Adaptable Architectures for Acceleration of Protocol Processing using FPGAs

Akagic Amila

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

School of Science for Open and Environmental Systems
Graduate School of Science and Technology
Keio University

September 2013

Abstract

The emergence of multi-Gigabit Ethernet and ever-increasing volume of network traffic on the Internet has begun outpacing server capacity to manage incoming data. In recent years, the network traffic exhibits constant increase, due to the confluence of many market trends. Today, data centers are considering employment of new technologies, such as 40- and 100-Gb Ethernet, however their adoption rate is still rather small. The major concern is that the potential for such high bandwidths would not be exploited, due to the communication overhead that consumes high levels of processor's processing power. One major source of processing overhead is the TCP/IP stack. This problem has been addressed in various methods. One method is to dedicate one or more cores for TCP/IP processing exclusively. However, with the new paradigm shift to multicore processors, it is hard to guarantee the high throughput for inherently sequential processes, such as cyclic redundancy checks. Other methods include protocol processing *offloading* onto a specialized hardware and using special large packets known as jumbo frames. This has been specially beneficial in storage applications that transfer large blocks of data.

The future networks also seem to take a new direction toward so called *programmable networks*, which will allow greater agility, programmability and flexibility. In this dissertation, we take another step in this direction by utilizing programmable hardware to achieve the same goals. At first, we target one of the challenging aspects of iSCSI processing, which is processing of digests or Cyclic Redundancy Checks (CRC). CRCs are often characterized as computationally intensive, and thus often substituted with less efficient error detection schemes. We propose a *non-adaptable* and *fully-adaptable CRC accelerators* based on a table-based algorithm, which has been rarely used in hardware implementations. The *non-adaptable* CRC accelerator is suitable for acceleration of a specific application, and has no ability to adapt to a new standard or an application. The *fully-adaptable CRC accelerator* has ability to process arbitrary number of input data and generates CRC for any known CRC standard during run-time. We modify table generation algorithm in order to decrease its space complexity.

We also address the problem of efficiently implementing IP-based iSCSI Offload Engine which operates on the top of the TCP/IP protocol stack. Based on the analysis of iSCSI traffic, CPU utilization and throughput of software-based Open-iSCSI, we propose a new architecture which offloads data transfer and related non-data functions to an FPGA based adapter. The resulting architecture relieves the host CPU from computational burden imposed by software

implementations. The iSCSI Offload Engine allows very low utilization on the host CPU of approximately 3%.

Our work is a step toward the goal of using hardware accelerators to enable higher levels of agility, programmability and flexibility in future networks.

Acknowledgments

There are a number of people without whom my doctoral study might not have been accomplished, and to whom I am greatly indebted.

Most of all, I would like to express my deepest gratitude to my supervisor, Professor Hideharu Amano. His constant, patient and generous guidance, suggestions and hopeful encouragements have been my endless source to pursue my work for three years I've spend in his laboratory. He has always given me encouraging words whenever I had a difficulty in my research life, and reviewed every paper that I have written with constructive comments. I am also grateful to all the members of "hlab" and ASAP group. Especially, Kazuei Hironaka, Keimei Miyajima, Takayuki Akamine and other h_superusers with technical support and significant knowledge.

For this dissertation, I would also like to thank my committee members: Naoaki Yamanaka, Iwao Sasase, and Fumio Teraoka for their time, interest, and helpful comments.

I wish to express my reverence and heartfelt gratitude to Prof. Walid Najjar from University of California, Riverside, for introducing me to this topic, which afterwards became my main focus of research. He has shared his great experience to enlarge my knowledge on computer architecture fields, and also provided me with the crucial evaluation environment for this study.

I'm grateful to Professor Novica Nosovic and Professor Adnan Salihbegovic, my former professors who had addressed evocative words on every milestone in my university life at University of Sarajevo and here at Keio. I would also like to express my reverence to many other colleagues and staff members from Faculty of Electrical Engineering, University of Sarajevo for being supportive during my studies.

My deepest reverence and unfathomable sense of gratitude goes to Zejnil Velic, Zikret Dzananovic, Redzep Husejnagic, Omer Mustafic and many other professors, who lost their lives in the Srebrenica Genocide. In the time of madness, you encouraged curiosity and a passion for learning, that has changed my life forever. I will never forget Zejnil Velic's colorful drawings from Biology class, who helped us memorize every lesson with ease. He thought me one of the most important concept in life: "*If you can't explain it simply, you don't understand it well enough*".

I gratefully acknowledge the funding sources that made my Ph.D. studies possible. First and foremost, I wish to express my sincere gratitude to Ministry of Education, Culture, Sports, Science, and Technology of Japan for awarding me Monbukagakusho Scholarship. This extraordinary opportunity enriched my life beyond all my expectations. I experienced a sense of

personal achievement that felt very different from professional achievements. I appreciate the opportunity to experience the wonders of Japanese cultures and to understand its history a little bit better. Second, I was funded by the Global COE (Center of Excellence) for the first 2 years and I was honored to be GCOE Research Assistant. My gratitude extends to GCOE's project leaders, Professor Naoaki Yamanaka and Professor Masayasu Yamaguchi, for their exceptional leadership and support during my studies. Third, my work was supported with grants from Keio Leading-edge Laboratory of Science and Technology. I'm thankful to all administrative staff from all three sources for their professionalism and helpfulness.

My time in Japan was made enjoyable in large part due to many friends and groups that became a part of my life. I'm especially grateful to KIND, MIFA and Keio Welcome Net groups for their efforts to introduce Japanese culture to foreign students. My unbounded thanks to all my friends, especially Alma Halilovic Okajima and Taro Okajima, who helped and supported my life in Japan countless times.

Lastly, I would like to thank my family for all their love and encouragement. For my parents Djulzida and Mirsad, who raised me with a love of science and supported me in all my pursuits, and my brother Adnan for his constant support. I'm especially grateful to my grandparents Behija Temim and Munib Talovic, whose values are deeply embedded in my character, and who have been my silent advisers and supporters whenever I felt lost.

And most of all for my loving, supportive, encouraging, patient and the best husband Emir Buza whose faithful support during all stages of my Ph.D. is so appreciated. Thank you.

Amila Akagic
Yokohama, Japan
September 2013

Contents

Abstract **i**

Acknowledgments **iii**

1 Introduction **1**

1.1 Background 1
1.2 Objective 3
1.3 Contribution 4
1.4 Dissertation Organization 5

2 Cyclic Redundancy Checks and iSCSI initiator **8**

2.1 Error Control Coding 9
2.1.1 Types of Errors 10
2.1.2 Types of Error protecting codes 11
2.1.3 Principles of Block Coding 11
2.1.4 Error Detection Schemes 15
2.2 Cyclic Redundancy Checks (CRC) 16
2.2.1 Algorithms for CRC Computation 16
2.2.2 CRC Standard 22
2.2.3 Related Work 23
2.3 Internet Small Computer System Interface (iSCSI) initiator 25
2.3.1 The iSCSI Protocol 25
2.3.2 Processing of iSCSI Read and Write commands 25
2.3.3 Implementation Approaches 28
2.3.4 Performance Analysis of Open-iSCSI 29
2.3.5 Related Work 30

3 High Performance Reconfigurable Computing **31**

3.1 Accelerator Based Computing with FPGAs 33
3.2 Classes of data processing architectures 34
3.2.1 Programmable Architectures 35

3.2.2 Reconfigurable Architectures 37
3.2.3 Application-Specific Architectures 40
3.3 Overview of Field Programmable Gate Arrays 41
3.3.1 FPGA Programming Technologies 41
3.3.2 FPGA Architecture 43
3.3.3 Configurable Logic Block 44
3.4 Hardware Design Flow 47

4 High-Speed Fully-Adaptable CRC Accelerators 49
4.1 Design of a CRC Accelerator 50
4.1.1 CRC Generation Module 52
4.1.2 Tables Generation Module 53
4.1.3 Effects of architecture's scalability 57
4.1.4 The IP* Core Interface 59
4.2 FPGA Implementation 59
4.3 Evaluation 60
4.3.1 Non-adaptable CRC accelerator core 60
4.3.2 Fully-adaptable CRC accelerator 61
4.3.3 Comparison to Related Work 64
4.4 Summary 65

5 Design and implementation of IP-based iSCSI Offload Engine on an FPGA 67
5.1 Design and implementation of iSCSI Offload Engine 69
5.1.1 The Reception Module (Rx) 73
5.1.2 The Transmission Module (Tx) 75
5.1.3 The CRC Generation Unit 76
5.1.4 The Control Module 77
5.1.5 Modification of the Open-iSCSI Initiator 79
5.2 Implementation Results and Analysis 82
5.2.1 iSCSI Offload Engine Board 82
5.2.2 Elapsed time of main operations 82
5.2.3 CPU Utilization and Throughput 83
5.2.4 Reconfiguration time 85
5.2.5 Resource Utilization 85
5.2.6 Comparison to Related Work 86
5.3 Summary 86

6 Conclusions 88
6.1 Concluding Remarks 88

Bibliography 93

Publications 99

List of Tables

2.1 A linear block code with k=4 and n = 7. 12
2.2 A list of parameters defined by a CRC standard. 23
2.3 A list of CRC standards with associated applications and protocols. 24

3.1 Acceleration benefits on Virtex FPGAs 34
3.2 A comparison between fine- and coarse-grained architectures. 39
3.3 A comparison summary of selected architecture domains. 39
3.4 Resource comparison between five Xilinx generations of FPGAs. 46

4.1 Resource utilization of R Modules in $(T_{R_i} + 1)$-stage pipelined architecture. 60
4.2 Resource utilization of *non-adaptable* Slicing–by–N_{32} and Slicing–by–N_{64}. 61
4.3 Resource utilization of *fully-adaptable* CRC accelerator. 63
4.4 Number of clock cycles and time required for re-generation of tables. 63
4.5 A summary of different CRC designs from Related Work and our implementations. 65

5.1 A minimum set of opcodes defined on an initiator and a target. 72
5.2 Elapsed time of main operations for a 1500-byte data packet. 83
5.3 Resource utilization of iSCSI Engine on Virtex-6 XC6VLX240T FPGA. 86

List of Figures

1.1 Dissertation organization 6
1.2 Positions and contributions of this dissertation. 7

2.1 Effects of noise on transmission lines. 10
2.2 A representation of n-bit codeword. 11
2.3 LSFR for polynomial $P(x) = x^4 + x^3 + x + 1$. 16
2.4 LSFR2 for polynomial $P(x) = x^4 + x^3 + x + 1$. 17
2.5 Layers of iSCSI packet. 26
2.6 Flow diagram for processing of a) Data-In PDU and b) Data-Out PDU. 27
2.7 Three major implementation choices for iSCSI. 28
2.8 The performance profile of processing Data-In PDUs. 29

3.1 Architecture domains as a function of efficiency/performance and flexibility. 35
3.2 An example of a) fine-grained and b) coarse-grained reconfigurable architecture. . 38
3.3 An implementation of 256-tap FIR filter in a) GPPs/ASIPs and b) FPGAs. 40
3.4 Two basic elements used in FPGA implementation technologies 42
3.5 FPGA Block Structure 43
3.6 Advancement in configuration of a CLB in latest Xilinx FPGAs. 45
3.7 An example of basic Xilinx Spartan-3 Configurable Logic Block. 46
3.8 Design flow of system hardware development. 47

4.1 Design overview of the non-adaptable and fully-adaptable CRC accelerators. . . . 51
4.2 The generic architecture of CRC Generator Module (CGM). 52
4.3 Pseudocode for generating contents of tables. 54
4.4 A schematic of various architectures of Table Generation Module. 55
4.5 Overlapped implementation of eight TGMs. 58
4.6 Throughputs of non-adaptable and fully-adaptable CRC accelerators. 62

5.1 Design overview of the proposed iSCSI Offload Engine. 70
5.2 An overview of two transfer directions during reading and writing processes. . . . 71
5.3 The structure of Reception and Transmission Modules in the iSCSI Offload Engine. 74

5.4 The architecture of CRC Generation Unit. 76
5.5 An exemplary exchange of information between the initiator and target. 78
5.6 The flow of information between modules in the iSCSI Offload Engine. 79
5.7 Unmodified and modified data-paths for creating a SCSI Command by tx_thread. 81
5.8 Comparison of throughput and CPU utilization for 1500 bytes MTU. 84

Chapter 1

Introduction

1.1 Background

The emergence of multi-Gigabit Ethernet and ever-increasing volume of network traffic on the Internet has begun outpacing server capacity to manage incoming data. In recent years, the network traffic exhibits constant increase, due to the confluence of many market trends. Consequently, there is a very high demand for faster transfer, processing, compilation, and storage of data. In the recent 2012 study [1], Cisco predicts that annual global IP traffic will reach 1.3 zettabytes by 2016. The level of traffic growth is driven by a number of factors, including (a) *an increasing number of devices*: 2.5 connections for each person on earth and an estimate of 50 billion devices by 2020 [2], (b) *more Internet users*: about 45% of the world's projected population, (c) *faster broadband speeds*: expected to increase nearly a fourfold, (d) *more video content*: 1.2M video minutes in every second and (e) *Wi-Fi growth*, which will account for the half of the world's Internet traffic by 2016. These factors are having a cumulative effect of putting new demands on IT.

