of performance and area between variable-radix and fixed-radix implementations.

mentations, was described. Using the variable-radix variable-wordlength architecture, an almost a continuous tradeoff in performance, area, latency and throughput is obtained, thus allowing designers to choose the most appropriate design for a given application.

Through the use of the variable-radix variable-wordlength architecture, higher area efficiencies were obtained as compared with the traditional digit-serial approach. Experiments show as much as 87% improved area efficiency has been achieved.

Compared with a worst-case analysis, runtime analysis showed that typical errors were several orders of magnitude smaller at the outputs (for example, 10^5 for the worst-case analysis, 10^{-2} for the runtime analysis). Given a sample dataset which is a representative of the input in practice, using runtime analysis for optimization may provide significantly larger room for wordlength reduction.

Using the module generation approach, a set of 70 implementations of the DCT of different radices and wordlengths were generated from a single description. These implementations had areas ranging from 366 to 4013 Virtex slices, with throughputs between 1.62×10^6 DCT operations per second and 120.50×10^6 DCT operations per second. Applying the DCT implementations to 2D image coding, the resultant images had SNRs between 19.67 dB and 44.52 dB.

Chapter 10

Conclusion

The main objective of this thesis was to develop techniques to improve the efficiency of FPGA-based coprocessor systems. A tool, called *fp*, was developed which provides automatic floating-point to fixed-point translation. It supports a generalized variable-radix digit-serial computation which provides the flexibility in controlling the degree of parallelism and the hardware requirements of the design. The *fp* tool greatly reduces design time and generates designs which are too tedious to be designed manually. A memory bus based interface used in a reconfigurable computer (RC) platform called *Pilchard* was shown to have a large performance improvement over the PCI Local Bus.

10.1 *fp*

Software programming and hardware designs being treated as distinct entities remains an obstacle in developing a FPGA-based coprocessor system. The design goal of *fp* was to bridge between these two entities in a way that software programs can be translated to hardware implementations with minimum additional effort.

Using this tool, designers may focus in the higher algorithmic issues and be less concerned with the details in a hardware implementation.

The primary input to *fp* is an algorithmic description which is a program written in a subset of the C language. The translation from an algorithmic description to a hardware implementation involves extraction of dataflow, determination of operator bit-width and radix, and generation of VHDL description. The output of *fp* is a synthesizable VHDL description that implements the algorithmic description with fixed-point arithmetic and satisfies user-specified performance requirements and constraints.

Bit-widths and radices of operators were determined via an optimization and simulation approach. Unlike previous approaches in which wordlengths of variables were optimized, *fp* optimizes the bit widths of operators. This approach is well suited to FPGAs that are rich in storage and routing resources (corresponding to variable wordlengths). In *fp*, fixed-point operators are parameterized. The optimizer tries different configurations of parameters and derives an implementation that minimizes a user supplied cost function reflecting the required tradeoff among various conflicting performance measures. The algorithm is simulated with a sample dataset in every trial, hence a runtime analysis of dynamic ranges and quantization errors can be extracted.

To further explore the tradeoff among area, latency and throughput, *fp* can generate a variable-radix variable-wordlength architecture. This architecture is considered as a generalization of the traditional digit-serial architecture, and is capable of providing higher resource utilization, parallelism and area efficiency.

Applications of the tool presented in this dissertation include a post-rendering 3D warping algorithm (Chapter 7), a parameterized electronic cochlea model (Chapter 8), and a systolic structure of the Discrete Cosine Transform (DCT)

(Chapter 9).

10.2 *Pilchard*

Improving the bandwidth and latency as well as simplifying the interface is particularly an important issue for FPGA-based coprocessors. To address this issue, an RC platform which communicates with the CPU via the memory bus called *Pilchard* was developed.

Measurements showed that *Pilchard* achieves approximately three times better write performance and five times better read performance as compared with the PCI Local Bus in the uncachable (UC) MTRR mode. If write-combining (WC) MTRR mode is used, the write performance can be as high as five times the that of the PCI Local Bus. Read and write performances can be further improved by using 64-bit data transfer, in which a ten times speedup in read performance and a six times speedup in write performance are obtained.

The applications presented in this dissertation that utilized the *Pilchard* platform include the International Data Encryption Algorithm (IDEA) cipher (Chapter 6) and the parameterized electronic cochlea model (Chapter 8).

10.3 Applications

The design methodologies developed in this dissertation were applied to four applications. These applications demonstrated how the proposed design methodologies improve upon existing design approaches for FPGA-based coprocessor. They also showed performance improvements upon purely software-based implementations.

The implementation of IDEA cipher uses a dedicated bit-serial design of the multiplication modulo $2^{16}+1$ operator to facilitate a bit-serial architecture. Enabled by a deeply-pipelined architecture, the implementation achieved a high system clock rate (150 MHz) and throughput (600 Mbytes/sec) on a Xilinx Virtex XCV300-6 FPGA. The key schedule can be changed via direct bitstream modification which saves on the interfacing logic. This implementation was tested on the *Pilchard* and PCI Local Bus platforms and a 3.74 times faster performance was achieved on *Pilchard*.

Post-rendering 3D warping involves arithmetic operations in which the bit widths affect the performance of the algorithm. With *fp*, a single C description of this algorithm can generate multiple implementations, each of which has a different tradeoff between area and performance. Two implementations of the post-rendering 3D warping algorithm were presented, one having a higher performance and the other having a smaller area. They respectively are 1.11 and 1.25 times faster than an optimized software implementation on a Xilinx XC4085XL-1 FPGA.

The proposed design methodologies were applied to develop a module generator for the Lyon and Mead cochlea filter. It makes use of the fixed-point modeling feature of *fp* to estimate the filter precision for a specific bit width, from which a designer may explore an application-optimized design. An implementation of the generated module on *Pilchard* showed 1.67 times higher performance than the same implementation on the PCI Local Bus and is 6.22 times faster than a software implementation.

The DCT algorithm discussed in this dissertation involves only additions and multiplications, and it is well-suited for a digit-serial implementation. Through the use *fp* and its variable-radix variable-wordlength architecture, a series of implementations of the DCT, each with a different tradeoff among various conflicting performance measures (precision, area, latency and throughput), were obtained from a single algorithmic description. These implementations have areas vary from 366 to 4013 Virtex slices and almost continuous throughputs between 1.62×10^6 and 120.50×10^6 DCT operations per second, with up to 87% improved area efficiency over the traditional digit-serial architectures.

10.4 Prospects for Research

This dissertation has focused on design methodologies that may more efficiently facilitate the FPGA-based coprocessor approach. It would be worth investigating the following refinements to the methodologies presented in this dissertation.

10.4.1 *fp*

Currently the *fp* system requires users' explicit identification of the inner loops for hardware implementation. This identification may possibly be automated using profiling techniques, in which the most expensive routines of a program can be identified and a justification of whether a hardware implementation may offer performance improvement can be made.

The simulation-based optimization procedure of *fp* utilizes a user supplied sample dataset. A careful selection of this dataset is needed and no guarantees that overflow will not occur for arbitrary inputs can be made. There may be more sophisticated methods to determine the relationship among the operators and wordlengths.

Recent FPGA devices have on-chip memory (such as the Virtex BlockRAM [Xil00c]) and multipliers (such as the Virtex-II 18-bit multiplier [Xil01a]). To utilize these resources, *fp* should support mapping operators to these resources.

10.4.2 *Pilchard*

The *Pilchard* board currently requires the bitstream to be downloaded from another machine. It would be feasible and more convenient to configure the FPGA via the memory bus interface. One possible way to achieve this is to use the SelectMAP configuration data pins on the FPGA [Xil00c], and to introduce external logic for the generation of appropriate control signals when certain data patterns appear on the memory bus.

Streaming SIMD Extension (SSE) instructions may produce memory burst read (and write) accesses [Int00a] without corrupting the cache and may be a more efficient method to transfer data to the FPGA board.

The design concept of *Pilchard* can be applied to the double data rate (DDR) SDRAM interface [Tra00a] whose bandwidth (2128 Mbytes/sec) is double that of the PC133 DIMM interface. The feasibility of employing a Rambus DRAM (RDRAM) inline memory modules (RIMM) interface also needs to be investigated. The RIMM interface has a series of standards (each of which corresponds to a different data rate), at one side being the RIMM1600 standard which has a 800 MHz data frequency and is 16-bit wide (a bandwidth of 1600 Mbytes/sec), and the other side being the RIMM8500 standard which has a 1066 MHz data frequency and is 64-bit wide (a bandwidth of 8532 Mbytes/sec) [Ram01].

It is envisaged that the *Pilchard* platform can enable many FPGA applications previously considered to not be practical due to bandwidth limitations. As an example, using the SelectLink [Xil00c] communications interface, very high I/O bandwidth (200 Mbytes/sec per pin) can be achieved and would be superior to the Myrinet (2 Gbits/sec) and Gigabit Ethernet (1 Gbit/sec) technologies in terms of bandwidth and latency for point-to-point connections.

10.5 Closing Remarks

A fundamental limitation of microprocessors is due to the nature that software programs are essentially sequences of operations chosen from the set of instructions supported by the microprocessor architecture. The use of FPGA-based custom computing machines is a potential solution to meet the strict performance requirements imposed by real-time and portable applications. An FPGA-based coprocessor system may offer significant performance improvements upon a purely software-based design, and with FPGA speed and density constantly improving and non-recurrent engineering costs for ASICs also increasing, this approach should become even more attractive in the future.

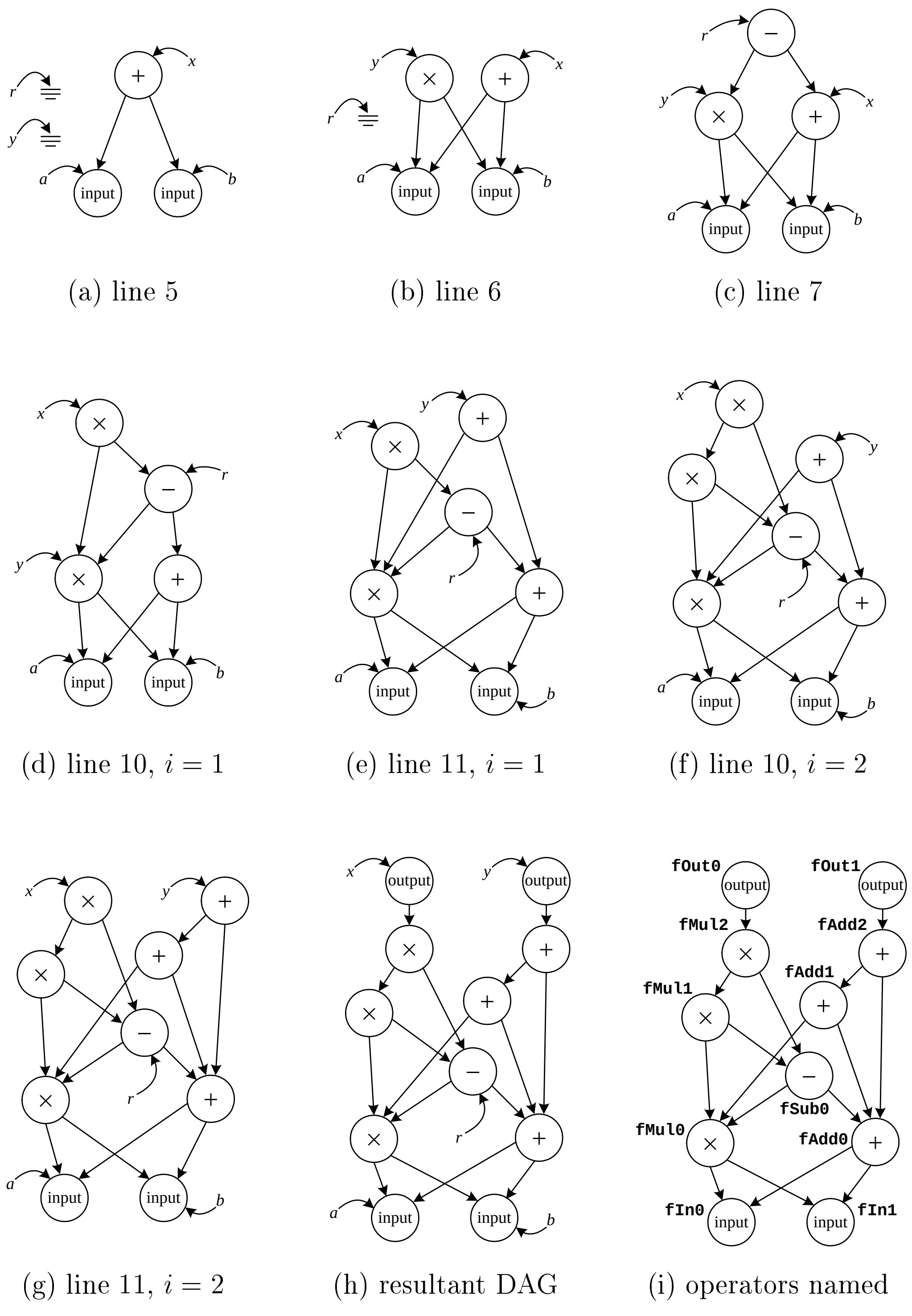
Appendix A

fp Implementation Details

This appendix is an extension to Chapters 3 and 4 which presents the implementation details of *fp*. Specifically, the algorithms used and the programming interface are described in detail.

A.1 Dataflow-Extraction Library

To examine how the compilation is carried out through the use of the dataflow-extraction library, the example program in Figure 3.1 is used. A directed acyclic graph (DAG) data structure is defined in the dataflow-execution library. Upon initialization of the program, the DAG structure is empty. There is a class variable in the `fixed` class which is a reference to a node in the DAG. This reference is initialized to `NULL` at the object constructor. In this program, a and b are inputs, x and y are outputs and r is a local variable. The `fixed` objects a and b are labeled as inputs by the caller before entering function g. This is achieved by calling an `input()` method. Similarly, the outputs x and y are indicated by a `output()` method.



(a) line 5 (b) line 6 (c) line 7

(d) line 10, $i = 1$ (e) line 11, $i = 1$ (f) line 10, $i = 2$

(g) line 11, $i = 2$ (h) resultant DAG (i) operators named

Figure A.1: Building a DAG from the algorithmic description in Figure 3.1.

After `a.input()` and `b.input()` calls, two input-type nodes are created correspondingly. The DAG node reference from a and b are updated to point to these input-type nodes. At line 5 of the program, the first `fixed` class operation which is the addition of a and b is issued. After its execution, a new add-type node is formed. This add-type node points to the nodes referenced by `fixed` objects a and b. Furthermore, the DAG node reference from x is updated to point to this new node. The intermediate DAG after the execution of line 5 is shown in Figure A.1(a). The intermediate DAGs after executing lines 6 and 7, as shown in Figures A.1(b) and A.1(c) respectively, are built in a similar manner.

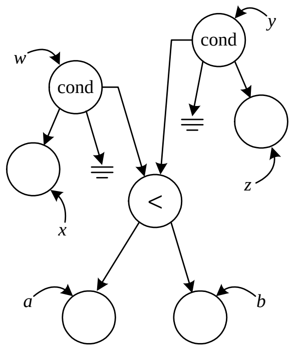
In lines 8 to 12 of the algorithmic function there is a for-loop with two iterations. However, the DAG is only updated in the loop body in lines 10 and 11 as these expressions involve operations on `fixed` objects. After the execution of line 10 in the first iteration, a new multiply-type node which points to the references of y (resolved by the conditional statement) and r is created, as shown in Figure A.1(d). The DAG node reference from x is then updated from the add-type node corresponding to $a + b$ to the new multiply-type node. In the next expression, there are two addition operations. The bracketed addition, $a + b$ is already computed in line 5. To minimize the number of operators and hence area of the resultant implementation, elimination of common subexpression is performed by the dataflow-extraction library for the node $a + b$ in the DAG built so far. The corresponding node will not be created, but a reference to an existing node is returned. Essentially, only one new add-type node is created after executing line 11 (Figure A.1(e)). The procedures follow in the second iteration of the loop, but in this iteration x is updated as $x \times r$ because of the result of the conditional statements (Figure A.1(f)). After all the expressions in the function are executed, a DAG as in Figure A.1(g) is formed. As a final step, after returning from function g `x.output()` and `y.output()` are executed. Two output-type nodes pointing to nodes referenced by x and y respectively are hence created.

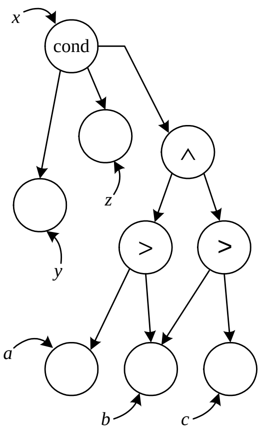
In the example, the conditional statement does not involve operations on `fixed` objects. To support conditional statements on `fixed` objects, the `fixed` class provides `ifthen()`, `ifelse()` and `ifthenelse()` methods. Compound conditional statements are supported, but they should not be specified with the logical-and (`&&`) operator in C. This is because in C, the execution of a sequence of and-ed conditional statements will be "short-circuited" when one of the conditional statements returns true, causing subsequent conditional statements not to be processed in forming the DAG. Lists of conditional statements are interpreted as and-ed conditional statements. By De Morgan's Law, or-ed conditional statements can be written as and-ed conditional statements. Examples of these methods are given in Table A.1.

Note that the example algorithmic function does not have loops with a variable number of iterations. For such cases, the graph representation must contain loops that a DAG cannot handle. Due to the choice of using a fully-pipelined architecture derived from a DAG for throughput maximization, *fp* only supports loops where the number of iterations are fixed during compile time.

The output of the dataflow-extraction process, which is a DAG description of the algorithmic function in C++ code, is generated by a `printdag()` function call after building the DAG. The parameters to the `printdag()` function call are the name of the functions, followed by the names of the inputs and outputs. For instance, the DAG description in Figure 3.4 is extracted by the `printdag("g", "a", "b", "x", "y")` function call.

Conditional statements	`fixed` class implementations	Corresponding DAG

`double`
`    a, b, w,`
`    x, y, z;`
`if (a > b)`
`    w = x;`
`else`
`    y = z;`
	`fixed`
	`    a, b, w,`
	`    x, y, z;`
	`w.ifthen(a > b, x);`
	`y.ifelse(a > b, z);`
		

`double`
`    a, b, c,`
`    x, y, z;`
`if (a > b && b > c)`
`    x = y;`
`else`
`    x = z;`
	`fixed`
	`    a, b, c,`
	`    x, y, z;`
	`x.ifthenelse(a > b,`
	`    b > c, y, z);`
		

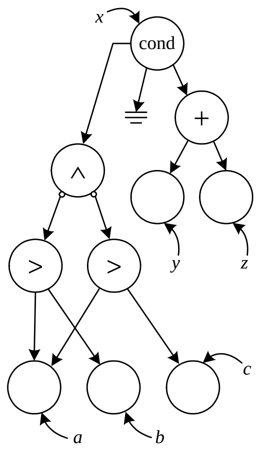
`double`
`    a, b, c,`
`    x, y, z;`
`if (a > b || a > c)`
`    x = y + z;`
	`fixed`
	`    a, b, c,`
	`    x, y, z;`
	`x.ifelse(`
	`    !(a > b),`
	`    !(a > c),`
	`    y + z);`
		

Table A.1: Support of conditional statements in `fixed` class.

A.2 Quantization and Overflow Models

The commonly-used quantization models, including rounding and truncation, are described in Table A.2. Overflow models, namely wrap-around and saturation, are described in Table A.3. In these tables, the precision format of a variable is represented as (w, f) (Section 3.4.3), where w is the total wordlength (sum of fractional and integer wordlengths) and f is the fractional wordlength.

A.3 Error Measurement and Range Estimation

The pseudocode in Figure A.2 is the procedure for extraction of dynamic ranges and quantization noise with a sample dataset $\mathbf{S} = \{\mathbf{V}_0, \mathbf{V}_1, \mathbf{V}_2, \ldots\}$, where $\mathbf{V}_i$ is an input vector of length equal to the number of module inputs, $i = \{0, 1, 2, \ldots\}$. A detailed explanation is given below:

1. The range of values for every input of the module (`FIn` objects) are determined by scanning the sample dataset but these ranges can also be overridden by user. The maximum quantization errors from the external data sources (the result of floating-point to fixed-point quantization) are also determined. These are set via `range()` and `error()` method calls to the input objects and are used as the initial values for worst-case range extraction and error estimation.

2. A worst-case analysis flag is set via a `WorstCaseAnalysis()` function call. Its effect is that subsequent operator calls (function calls beginning with an "`F`") do not instantiate operators. Instead, the following operations are carried out:

   (a) The inputs of the operator copies the precision format from the outputs

Model	Description
Rounding	The quantized value of $q = (q_{w_q-1}q_{w_q-2}\ldots q_1q_0)$ (precision format (w_q, f_q)) is obtained by rounding the original value of $p = (p_{w_p-1}p_{w_p-2}\ldots p_1p_0)$ (precision format (w_p, f_p)) to the nearest f_q binary digits ($w_p > w_q$, $f_p > f_q$ and $w_p - f_p = w_q - f_q$). The quantization algorithm is $$\begin{aligned} q &= (q_{w_q-1}q_{w_q-2}\ldots q_1q_0) \\ &= \begin{cases} (p_{w_p-1}p_{w_p-2}\ldots p_{w_p-w_q+1}p_{w_p-w_q}) & \text{if } p_{w_p-w_q-1} = 0, \\ (p_{w_p-1}p_{w_p-2}\ldots p_{w_p-w_q+1}p_{w_p-w_q}) + 2^{-f_q} & \text{if } p_{w_p-w_q-1} = 1, \end{cases} \end{aligned}$$ which can be implemented by a w_q-bit adder. The maximum absolute error produced by quantization, E_{qmax}, is $2^{-f_q-1} - 2^{-f_p}$.
Truncation	The quantized value of $q = (q_{w_q-1}q_{w_q-2}\ldots q_1q_0)$ (precision format (w_q, f_q)) is obtained by truncating the original value of $p = (p_{w_p-1}p_{w_p-2}\ldots p_1p_0)$ (precision format (w_p, f_p)) by $f_p - f_q$ LSB-side bits ($w_p > w_q$, $f_p > f_q$ and $w_p - f_p = w_q - f_q$). The quantization algorithm is $$\begin{aligned} q &= (q_{w_q-1}q_{w_q-2}\ldots q_1q_0) \\ &= (p_{w_p-1}p_{w_p-2}, \ldots p_{w_p-w_q+1}p_{w_p-w_q}), \end{aligned}$$ which can be implemented by a omitting the lower $f_p - f_q$ bits of the datapath. The maximum absolute error produced by quantization, E_{qmax}, is $2^{-f_q} - 2^{-f_p}$.

Table A.2: Roundoff and truncation quantization models.

Model	Description
Wrap-around	When a variable overflows, its overflowed MSBs are omitted. The overflowed value of $q = (q_{w_q-1}q_{w_q-2}\dots q_1q_0)$ (precision format of q is (w_q, f_q)) is obtained from the lower w_q bits of the original value $p = (p_{w_p-1}p_{w_p-2}\dots p_1p_0)$ (precision format of p is (w_p, f_p)) ($w_p > w_q$, $f_p = f_q$). The overflow algorithm is $$\begin{aligned} q &= (q_{w_q-1}q_{w_q-2}\dots q_1q_0) \\ &= (p_{w_q-1}p_{w_q-2}\dots p_1p_0), \end{aligned}$$ which can be implemented by omitting the upper $w_p - w_q$ bits of the datapath. The error produced by overflow is unbounded.
Saturation	When a variable overflows, the maximum or minimum value which can be represented is used. The overflowed value of $q = (q_{w_q-1}q_{w_q-2}\dots q_1q_0)$ (precision format (w_q, f_q)) is obtained from the lower w_q bits of the original value of $p = (p_{w_p-1}p_{w_p-2}\dots p_1p_0)$ (precision format (w_p, f_p)), or the maximum or minimum value which can be represented. The overflow algorithm is $$\begin{aligned} q &= (q_{w_q-1}q_{w_q-2}\dots q_1q_0) \\ &= \begin{cases} (011\dots1) & \\ \quad \text{if } p_{w_p-1} = 0,\ (p_{w_p-2}p_{w_p-3}\dots p_{w_q}) \neq (00\dots0), & \\ (100\dots0) & \\ \quad \text{if } p_{w_p-1} = 1,\ (p_{w_p-2}p_{w_p-3}\dots p_{w_q}) \neq (11\dots1), & \\ (p_{w_q-1}p_{w_q-2}\dots p_1p_0) & \text{otherwise,} \end{cases} \end{aligned}$$ which can be implemented by a comparator followed by a multiplexer. The error produced by overflow is unbounded.

Table A.3: Wrap-around and saturation overflow models.

```
procedure ErrorMeasurementAndRangeEstimation()
    input $\mathbf{V}$, $minRange$, $maxRange$, $extError$
    output $error$, $range$
begin
foreach $x$ in Fin objects
    $x$.range($minRange_x$, $maxRange_x$);
    $x$.error($extError_x$);
end for                  // set ranges and quantization errors of every input
WorstCaseAnalysis();
DAGDescription();                                  // worst-case analysis run
 while true
    if OperatorTrimming() = 0
        break;
    end if
 end while
RuntimeAnalysis();
foreach $\mathbf{V}$ in $\mathbf{S}$                          // $\mathbf{V}$ is an input vector in $\mathbf{S}$
    foreach $x$ in Fin objects
        $x$.sample($v_x$);
    end for               // $v_x$ is the value in $\mathbf{V}$ corresponding to input $x$
    DAGDescription();                                // runtime analysis runs
    UpdateError($error$);               // record runtime quantization errors
    UpdateRange($range$);    // update runtime maximum dynamic ranges
end for
end procedure
```

Figure A.2: The pseudocode for error measurement and range estimation.

of the preceding operators to which they connect. If the input "truncation" parameters are non-zero, the precision formats are adjusted accordingly (Section 3.4.4). The quantization errors at the inputs are recorded.

(b) The arithmetic core (Section 3.4.1) derives the dynamic range of this operation and the minimum precision format to produce bit-exact results.

(c) The outputs of the operator copy the precision formats from the arithmetic core. If the output "truncation" or "expansion" parameters are non-zero, the precision formats are adjusted accordingly (Section 3.4.4). The quantization errors at the outputs are recorded.

Subsequent `ModuleBegin()` and `ModuleEnd()` function calls are ignored.

3. The DAG description is executed via `DAGDescription()` and a worst-case analysis of the algorithmic function based on the sample dataset is produced.

4. Operator trimming is performed (Section 3.5.2).

5. A runtime analysis flag is set via a `RuntimeAnalysis()` function call. Note that upon reaching this step the precision formats of all the inputs and outputs are already derived. In the case that the flag is set, subsequent operator calls will carry out the following operations:

   (a) The inputs of the operator take values from the outputs of the preceding operators to which they connect, and reformat the representation of the value to the precision format that is already set.

   (b) The arithmetic core performs the required computation. The bit width of the operation is the minimally required one which produce bit-exact results with respect to the inputs.

(c) The outputs copy the intermediate results from the arithmetic core, and reformat the representation of the value to the precision format that is already set.

Subsequent `ModuleBegin()` and `ModuleEnd()` function calls are ignored.