The Internet is evolving from the *Internet of Things (IoT)* [3], with 10 billions of devices connected to the Internet every day, to the *Internet of Everything (IoE)*, also referred to as *networks of networks* [4], with more than 50 billion of connected *things*. The principal difference between these concepts is that IoT focuses on the *volume* of connected things, whereas IoE focuses on the actual *connections*. In the future, billions of new devices and smart sensors will interact with one another without any human interaction. Hence, they will generate an enormous amount of data at an unprecedented scale and resolution. It is obvious that the number of users is ever-increasing factor that has an effect on a network. This effect is defined by *Metcalfe's law*.[1] Those organizations who put the highest effort to harness capacities offered by the new *networks of networks* will have the competitive edge.

Thus, there will be many unprecedented opportunities as well as challenges to face in the

[1] The Metcalfe's law states that the value of a network increases proportionally to the square of the number of users.

next decade. One of the goals of both IoT and IoE is to bring applications available anytime and anywhere, which will allow users to move with greater agility and speed. In order to support this idea, data centers have become the connecting technology between users and online applications. The entire infrastructure of data centers is undergoing a major transformation. The major initiators of this change are migration of applications to private and public clouds, networks getting faster and adoption of virtualization. In terms of network connections, data centers today tend to employ 10-gigabit Ethernet (10GbE). However, they are heavily virtualized or handling large-scale streaming audio/video applications [5], thus it became apparent that 10GbE isn't fast enough. Even though there are numerous network adapter cards that operate at 40- or 100-gigabit Ethernet, their adoption rate is still rather small. The reasons for this are many, but two major reasons are the price of the equipment and new wiring connectors.

Another aspect to consider is the processing power or CPU utilization of data center servers. The potential for high bandwidth has little value if communication overheads consume all the CPU processing power. In such a case, very little processing power is left for processing other applications. Thus, CPU utilization is as important as bandwidth. It has been shown that a GbE link in a *single-CPU server* can consume close to a half of server's processing cycles [6,7]. Thus, the CPU's processing power would become a bottleneck if architecture of the network interface card (NIC) would not change. A consequence would be severely limited throughput. One of the solutions for this problem is to dedicate one or more cores for TCP/IP processing [8]. However, with maximum operating speed of little above 3 GHz it is hard to guarantee high throughput[2]. Other methods to minimize CPU utilization include *offloading* protocol processing onto specialized hardware [10–14] and using special large packets known as jumbo frames. This has been especially beneficial in storage applications that transfers large blocks of data.

The future networks also seem to take another direction, as has been recently presented by the CEO of Cisco, John Chambers. He predicts that networks will become *programmable* and not just in the data centers, but throughout the network. This will allow greater agility, programmability and flexibility. One step in this direction is so called *Software Defined Networking*, which separates data and control planes with well-defined protocol [15]. In this concept, the control functionalities are taken out of the equipment and moved to a centralized or distributed system, while data plane is retained in the equipment. Cisco already started developing a prototype solution which they call Cisco Open Network Environment (Cisco ONE) [16].

In this dissertation, we take another step in this direction by utilizing *programmable hardware* to achieve greater agility, programmability and flexibility. We utilize the Field Programmable Gate Array (FPGA) which has been commonly used to speed up computationally intensive applications.

[2]A general rule of thumb is that it takes 1 MHz of CPU processing power to handle 1 Mb of network bandwidth [9]. Thus, a processor operating on 10 GHz would be required in order to process 10 Gbps.

1.2 Objective

The objective of this dissertation is to study high performance reconfigurable architectures for protocol processing. The study is based on the viewpoint that future computer systems will integrate small FPGAs with general purpose processors, whose role will be to accelerate computationally intensive kernels.

At first, we address the problem of efficient implementation of Cyclic Redundancy Check, which has been identified as one of major bottlenecks in iSCSI protocol implementation. Traditionally, CRCs have been used in numerous applications for various types of network data transmissions, data compression (e.g. gzip and bzip2) and data encryption. Howerver, they are often characterized as computationally intensive, and thus often substituted with less efficient error detection schemes. CRC plays an important role in the implementation of iSCSI (Internet Small Computer System Interface) protocol in Storage Area Networks (SANs) for detecting errors which occur between protocol transitions. When CRC is disabled, the network must rely on other mechanisms to detect corrupted data, such as TCP and Ethernet error detection mechanisms. Unfortunately, these mechanisms cannot detected data corruption between upper layer protocol transitions. This can lead to various problems such as failed integrity check of a database. Therefore, our goals are to (1) reduce the computational burden, (2) make architecture generic enough to support a variety of applications, (3) make architecture scalable so it can process arbitrary number of data input (4) achieve significant improvements in throughput and (5) make it area efficient.

The goal of traditional methods for designing CRC accelerators is acceleration of a specific application. In such accelerators, the resulting CRC value is determined by a CRC standard deployed by the application, which is usually fixed at the design time. We call these accelerators *non-adaptable*. If accelerator does not have ability to adapt to a new standard or an application, its usability is very limited. Thus, we propose *adaptable CRC accelerator*, which has ability to generate CRC for a variety of CRC standards and thus support a wide range of applications. Such accelerator eliminates the need for many non-adaptable CRC implementations. It also has ability to process arbitrary number of input data and generates CRC for all currently defined CRC standards during run-time.

Cyclic Redundancy Check plays an important role in IP-based storage systems or precisely iSCSI initiator [17]. IP-based storage systems often require bandwidth intensive access to storage devices, thus they exhibit high CPU utilization and low throughput when executed in a principally software implementation. This is especially evident for multi-Gbps networks where the impact of computational overhead is so pronounced that the current state of the art processors cannot take advantage of the capacity of the network. The major concern is the processing of iSCSI digests or CRC [18], thus it is common practice to disable data digests [19]. Commercial hardware iSCSI solutions have been implemented by using TCP/IP Offload Engines (TOE) or

iSCSI host bus adapters (HBA). There has been only one attempt to offload iSCSI protocol to an FPGA [14]. However, the maximum reported throughput is less than 100 Mbps, which is not adequate for new multi-Gbps networks. There are three primary reasons why we believe offloading the iSCSI protocol is challenging. First, the scope of iSCSI code is too large and requires a lot of programming effort and time. Second, some functions such as authentication, authorization and security are challenging to implement in hardware. Third, it is thought that operating frequency of FPGAs is not enough to accomplish required throughput for high-speed networks. The performance of software initiators is limited by the processing power of a general purpose processor, especially for the multi-Gbps networks [20]. The biggest concern is high level of CPU utilization that it causes. This has led to extensive research of offloading protocol processing to hardware.

In the second part of this work, we address the problem of efficiently implementing IP-based iSCSI Offload Engine which operates on the top of the TCP/IP protocol stack. Even though processing iSCSI digests have been identified as the most computationally intensive part of iSCSI protocol processing, it is not enough to offload only iSCSI digests. In such a case, the communication overhead between software and hardware parts might undermine all the performance gain. On the other hand, it is challenging to offload all iSCSI processes onto an FPGA. The target applications are mission-critical applications which require high data integrity, such as those of financial and banking transactions where database integrity failures might lead to lost funds, inaccurate stock exchange or credit card transactions. In these systems it is required to enable header and data digests, which adversely affects overall performance.

To address these issues, we propose to combine two types of computing engines, general purpose processors and FPGAs, in order to satisfy the current and future performance demands. The advantage of FPGAs is that they allow speedup of slow sequential algorithms by efficient hardware implementations. Thus, an algorithms can be partitioned into smaller units and executed in parallel on an FPGA. This means that in every clock cycle it is possible to execute all the units with no delay. FPGAs can also be re-programmed to obtain different hardware capabilities at various times and this characteristic is knows as *reconfigurability*. Another advantage is so called *dynamic reconfigurability*, which allows FPGAs to modify operation during run-time. One drawback of FPGAs is their low clock speed when compared with ASICs or CPUs, which is usually an order slower. Thus, not every algorithm is suitable for execution on FPGAs. Some algorithms might present a set of challenges due to the complexity and the volume of the code.

1.3 Contribution

In this dissertation, we address the problem of efficiently implementing 1) Cyclic Redundancy Checks accelerators and 2) IP-based iSCSI Offload Engine which operate on the top of the TCP/IP protocol stack. One major concern for CRC implementations is high level of CPU utilization and low throughput when executed in a principally software implementation. This has

led to substituting CRC with less efficient error detection schemes. We describe the design and implementation of non-adaptable and fully-adaptable CRC accelerators based on a table-based algorithm which is suited for the flexible implementation. Although the table-based algorithm has been used in software, it has been rarely implemented in hardware as its performance is believed to be lower than traditional implementation. We prove that this approach can be successfully implemented on an FPGA and achieve significant performance improvements over related work.

Our contributions are as follows:

1) We design non-adaptable CRC accelerators with sufficient performance and reasonable resource utilization using a table-based algorithm. Based on this design, a *fully-adaptable* CRC accelerator is proposed by integrating algorithm for generating CRCs and algorithm for generating contents of tables. Resulting architecture generates CRC for any known CRC standard during run-time. It achieves throughput of up to 418.8 Gbps, when the number of input bits M is 1024. Additionally, we modify table generation algorithm in order to decrease its space complexity from $O(nm)$ to $O(n)$, where n is a number of tables, and m is a number of bits in a slice[3]. Design of our architectures guarantees scalability/expandability by processing arbitrary number of input bits M at minimal area cost. In order to show efficiency of our architecture in terms of area utilization and throughput, we design five implementations, where $M \in 64, 128, 256, 512, 1024$.

2) We analyze iSCSI traffic and identify the most commonly used functions. We measure and analyze CPU utilization and throughput of Open-iSCSI [21], which is an open source software based iSCSI initiator. Based on this analysis, we offload data transfer and related non-data functions to an FPGA based adapter. Data transfer functions are the most computationally intensive and the most executed functions in a common case scenario. Other functions which do not affect performance are implemented in software on a general purpose processor. The resulting architecture relieves the host CPU from computational burden imposed by the software implementation. It is proved that the new architecture overcomes the performance limitations imposed by a single processor which operates on 15 times higher frequency than our FPGA implementation. The iSCSI Offload Engine allows very low utilization on the host CPU of approximately 3%. Our architecture guarantees flexibility, since many functions are implemented on a general purpose processor. Any new feature, such as security functions, specification updates, CRC standards, etc., can be easily implemented.