6. For every input of the module to be simulated, a `sample()` method is made and the next sample input vector is taken.

7. The procedure `DAGDescription()` is executed again. Since the function call `RuntimeAnalysis()` is dispatched, the operator calls carries out runtime analysis of the algorithmic function with an entry from the sample dataset.

8. The same set of operations as in the DAG description is performed in double-precision floating-point representations. The error produced (the absolute difference between the fixed-point and the floating-point values) are computed. The error at the inputs and outputs of all the operators are recorded.

9. The inputs and outputs contain variables which states the maximum run-time dynamic ranges so far exhibited. These ranges are updated accordingly.

10. Steps 6 to 9 are repeated for every entry in the sample dataset.

11. After all the entries in the sample dataset are processed, the inputs and outputs of the operators contain the worst-case and runtime analyses of dynamic ranges and quantization errors using the supplied set of parameters and the sample dataset as inputs.

A.4 Trimming of Operators

As described in Section 3.5.2, trimming of operators uses an iterative approach. The *fp* kernel invokes the operator trimming procedure by consecutive calls to `OperatorTrimming()` until the function returns zero. The pseudocode of the operator trimming procedure for a module m is shown in Figure A.3. The procedure first checks on each operator whether its outputs are all of zero wordlength or are connected to trimmed operators. If the first check fails, the procedure continues to check whether the inputs of the operator are all of zero wordlength or are connected to trimmed operators. If either case is satisfied, the operator is trimmed, else, if only one input is of non-zero wordlength and the operator class is transformable, the transformation is carried out and the operator is also trimmed. Otherwise, the operator remains untrimmed.

An operator trimming procedure that returns zero indicates that no further trimming can be made. Afterwards, the simulation DAG is updated accordingly, so that in subsequent stages of simulation the trimmed operators are neglected.

A.5 Latency and Throughput Calculation

To calculate the latency and throughput for an implementation of the algorithmic function, the same DAG description as in error extraction and range estimation is used. However, prior to calling this DAG description, the function call `LatencyAndThroughputCalculation()` (Figure A.4) is performed for every operator. The procedure queries the module library for the latency and throughput of the operator, lat_{op} and tp_{op}, and progressively calculates the latency LAT and throughput TP of the whole module. LAT and TP are global variables, which are initialized to zero as the first step of this procedure.

```
procedure OperatorTrimming()
    input $m$
    output $trimmed$
begin
$trimmed$ = 0;
foreach $i$ in $m$.operators                         // $i$ is an operator of $m$
    $trim$ = true;
    foreach $j$ in $i$.outputs
        if $j$.wordlength $\neq$ 0
            foreach $k$ in $j$.succeed   // $k$ is a succeeding operator of $i$
                if $k$.trimmed = false
                    $trim$ = false;
                end if
            end for
        end if
    end for
    if $trim$ = false
        $count$ = 0;
        foreach $j$ in $i$.inputs
            if $j$.wordlength $\neq$ 0
            and $j$.precede.trimmed = false
                                  // $j$.precede is the preceding operator of $i$
                $count$ = $count + 1$;
            end if
        end for
    end if
```

Figure A.3: The pseudocode for operator trimming.

```
        **if** $trim$ = **true or** $count$ = 0
        **or** ($count$ = 1 **and** $i$.transformable = **true**)
            $trimmed$ = $trimmed + 1$;
            $i$.trimmed = **true**;                         // *operator trimmed*
            **if** $count$ = 1
                $i$.transform();
            **end if**
        **end if**
    **end for**
    **return** $trimmed$;
    **end procedure**
```

Figure A.3 (continued): The pseudocode for operator trimming.

In the latency and throughput calculation procedure, the accumulated latency of an operator is the number of cycles taken from the instant an input vector enters the module input to the instant the operator's output result becomes valid.

As an example, suppose an input vector enters the module at the zeroth cycle. The intermediate results corresponding to this input vector arrives at the inputs of an operator at the m-th clock cycle. If this operator has an n-cycle latency, then the outputs from this operator which corresponds to that input vector is valid at the $(m+n)$-th clock cycle. In this case, this operator has an accumulated latency of $m + n$ clock cycles.

A detailed description of the pseudocode in Figure A.4 is given below:

1. The class name of the operator *op* and its current parameters are passed to the module library. The module library returns the latency and the

```
**procedure** LatencyAndThroughputCalculation()
    **input** $op$
**begin**
$\{lat_{op}, tp_{op}\}$ = Enquire($op$.class, $op$.param);
                                        // *query the module library*
$align$ = 0;
**foreach** $i$ **in** $op$.inputs
    $align$ = Max($i$.precede.acclatency, $align$);
// $i$.precede.acclatency *is the accumulated latency of the preceding operator*
**end for**
**foreach** $i$ **in** $op$.inputs
    $i$.InsertStageLatch($align - i$.precede.acclatency);
                    // *insert stage latches for every input whenever appropriate*
**end for**
$op$.acclatency = $align + lat_{op}$;     // *update the latencies of the operator*
$LAT$ = Max($LAT$, $align + lat_{op}$);
$TP$ = Max($TP$, $tp_{op}$);
                         // *update the latencies and throughput of the module*
**end procedure**
```

Figure A.4: The pseudocode for the calculation of latency and throughput cycles.

throughput of *op*.

2. There is a class variable (*acclatency*) in every operator class indicating the accumulated latency of every operator in the current configuration. The procedure interrogates every preceding operator of *op* and determines the maximum accumulated latency among these operators (*align*).

3. Stage latches are inserted individually between every pairs of inputs and their preceding outputs whenever the operands are not time-aligned.

4. The accumulated latency of *op* is updated. Since its inputs are valid at the *align*-th cycle (*align* is a local variable in the procedure) after an input vector enters the module, the accumulated latency of *op* should be *align* plus lat_{op}, the latency of *op* itself.

5. The latency of the module, LAT, should have the same value as the maximum accumulated latency among all its operator.

6. The throughput of the module, TP, is determined by the operator with the smallest throughput (the largest number of clock cycles between consecutive inputs). Therefore, TP should updated if TP is less then tp_{op}.

7. As the final step, if the module has more than one outputs, stage latches are inserted at the outputs so as to make them time-aligned.

A.6 Insertion of Stage Latches

The time-alignment process, achieved by inserting stage latches between inputs and outputs of operators, ensures all the operands (the first digit of all the operands in a digit-serial architecture) enter an operator at the same clock cycle. The pseudocode for the stage latch insertion procedure is given in Figure A.5.

In this procedure, sharing of stage latches among sibling operators (those with common preceding operator) is carried out in order to reduce area.

A.7 Cost Functions Using the Bessel Function

Referring to Equation 3.3,

$$f'_{cost}(\mathbf{V}') = f'_E(\sum_{z\in\mathbf{Z}} \frac{k_z}{SNR_z}) + f'_A(A),$$

f'_E should be minimum when the errors at outputs are at the user specified precisions. f'_E is increasing along all directions from the minimum point. f'_A should be monotonically increasing with A for $A \geq 0$ since the estimated area cannot be negative.

Cost functions that have the properties described above can be constructed by the Bessel functions [AS65]. The plot of the Bessel function of the first kind $J_v(x)$ for $v, x = [0.0, 10.0)$ is shown in Figure A.6.

Consider f'_E and f'_A independently. For some values of x, $J_v(x)$ gives proper functions for f'_E or f'_A. Specifically, $-J_v(4.0)$ and $-J_v(0.9)$, $v \in \mathbb{R}^+$, as shown in Figure A.7, for the constructions of f'_E and f'_A respectively, are of interests. The values of x for the Bessel functions were obtained by experiments. Denote

$$E = \sum_{z\in\mathbf{Z}} \frac{k_z}{SNR_z},$$

by shifting and scaling of $-J_v(4.0)$ and $-J_v(0.9)$ as

$$\begin{aligned} f'_{cost}(\mathbf{V}') &= f'_E(E) + f'_A(A) \\ &= k_E(0.5 - J_{k_eE}(4.0)) + k_A(1.0 - J_{k_aA}(0.9)) \end{aligned} \tag{A.1}$$

where k_E and k_A are respectively the weighting factors of the precision and area components, k_e and k_a are respectively the scaling factors for the precision and

```
procedure InsertStageLatch()
    input cycles                          // cycles of stage latch to be inserted
begin
sharewith = this.precede;
sharedcycles = 0;
toshare = null;
tosharecycles = $+\infty$;
foreach i in this.precede.succeed                  // i is a sibling operator
    if i.inserted = true
        if $sharedcycles < i.\texttt{cycles} < cycles$
            sharewith = i;
            sharedcycles = i.cycles;
        end if
          // find the sibling operator that offers maximal resource sharing
        if $cycles < i.\texttt{cycles} < tosharecycles$
            toshare = i;
            tosharecycles = i.cycles;
        end if
          // find the sibling operator that should be given resource sharing
    end if
end for
```

Figure A.5: The pseudocode for the insertion of stage latches.

```
    Connect( this, $sharewith$, $cycles - sharedcycles$);
    if $toshare \neq$ null
        Connect($toshare$, this, $tosharecycles - cycles$);
    end if                    // connect ports and insert stage latches in-between
    this.cycles = $cycles$;
    this.inserted = true;
end procedure
```

Figure A.5 (continued): The pseudocode for the insertion of stage latches.

area measures, $k_E, k_A, k_e, k_a \in \mathbb{R}^+$, the resultant function f'_{cost} is suitable for representing the tradeoff between area and precision. For instance, if a module has only one output z and this output must have an SNR greater than 40 dB and the optimization objective is to minimize the area, one can set $k_z = 1.0$, $k_E = 1.0$, $k_A = 1.0$, $k_e = 6.7 \times 10^{-2}$ and $k_a = 1.0 \times 10^{-3}$. A plot of Equation A.1 with these coefficients is shown in Figure A.8. It can be seen that when $E = 40$, corresponding to a SNR of 40 dB at the output, and $A \to 0$, the cost function is minimized. To prevent implementations with output SNR less than 40 dB being chosen, it is often necessary to specify an error constraint as well (Section 3.6.2).

A.8 Latency of Conversion Modules

The pseudocode for calculating the latency of a conversion module is shown in Figure A.9. The procedure takes the input format (w_a, f_a, d_a), the output format (w_b, f_b, d_b) and the number of digits n as inputs. The goal of the procedure is to determine which input bits are present on every clock cycle and to compare

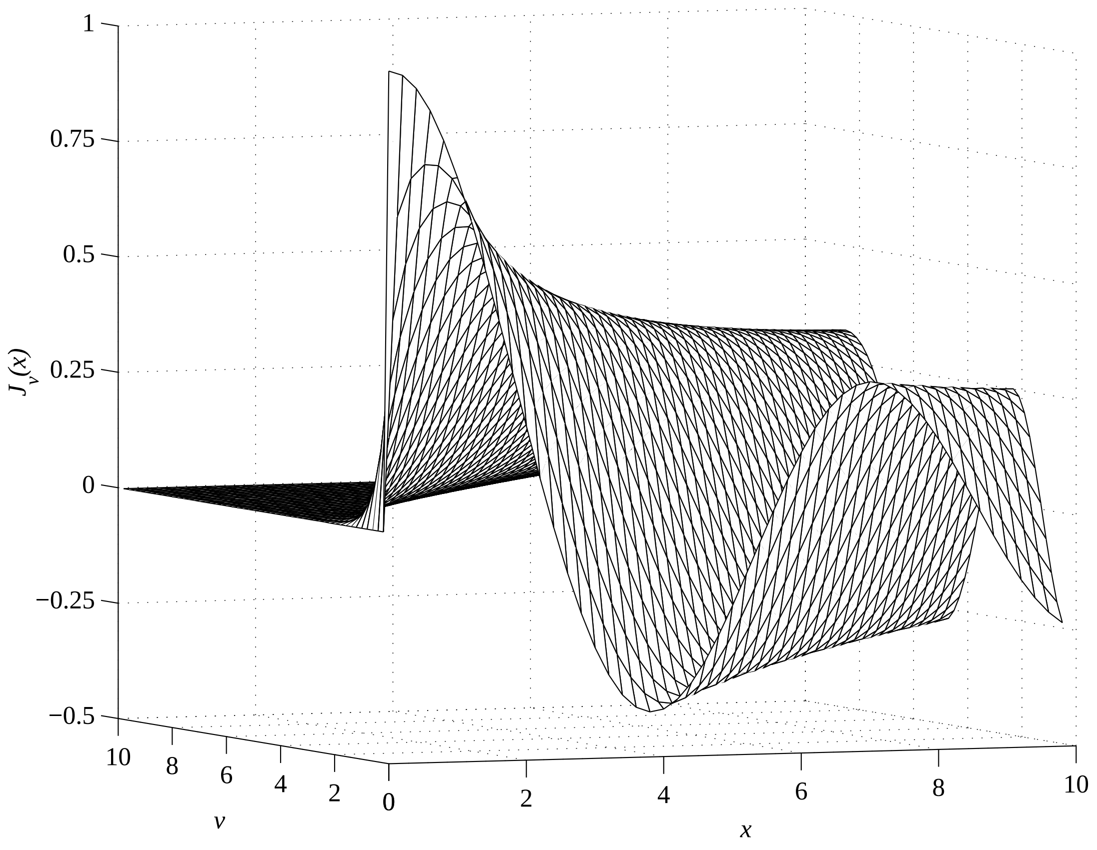


Figure A.6: The Bessel function of the first kind $J_v(x)$.

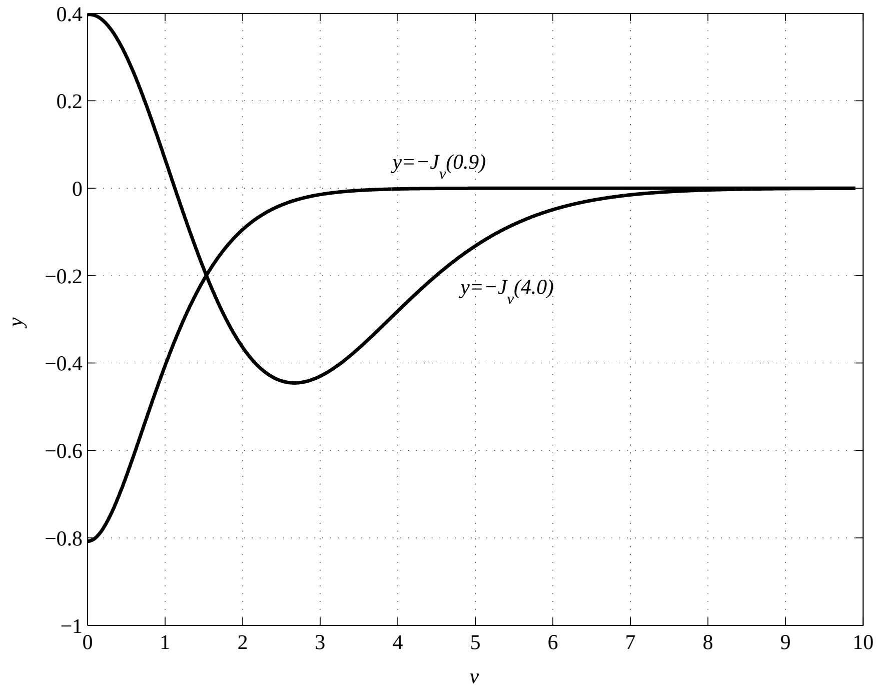


Figure A.7: Plots of the Bessel functions selected for constructing functions f'_E and f'_A, namely $-J_v(4.0)$ and $-J_v(0.9)$.

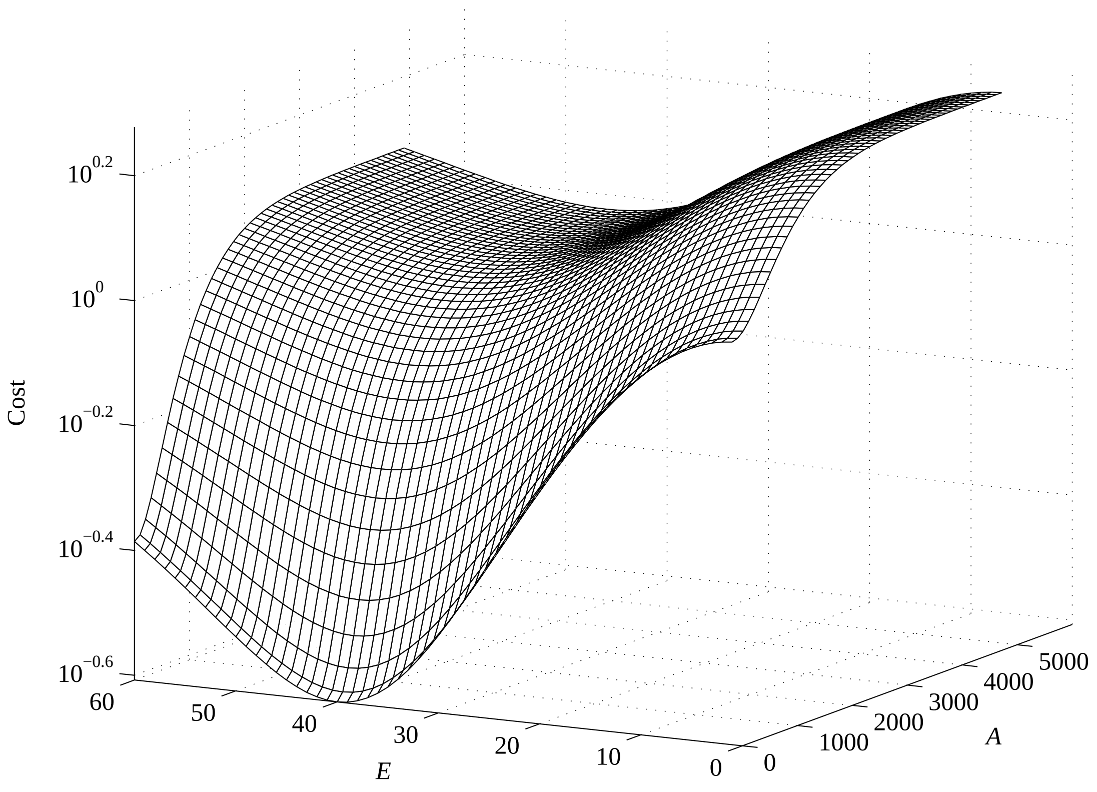


Figure A.8: The cost function f'_{cost} with coefficients $k_z = 1.0$, $k_E = 1.0$, $k_A = 1.0$, $k_e = 6.7 \times 10^{-2}$ and $k_a = 1.0 \times 10^{-3}$.

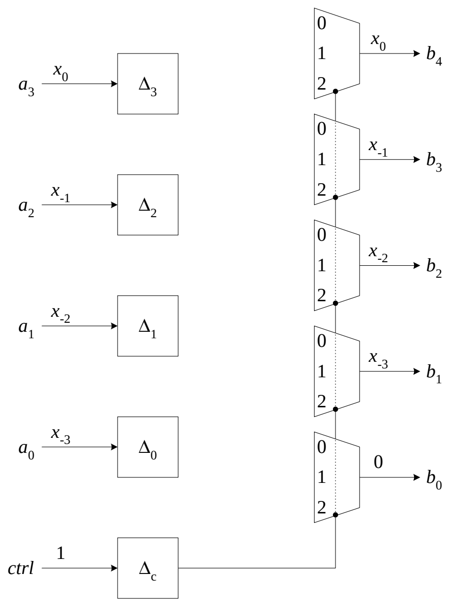
them with those required at the outputs. The procedure begins by assuming the latency is zero. If it is found that at any clock cycle the required output bits are not yet available, the latency is incremented. The procedure continues until all the output bits are assigned.

To illustrate the construction of a conversion module, the example in Section 4.3 is used (from $(12, 3, 4)$ to $(15, 4, 5)$). Its latency was found to be one cycle. Therefore, referring Figure 4.3, when *ctrl* is high, $a = \{x_0, x_{-1}, x_{-2}, x_{-3}\}$. In the next two clock cycles, $a = \{x_4, x_3, x_2, x_1\}$ and $a = \{a_8, a_7, a_6, a_5\}$ respectively.

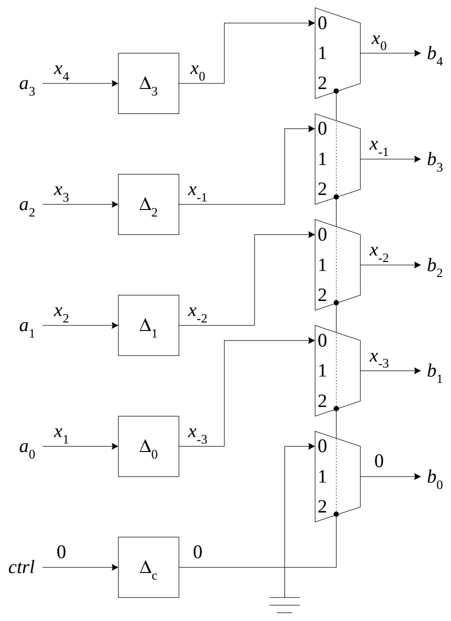
The construction of this conversion module begins by introducing a shift register which is 4-bit wide (input digit size is four), one-cycle long (latency is one cycle), and adding three-to-one multiplexers (number of digits is three) before every output (Figure A.10(a)). The control signal, *ctrl*, being set high in the zero clock cycle resets the counter Δ_c to zero in the first clock cycle. On the first clock cycle, the contents of the shift register and the input corresponds to $\{x_0, x_{-1}, x_{-2}, x_{-3}\}$ and $\{x_4, x_3, x_2, x_1\}$ respectively, hence the correct bits are connected to the zeroth input of the multiplexers (Figure A.10(b)). This process continues for the second and the third clock cycle (Figures A.10(c) and A.10(d)). In this example the shift registers and the multiplexers cannot be further simplified but in other cases, particularly when the number of digits is large, simplifications can be made.

```
procedure LatencyOfConversionModule()
    input $(w_a, f_a, d_a), (w_b, f_b, d_b), n$
    output $latency$
begin
$latency = 0$;
$i = 0$;
$x = -f_a + d_a - 1$;
$y = -f_b + d_b - 1$;
while $i \neq n - 1$
    if $y > x$
        $latency = latency + 1$;
          // running out of input bits, need an additional cycle of latency
    else
        $i = i+1$; // found all required output bits in registers or input bits
    end if
    $x = x + d_a$;
    $y = y + d_b$;
end while
end procedure
```

Figure A.9: The pseudocode for calculating the latency of a conversion module.

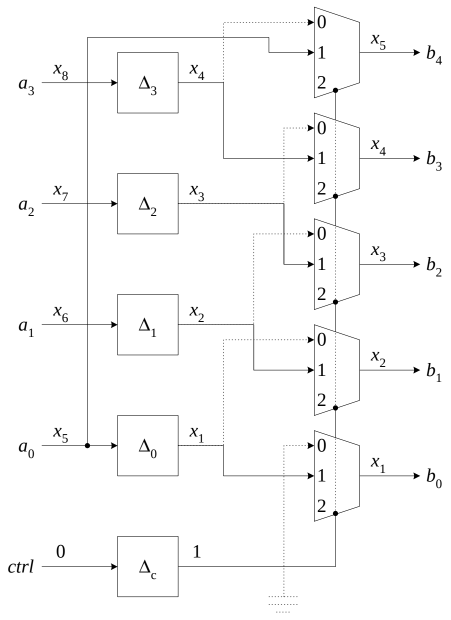


(a) the zeroth clock cycle



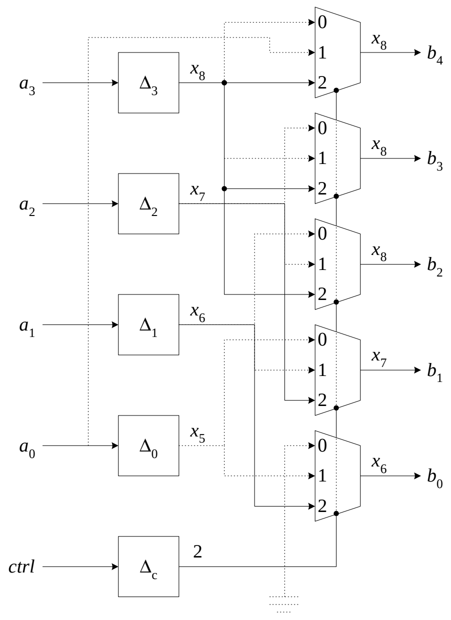
a_3 x_4 Δ_3 x_0
a_2 x_3 Δ_2 x_{-1}
a_1 x_2 Δ_1 x_{-2}
a_0 x_1 Δ_0 x_{-3}
ctrl 0 Δ_c 0
x_0 b_4
x_{-1} b_3
x_{-2} b_2
x_{-3} b_1
0 b_0

(b) the first clock cycle



a_3 x_8 Δ_3 x_4
a_2 x_7 Δ_2 x_3
a_1 x_6 Δ_1 x_2
a_0 x_5 Δ_0 x_1
ctrl 0 Δ_c 1
x_5 b_4
x_4 b_3
x_3 b_2
x_2 b_1
x_1 b_0

(c) the second clock cycle



(d) the third clock cycle

Figure A.10: Construction of the conversion module from a precision format of $(12, 3, 4)$ to $(15, 4, 5)$.

Appendix B

Direct Modification of FPGA Design Bitstreams

There are many situations in which FPGA designs can be efficiently reused. For example, suppose there is an FPGA design for solving a particular problem, in which the parameters to the problem are defined in the FPGA logic blocks. To solve the same problem with a different set of parameters, it may not be necessary to re-synthesize the design. Instead, the same result may be achieved by modifying the contents of the corresponding FPGA design bitstream. This reduces the turnaround time by several orders of magnitude. In this appendix, a technique for modifying FPGA bitstreams is presented.

The appendix begins with a review of the FPGA design flow and the motivation for directly modifying the FPGA design bitstream. First, a description of how an FPGA configuration is encoded by a design bitstream is given. Following this, a detailed description is given of a method to modify the FPGA bitstream. The utility of this technique is explored in a few empirical examples.

B.1 Background

An FPGA must be configured before it can be used. Configuration is the process of loading a design bitstream into the FPGA internal configuration memory, so that the logic resources of the FPGA are programmed to implement a designated function. The design bitstream defines the programmable contents of all the components in an FPGA, including LUTs, registers, routing, I/O pads, delay-locked loops (DLLs) and possibly on-chip memory (such as the Xilinx Virtex series BlockRAM components [Xil00c]) and multipliers (such as the Xilinx Virtex-II series 18×18-bit multipliers [Xil01a]).

The FPGA design flow consists of a number of steps, namely high-level synthesis, technology mapping, placement and routing (P&R), and bitstream generation. Most of these procedures are computationally expensive, particularly for P&R which is a non-deterministic polynomial-time (NP) complete problem. The execution time and memory requirements for a typical execution of the procedures in the FPGA design flow are listed in Table B.1. Results in this table were obtained from an 1.5 GHz Intel Pentium-IV machine with 512 Mbytes memory for a design which utilizes 42% resources of a Xilinx Virtex XCV-1000 device (an implementation of the electronic cochlea model described in Chapter 8). The synthesis and implementation tools used were Synopsys FPGA Express 3.5 and Xilinx Foundation 3.3i respectively.