1.4 Dissertation Organization

The dissertation organization is provided in Fig. 1.1 and here we provide short description of every chapter. In Chapter 2 we introduce two problems that we addressed in this dissertation. We survey theory behind error conrol coding, commonly used CRC algorithms and overview

[3]A slice is formed when a binary number is sliced into two or more constituent. Here, the slice is referred to Slicing-by-N algorithm, and not to FPGA slice.

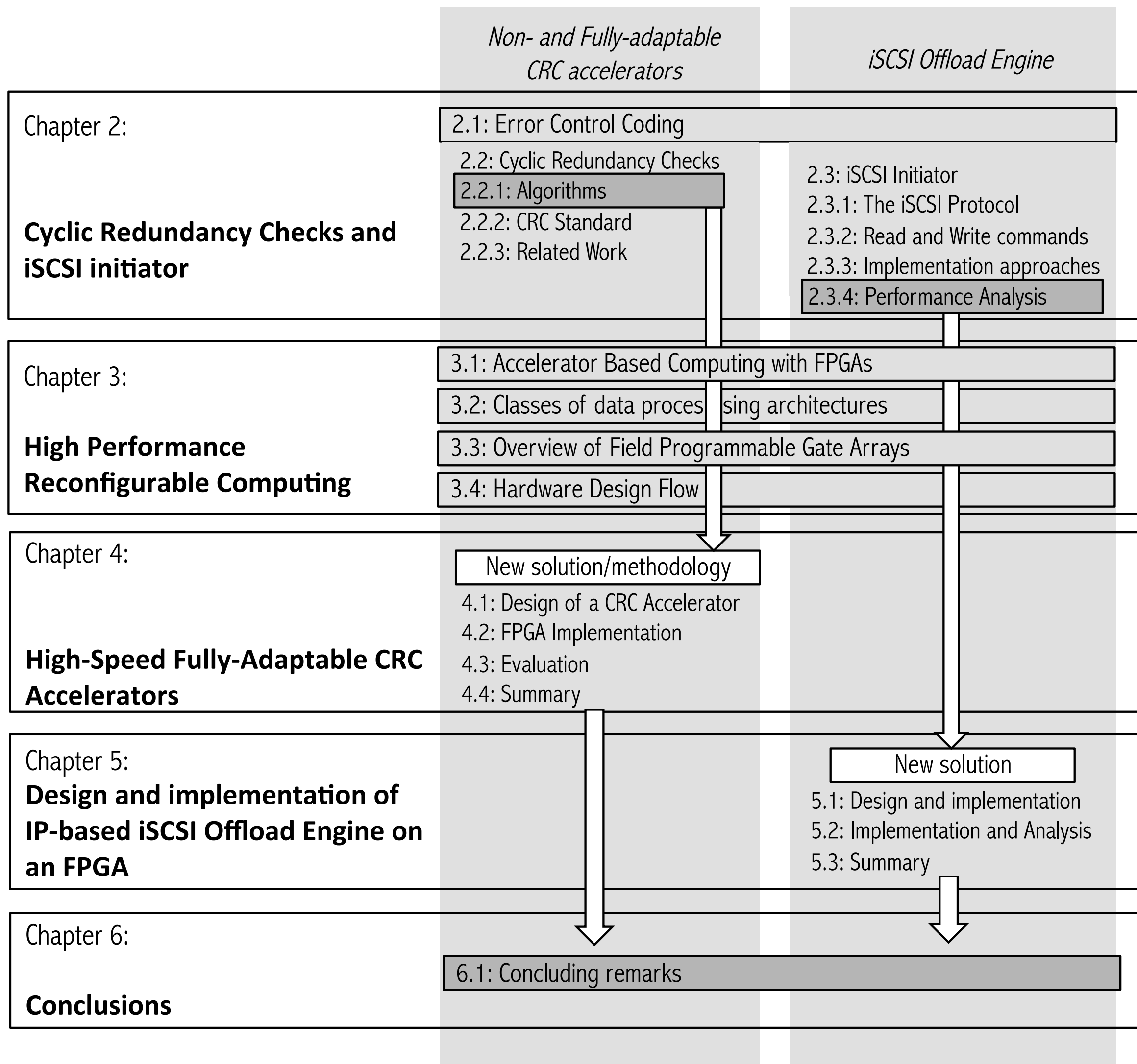

Figure 1.1: Dissertation organization

Internet Small Computer System Interface protocol. We highlight problems related to these two topics. We present results of performance analysis of software-based Open-iSCSI Initiator and highlight its bottlenecks. Chapter 3 explains current state of high performance reconfigurable technologies and gives short introduction to Field Programmable Gate Array (FPGA), which we use to solve problems related with conventional approaches. Chapter 4 proposes a new methodology for designing non-adaptable and fully-adaptable CRC accelerators. Chapter 5 proposes an iSCSI Offload Engine based on a FPGA adapter. The new architecture relieves the host CPU from computational burden imposed by the software implementation. We compare our results with similar technologies. Chapter 6 summarizes and concludes this dissertation.

In Fig. 1.2 we present positions of related work and contributions of this dissertations.

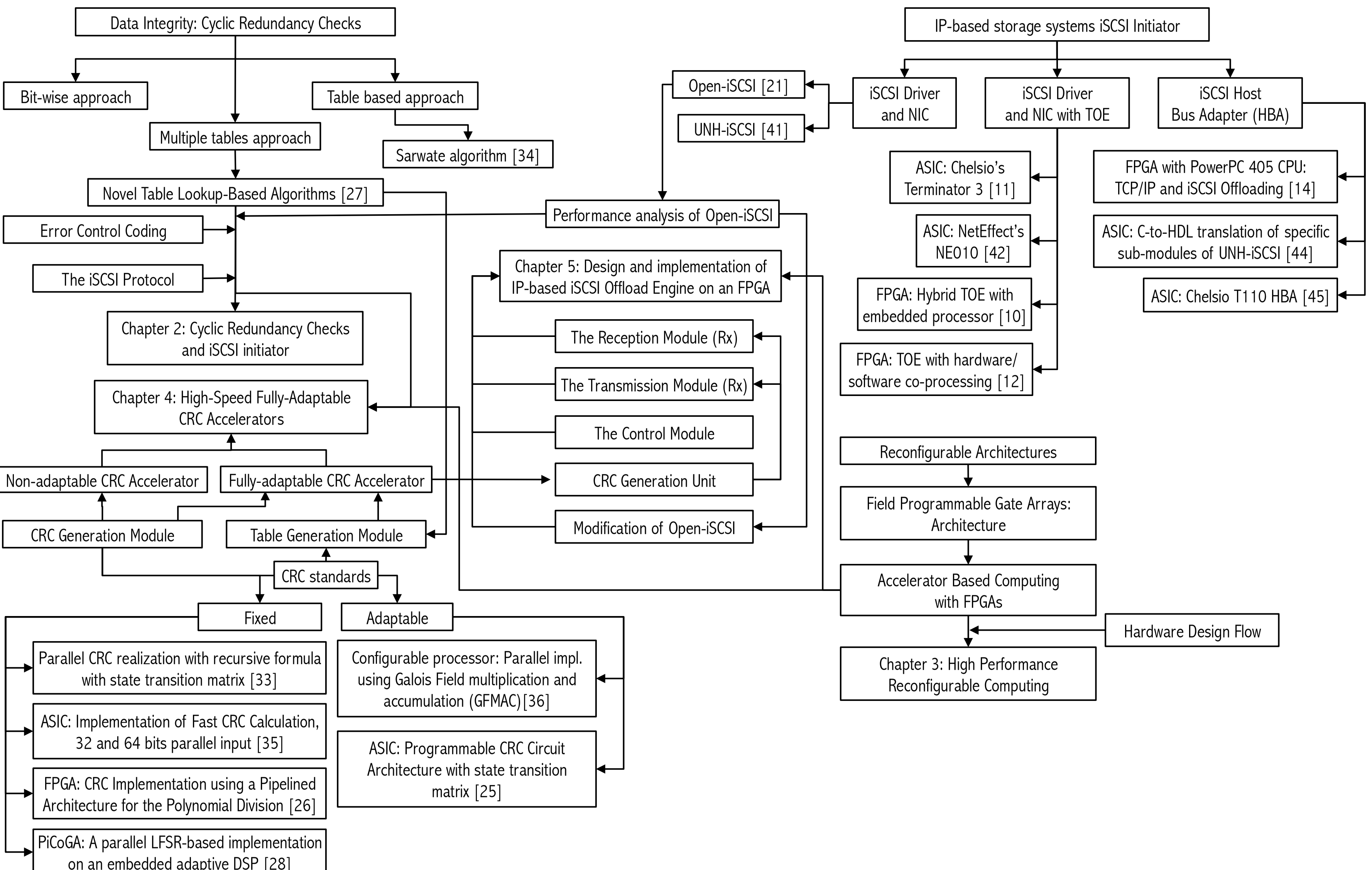

Figure 1.2: Positions and contributions of this dissertation.

Chapter 2

Cyclic Redundancy Checks and iSCSI initiator

Cyclic Redundancy Check (CRC) is a well known error detection scheme used to detect corruption of digital content in digital networks and storage devices. Numerous applications use different CRC standards and algorithms for various types of network data transmissions, data compression (e.g. gzip and bzip2) and data encryption. The simplest CRC algorithm imitates the standard hand calculations, which are repeated shifts and conditional subtracts. In this case, input bits are processed one at a time. One way to speed up this process is to merge a number of shift and conditional subtract operations together within a single clock cycle, which is commonly referred to as parallel CRC generation [22].

Traditionally, CRCs are implemented using *hardware* or *software* methods. Early hardware designs are based on the serial Linear Feedback Shift Register (LFSR), which performs the computations by handling one bit at the time [23, 24]. However, some parallel implementations are proposed in [25, 26]. In [25], *Toal et al.* proposed ASIC based architecture for parallel CRC generation, based on a method of merging shift and conditional subtract operations together within a single clock cycle [22]. While in [26], *Monteiro et al.* proposed pipelined architecture for polynomial division by using FPGAs. The basic idea is to perform a number of successive multiplications and divisions in a shift-register structure.

Software implementations can handle word-size data, thus they became more convenient and, until recently, very fast. They are based on algorithms with pre-computed remainders stored in table(s), which are referred to as *table-based* approaches. The most recent algorithms, Slicing-by-4 and Slicing-by-8, are based on *multiple tables approach*. Their main advantage is the ability to process arbitrary number of data at a time. However, with the new paradigm shift to multicore processors, it is challenging to exploit available parallelism due to a sequential nature of cyclic redundancy checks algorithms. Hence, when CRCs are implemented in principally software implementations, they provide modest throughput of up to 3.6 Gbps [27], which is not suitable for multi-Gbps networks. One additional problem is that they consume significant portion of CPU processing power, thus very little to no resources are left for execution of other applications. Hence, the major concern is that the potential of new multi-Gbps networks will

not be exploited, due to the communication overhead that consumes high levels of processor's processing power. One obvious advantage of software implementations is that they are very flexible, since very little effort is required to adapt algorithm to different CRC standards.

Even though many CRC implementations exist, there are two concerns that need to be addressed. The first one is the potential to processes data at current high-speed line rates, such as 10, 40 or more Gbps. Based on our review of current implementations, it is evident that almost all implementations provide less than 10 Gbps throughput. Only one ASIC-based implementation [28] provides around 25 Gbps of throughput. As we mentioned earlier, the most recent software implementation provides only up to 3.6 Gbps [27]. The second concern is ability to *adapt* to different CRC standards. This may become concern when specification of an application or a protocol changes, or when only one CRC accelerator is available. If the accelerator is fixed to only one CRC standard, it cannot be used with other applications or protocols. This is especially evident in ASIC implementations which tend to have fixed structure.