For some problems, the parameters to the algorithm can be solely defined in the LUTs of an FPGA implementation. Example problem parameters include the encryption/decryption key of a cryptographic algorithm and the microcode of a processor. To change these parameters in the high-level description requires a re-execution of the entire FPGA design flow which is very time-consuming. An alternative is to directly manipulate the bitstream so that the parameters

Procedure	Execution time	Memory requirements
High-level synthesis	2000 seconds	420 Mbytes
Technology mapping	100 seconds	160 Mbytes
Placement and routing	1500 seconds	260 Mbytes
Bitstream generation	45 seconds	165 Mbytes

Table B.1: Execution time and memory requirements for typical executions of the procedures in the FPGA design flow.

encoded in the logic are modified accordingly. This is achievable provided that, first, the format of the design bitstream is known and second, the mapping of the logic to the physical locations in the logic cell (LC) array of the FPGA is known. Using direct bitstream modification, a design needs to go through the procedures in Table B.1 only once. Intuitively, bitstream modification should take much less time because the bitstream is relatively small in size (for example, a Xilinx XCV1000 bitstream has 823515 bytes) and the process usually involves simple modifications to a small number of locations.

Previously the format of most design bitstreams was not documented, mainly to prevent reverse engineering of designs, with the exception of the Xilinx XC6200 series FPGA. The Xilinx XC6200 series FPGA supports runtime reconfiguration, either partial or non-partial, and its bitstream format is documented [Xil97]. The idea of runtime reconfiguration is to divide an application into a series of stages and implement them as separate circuit modules executing sequentially on the FPGA device. In the case of partial reconfiguration, transition of configurations can be accomplished by downloading only the concerned portion of the updated bitstream. Design methodologies for runtime reconfigurable systems have been an active research topic [HH95, HW95b, BDH+97, LSC96, LSC97, LM98]. In these work, the primary focus is to derive efficient strategies of scheduling the circuit modules for executing on the FPGA device.

An advantage of runtime reconfiguration is reduced turnaround time because different pre-synthesized modules can be chosen to form an implementation and re-synthesis is therefore not needed. The technique of direct bitstream modification has the same objective. It is similar to runtime reconfiguration, in which the bitstream are directly manipulated to reduce turnaround time, but the former changes the contents of bitstreams before downloading to the FPGA device while the latter changes part of the FPGA configuration during runtime by downloading a partial reconfiguration bitstream. Direct bitstream modification is applicable to any FPGA device which its bitstream format is known. In contrast, partial reconfiguration is applicable only to FPGA devices which have a reconfiguration interface.

There has been some research work that makes use of the knowledge concerning the formats of bitstreams and directly modifies the bitstreams so as to eliminate the need of re-synthesis. Previous work of bitstream modification for the Xilinx XC6200 series FPGAs includes the implementation of the global satisfiability (GSAT) algorithm by Wong et. al. [WYLL99] and the satisfiability solver by Abramovic, Sousa and Saab [ASS99]. Both the implementations have the Boolean equations (known as clauses) stored in LUTs. If the problem itself does not require as many clauses and variables as the hardware supports, the unused hardware can be left on the device without affecting the functionality. This is to say, it is only required to modify the part of the design bitstream that are related to the LUTs which encoded the Boolean equations.

The bitstream configurable clause evaluator for satisfiability (SAT) problems on Xilinx XC4000 series FPGAs by Leong and Chung [LC99] uses an approach similar to the above works, in which the Boolean equations are encoded solely in

LUTs. As the design bitstream format of Xilinx XC4000 series FPGAs was not disclosed, in this work the relevant locations in the bitstream to be modified were located by matching the resultant bitstream with the high-level description. The Data Encryption Standard (DES) implementation by Patterson [Pat00] stored the encryption key in LUTs and modified the bitstream of a Xilinx Virtex device via the JBits Application Programming Interface (API) [GLS99].

Unfortunately, the Xilinx XC6200 series has been discontinued and the largest capacity it offered is insufficient for many applications (the largest Xilinx XC6200 series device, XC6264, has 82000 system gates versus four million for the Xilinx Virtex-E XCV3200E). The bitstream format of Xilinx Virtex series is documented [Xil00e] and it is relatively easy to modify design bitstreams directly to reduce turnaround time. The research that utilized this technique include an implementation of the walk-SAT (WSAT) algorithm [LSW$^+$01] and two implementations of the International Data Encryption Algorithm (IDEA) [CTLL01]. In this dissertation, one of these IDEA implementations (the bit-serial implementation) was presented in Chapter 6.

B.2 Configuration Bitstream

The configuration bitstream of an FPGA is essentially a concatenation of the configuration data of all programmable components. In practice, the bitstream also has tags, control words and cyclic redundancy checksums (CRCs) for validation, so that the FPGA device can identify malformed bitstreams and prevent them from being used and possibly damaging the device.

Taking the Xilinx Virtex series as an example, its configuration bitstream is organized in frames. Frames are read and written sequentially during download and readback operations. The size of a frame depends on the capacity of the device. Multiple frames form a configuration column and the concatenation of these columns forms the main body of a design bitstream. The bitstream header includes the configuration rate, readback capability, the clock to be associated during bitstream download and other miscellaneous information. At the end is a 16-bit CRC checksum of the polynomial $X^{16} + X^{15} + X^2 + 1$. To minimize the area overhead for the configuration logic, the configuration memory of adjacent components are linked to form a large shift register during configuration. This also explains why the design bitstream is organized as a concatenation of configuration rows [Xil00f, Xil00e].

B.3 Bitstream Modification

Bitstream modification, as described in this dissertation, involves modifying the parameters to a problem. There are two stages in bitstream modification. In the first stage, the locations of the LUTs and BlockRAMs to be modified in the bitstreams are extracted, while in the second stage these locations are updated with the new parameters and the CRC checksum is recomputed.

B.3.1 Location Extraction

The stage of location extraction proceeds as follows:

1. The vendor's implementation tools are used to generate a circuit description of the design suitable for machine parsing. For the Xilinx tools, such a tool is called *ncdread* and the circuit descriptions have an extension *.ncd*. The circuit description contains the mapping between variables in the HDL or schematic and the physical locations of their corresponding LUTs [Xil00b].

2. Locate the physical locations of the LUTs corresponding to the variables which need to be changed. This is achieved by referring to the circuit description.

3. During routing, the four inputs of a LUT may be permuted for optimization purposes. The set of new data, if necessary, should be permuted in the same way as the router did before modifying the bitstream.

To illustrate this procedure, the design bitstream for the bit-serial implementation of the IDEA cipher (Chapter 6) is used as an example. The target device of this design is a Xilinx XCV300-6 FPGA. In this implementation, the variable `round_0/addconst_0/k` is the output of a shift register LUT (SRL) primitive, and this LUT stores one of the 16-bit subkeys of the IDEA cipher with an original value of 0xE81A. The relevant output of *ncdread* is shown in Figure B.1. Note that the name of the output is derived from the hierarchical VHDL description.

From the output of *ncdread*, it can be observed that this SRL primitive is located at the zeroth slice of the nineteenth row, twenty-eighth column of the configurable logic block (CLB) array. The initial content of an SRL primitive is always stored at LUT G (corresponds to pin Y, as shown in line 20). The content of the LUT in this bitstream is shown in line 7, after the string `G:#RAM:D=`. This LUT location is referred as $CLB\_R19C28.S0.G$.

If the storage element concerned is not an SRL primitive but is a ROM primitive, then it is necessary to identify whether the required content is stored in LUT F or G. This can also be achieved by parsing the output of *ncdread*. An extract of the output of *ncdread*, as shown in Figure B.2, is used as an example. Lines 22 and 23 shows the mappings between the variable names in high-level description and the output pins of the Virtex slices. Outputs X and Y correspond to LUTs F and G respectively.

```
NC_COMP:2969 - <round_0/addconst_0/k>
site = CLB_R19C28.S0
        Config String: <CYSELF:#OFF CYSELG:#OFF
        CKINV:1 COUTUSED:#OFF YUSED:0 XUSED:0 XBUSED:#OFF
        F5USED:#OFF YBMUX:#OFF CYINIT:#OFF DYMUX:#OFF
        DXMUX:#OFF CY0F:#OFF CY0G:#OFF F:#LUT:D=0
        G:#RAM:D=0xE81A RAMCONFIG:1SHIFT REVUSED:#OFF
        BYMUX:BY BXMUX:#OFF CEMUX:#OFF SRMUX:SR GYMUX:G
        FXMUX:F SYNC_ATTR:#OFF SRFFMUX:#OFF INITY:#OFF
        FFX:#OFF FFY:#OFF INITX:#OFF>
    23 pins -
      pin 1 - BY: <round_0/addconst_0/k>
      pin 4 - CLK: <clk_BUFGPed>
      pin 12 - G1: <GLOBAL_LOGIC1_110>
      pin 13 - G2: <GLOBAL_LOGIC1_110>
      pin 14 - G3: <GLOBAL_LOGIC1_105>
      pin 15 - G4: <GLOBAL_LOGIC1_105>
      pin 16 - SR: <round_0/addconst_0/const_0/ken>
      pin 17 - X: <GLOBAL_LOGIC0_119>
      pin 20 - Y: <round_0/addconst_0/k>
```

Figure B.1: An extract of the output of *ncdread* showing the mapping of a variable in the high-level description and its physical location. The storage element concerned is an SRL primitive.

```
NC_COMP:207 - <round_0/w2> site = CLB_R20C28.S0
        Config String: <CYSELF:#OFF CYSELG:#OFF
        CKINV:1 COUTUSED:#OFF YUSED:#OFF XUSED:#OFF
        XBUSED:#OFF F5USED:#OFF YBMUX:#OFF CYINIT:#OFF
        DYMUX:1 DXMUX:1 CY0F:#OFF CY0G:#OFF
        F:#LUT:D=(((~A1*A2)+(A1*(A4@~A2)))@A3)
        G:#LUT:D=((~A1*(A3*A4))+(A1*((~A2*(A3+A4))+
        (A2*(A3*A4))))) RAMCONFIG:#OFF REVUSED:#OFF
        BYMUX:#OFF BXMUX:#OFF CEMUX:#OFF SRMUX:SR
        GYMUX:G FXMUX:F SYNC_ATTR:ASYNC SRFFMUX:0
        INITY:LOW FFX:#FF FFY:#FF INITX:LOW>
    23 pins -
      pin 4 - CLK: <clk_BUFGPed>
      pin 6 - F1: <round_0/addconst_0/add_0/acc<1>>
      pin 7 - F2: <round_0/addconst_0/k>
      pin 8 - F3: <w12>
      pin 9 - F4: <ctrl<0>>
      pin 12 - G1: <round_0/addconst_0/add_0/acc<1>>
      pin 13 - G2: <ctrl<0>>
      pin 14 - G3: <w12>
      pin 15 - G4: <round_0/addconst_0/k>
      pin 16 - SR: <N_reset>
      pin 19 - X: <round_0/w2>
      pin 22 - Y: <round_0/addconst_0/add_0/acc<1>>
```

Figure B.2: An extract of the output of *ncdread* showing the mapping of a variable in the high-level description and its physical location. The storage element concerned is a ROM primitive.

To verify the physical location which has been extracted, the contents of the LUT can be retrieved using *readlut*, a tool developed in this research work. For instance, executing `readlut idea.bit 4 11 0 F` would return the value of the LUT at location $CLB\_R4C11.S0.F$.

To locate the bits in a design bitstream correspond to a LUT, the tool first analyzes the header of the design bitstream to identify which Virtex device the design bitstream is intended for. From this, the tool looks up the number of words in a frame, FL, for the bitstream of this specific device. Next, using FL, the row and column numbers, and the slice and LUT numbers, it calculates the frame major and minor addresses, the frame word index and the bit index (MJA, MNA, fm_wd, $fm\_wd\_bit\_idx$). With these values, the actual offsets of the bits corresponding to the desired LUT are computed [Xil00e]. Note that this offset refers to the start of the configuration data in the design bitstream. The design bitstream also contains packet headers and other non-configuration packet data. The tool ignores these non-configuration bits to identify the actual offsets of the bits with respect to the design bitstream [Xil00f]. As an example, the bit offsets (with respect to the design bitstream file) of the bits corresponding to $CLB\_R17C23.S0.G$ on a Xilinx Virtex XCV300 device are $125083 + i \times 672$, $i = \{0, 1, \ldots, 15\}$.

If the storage element is a ROM primitive, the design tools may permute the inputs of the LUT for optimization purposes (the ROM contents are also changed accordingly). To address this problem, the LUT content may be set to a special value, such as 0x3569, during synthesis. For four-input LUTs, there are $4! = 24$ possible permutations. As shown in Table B.2, with this special value any permutation can be identified by a lookup of the value stored in the bitstream. For instance, if the value stored in the bitstream is 0x16AD, then the first and the third inputs are swapped, and the second and the fourth inputs are swapped.

Permutation	LUT content	Permutation	LUT content
$\{a, b, c, d\}$	0x3569	$\{a, b, d, c\}$	0x3659
$\{a, c, b, d\}$	0x1D69	$\{a, c, d, b\}$	0x16D9
$\{a, d, b, c\}$	0x1E65	$\{a, d, c, b\}$	0x16E5
$\{b, a, c, d\}$	0x5369	$\{b, a, d, c\}$	0x5639
$\{b, c, a, d\}$	0x4769	$\{b, c, d, a\}$	0x4679
$\{b, d, a, c\}$	0x562D	$\{b, d, c, a\}$	0x526D
$\{c, a, b, d\}$	0x1B69	$\{c, a, d, b\}$	0x16B9
$\{c, b, a, d\}$	0x2769	$\{c, b, d, a\}$	0x2679
$\{c, d, a, b\}$	0x16AD	$\{c, d, b, a\}$	0x1A6D
$\{d, a, b, c\}$	0x1E63	$\{d, a, c, b\}$	0x16E3
$\{d, b, a, c\}$	0x364B	$\{d, b, c, a\}$	0x346B
$\{d, c, a, b\}$	0x16CB	$\{d, c, b, a\}$	0x1C6B

Table B.2: The contents of a LUT before and after permutation of input bits, with 0x3569 being the original LUT value and $\{a, b, c, d\}$ being the original permutation.