In this chapter we first review basic theory behind Error Control Coding, with the special focus on Cyclic Redundancy Checks. Then, we review existing algorithms for CRC generation and we highlight their bottlenecks. Then, we address one practical problem where CRCs have been identified as the major bottleneck of processing, especially when executed in principally software implementations. That is *Internet Small Computer System Interface (iSCSI) initiator*, which is one of the mostly used components in IP-based storage systems.

2.1 Error Control Coding

Error control coding is a method which provides the means to detect or correct transmission errors by introducing redundancies into the data to be transmitted. Error control coding usually refers to error *detection* and *correction*. *Error detection* is the ability to detect errors caused by noise or other impairments during transmission from the transmitter to the receiver [29]. *Error correction* has an additional feature that enables identification and correction of the errors. Error detection always precedes error correction. In this dissertation, we focus only on detection, because most network applications do not use error correcting codes. The reason is that they are expensive in terms of computational power. Also, they can correct only a few bits of errors, thus they are not effective for more common burst errors. Instead of error correction, network applications rather ask sender to re-transmit the correct bits. It is important to note that today's fiber optic network have very small error rates. If a packet is lost it is often due to the lack of buffering in routers. In a case where a small number of bit errors occur, the error correcting codes do not help. In the case of magnetic disks, they have to use error correcting codes because they cannot depend on re-transmission for error correction.

All error detection codes transmit more bits than it was in the original data. These bits are derived by some deterministic algorithm, that will produce the same output if applied on the

original data. In the case of error detection, a receiver can apply the same method as sender and check if the output is the same as received bits. These additional bits are referred as *redundancy*, *check bits* or *parity bits*. The method to generate these bits is called error detection scheme. The effectiveness of an error detection scheme is measured by the error scenario that results in the most undetected errors [30]. Its purpose is to reduce the rate of undetected errors to a level acceptable to users.

2.1.1 Types of Errors

Errors on transmission lines can occur due to *inter-symbol interference* and *noise* (Fig. 2.1). The inter-symbol interference refers to a case when the energy from a previous bit causes a bit to be wrongly interpreted. The noise, on the other hand, causes a change of a 0 level into a 1 level. The *raw error rate* is the fraction of incorrect senseword symbols (output symbols from channel demodulator), which is expressed as:

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} P(x_i \neq y_i), \tag{2.1}$$

where x_i represents input bits into a channel, and y_i output bits from the channel (Fig. 2.1).

There are three types of errors that occur on transmission lines: *random*, *burst* and *catastrophic* errors. The *random errors* are independent noise symbols. They are caused by thermal noise that is always present on transmission lines. Each noise event affects isolated symbols. The *burst errors* occur when noise event causes a contiguous sequence of unreliable symbols. They also occur due to incorrect synchronization at the physical layers (e.g. when a connector is being plugged in). The *catastrophic errors* occur when a channel becomes unusable for a period of time comparable to or longer than a data packet (e.g. ethernet collisions). In this case, re-transmission is needed because packets are very corrupted.

Definition 1 *A burst error of length l is an n-tuple whose nonzero symbols are confined to a span of l symbols and no fewer.*

The intermediate bits may or may not be corrupted. The burst error detecting ability of every (n, k) block code is less or equal to $n - k$. An ideal error detection code should be able to detect large localized burst errors and also be able to detect as many random bit errors as possible.

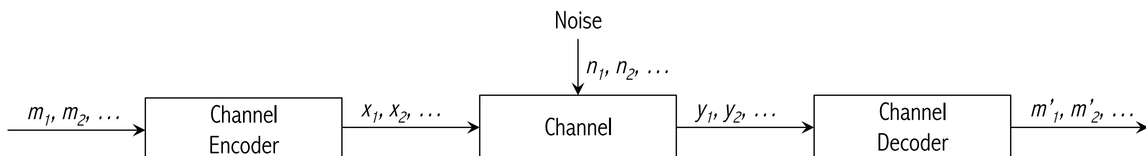

Figure 2.1: Effects of noise on transmission lines.

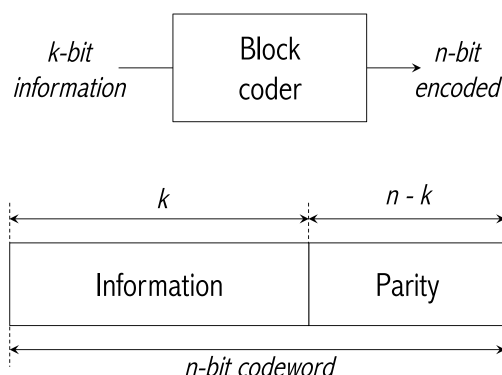

Figure 2.2: A representation of n-bit codeword.

2.1.2 Types of Error protecting codes

There are two types of error protecting codes: *block* and *convolutional* codes. In the block coding, a data is blocked into k-vectors of information digits, then encoded into n-digit codewords ($n \geq k$) by adding $p = n - k$ redundant check digits [31, 32]. The encoding of each data block is independent of past and future blocks. Since cyclick redundancy checks are a type of block codes, we focus only on block codes and provide more information in section 2.1.3.

The convolutional code is time-invariant encoding scheme, where each n-bit codeword depends on the current information digits on the past m information blocks [31, 32]. This means that the information bits are spread along the sequence. The information is mapped to code bits, instead of blocks.

2.1.3 Principles of Block Coding

In the block coding, a binary information sequence is segmented into message blocks of fixed length, where each block u consists of k bits. Thus, in the case of binary information from the field of $GF(2)$[1], there are a total of 2^k distinct messages.

In more general terms, elements from a bigger field $GF(q)$ might be considered, thus there are a total of q^k distinct messages. A message of n symbols associated with an input block is called a *codeword*. An alternative approach to repeatedly calculating code every time it is needed is to have a *lookup* table with k inputs and n outputs. This approach is feasible when k is small, however this might be infeasible when k gets large.

A *block code* is a code in which k bits (also referred as symbols) are input and n bits are output (Fig. 2.2). This code is designated as an (n, k) code. A binary block code is linear if and

[1] A field with a finite number of elements is called a *finite field* or *Galois field* (GF). In the case of GF(2), the field has only two elements {0,1}, and it is called *binary field*.

Table 2.1: A linear block code with k=4 and n = 7.

Messages	Codewords
(0 0 0 0)	(0 0 0 0 0 0 0)
(1 0 0 0)	(1 1 0 1 0 0 0)
(0 1 0 0)	(0 1 1 0 1 0 0)
(1 1 0 0)	(1 0 1 1 1 0 0)
(0 0 1 0)	(1 1 1 0 0 1 0)
(1 0 1 0)	(0 0 1 1 0 1 0)
(0 1 1 0)	(1 0 0 0 1 1 0)
(1 1 1 0)	(0 1 0 1 1 1 0)
(0 0 0 1)	(1 0 1 0 0 0 1)
(1 0 0 1)	(0 1 1 1 0 0 1)
(0 1 0 1)	(1 1 0 0 1 0 1)
(1 1 0 1)	(0 0 0 1 1 0 1)
(0 0 1 1)	(0 1 0 0 0 1 1)
(1 0 1 1)	(1 0 0 1 0 1 1)
(0 1 1 1)	(1 0 1 0 1 1 1)
(1 1 1 1)	(1 1 1 1 1 1 1)

only if the modulo-2 sum of two code words is also a code word. An example of linear block code (7, 4) is given in Table 2.1.

Definition 2 *A block code C of length n with 2^k code words is called a **linear** (n, k) code if and only if its 2^k code words form a k-dimensional subspace of the vector space of all n-tuples over the field GF(2).*

Since an (n, k) linear code C is a k-dimensional subspace of the vector space V_n of all the binary n-tuples, it is possible to find k linearly independent code words $g_0, g_1, ..., g_{k-1}$ in C such that every *codeword* v is a linear combination of these k *codewords*:

$$v = u_0 g_0 + u_1 g_1 + ... + u_{k-1} g_{k-1}, \tag{2.2}$$

where $u_i = 0$ or 1 for $0 \leq i \leq k$. These linearly independent *codewords* can be arranged in rows of a $k \times n$ matrix as shown in equation 2.3.

$$G = \begin{bmatrix} g_0 \\ g_1 \\ . \\ . \\ . \\ g_{k-1} \end{bmatrix} = \begin{bmatrix} g_{00} & g_{01} & g_{02} & \cdots & g_{0,n-1} \\ g_{10} & g_{11} & g_{12} & \cdots & g_{1,n-1} \\ . & . & . & & . \\ . & . & . & & . \\ . & . & . & & . \\ g_{k-1,0} & g_{k-1,1} & g_{k-1,2} & \cdots & g_{k-1,n-1} \end{bmatrix}, \tag{2.3}$$

where $g_i = (g_{i0}, g_{i1}, ..., g_{i,n-1})$ for $0 \leq i \leq k$. If $u = (u_0, u_1, ..., u_{k-1})$ is the message to be encoded, the corresponding *codeword* is given as follows:

$$v = u \times G = (u_0, u_1, ..., u_{k-1}) \begin{bmatrix} g_0 \\ g_1 \\ . \\ . \\ . \\ g_{k-1} \end{bmatrix} = u_0 g_0 + u_1 g_1 + ... + u_{k-1} g_{k-1}. \tag{2.4}$$

The matrix G is called a *generator matrix* for C. Its rows generate the (n, k) linear code C. Thus, the coding operation can be represented as matrix multiplication. The linear code (n, k) is completely specified by the k rows of a generator matrix G, therefore the encoder can store the k rows of G and form a linear combination based on the input message. All-zero sequence must be a codeword. Therefore, the minimum distance of the code C is the codeword of smallest weight.

The generator matrix for the (7, 4) code from Table 2.1 can be formed as follows:

$$G = \begin{bmatrix} g_0 \\ g_1 \\ g_2 \\ g_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.5}$$

Thus, the codeword for a message $u = (1\ 1\ 0\ 1)$ is calculated as:

$$\begin{aligned} v &= 1 \times g_0 + 1 \times g_1 + 0 \times g_2 + 1 \times g_3 \\ &= (1\quad 1\quad 0\quad 1\quad 0\quad 0\quad 0) \\ &\quad + (0\quad 1\quad 1\quad 0\quad 1\quad 0\quad 0) \\ &\quad + (1\quad 0\quad 1\quad 0\quad 0\quad 0\quad 1) \\ &= (0\quad 0\quad 0\quad 1\quad 1\quad 0\quad 1). \end{aligned}$$

Associated with every linear block code generator G is a matrix H called the *parity check matrix* whose rows span the nullspace of G. If v is a codeword, then it is orthogonal to each row of H, which is expressed by the following equation:

$$v \times H_T = 0. \quad (2.6)$$

From this equation, follows the following:

$$G \times H_T = 0. \quad (2.7)$$

It is called *dual code* and it has H as its generator matrix. This code is denoted as $C^{\perp}$. If G is the generator for an (n, k) code, then H is the generator for an $(n, n-k)$ code. From the example given above, the parity check matrix can be calculated as:

$$H = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}.$$

This is called the generator of an $(7, 3)$ code. And, from here it is possible to calculate the coderwords for dual code:

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{matrix}$$

It may be verified that every codeword in C is orthogonal to every codeword in $C^{\perp}$.