These special values can be obtained by picking four hexadecimal digits from 3, 5, 6, 9, A and C (the four-digit binary representation of these values have two ones and two zeros), and concatenate them in any order to form a four-digit hexadecimal number.

The output of the first stage is a table containing the physical locations and permutations (if any), for every entry in the list of LUTs to be modified.

B.3.2 Bitstream Updating

The steps involved in updating the bitstream are as follows:

1. Obtain the physical locations of the set of LUTs to be modified from the first stage. Obtain also the set of input permutations (if necessary).
2. Write the new data to the bitstream. The offsets in the bitstream to be written are calculated by the documentation provided by the vendor [Xil00e].
3. Re-compute the CRC checksum and write it back to the bitstream.

Having the table containing the physical locations and permutations of the list of LUTs to be modified, the second stage is relatively a straightforward process. A tool called *writelut* was developed to perform this task. For example, executing `writelut idea.bit 13 2 1 G A47D` would put the value 0xA47D to the LUT at $CLB\_R12C2.S1.G$. The mechanism of locating the bits that correspond to a LUT in *writelut* is similar to that of *readlut*. When modifying the last LUTs, a `crc` parameter should be supplied to the *writelut* tool, such as `writelut idea.bit 13 2 1 G A47D crc`. This will update the LUT concerned, as well as updating the 16-bit CRC of the bitstream. The algorithm for computing the CRC checksum uses the algorithm provided by Xilinx [Xil00e].

B.3.3 Experiments and Applications

The extraction stage is carried out once for every design and the updating stage is carried out whenever the design should be modified with a new set of parameters. Experiments on a 333 MHz Intel Pentium II machine show that the first stage takes approximately five seconds to locate the physical locations and permutations of 2200 LUTs. The second stage requires less than one second. Compared with the traditional synthesis and implement procedures which requires time in the order of tens of minutes, bitstream modification enjoys a three orders of magnitude speedup.

The four-input LUTs in a Virtex LC can be configured as a 16-bit ROM primitive. A Virtex slice, which includes two LCs and two storage elements, can be configured as two 16-bit SRL primitives [Xil00g]. In a traditional FPGA design, these primitives are considered as the storage elements for constants. In this work, these primitives were identified as changeable cells through direct bitstream modifications.

The WSAT implementation [LSW$^+$01] mentioned in Section B.3 utilizes an array of LCs configured as ROMs, each of which formulates a function of three literals. This architecture enabled the modification of the design bitstream to solve other 3-SAT problems (SAT problems with each clause containing at most 3 literals). The IDEA implementations [CTLL01] (the bit-serial implementation was presented in Chapter 6) have the key schedule initially stored in SRL primitives. Hence, an implementation with a different key schedule can be obtained by modifying the initial SRL contents.

B.4 Summary

The use of bitstream modification to improve turnaround time in the FPGA design flow was presented. This technique first constructs a “template” which contains the algorithm to be implemented and with the parameters locked to specific LUTs. Using vendor’s tools the mapping between the variables in the high-level design and the low-level LUT physical location are be tracked. Hence, it is possible to modify the bitstream directly to obtain an implementation for the same problem with a different set of parameters. This technique obviates the need for the time-consuming FPGA design flow for any compilation following the first one. Experiments showed a three orders of magnitude speed improvement can be obtained.

Appendix C

Distributed Arithmetic

Distributed arithmetic (DA) is a computational algorithm that perform multiplications with LUTs. Specifically, DA is targeted for sum of products (SOP) operations with one variable unchanged during execution, an operation widely used in DSP filtering and frequency transforming functions [Xil96].

To derive the DA algorithm, consider the SOP, S, of N terms

$$S = \sum_{i=0}^{N-1} k_i x_i \tag{C.1}$$

where k_i is the fixed weighting factor and x_i is the input. For two's complement fractions, the numerical value of $x_i = \{x_{i0} x_{i1} \ldots x_{i(n-1)}\}$ is

$$x_i = -x_{i0} + \sum_{b=1}^{n-1} x_{ib} \times 2^{-b}. \tag{C.2}$$

Substituting Equation C.2 into Equation C.1 yields

$$\begin{aligned} S = &- (x_{00} \times k_0 + x_{10} \times k_1 + \ldots + x_{(N-1)0} \times k_{N-1}) \times 2^0 \\ &+ (x_{01} \times k_0 + x_{11} \times k_1 + \ldots + x_{(N-1)1} \times k_{N-1}) \times 2^{-1} \\ &+ (x_{02} \times k_0 + x_{12} \times k_1 + \ldots + x_{(N-1)2} \times k_{N-1}) \times 2^{-2} \\ &\quad \vdots \\ &+ (x_{0(n-1)} \times k_0 + x_{1(n-1)} \times k_1 + \ldots + x_{(N-1)(n-1)} \times k_{N-1}) \times 2^{-(n-1)} \end{aligned} \tag{C.3}$$

The organization of the input variables are in a bit-serial, LSB first format. Since $x_{ij} \in \{0,1\}$ $(i = 0,1,\ldots,N-1,\ j = 0,1,\ldots,n-1)$, each term within the brackets of Equation C.3 is the sum of weighting factors $k_0, k_1, \ldots, k_{N-1}$. On every clock cycle, one of the bracketed terms of S can thus be computed by applying $x_0, x_1, \ldots, x_{N-1}$ as the address inputs of a 2^N-entry LUT. The contents of the LUT are pre-computed from the constants k_i $(i = 0,1,\ldots,N-1)$, as shown in Table C.1. The output of the LUT is multiplied by a power of two (a shift operation) and then accumulated. After n cycles, the accumulator contains the value of S.

DA is particularly suitable and lead to a very high area efficiency for FPGA architectures bacause:

- The DA LUTs can be efficiently implemented using the embedded memory (distributed RAM for the Xilinx FPGAs) on an FPGA (particularly true for the SOP of four or five terms, in which their implementations can use ROM16X1 and ROM32X1 primitives respectively).
- A DA operator uses minimal area (the use of LUTs saves multipliers).
- Bit-level parallelism is elaborated.

Input bits ($b_{N-1} \ldots b_2 b_1 b_0$)	LUT address	LUT content
$0 \ldots 000$	0	0
$0 \ldots 001$	1	k_0
$0 \ldots 010$	2	k_1
$0 \ldots 011$	3	$k_0 + k_1$
$0 \ldots 100$	4	k_2
$0 \ldots 101$	5	$k_0 + k_2$
$0 \ldots 110$	6	$k_2 + k_1$
$0 \ldots 111$	7	$k_0 + k_1 + k_2$
$\vdots$	$\vdots$	$\vdots$
$1 \ldots 111$	2^{N-1}	$k_0 + k_1 + \ldots + k_{N-1}$

Table C.1: Contents of a DA LUT. For each address, the terms k_i for which $b_i = 1$ are summed to yield a partial sum.

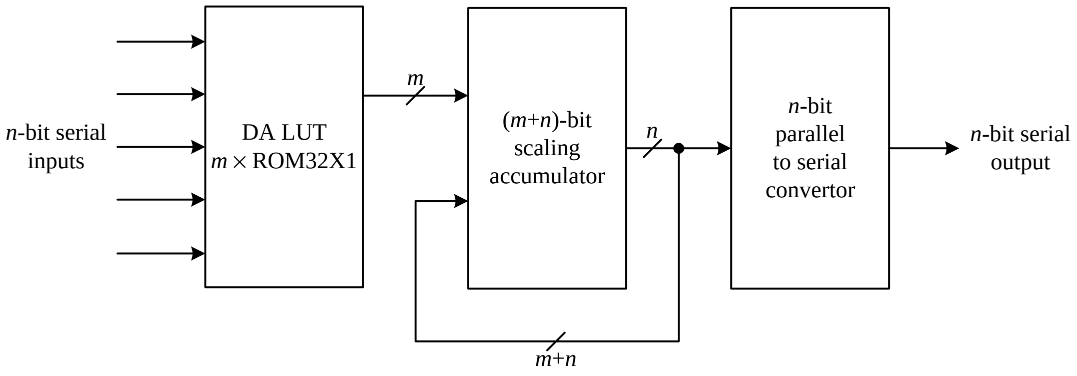

n-bit serial inputs
DA LUT
m × ROM32X1
m
(m+n)-bit scaling accumulator
n
n-bit parallel to serial convertor
n-bit serial output
m+n

Figure C.1: Implementation of the SOP of five terms using DA.

As an example, Figure C.1 shows the implementation of the SOP of five terms using DA. Let the inputs and outputs be n bits and the width of the DA LUT be m bits (the LUT content in Table C.1 are quantized to m bits). The scaling accumulator sums over the m-bit partial products for n cycles, hence its bit width is $m+n$ bits. At the n-th cycle after the inputs, the most significant n bits of the scaling accumulator are latched to a parallel to serial converter. As a result, this operator has a latency of n cycles and its critical path is its $(m+n)$-bit scaling accumulator.

Bibliography

[Alf96] P. Alfke. *Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators.* Xilinx, Inc., July 1996. Application Note XAPP052, Version 1.1.

[Ann99a] Annapolis Micro Systems, Inc. *Wildcard Reference Manual*, 1999. Revision 1.1.

[Ann99b] Annapolis Micro Systems, Inc. *Wildforce Reference Manual*, 1999. Revision 3.4.

[Ann00] Annapolis Micro Systems, Inc. *Wildstar Reference Manual*, 2000. Revision 3.3.

[ANR74] N. Ahmed, T. Natarajan, and K. R. Rao. Discrete cosine transform. *IEEE Transactions on Computers*, 23:90–93, 1974.

[ANS85] ANSI/IEEE, New York. *IEEE Standard for Binary Floating-Point Arithmetic*, 1985. Std 754-1985.

[AS65] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* Dover Publications, Inc., 1965.

[Asc99a] Ascom. *IDEA C Source Code*, 1999. Version 2.1 (`http://www.ascom.ch/infosec/downloads/ideaplus.zip`).

[Asc99b] Ascom. *IDEACrypt Coprocessor Data Sheet*, 1999. (`http://www.ascom.ch/infosec/downloads/IDEACrypt_Coprocessor.pdf`).

[Asc99c] Ascom. *IDEACrypt Kernel Data Sheet*, 1999. (`http://www.ascom.ch/infosec/downloads/IDEACrypt_Kernel.pdf`).

[ASS99] M. Abramovic, J. T. Sousa, and D. Saab. A massively-parallel easily-scalable satisfiability solver using reconfigurable hardware. In *Proceedings of the 36th Design Automation Conference*, pages 684–690, 1999.

[ASU85] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1985.

[BC97] N. W. Bergmann and Y. Y. Chung. Efficient implementation of DCT-based video compression on custom computers. *Conference Record of the 31st Asilomar Conference on Signals, Systems and Computers*, 2:1532–1536, 1997.

[BCF$^+$91] H. Bonnenberg, A. Curiger, N. Felber, H. Kaeslin, and X. Lai. VLSI implementation of a new block cipher. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computer and Processors*, pages 501–513, 1991.

[BDH$^+$97] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. de Wit. A dynamic reconfiguration run-time system. In *Proceedings of the IEEE*

Symposium on Field-Programmable Custom Computing Machines, pages 66–75, 1997.

[BNS$^+$98] M. Brucke, W. Nebel, A. Schwarz, B. Mertsching, M. Hansen, and B. Kollmeier. Digital VLSI-implementation of a psychoacoustically and physiologically motivated speech preprocessor. In *Proceedings of the NATO Advanced Study Institute on Computational Hearing*, pages 157–162, 1998.

[Bor97] J. Borst. Differential-linear cryptanalysis of IDEA. ESAT–COSIC Technical Report 96–2, Department of Electrical Engineering, Katholieke Universiteit Leuven, February 1997.

[BW96] J. C. Bor and C. Y. Wu. Analog electronic cochlea design using multiplexing switched-capacitor circuits. *IEEE Transactions on Neural Networks*, 7(1):155–166, 1996.

[CBK91] A. V. Curiger, H. Bonnenberg, and H. Kaeslin. Regular VLSI architectures for multiplication modulo $2^n + 1$. *IEEE Journal of Solid-State Circuits*, 26(7):990–994, July 1991.

[CBZ$^+$93] A. Curiger, H. Bonnenberg, R. Zimmerman, N. Felber, H. Kaeslin, and W. Fichtner. VINCI: VLSI implementation of the new secret-key block cipher IDEA. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 15.5.1–15.5.4, 1993.

[CC99] A. G. M. Cilio and H. Corporaal. Floating point to fixed point conversion of C code. In *European Joint Conference on Theory and Practice of Software*, pages 229–243, March 1999.

[CCL99] G. A. Constantinides, P. Y. K. Cheung, and W. Luk. Truncation noise in fixed-point SFGs. *Electronics Letters*, 35(23):2012–2014, November 1999.

[CCL00] G. A. Constantinides, P. Y. K. Cheung, and W. Luk. Multiple precision for resource minimization. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 307–308, April 2000.

[CCL01] G. A. Constantinides, P. Y. K. Cheung, and W. Luk. The multiple wordlength paradigm. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, May 2001.

[CJ90] C. Chakrabarti and J. JáJá. Systolic architectures for the computation of the discrete hartley and the discrete cosine transforms based on prime factor decomposition. *IEEE Transactions on Computers*, 39(11):1359–1368, November 1990.

[CRS$^+$99] R. Cmar, L. Rijnders, P. Schaumont, S. Vernalde, and I. Bolsens. A methodology and design environment for DSP ASIC fixed point refinement. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 271–276, 1999.

[CS92] Y. H. Chan and W. C. Siu. On the realization of discrete cosine transform using the distributed arithmetic. *IEEE Transactions on Circuits and Systems*, 39:705–712, September 1992.

[CTLL01] O. Y. H. Cheung, K. H. Tsoi, P. H. W. Leong, and M. P. Leong. Tradeoffs in parallel and serial implementations of the international data encryption algorithm IDEA. In *Lecture Notes in Computer*

Science – Workshop on Cryptographic Hardware and Embedded Systems, May 2001. accepted for publication.

[CW91] L. W. Chang and M. C. Wu. A unified systolic array for discrete cosine and sine transform. *IEEE Transactions on Signal Processing*, 39:192–194, January 1991.

[DeH00] André DeHon. The density advantage of configurable computing. *Computer*, 33(4):41–49, April 2000.

[Ele99] Electronic Frontier Foundation. DES challenge III broken in record 22 hours, January 1999. (`http://www.eff.org/pub/Privacy/Crypto_misc/DESCracker/HTML/19990119_deschallenge3.html`).

[FCF93] G. Fettweis, J. Chiu, and B. Fraenkel. A low-complexity bit-serial DCT/IDCT architecture. In *Technical Program, Conference Record, IEEE International Conference on Communications*, volume 1, pages 217–221, May 1993.

[FH95] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Addison Wesley, 1995.

[FW92] E. Feig and S. Winograd. Fast algorithms for the discrete cosine transform. *IEEE Transactions on Signal Processing*, 40:2174–2193, September 1992.

[Gal91] D. Le Gall. MPEG: A video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, April 1991.

[GL95] S. Guo and W. Luk. Compiling ruby into FPGAs. In W. Moore and W. Luk, editors, *Lecture Notes in Computer Science 975 – Field Programmable Logic and Applications*, pages 188–197. Springer-Verlag, 1995.

[GLS99] S. Guccione, D. Levi, and P. Sundararajan. JBits: Java based interface for reconfigurable computing. In *Second Annual Military and Aerospace Applications of Programmable Devices and Technologies*, September 1999.

[Goo99] R. Gooch. *MTRR (Memory Type Range Register) Control*, June 1999. Linux kernel Documentation, `Documentation/mtrr.txt`.

[Gos95] G. R. Goslin. *A Guide to Using Field Programmable Gate Arrays (FPGAs) for Application-Specific Digital Signal Processing Performance.* Xilinx, Inc., 1995. Application Note.

[GSB⁺00] S. C. Goldstein, H. Schmit, M. Budiu, M. Moe, and R. R. Taylor. PipeRench: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, April 2000.

[Har76] R. M. Haralick. A strong efficient way to implement the discrete cosine transform. *IEEE Transactions on Computers*, 25:333–341, 1976.

[HB01] B. V. Herzen and J. Brunetti. *Multi-Channel 622 MB/s LVDS Data Transfer for Virtex-E Devices.* Xilinx, Inc., January 2001. Application Note XAPP233, Version 1.2.

[HH95] J. D. Hadley and B. L. Hutchings. Design methodologies for partially reconfigured systems. In *Proceedings of the IEEE Sympo-*

sium on Field-Programmable Custom Computing Machines, pages 78–84, 1995.

[HL94] M. Hellman and S. Langford. Differential-linear cryptanalysis. In *Advances in Cryptology, Proceedings of Eurocrypt 1994*, pages 26–36, 1994.

[HM98] J. Hunter and J. McCanny. Discrete cosine transform generator for VLSI synthesis. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 5, pages 2997–3000, 1998.

[Hoa93] D. T. Hoang. Searching genetic databases on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, 1993.

[HP95] R. Hartley and K. K. Parhi. *Digit-Serial Computation.* Kluwer Academic Publishers, 1995.

[HRR93] S. Hadinger and P. Raynaud-Richard. Résolution numérique des équations de Laplace et de la chaleur. Rapport d'option, Ecole Polytechnique, 91128 Palaisau Cedea, France, 1993.

[HW95a] Y. H. Hu and Z. Wu. An efficient CORDIC array structure for the implementation of discrete cosine transform. *IEEE Transactions on Signal Processing*, 43:331–336, January 1995.

[HW95b] B. L. Hutchings and M. J. Wirthlin. Implementation approaches for reconfigurable logic applications. In W. Moore and W. Luk, editors, *Lecture Notes in Computer Science 975 – Field Programmable Logic and Applications*, pages 419–428. Springer-Verlag, 1995.

[IBM98] IBM and Reliance Computer Corporation. *PC133 SDRAM Registered DIMM*, August 1998. Revision 1.1.

[Int98a] Intel Corporation. *Accelerated Graphics Port Interface Specficiation*, May 1998. Revision 2.0.

[Int98b] Intel Corporation. *Intel 440BX AGPset: 82443BX Host Bridge/Controller Datasheet*, April 1998.

[Int98c] Intel Corporation. *Intel 440LX AGPset: 82443LX PCI AGP Controller (PAC) Datasheet*, January 1998.

[Int98d] Intel Corporation. *PC SDRAM Registered DIMM Design Support Document*, 1998. Revision 1.2.

[Int00a] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, 2000. Volume 1: Basic Architecture.

[Int00b] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, 2000. Volume 3: System Programming Guide.

[Int00c] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, 2000. Volume 2: Instruction Set Reference Manual.

[Int01a] Intel Corporation. *Intel 815 Chipset Family: 82815EP and 82815P Memory Controller Hub (MCH) Datasheet*, March 2001.

[Int01b] Intel Corporation. *Intel Pentium III processors with 512 KB L2 cache at 1.13 GHz Datasheet*, June 2001.

[ITU95] ITU. *Video Coding for Low Bitrate Communication*, 1995. ITU-T Draft Recommendation H.263.

[JD99] D. Johnson and M. Dofossez. Programming a Xilinx FPGA in "C". *Xcell*, 34(4):26–30, 1999.

[JED] JEDEC. *Configurations for Solid State Memories*. Standard 21-C, Release 7r8r9.

[Jin01] C. T. Jin. *Spectral Analysis and Resolving Spatial Ambiguities in Human Sound Localization*. PhD thesis, University of Sydney, 2001.

[KFM+96] T. Kuroda, T. Fujita, S. Mita, T. Nagamatsu, S. Yoshioka, K. Suzuki, F. Sano, M. Norishima, M. Murota, M. Kako, M. Kinugawa, M. Kakumu, and T. Sakurai. A 0.9V, 150-MHz, 10-mW, 4 mm^2, 2-D discrete cosine transform core processor with variable threshold-voltage (VT) scheme. *IEEE Journal of Solid-State Circuits*, 31(11):1770–1779, November 1996.

[KKS98] S. Kim, K. I. Kum, and W. Sung. Fixed-point optimization utility for C and C++ based digital signal processing programs. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 45:1455–1464, November 1998.

[KKS99] K. Kum, J. Kang, and W. Sung. A floating-point to integer C converter with shift reduction for fixed-point digital signal processors. In *Processings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 2163–2166, April 1999.

[KKS00] K. Kum, J. Kang, and W. Sung. AUTOSCALER for C: An optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Transactions on Circuits and*

Systems II: Analog and Digital Signal Processing, 47(9):840–848, September 2000.

[Kna98] S. K. Knapp. *Using Programmable Logic to Accelerate DSP Functions.* Xilinx, Inc., 1998.

[Knu95] L. R. Knudsen. Truncated and higher order differentials. In *Proceedings of the Second International Workshop on Fast Software Encryption*, pages 196–211, 1995.

[Kor95] I. Koren. *Computer Arithmetic Algorithms.* Prentice-Hall, 1995.

[KRDP98] H. Kropp, C. Reuter, T. T. Do, and P. Pirsch. A generator for pipelined multipliers on FPGAs. In *Proceedings of the 9th International Conference on Signal Processing Applications and Technology*, pages 669–673, September 1998.

[KS94] S. Kim and W. Sung. A floating-point to fixed-point assembly program translator for the TMS 320C25. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 41(11):730–739, November 1994.

[LC99] P. H. W. Leong and C. K. Chung. FPGA-based runtime configurable clause evaluator for SAT problems. *Electronics Letters*, 35(19):1618–1619, September 1999.

[Lee84] B. G. Lee. A new algorithm to compute the discrete cosine transform. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 32:1243–1247, 1984.

[Lee91] W. Lee. A new algorithm to compute the DCT and its inverse. *IEEE Transactions on Signal Processing*, 39:1305–1313, June 1991.

[LFP94] W. Luk, D. Ferguson, and I. Page. Structured hardware compilation of parallel programs. In W. Moore and W. Luk, editors, *More FPGAs*. Abingdon EE&CS Books,, 1994.

[Lio91] M. Liou. Overview of the $p \times 64$ kbits/s video coding standard. *Communications of the ACM*, 34(4):59–63, April 1991.

[Lip98] H. Lipmaa. IDEA: A cipher for multimedia architectures? In S. Tavares and H. Meijer, editors, *Lecture Notes in Computer Science 1556 – Selected Areas in Cryptography*, pages 253–268. Springer-Verlag, August 1998.

[LLCL98] J. G. Liu, H. F. Li, F. H. Y. Chan, and F. K. Lam. Fast discrete cosine transform via computation of moments. *Journal of VLSI Signal Processing*, 19:257–268, 1998.

[LM88] R. F. Lyon and C. Mead. An analog electronic cochlea. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 36(7):1119–1134, July 1988.

[LM89a] J. P. Lazzaro and C. A. Mead. Silicon models of auditory localization. *Neural Computation*, 1:47–57, 1989.

[LM89b] J. P. Lazzaro and C. A. Mead. Silicon models of pitch perception. In *Proceedings National Academy of Sciences*, volume 86, pages 9597–9601, 1989.

[LM89c] R. F. Lyon and C. Mead. *Electronic Cochlea*, chapter 16 in Analog VLSI and Neural Systems. Addison Wesley, 1989.

[LM90] X. Lai and J. Massay. A proposal for a new block encryption standard. In *Advances in Cryptology, Proceedings of Eurocrypt 1990*, pages 389–404, 1990.

[LM98] W. Luk and S. McKeever. Pebble: A language for parameterised and reconfigurable hardware design. In R. W. Hartenstein and A. Keevallik, editors, *Lecture Notes in Computer Science 1482 – Field Programmable Logic and Applications*, pages 59–68. Springer-Verlag, 1998.

[LMM91] X. Lai, J. Massay, and S. Murphy. Markov ciphers and differential cryptanalysis. In *Advances in Cryptology, Proceedings of Eurocrypt 1991*, pages 17–38, 1991.

[LO98] K. Lenwchasatit and A. Ortega. DCT computation based on variable complexity fast approximations. In *Proceedings of the International Conference on Image Processing*, volume 3, pages 95–99, October 1998.

[LSC96] W. Luk, N. Shirazi, and P. Y. K. Cheung. Modelling and optimising run-time reconfigurable systems. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 167–176, 1996.

[LSC97] W. Luk, N. Shirazi, and P. Y. K. Cheung. Compilation tools for run-time reconfigurable designs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 56–65, April 1997.

[LSW+01] P. H. W. Leong, C. W. Sham, W. C. Wong, H. Y. Wong, W. S. Yuen, and M. P. Leong. A bitstream reconfigurable FPGA implementation of the WSAT algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(1):197–200, February 2001.

[LTJM97] S. C. Lim, A. R. Temple, S. Jones, and R. Meddis. VHDL-based design of biologically inspired pitch detection system. In *Proceedings of the IEEE International Conference on Neural Networks*, volume 2, pages 922–927, 1997.

[LW94] W. Luk and T. Wu. Towards a declarative framework for hardware-software codesign. In *Proceedings of the Third International Workshop on Hardware/Software Codesign*, pages 181–188, 1994.

[LWK94] J. P. Lazzaro, J. Wawrznek, and A. Kramer. Systems technologies for silicon auditory models. *IEEE Micro*, 14(3):7–15, 1994.

[LWL97] J. P. Lazzaro, J. Wawrznek, and R. P. Lippmann. A micropower analog circuit implementation of hidden markov model state decoding. *Journal of Solid State Circuits*, 32(8):1200–1209, 1997.

[Lyo76] R. F. Lyon. Two's complement pipeline multipliers. *IEEE Transactions on Communications*, 12:418–425, April 1976.

[Lyo91] R. F. Lyon. Analog implmentations of auditory models. In *Proceedings of the DARPA Workshop on Speech and Natural Language*. Morgan Kaufmann, 1991.

[MAL91] C. A. Mead, X. Arreguit, and J. Lazzaro. Analog VLSI model of binaural hearing. *IEEE Transactions on Neural Networks*, 2:230–236, 1991.

[Mar00] M. Mares. *PCI Utilities*, May 2000. Version 2.1.8, `http://atrey.karlin.mff.cuni.cz/~mj/pciutils.html`.

[MB95] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. In *Proceedings of SIGGRAPH'95*, pages 39–46, August 1995.

[MB97] W. Mark and G. Bishop. Memory access patterns of occlusion-compatible 3D image warping. In *Proceedings of the 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 35–44, August 1997.

[Mea89] C. Mead. *Analog VLSI and Neural Systems.* Addison Wesley, 1989.

[MHD$^+$97] J. V. McCanny, Y. Hu, T. J. Ding, D. Trainor, and D. Ridge. Rapid design of DSP ASIC cores using hierarchical VHDL libraries. In *Conference Record of the Conference 30th Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 1344–1348, 1997.

[Mic99] Micron Semiconductor Inc. *Micron MT48LC16M8A2 Synchronous DRAM Datasheet*, November 1999.