When original data needs to be explicitly evident in the corewords, that coding is called *systematic encoding*. Then, G is denoted as:

$$G = [P|I_k], \quad (2.8)$$

where I_k is $k \times k$ identity matrix and P is $k \times n - k$ matrix. A linear systematic (n, k) code is completely specified by a $k \times n$ matrix G in the following form:

$$G = \begin{bmatrix} g_0 \\ g_1 \\ g_2 \\ . \\ . \\ . \\ g_{k-1} \end{bmatrix} = \begin{bmatrix} p_{00} & p_{01} & \dots & p_{0,n-k-1} & | & 1 & 0 & 0 & \dots & 0 \\ p_{10} & p_{11} & \dots & p_{1,n-k-1} & | & 0 & 1 & 0 & \dots & 0 \\ p_{20} & p_{21} & \dots & p_{2,n-k-1} & | & 0 & 0 & 1 & \dots & 0 \\ & & & & | & & & & \dots & \\ & & & & | & & & & \dots & \\ & & & & | & & & & \dots & \\ p_{k-1,0} & p_{k-1,1} & \dots & p_{k-1,n-k-1} & | & 0 & 0 & 0 & \dots & 1 \end{bmatrix}. \quad (2.9)$$

Similarly, H parity matrix of an (n,k) linear code can take the following form:

$$H = [I_{n-k}|P^T] = \begin{bmatrix} 1 & 0 & 0 & ... & 0 & | & p_{00} & p_{01} & \cdots & p_{k-1,0} \\ 0 & 1 & 0 & ... & 0 & | & p_{10} & p_{11} & \cdots & p_{k-1,1} \\ 0 & 0 & 1 & ... & 0 & | & p_{0} & p_{21} & \cdots & p_{k-1,2} \\ & & & ... & & | & & & & \\ & & & ... & & | & & & & \\ & & & ... & & | & & & & \\ 0 & 0 & 0 & ... & 1 & | & p_{0,n-k-1} & p_{1,n-k-1} & \cdots & p_{k-1,n-k-1} \end{bmatrix}. \quad (2.10)$$

2.1.4 Error Detection Schemes

There are several schemes for error detection. We will review some of them, with emphasize on CRC (section 2.2).

1. **Repetition schemes**: The data is broken up into blocks of bits and each block is sent predetermined number of times. This scheme is not efficient since all repeated blocks of data might be corrupted, thus the receiver might think that the data is correct.

2. **Parity schemes**: The data is broken up into blocks of bits and the number of 1 bits is counted. The extra bit is added during the transmission. This bit is called *parity bit*. There are two type of parity scheme: even and odd parity scheme. In the *Even scheme*, if the number of "1" bits is even (including parity bit), then the parity bit is set to "0", else to "1". In the *Odd scheme*, if the number of "1" bits is odd (including parity bit), then the parity bit is set to "1", else to "0". The problem arises when there are even number of error bits. In this case, the parity scheme can not detect errors. This scheme also can not identify the error bit's position.

3. **Checksum**: The data is broken up into blocks of bits whose binary values are added to form a *checksum*. The checksum is appended to the end of the transmitted message. On the other end of the communication channel, this process is repeated and the result is compared with the existing checksum. A non-match indicates an error, while a match only indicates that the algorithm did not detect any errors. The checksum can not detect all types of errors, some of which are the following: reordering of the bytes, inserting or deleting zero-valued bytes and multiple errors that cancel each other out.

4. **Cyclic redundancy checks**: This error detecting scheme is almost universally used in networks. This is because CRCs have good burst error detection properties and good random bit error detection properties. It has high probability of detection for most larger random bit errors. Its advantage over other detecting scheme is that it can detect almost all errors with relatively low number of bits of redundancy. It is detailed in section 2.2.

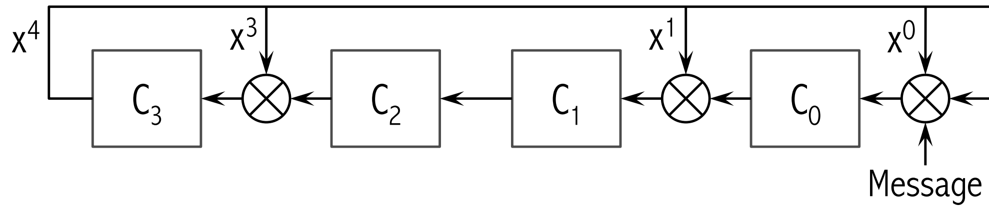

Figure 2.3: LSFR for polynomial $P(x) = x^4 + x^3 + x + 1$.

2.2 Cyclic Redundancy Checks (CRC)

CRC is calculated by performing long division operation between input message and a generator polynomial. At first, a message M is multiplied by x^w (x is a dummy variable, while w is the length of a generator polynomial), which is equivalent to shifting to left by a polynomial length. This value is then divided by a generator polynomial $G(x)$, and the remainder is called CRC as shown in equation 2.11. The CRC is affixed to the original message M and transmitted to a receiver.

$$CRC(M) = M(x) \times x^w \bmod G(x) \tag{2.11}$$

An input data or a message M is treated as a polynomial, where bit values are coefficients of a dummy variable x. The coefficients are all either 0 or 1, while the power of x corresponds to the bit position. For example, the message "01010100" is represented as $0 \times x^7 + 1 \times x^6 + 0 \times x^5 + 1 \times x^4 + 0 \times x^3 + 1 \times x^2 + 0 \times x^1 + 0 \times x^0$. If the length of M is defined as l, then M can be represented as:

$$M(x) = m_{l-1}x^{l-1} + m_{l-2}x^{l-2} + ... + m_0 \tag{2.12}$$

where m_{l-1} is the most significant bit of a message M and m_0 is the least significant bit. A generator polynomial of length w is represented in the same manner:

$$G(x) = g_w x^w + g_{w-1}x^{w-1} + g_{w-2}x^{w-2} + ... + g_0 \tag{2.13}$$

Due to interference during transmission, data might be corrupted during transport. Errors will be detected on a receiver's side by performing similar process as a sender. At first, a receiver will remove received CRC, then it will perform long division operation with the same generator polynomial specified by a protocol used. Then it will compare received CRC value and its own. Any discrepancy between these two values indicates the presence of transmission errors in the received pair. In this case, a receiver will discard the message and request re-transmission of the data.

2.2.1 Algorithms for CRC Computation

In this section we overview widely used approaches for generating CRC with emphasize on newly proposed ones [27]. We highlight major overheads of these approaches.

2.2.1.1 Bitwise approach

In this approach, CRC is calculated with N shifts and XOR operations for N-bit input message, which makes this algorithm computationally intensive. Early hardware implementations were based on this algorithm, and implemented using linear feedback shift registers (LFSR) [24] (illustrated in Fig. 2.3). Input message is fed *serially* into a circuit, hence if implemented on an FPGA throughput would be limited by operating frequency of an FPGA (e.g. 200 MHz limits throughput to 200 Mbps, thus will not be suitable for high-speed links). Some level of parallelism must be introduced to gain higher throughput.

Traditional implementation with LFSR can be modified to combine the message with the most significant register bit to form the feedback, as is illustrated in Fig. 2.4. This optimization has been referred to as LFSR2. The advantage is that no zeros are needed to be shifted at the end, thus the CRC can be generated in w clock cycles earlier (w is the length of a polynomial).

2.2.1.2 Parallel CRC Computation

In [33], Campobello et al. proposed parallel implementation of the CRC based on the circuit in Fig. 2.4. The initial assumption is that the degree of the polynomial (w) and the length of the message are both multiples of the number of bits processed in parallel (M). From linear systems theory, time-invariant linear system can be expressed as follows:

$$\begin{aligned} X(i+1) &= FX(i) + GU(i) \\ Y(i) &= HX(i) + JU(i), \end{aligned} \tag{2.14}$$

where X is the state of the system, U input, Y output, and i ranges from 2 to M. F, G, H, J are used to denote matrices, and X, Y, and U column vectors. Thus, $X(i)$ can be derived from previous equation as:

$$X(i) = F^i X(0) + [F^{i-1}G \cdots FG\ G][U(0) \cdots U(i-1)]^T. \tag{2.15}$$

Matrix F and G are chosen according to the equations of serial LFRS, while H and J are the identity and zero matrices, respectively. Thus, X and H are:

$$X = [x_{w-1} \cdots x_1 x_0]^T$$
$$H = I_w.$$

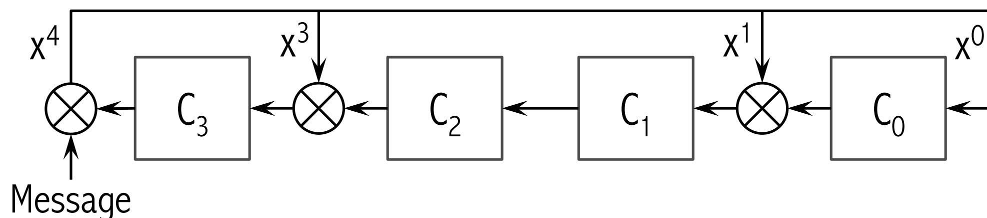

Figure 2.4: LSFR2 for polynomial $P(x) = x^4 + x^3 + x + 1$.

If the identity matrix is of size $w \times w$, then

$$J = [00 \cdots 0]^T,$$
$$U = d,$$
$$G = [00 \cdots 1]^T,$$

$$F = \begin{bmatrix} G_{w-1} & 1 & 0 & ... & 0 \\ G_{w-2} & 0 & 1 & ... & 0 \\ ... & ... & ... & ... & ... \\ G_1 & 0 & 0 & ... & 1 \\ G_0 & 0 & 0 & ... & 0 \end{bmatrix}.$$

When $i = M$, then $X(M)$ can be derived as:

$$X(M) = F^M \otimes X(0) \oplus [0 \cdots 0|d(0) \cdots d(M-1)]^T. \tag{2.16}$$

From here it is possible to obtain a *recursive formula*:

$$X' = F^M \otimes X \oplus D, \tag{2.17}$$

where X' and X represent the next and the present state of the system, while $D = [d_{w-1} \cdots d_1 d_0]^T$ assumes the following values: $[0 \cdots 0|b_0 \cdots b_{M-1}]^T$, $[0 \cdots 0|b_M \cdots b_{M-1}]^T$, where b_i are bits of original input data followed by a sequence of w zeros.

For every generator polynomial, new F^w matrix must be derived, and from there it is possible to derive matrix F^M. F^w matrix is constructed recursively from the following formula:

$$F^i = \left[F^{i-1} \otimes \begin{bmatrix} p_{m-1} \\ ... \\ p_1 \\ p_0 \end{bmatrix} \;\middle|\; \text{the first } m-1 \text{ columns of } F^{i-1} \right]. \tag{2.18}$$

F^M is defined as:

$$F^M = \left[F^{M-1} \otimes P' ... F \otimes P' \; P' \; | \tfrac{I_{m-w}}{0} \right]. \tag{2.19}$$

While, F^w is:

$$F^w = \left[F^{w-1} \otimes P' | ... | F \otimes P' | P' \right]. \tag{2.20}$$

Here, we show four examples of F^w matrices based on four polynomials. In order to improve the readability, matrices F^w are reported as a column vector in which each element is the hexadecimal representation of the binary sequence obtained from the corresponding row of F^w, where the first bit is the most significant. G denotes generator polynomial.

CRC-12: G = $\{1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1\}$

$F^{12}_{CRC-12} = [CFF\ 280\ 140\ 0A0\ 050\ 028\ 814\ 40A\ 205\ DFD\ A01\ 9FF]^T$

CRC-16: G = $\{1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1\}$

$$F^{16}_{CRC-16} = [2DFFF\ 3000\ 1800\ 0C00\ 0600\ 0300\ 0180\ 00C0\ 0060\ 0030\ 0018\ 000C\ 8006\ 4003\ 7FFE\ BFFF]^T.$$

CRC-CCITT: G = $\{1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1\}$

$$F^{16}_{CRC-CCITT} = [0C88\ 0644\ 0322\ 8191\ CC40\ 6620\ B310\ D988\ ECC4\ 7662\ 3B31\ 9110\ C888\ 6444\ 3222\ 1911]^T.$$

CRC-32: G = $\{1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1\}$

$$F^{32}_{CRC-32} = [FB808B20\ 7DC04590\ BEE022C8\ 5F701164\ 2FB808B2\ 97DC0459\ B06E890C\ 58374486\ AC1BA243\ AD8D5A01\ AD462620\ 56A31310\ 2B518988\ 95A8C4C4\ CAD46262\ 656A3131\ 493593B8\ 249AC9DC\ 924D64EE\ C926B277\ 9F13D21B\ B409622D\ 21843A36\ 90C21D1B\ 33E185AD\ 627049F6\ 313824FB\ E31C995D\ 8A0EC78E\ C50763C7\ 19033AC3\ F7011641]^T.$$

2.2.1.3 Table based approach

The main idea behind this approach is to pre-compute remainders for a specific input and store them into a table. Widely used algorithm with this approach is known as Sarwate algorithm [34]. This algorithm has been designed when computer architectures supported XOR operation with only *eight* bits, but it is still used today in low-performance implementations. In general, to process a message M of length l, Sarwate algorithm requires a table of $2^l \times (w-1)$ pre-computed remainders (w is the length of a generator polynomial). Today's processors support operations with 32 and 64 bit values, thus if this algorithm is to be extended it would require lookup tables of $2^{32} \times (w-1)$ and $2^{64} \times (w-1)$ for processing 32 or 64 bit input data, respectively.

Number of bits processed in parallel (l)	Size of a table
8	$2^8 \times (w-1)$
16	$2^{16} \times (w-1)$
32	$2^{32} \times (w-1)$
64	$2^{64} \times (w-1)$

The problem arises when one wants to extend the number of bits processed in parallel l, as is shown in the table bellow. The size of the table grows very quickly, and it would be impossible to implement this algorithm efficiently in software or hardware. These tables cannot fit into a

cache so their contents have to be constantly fetched from the main memory, causing significant performance drop.

2.2.1.4 Multiple tables approach

Looking to overcome limitations of processing only 8 bits of data at a time, two new algorithms have been proposed and evaluated in [27]: ***Slicing–by–4*** and ***Slicing–by–8***. The main advantage is that they can read and process *arbitrarily large amounts* of data at a time. The algorithms are based on two principles associated with modulo-2 arithmetic: 1) *bit-slicing* and 2) *bit replacement*. The bit-slicing principle suggests that if a binary number is sliced into two or more constituent terms, then the CRC value can be calculated as a function of the CRC values of its constituent terms. The bit replacement principle suggests that the amounts of bits from bitstreams can be replaced by potentially much smaller in length binary numbers producing the same CRC values.

The Slicing–by–4 and Slicing–by–8 are based on the algorithmic framework which distinguishes the first and all subsequent steps. The reason why these steps are different is because the length on input stream may not be a multiple of the amount of bits that are read at a time q. Thus, different set of tables may be needed than in all other subsequent steps. At first, we provide details of the *first step* and then every subsequent step k.

First step differs from all subsequent steps. Let $B = [b_1 b_2 ... b_l]$ be the input bitstream, $P = [b_1 b_2 ... b_p]$ the initial p most significant bits of B, and b_1 the most significant bit of P and B. The length l of B is $l > p$, and the length g of generator polynomial $G(x)$ is $g < l$. Then, the $l - g + 1$ most significant bits of B are the information bits that are being encoded, and the $g - 1$ bits of B are equal to zero.

In order to be able to read potentially large amounts of data without the need to access a lookup table of 2^p entries, P is sliced into m slices, which are symbolized as $P_1, P_2, ..., P_m$ with lengths $p_1, p_2, ..., p_m$. Thus, the binary number P and its length p are expressed as $P = [P_1 : P_2 : ... : P_m]$ and $p = \sum_i p_i$ for every $i \in [1, m]$.

Each slice is associated with separate table. Thus, there are m different tables $T_1, T_2, ..., T_m$ of sizes equal to $2^{p_1}, 2^{p_2}, ..., 2^{p_m}$, respectively. Each table T_i contains the remainders from the long division of all possible values of slice P_i shifted by an offset o_i. The divisor used is the generator polynomial. The offset o_i is given by

$$o_i = \sum_{j=i+1}^{m} p_j. \tag{2.21}$$

The remainders during the first step are calculated with

$$R_i^{(1)} = P_i \cdot 2^{o_i} \text{ mod } G, \text{ where } i \in \{1, 2, ..., m\}. \tag{2.22}$$

Then, the $R^{(1)}$ is defined as the result of XOR operation ($\oplus$) between all $R_i^{(1)}$ values returned from tables:

$$R^{(1)} = \oplus_{i=1}^{m} R_i^{(1)}. \tag{2.23}$$

If $Q^{(1)}$ represents the next q bits of the bitstream, which are positioned after the initial p bits, then $Q^{(1)}$ is defined by

$$Q^{(1)} = [b_{p+1} b_{p+2} ... b_{p+q}]. \tag{2.24}$$

From here, it is possible to define the binary number $S^{(1)}$:

$$S^{(1)} = [R^{(1)} : Q^{(1)}] = R^{(1)} \cdot 2^q \oplus Q^{(1)}. \tag{2.25}$$

The *First step* ends with derivation of the binary number $S^{(1)}$. In each subsequent step k, the algorithms operate on a binary number $S^{(k-1)}$ produced during the previous step $k-1$.

Step k: The binary number $S^{(k-1)}$ of length s is sliced into n slices, which are symbolized as $S_1^{(k-1)}, S_2^{(k-1)}, ..., S_n^{(k-1)}$ with corresponding slice lengths $s_1, s_2, ..., s_n$. Hence, $S^{(k-1)} = [S_1^{(k-1)} : S_2^{(k-1)} : ... : S_n^{(k-1)}]$, while $s = \sum_i s_i$ for $i \in [1, n]$. Each slice is used to perform a table lookup. The number of tables equals number of slices and every step $k > 1$ uses the same set of tables. For n slices, there are n tables symbolized as $T_1^{'}, T_2^{'}, ..., T_n^{'}$ with sizes equal to $2^{s_1}, 2^{s_2}, ..., 2^{s_n}$ entries, respectively. Each table $T_i^{'}$ contains the remainders from the long division of S_i^{k-1} shifted by an offset f_i. Similar to equation 2.21, the offset f_i is defined as:

$$f_i = \sum_{j=i+1}^{n} s_j. \tag{2.26}$$

The remainders during this step are calculated by

$$R_i^{(k)} = S_i^{k-1} \cdot 2^{f_i} \text{ mod } G, \text{ where } i \in \{1, 2, ..., m\}. \tag{2.27}$$

Then, the $R^{(k)}$ is defined as the result of XOR operation ($\oplus$) between all $R_i^{(k)}$ values returned from tables:

$$R^{(k)} = \oplus_{i=1}^{n} R_i^{(1)}. \tag{2.28}$$

Subsequently, the binary number $S^{(k)}$ is calculated from $R^{(k)}$, and the next q bits of the bitstream are formed:

$$S^{(k)} = [R^{(k)} : Q^{(k)}] = R^{(k)} \cdot 2^q \oplus Q^{(k)}. \tag{2.29}$$

The step k ends with derivation of the binary number S^k. In subsequent iteration, the same procedure of bit slicing and parallel table lookups is repeated until all the bits in the bitstream are processed. The total number of steps N that are required for the calculation of a CRC value for bitstream B of length l is equal to

$$N = \left\lfloor \frac{l}{q} \right\rfloor + 1, \tag{2.30}$$

where q is the number of bits read at a time. It is assumed that the number of bits initially read p is not the same as q.

The difference between two algorithms and comparison with Sarwate algorithm is given below. The *Slicing–by–4* algorithm reads and processes 32 bits at a time, and it doubles the performance of existing implementations of Sarware algorithm. The algorithm deploys *four* tables with pre-computed remainders, which are accessed in parallel by using *four* 8 bit slices. A slice is formed by slicing input binary number into four constituent terms. Each table requires 1 KB of data in the cache (256×32 bits values for 33 bit generator polynomial), thus *Slicing–by–4* requires 4 KB of data in memory. This amount of data can easily fit in today's cache, resulting in faster execution, but it is still limited by the speed of a processor.

Similarly, the *Slicing–by–8* triples the performance of the Sarware algorithm. It reads and processes 64 bits at a time, and it deploys *eight* look-up tables accessed by *eight* 8 bit slices. The algorithm requires 8 KB of data in the cache.

2.2.2 CRC Standard

The content of a pre-computed table depends on specific parameters of a CRC standard, as well as on the position of a byte in the input stream that is being processed. A CRC standard is defined by 8 parameters, as shown on the example of CRC32c standard in Table 2.2.

$Width$ defines width of the algorithm. $Poly$ defines hexadecimal value of a generator polynomial, with top bit omitted, since its value is always 1. $Init$ defines initial value of a CRC register used only in the first iteration of a CRC algorithm. Input message is reflected before performing long division operation if parameter $RefIn$ is true, e.g. 8 bit value will be processed with bit 7 being treated as the least significant bit (LSB) and bit 0 as most significant bit (MSB). If $RefIn$ is false, input bits will not be reflected. Similarly, if $RefOut$ parameter is defined as true, the remainder is reflected before writing it into the table. $XorOut$ parameter is defined as a hexadecimal value and it is used in a final stage before the value is returned as the official checksum. $Check$ parameter is defined as hexadecimal value that represents CRC value of the ASCII string "123456789". It is used as a weak validator of implementations of the algorithm. The parameters $Name$ and $Check$ are not of any use for our implementation, thus we omit them.

Table 2.2: A list of parameters defined by a CRC standard shown on the example of CRC32c standard.

Parameter	Value
Name	CRC-32C
Width	32
Poly	1EDC6F41
Init	FFFFFFFF
RefIn	True
RefOut	True
XorOut	FFFFFFFF
Check	E3069283

In the server-type workload, CRC standard can be changed often. This is related to a number and a type of applications and protocols usually executed/employed on the server. Some applications/protocols share a common CRC standard, in which case it is not required to change functionality. However, many applications/protocols employ different CRC standards. Some examples are listed in Table 2.3.

CRC standard is sometimes changed in the specification of an application or a protocol, due to the policy change or the need for more effectiveness. In some cases, it is important to have ability to adapt fast. When CRC is implemented in software, then adaptation is very fast (the code is modified, compiled and ran). However, in the case of ASIC solutions, the entire chip or often the accelerator card has to be replaced, which affects operational cost of the system.

2.2.3 Related Work

Implementations of CRC accelerators with fixed CRC Standards are presented in [26,28,33,35]. In [33] authors identified a recursive formula in serial implementation from which they derived parallel implementation, achieving maximum of 4.38 Gbps while processing 32 bits at a time, occupying only 162 LUTs. In [35] authors design a circuit with two parallel calculation units, capable of processing 32 and 64 bits of input in two different implementations. They operate on 180 MHz in 0.35 micron technology, with maximum throughput of 5.76 Gpbs for 32 and 64 bits processed every clock cycle. In [26] CRC is implemented in a pipeline structure, with a number of successive multiplications and divisions. The maximum reported throughput for processing 32 bits at a time with 16 bits generator polynomial is 4.585 Gbps with clock of 153.84 MHz, and 2.838 Gbps for processing 32 bits with 32 bits polynomial, with clock of 95.23 MHz on Altera FLEX 10KE

Table 2.3: A list of CRC standards with associated applications and protocols.