[MMB97] W. Mark, L. McMillan, and G. Bishop. Post-rendering 3D warping. In *Symposium on Interactive 3D Graphics*, pages 7–16, 1997.

[MMF98] O. Mencer, M. Morf, and M. J. Flynn. Hardware software tri-design of encryption for mobile communication units. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 5, pages 3045–3048, May 1998.

[Mol93] L. Moll. Implantation d'un algorithme de stéréovision par corrélation sur mémoire active programmable PeRLe-1. Rapport de stage, Ecole des Mines de Paris, Centre de Mathématiques Appliquées, 06904 Sophia-Antipolis, France, 1993.

[MRHH97] J. McCanny, D. Ridge, Y. Hu, and J. Hunter. Hierarchical VHDL libraries for DSP ASIC design. In *Processings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 1, pages 675–678, April 1997.

[MYSA00] Z. Mohd-Yusof, I. Suleiman, and Z. Aspar. Implementation of two dimensional forward DCT and inverse DCT using FPGA. In *Proceedings of TENCON 2000*, volume 3, pages 242–245, 2000.

[MZ91] C. Meier and R. Zimmerman. A multiplier modulo $(2^n + 1)$. Diploma thesis, Institut für Integrierte Systeme, ETH, Zürich, Switzerland, February 1991.

[Nal] Nallatech Limited. *Ballyinx PCI64 PCI Prototyping Board*. `http://www.nallatech.com/products/dime_professional/ballyinx/ballyinx_issue_1.pdf`.

[NJLDGG00] L. Naviner, C. Laurent J. L. Danger, and A. Garcia-Garcia. Efficient implementation for high accuracy DCT processor based on FPGA. In *Proceedings of the 42nd Midwest Symposium on Circuits and Systems*, volume 1, pages 508–511, 2000.

[NM65] J. Nelder and R. Mead. A simplex method for function minimization. *Computer*, 7:308–313, 1965.

[Pag96] I. Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996.

[Pat00] C. Patterson. High performance DES encryption in Virtex FPGAs using JBits. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 113–121, April 2000.

[PCI] PCI SIG. *PCI Local Bus Specification.* Revision 2.2.

[PD00] I. Page and R. Dettmer. Software to silicon. *IEE Review*, 46(5):15–19, September 2000.

[Pic88] J. O. Pickles. *An Introduction to the Physiology of Hearing.* Academic Press, 1988.

[PL91] I. Page and W. Luk. Compiling occam into field-programmable gate arrays. In W. Moore and W. Luk, editors, *FPGAs*, pages 271–283. Abingdon EE&CS Books,, September 1991.

[PLL93] I. Page, W. Luk, and H. Lau. Hardware compilation for FPGAs: Imperative and declarative approaches for a robotics interface. In *IEE Colloquium on Field Programmable Gate Arrays - Technology and Applications*, pages 9/1–9/4, 1993.

[PM93] W. B. Pennebaker and J. L. Mitchell. *JPEG – Still Image Data Compression Standard.* New York: Van Nostrant Reinhold, 1993.

[PSB99] W. Pan, A. Shams, and M. A. Bayoumi. NEDA: A new distributed arithmetic architecture and its application to one dimensional discrete cosine transform. In *IEEE Workshop on Signal Processing Systems*, pages 159–168, 1999.

[PW90] K. K. Parhi and C. Wang. Digit-serial DSP architectures. In *Proceedings of the International Conference on Applications Specific Array Processors*, pages 341–351, September 1990.

[Ram01] Rambus, Inc. *Rambus 32 and 64 bit RIMM Module Technology Summary*, June 2001. Version 0.0.

[RAPL98] M. M. Rafferty, D. G. Aliaga, V. Popescu, and A. A. Lastra. Images for accelerating architectural walkthroughs. *IEEE Computer Graphics and Applications*, 18(6):38–48, November/December 1998.

[RB98] P. Rademacher and G. Bishop. Multiple-center-of-projection images. In *Proceedings of SIGGRAPH'98*, pages 199–206, July 1998.

[RCHP91] J. M. Rabaey, C. Chu, P. Hoang, and M. Potkonjak. Fast prototyping of datapath-intensive architectures. *IEEE Design and Test of Computers*, pages 40–51, June 1991.

[RFLC90] J. Rose, R. J. Francis, D. Lewis, and P. Chow. Architecture of field-programmable gate arrays: The effect of logic block functionality on area efficiency. *IEEE Journal of Solid-State Circuits*, 25(5):1217–1225, October 1990.

[RJR95] N. K. Ratha, A. K. Jain, and D. T. Rover. An FPGA-based point pattern matching processor with application to fingerprint matching. In *Proceedings of the Conference on Computer Architectures for Machine Perception*, pages 394–401, 1995.

[SAF98] S. L. C. Salomao, V. C. Alves, and E. M. C. Filho. HiPCrypto: A high-performance VLSI cryptographic chip. In *Proceedings of the Eleventh Annual IEEE ASIC Conference*, pages 7–11, 1998.

[Sch96] B. Schneier. *Applied Cryptography.* John Wiley & Sons, second edition, 1996.

[SCP98] H. Suzuki, Y. N. Chang, and K. K. Parhi. Performance tradeoffs in digit-serial DSP systems. In *Conference Record of the Conference 32nd Asilomar Conference on Signals, Systems and Computers*, November 1998.

[SGHS98] J. Shade, S. Gortler, L. W. He, and R. Szeliski. Layered depth images. In *Proceedings of SIGGRAPH'98*, pages 231–242, July 1998.

[SK94] W. Sung and K. I. Kum. Word-length determination and scaling software for a signal flow block diagram. In *Processings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 457–460, April 1994.

[SK95] W. Sung and K. I. Kum. Simulation-based word-length optimization method for fixed-point digital signal processing systems. *IEEE Transactions on Signal Processing*, pages 3087–3090, December 1995.

[Sku90] M. Skubiszewski. A hardware emulator for binary neural networks. In *Proceedings of the 1990 International Neural Network Conference*, volume 2, pages 555–5581, 1990.

[Sku92] M. Skubiszewski. An exact hardware implementation of the boltzmann machine. In *Proceedings of the 1992 International Conference on Application-Specific Array Processors*, 1992.

[SL92] C. D. Summerfield and R. F. Lyon. ASIC implementation of the Lyon cochlea model. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 673–676, 1992.

[Sla98] M. Slaney. *Auditory Toolbox: A MATLAB Toolbox for Auditory Modeling Work*. Interval Research Corporation, 1998. Technical Report #1998-010, Version 2.

[Sun91] W. Sung. An automatic scaling method for the programming of fixed-point digital signal processors. *IEEE Transactions on Circuits and Systems*, 1:37–40, 1991.

[SVR$^+$98] P. Schaumont, S. Varnalde, L. Rijnders, M. Engels, and I. Bolsens. A programming environment for the design of complex high speed ASICs. In *Proceedings of the 35th Design Automation Conference*, pages 315–320, 1998.

[THW97] D. W. Trainor, J. P. Heron, and R. F. Woods. Implementation of the 2D DCT using a Xilinx XC6264 FPGA. In *IEEE Workshop on Signal Processing Systems*, pages 541–550, 1997.

[Tra00a] J. Tran. *Synthesizable 1.6 GBytes/s DDR SDRAM Controller*. Xilinx, Inc., March 2000. Application Note XAPP200, Version 2.3.

[Tra00b] T. D. Tran. The BinDCT: Fast multiplierless approximation of DCT. *IEEE Signal Processing Letters*, 7:145–149, June 2000.

[UIT$^+$92] S. Uramoto, Y. Inoue, A. Takabatake, J. Takeda, Y. Yamshita, H. Terane, and M. Yoshimoto. A 100-MHz 2-D discrete cosine transform core processor. *IEEE Journal of Solid-State Circuits*, 27(4):492–499, April 1992.

[VBR$^+$96] J. E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. M. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(1):56–69, March 1996.

[vSFV97] A. van Schaik, E. Fragnière, and E. Vittoz. Improved silicon cochlea using compatible lateral bipolar transistors. In *Advances in Nerual Information Processing Systems 8*. MIT Press, 1997.

[vSM99] A. van Schaik and R. Meddis. Analog very large-scale integration (VLSI) implementation of a model of amplitude-modulation sensitivity in the auditory brainstem. *Journal of the Acoustical Society of America*, 105:811–821, February 1999.

[WBGM97] M. Willems, V. Bürsgens, T. Grötker, and H. Meyr. FRIDGE: An interactive code generation environment for HW/SW codesign. In *Processings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 287–290, April 1997.

[WBK$^+$97] M. Willems, V. Bürsgens, H. Keding, T. Grötker, and H. Meyr. System level fixed-point design based on an interpolative approach. In *Proceedings of the 34th Design Automation Conference*, pages 293–298, 1997.

[Wei97] M. Weinhardt. Compilation and pipeline synthesis for reconfigurable architectures. In *Reconfigurable Architectures Workshop*, 1997.

[Wir98] N. Wirth. Hardware compilation: Translating programs into circuits. *Computer*, 31(6):25–31, June 1998.

[WKLM92] L. Watts, D. A. Kerns, R. F. Lyon, and C. A. Mead. Improved implementation of the silicon cochlea. *IEEE Journal of Solid State Circuits*, 27(5):692–700, May 1992.

[WL99] M. Weinhardt and W. Luk. Pipeline vectorization for reconfigurable systems. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 52–62, April 1999.

[WMSL95] S. Wolter, H. Matz, A. Schubert, and R. Laur. On the VLSI implementation of the international data encryption algorithm IDEA. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, volume 1, pages 397–400, 1995.

[WYLL99] H. Y. Wong, W. S. Yuen, K. H. Lee, and P. H. W. Leong. A runtime reconfigurable implementation of the GSAT algorithm. In *Proceedings of the Field Programmable Logic and Applications Wordshop*, pages 526–531, 1999.

[XC00] T. Xanthopoulos and A. P. Chandrakasan. A low-power DCT core using adaptive bitwidth and arithmetic activity exploiting signal correlations and quantization. *IEEE Journal of Solid-State Circuits*, 35(5):740–750, May 2000.

[Xil96] Inc. Xilinx. *The Role of Distributed Arithmetic in FPGA-Based Signal Processing*, November 1996. (`http://www.xilinx.com/appnotes/theory1.pdf`).

[Xil97] Xilinx, Inc. *XC6200 Programmable Gate Arrays Datasheet*, 1997.

[Xil00a] Inc. Xilinx. *Modeling and Implementation of DSP FPGA Solutions*, 2000.

(`http://www.xilinx.com/products/logicore/dsp/matlab_final.pdf`).

[Xil00b] Xilinx, Inc. *Development System Reference Guide*, 2000. Version 3.1i.

[Xil00c] Xilinx, Inc. *The Programmable Logic Data Book*, 2000.

[Xil00d] Xilinx, Inc. *Using the Virtex Delay-Locked Loop*, September 2000. Application Note XAPP132, Version 2.3.

[Xil00e] Xilinx, Inc. *Virtex Configuration Architecture Advanced Users' Guide*, February 2000. Application Note XAPP151, Version 1.3.

[Xil00f] Xilinx, Inc. *Virtex FPGA Series Configuration and Readback*, February 2000. Application Note XAPP138, Version 2.0.

[Xil00g] Xilinx, Inc. *Xilinx Foundation Series 3.1i Libraries Guide*, 2000.

[Xil01a] Xilinx. *Virtex-II Platform FPGA User Guide*, 2001. Version 1.1.

[Xil01b] Xilinx, Inc. *LogiCORE PCI 64 Virtex/Spartan-II Interface*, April 2001. LogiCORE Data Sheet, Version 3.0.

[ZCB+94] R. Zimmermann, A. Curiger, H. Bonnenberg, H. Kaeslin, N. Felber, and W. Fichtner. A 177 Mb/sec VLSI implementation of the international data encryption algorithm. *IEEE Journal of Solid-State Circuits*, 29(3):303–307, March 1994.

[ZL99] X. Zhu and B. Lin. Hardware compilation for FPGA-based configurable computing machines. In *Proceedings of the 36th Design Automation Conference*, pages 697–702, 1999.

Publications

Journals

- P. H. W. Leong, C. W. Sham, W. C. Wong, H. Y. Wong, W. S. Yuen, and M. P. Leong, “A bitstream reconfigurable FPGA implementation of the WSAT algorithm,” *IEEE Transactions on VLSI Systems*, 9(1):197–200, February 2001.
- M. P. Leong and P. H. W. Leong, “A variable-radix digit-serial design methodology and its applications to the discrete cosine transform,” *IEEE Transactions on VLSI Systems*, 2001, in review.

Conference Papers

- M. P. Leong, M. Y. Yeung, C. K. Yeung, C. W. Fu, P. A. Heng, and P. H. W. Leong, “Automatic floating to fixed point translation and its application to post-rendering 3D warping,” in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1999, pp. 240–248.
- M. P. Leong, O. Y. H. Cheung, K. H. Tsoi, and P. H. W. Leong, “A bit-serial implementation of the international data encryption algorithm (IDEA),” in

Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, April 2000, pp. 122–131.

- M. P. Leong, C. T. Jin, and P. H. W. Leong, "Parameterized module generator for an FPGA-based electronic cochlea," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, May 2001, to appear.
- P. H. W. Leong, M. P. Leong, O. Y. H. Cheung, T. Tung, C. M. Kwok, M. Y. Wong, and K. H. Lee, "Pilchard – a reconfigurable computing platform with memory slot interface," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, May 2001, to appear.
- O. Y. H. Cheung, K. H. Tsoi, P. H. W. Leong, and M. P. Leong, "Tradeoffs in parallel and serial implementations of the international data encryption algorithm IDEA," in *Lecture Notes in Computer Science – Workshop on Cryptographic Hardware and Embedded Systems*, May 2001, to appear.