CRC Standard	Applications and Protocols
CRC-16-CCITT	Bluetooth, HDLC FCS, SD
CRC-16-IBM	USB
CRC-24	OpenPGP
CRC-32	Serial ATA, Gnuradio, HDLC, Ethernet, MPEG-2, Gzip, Bzip2
CRC-32C	iSCSI, Btrfs, ext4, SCTP
CRC-40	GSM control channel
CRC-64-ISO	HDLC
CRC-64-ECMA-182	XZ Utils

device. In [28], the implementation of parallel LFSR-based applications on an adaptive DSP featuring a Pipelined Configurable Gate Array (PiCoGA) has been presented, with Ethernet's 32 bits CRC as a test-case. PiCoGA is integrated in the embedded digital signal processor based on run-time reconfigurable technology (named DREAM), featuring a working frequency of 200 MHz. On the target architecture, CRC circuit achieves up to 25 Gbps throughput with a parallel LFSR processing 128 bits at time.

There is relatively little work in the area of fully-adaptable CRCs on FPGAs. We found only two other hardware implementations [25, 36] that can support very limited number of generator polynomials. The re-generation in [36] is achieved with Galois Field Multiplication and Accumulation (GFMAC) with *soft-coded* and *hard-coded* generator polynomials. Soft-coded implementation is much slower than fixed hard-coded counterpart. The maximum throughput of soft-coded design with 32 bit CRC is 1.3 Gbps for a 128 bit message. Unfortunately, the reconfiguration time is not provided. The implementation [25] can process variable number of 32 bit generator polynomials, and can be modified to support 64 bit, but cannot support both at the same time in one circuit. The maximum throughput for processing 32 bits with a 32 bit generator polynomial is 4.92 Gbps. It is generic in its design, thus it can be scaled to 64, 128 or 256 bits, with maximal theoretical throughput of 40 Gbps at 256 bits. The reconfiguration time is not very specific - under 1μs.

The mostly used software solution [27] achieves maximum of 3.6 Gbps for processing 64 bits at a time on Intel Pentium 1.7 GHz. In [37] we measured performance of the same algorithm on the state-of-the-art Xeon 3 GHz processor with 4MB of L2 cache, and the throughput was 9.58 Gbps while processing 512 bits at a time, in idealized conditions without cache misses. There is no research about CRC circuits which support 64 bit generator polynomials.

2.3 Internet Small Computer System Interface (iSCSI) initiator

In this section, we overview *Internet Small Computer System Interface (iSCSI) initiator*, which is one of mostly used components in IP-based storage systems. The target applications are mission-critical applications which require high data integrity, such as those of financial and banking transactions where database integrity failures might lead to lost funds, inaccurate stock exchange or credit card transactions. In these systems it is required to enable header and data digests (CRC), which adversely affects overall performance. IP-based storage systems often require bandwidth intensive access to storage devices, thus *they exhibit high CPU utilization and low throughput* when executed in a principally software implementation. Even though, CRC has been identified as one of the major bottlenecks in iSCSI implementations, it is not enough to offload only CRC to an accelerator card. In this case, the overhead of communication between software and hardware parts might undermine all the effort.

Hereafter, we overview the iSCSI Protocol, the common approaches of implementation, and we provide performance analysis of commonly used software-based Open-iSCSI. We highlight bottlenecks of this implementation. At the end of this section, we provide related work. In Chapter 5, we propose to offload frequently executed operations to an FPGA-based accelerator card to address the problems of high CPU utilization and low throughput.

2.3.1 The iSCSI Protocol

The iSCSI protocol is a transport for SCSI packets over TCP/IP infrastructure. The information exchange is based on a client/server model where the client is called *initiator*, and server *target*. The initiator and target divide their communications into messages, which are called Protocol Data Units (PDUs). Typically, an initiator issues commands to a SCSI target to request transfer of data to/from I/O devices. The group of TCP connections that link an initiator with a target form a session. A session has two phases: Login and Full Feature Phase. In the Login Phase, an initiator and a target negotiate protocol and security parameters, and authenticate each other for the rest of the session. The session then transitions to the Full Feature Phase. In this phase, an initiator may send SCSI commands and data to various SCSI devices on the target. The majority of protocol processing load happens in the second phase.

2.3.2 Processing of iSCSI Read and Write commands

The principal layers of the storage networking model based on iSCSI are shown in Fig. 2.5. The data segment represents the SCSI command set for communication with SCSI devices. iSCSI layer is responsible for transmitting and receiving SCSI commands over TCP/IP infrastructure. The TCP layer is used as end-to-end protocol to establish a reliable session, and for delivering in-order TCP segments to the iSCSI layer. The IP layer is used to route the datagrams between

network devices, and the Ethernet layer is used as MAC protocol handler to transfer Ethernet frames across the physical link.

Fig. 2.6 illustrates an exemplary flow diagram for processing a) incoming Data-In PDU (part of *READ* operation) and b) outgoing Data-Out PDU (part of *WRITE* operation). These units are two main vehicles by which SCSI data payload is transmitted between an initiator and a target.

In the case of Fig. 2.6a), the initiator first sends the request for reading data in the form of SCSI READ commands to a target. Then, the target sends requested data. Ethernet, IP and TCP layers are first to be processed by either TCP/IP Offload Engine (TOE) or some software implementation. Then, if the frame does not correspond to an iSCSI PDU, it is forwarded either to a different network processor or to the main memory. Else, the header is validated by calculating its digest in the second phase. If the newly calculated digest is not the same as the received one, the PDU is dropped and re-transmission request is sent. In the third phase, the information in the iSCSI frame is identified and corresponding operations are performed. In the final phase, the digest of data segment is calculated and compared with the received data digest. If two digest values are equal, the data segment is copied to the main memory. If not, the frame is dropped and re-transmission is requested.

In the case of Fig. 2.6b), the initiator first sends the request for writing data in the form of SCSI WRITE commands to a target. Then, the target sends *R2T PDU* informing the initiator that it is ready to transmit. When the initiator receives R2T, transmission of data-out PDU may begin. The formation of Data-Out PDU begins with construction of its header. Then, the header and data segment digest are calculated, respectively. The header, header digest, data segment and data digest are then encapsulated with TCP, IP and Ethernet layers to form a Data-Out PDU and sent to a target.

Figure 2.5: Layers of iSCSI packet. The formation of the packet begins with data segment, creation of header and data digests, and an appropriate iSCSI header. Then, the packet is build out through TCP, IP and Gigabit Ethernet layers. (Optional fields are marked with *.)

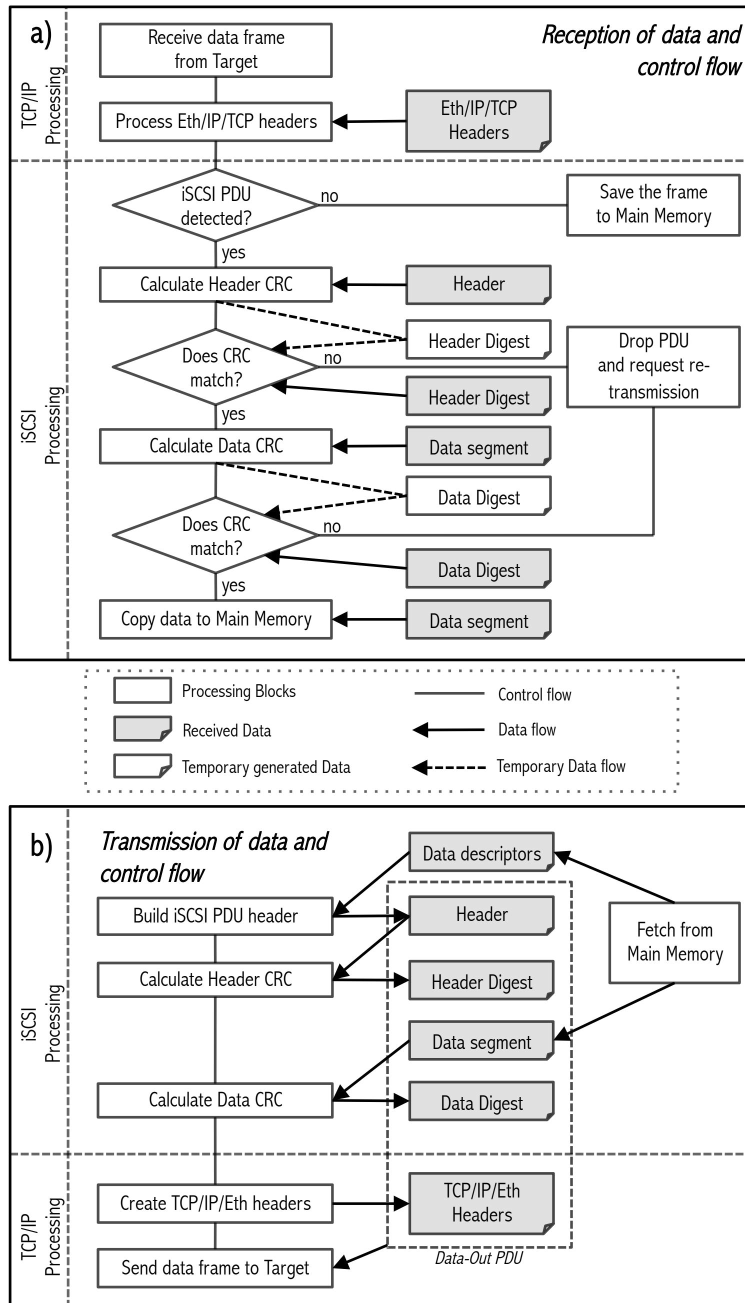

Figure 2.6: Flow diagram for processing of a) Data-In PDU on the initiator, which performs SCSI read on the target; b) Data-Out PDU on the initiator, which performs SCSI write on the target.

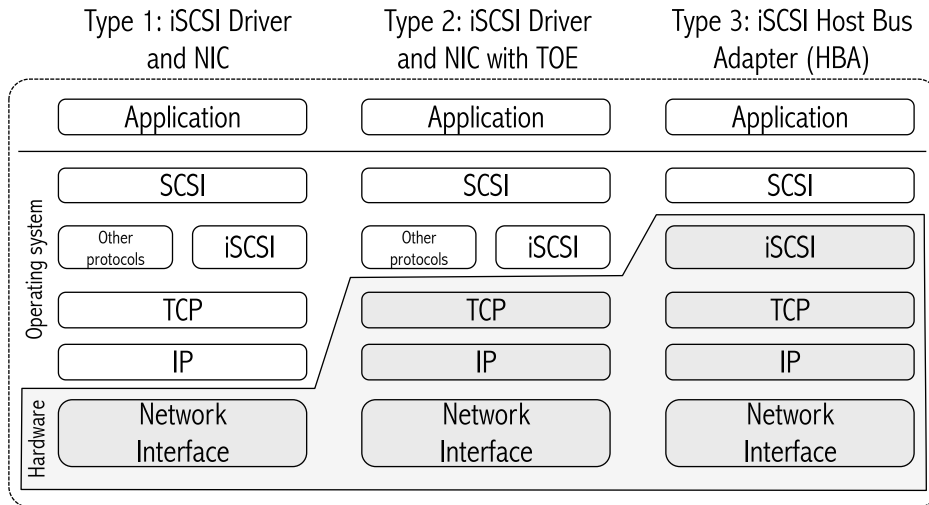

Figure 2.7: Three major implementation choices for iSCSI.

2.3.3 Implementation Approaches

In recent studies we found three major implementation choices for iSCSI (Fig. 2.7):

Type 1: *iSCSI Driver with NIC*: coupled with a generic Ethernet NIC with software implementation of iSCSI initiator.

Type 2: *iSCSI Driver with TCP offload engine*: entire TCP/IP stack is offloaded onto a special purpose hardware accelerators coupled with operating system based iSCSI initiator.

Type 3: *iSCSI Host Bus Adapter*: TCP/IP and iSCSI initiator functions are offloaded to a special purpose hardware.

For some applications, software initiators (Type 1) will suffice, but more-demanding applications require offloading of iSCSI processing to hardware initiators. The main advantage of software based iSCSI initiators is their ability to easily adapt to modifications in the protocol. There are two types of iSCSI hardware initiators. Type 2 only offloads TCP/IP processing from the system's CPU to a specialized Ethernet card which is called TCP Offload Engine. Type 3 offloads both TCP/IP and iSCSI processing from the system CPU to a specialized adapters known as iSCSI Host Bus Adapters. They are usually implemented as ASIC solutions with superior performance when compared to performance of Type 1, but they lack flexibility. Examples of existing implementations are reviewed in section 2.3.5.

2.3.4 Performance Analysis of Open-iSCSI

We analyzed iSCSI traffic with Wireshark [38], the open source network packet analyzer. We measured traffic between a software initiator and a target by using a set of microbenchmarks. The microbenchmarks transmitted arbitrary number of data in both directions. The iSCSI commands are issued to read/write from/to the same disk block address multiple times in order to minimize the number of cache misses. We setup a software initiator with Open-iSCSI [21] on Intel Core2 CPU 2.40 GHz with 8 GB of RAM, and a target by using Linux SCSI target framework [39] on the Intel Core2 Quad CPU 2.83 GHz with 8 GB of RAM. The operating system on both CPUs was based on Linux kernel 2.6.34.

In the most common case transmission, 60-70% of instructions were Data-In and/or Data-Out PDUs, following by R2T, SCSI Commands and Responses with 10-20%. The remaining instructions were related to mostly Login Phase, connection cleanup and connection termination. Then, we analyzed number of instructions and CPU utilization with Oprofile [40], which is a system-wide profiler for Linux kernel. Fig. 2.8 shows the performance profile of processing Data-In PDUs when header and data digests are enabled. The cost of data digest processing (with kernel's CRC32c module) represents about 50% of the total number of instructions for 8KB workload size, while the iSCSI protocol processing is only 4%. As expected, the data digest processing increases linearly with I/O workload size, while processing cost is decreasing. During our experiments, CPU utilization was the highest when data digests were enabled, varying from 33% to 70% of processor's resources. Thus, little or no processing resources are left for other applications. When data digest is disabled, the cost of header digest processing is indistinguishable with 1% of the total number of instructions.

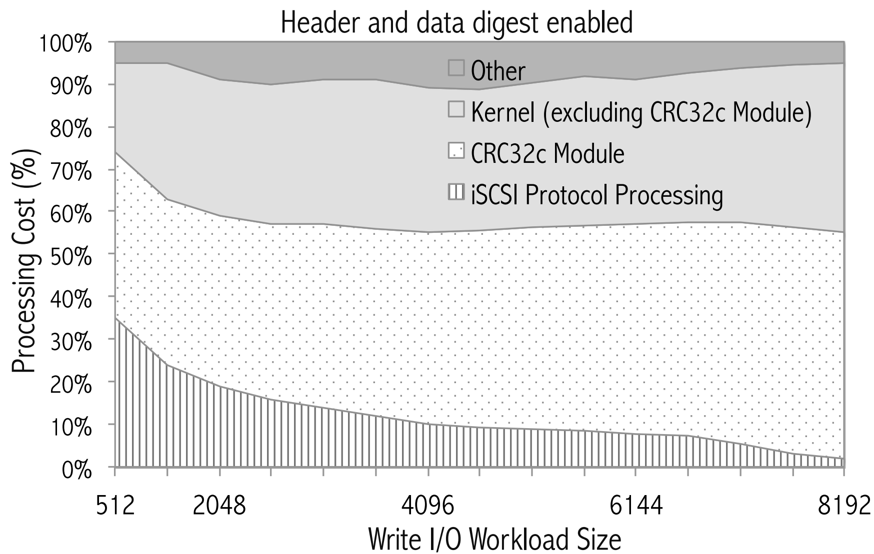

Figure 2.8: The performance profile of processing Data-In PDUs on Intel Core2 CPU 2.40 GHz, when header and data digest are enabled.

2.3.5 Related Work

The most common Type 1 implementations in the research community are open source Open-iSCSI [21] and UNH-iSCSI projects [41]. Examples of Type 2 are ASIC-based *10GbE* TOEs: Chelsio's Terminator 3 chip [11] and NetEffect's NE010 adapter [42]. Both adapters show low CPU utilization and near 10 Gbps performance, especially for larger data sizes. However, very little information is available concerning their architectures. There is some research about TOEs on FPGAs. In [10], Wu et al. introduced a hybrid TOE which processes IP, ARP, and ICMP protocols on an FPGA, and TCP on an embedded processor by using software. In [12], Jang et al. presented the design and implementation of a TOE by means of hardware/software co-processing. Both implementations focus on decreasing CPU utilization by offloading TCP/IP processing to an FPGA. The maximum reported throughput is bellow 1 Gbps: Wu et al. reported 300 Mbps [10], Jang et al. reported 673 and 551 Mbps [12]. However, two companies recently announced FPGA based TOEs which operate at full 10 Gbps [13, 43] line rate. As we argued in the previous section, digest processing occupies significant amount of CPU utilization. Thus, it is not enough to offload TCP/IP processing to a special purpose hardware. This is especially evident for multi-Gbps networks where the impact of computational overhead is so pronounced that the current state of the art processors cannot take advantage of the capacity of the network.

Examples of Type 3 are [14, 44, 45]. In [14], Han-Chiang Chen et al. proposed offloading of TCP/IP and iSCSI to an embedded OS on a PowerPC 405 CPU, which is part of Xilinx FPGA embedded platform. This is the only attempt to offload iSCSI to an FPGA. The implementation does not have any hardware accelerated modules, and it only consists of running unmodified software initiator on the PowerPC 405 CPU. The maximum reported throughput is 86 Mbps without digests, and 31.84 Mbps with digests. The low throughput is attributed to the low frequency of PowerPC 405 CPU (300 MHz). The CPU utilization was 1.5%. In [44], Chung-Ho Chen et al. proposed a hardware accelerator for data transfer iSCSI functions. The accelerator is designed with direct C-to-HDL translation of specific sub-modules of UNH-iSCSI software. The design is evaluated with UMC 0.18μ technology with 100 MHz system clock. The accelerator is able to meet the requirements of 1 Gbps network when the average PDU size is greater than 125 bytes. In [45], the peak throughput performance of Chelsio T110 (10 Gbps iSCSI ASIC-based HBA) is 6.69 Gbps without digests, and 5.9 Gbps with digests. The average CPU utilization is 30% without digests, and 37% with digests.

Chapter 3

High Performance Reconfigurable Computing

High Performance Reconfigurable Computing (HPRC) is relatively new concept in computing. It represents synergistic systems consisting of conventional processors and field-programmable gate arrays (FPGAs). In recent years, HPRC has shown orders of magnitude improvements in performance, power, size and cost over conventional high performance computers in some computationally intensive integer applications, thus it has received widespread attention of High Performance Computing[1] (HPC) community. HPC is a broad term that at its core represents computationally intensive applications that need acceleration.

The importance of HPC systems has been highlighted as a crucial asset for many governments, such as the example of recent report [46] from the European Commission (EC) in 2012. In the report, HPC has been identified as a crucial asset for the European Union's innovation capacity. Thus, EU joined the world-wide race for leadership in HPC systems, which is driven by the need to address societal and scientific grand challenges more effectively. The widespread use of HPC systems in EU covers various areas such as early detection and treatment of diseases (e.g. Alzheimer's), deciphering the human brain [47], forecasting climate evolution or preventing and managing large-scale catastrophes. The importance of HPC systems was first recognized by US and Japan, following China and Russia who declared HPC an area of strategic priority and massively increased their efforts in recent years.

Thus, a new window of opportunities is opening for HPC systems since the field is currently undergoing a major change as the next generation of computing systems (referred to as "exa-scale" systems) are set to emerge by 2020. Experts predict that new systems will be capable of 10^{18} floating point operations per second (flops), which is 1000 times more than the most powerful machines in 2010 (referred to as "peta-flop" or 10^{15} flops). In mid-2007, DARPA/IPTO has sponsored a series of studies intended to understand the future course of mainstream computing technology and determine whether or not it would allow a 1000 times increase in the computational capabilities of computing systems by the 2015 time frame. The resulting two reports [48] describe technology and software challenges in details. The four major technology challenges

[1] Also known as high-end computing or supercomputing.

were identified in [49] and [50] (updated version):

- The **Energy and Power Challenge** is the most pervasive of the four. The key observation is that it may be easier to solve the power problem associated with base computation than it will be to reduce the problem of transporting data from one site to another. Based on today's architectures and concepts it is assumed that an exa-scale system would require 100 MW of power in year 2018. Thus, the use of a common nuclear power-plant would be required for running a supercomputer exclusively, which is unacceptable.
- The **Memory and Storage Challenge** concerns the lack of currently available technology to *retain data at high enough capacities*, *access data at high enough rates* and *fit within an acceptable power envelope*. One of the solutions are additional layers in the memory hierarchy such as higher-level CPU-caches or flash memory as disk caches.
- The **Concurrency and Locality Challenge** is evident through the end of increasing single thread performance and the growth of levels of parallelism that makes it challenging for users to exploit the system. The projections for the data center class systems, in particular, indicate that applications may have to support upwards of a billion separate threads to efficiently use the hardware.
- The **Resiliency Challenge** concerns the increasing number of components in HPC systems and ability of a system to continue operation in the presence of either faults or performance fluctuations.

Traditionally, a HPC system consists of general purpose processors and high-speed interconnect links which connect the processors. Until the early 2000s, HPC relied on a processing power of inexpensive singe-core CPUs, whose performance scaled with frequency in line with Moore's Law.[2] However, the power dissipation escalated to impractical levels with the increase of CPU's frequency. Thus, general purpose CPU (GPCPU) vendors changed the course in the mid-2000s to multicore architectures to meet high-performance demands. This paradigm shift also forced adoption of a parallel programming model in order to take full advantage of underlining performance.

There are two primary methods for the execution of algorithms in conventional computing [52]:

- Hardwired Technology: an Application Specific Integrated Circuit (ASIC) or a group of individual components forming a board-level solution. ASICs are designed to perform only

[2]The *Moore's law [51]* postulates that the level of chip complexity that can be manufactured for minimal cost is an exponential function that *doubles* in a period of time. Surprisingly, Gordon Moore also predicted that by the end of this decade (2020), society will have reached maximum chip-computer power.

a given computation, which makes them very fast and efficient. However, the circuit cannot be altered after fabrication. This forces a redesign and refabrication of the chip if any part of its circuit requires modification. Board-level circuits are also in-flexible to some degree. They frequently require a board redesign and replacement.

- Software-programmed Microprocessors: Processors execute a set of instructions to perform a computation. Instructions are read from the memory, decoded and then executed. Even though very flexible, performance may suffer in clock speed or in work rate, and is far below that of an ASIC. The result is high execution overhead per instruction.

One promising alternative to scaling the number of cores in a system is to apply custom accelerators which has received a widespread attention. In the current TOP500 list, 40% of the top 10 systems are already equipped with accelerator cards [50]. Examples of modern accelerator technologies are general purpose GPUs (GPGPU), Field Programmable Gate Array and Cell processor. In this dissertation, we focus on acceleration methods using FPGAs, since they have ASIC-like performance and power consumption, and GPGPU-like flexibility. Hence, they can accelerate computationally intensive applications and exhibit high performance, while being able to adapt to future changes.

3.1 Accelerator Based Computing with FPGAs

Accelerator based computing aims at moving the computationally demanding tasks of the application to a special purpose processor optimized to perform certain computations very efficiently. The main purpose of an accelerator is to speed-up the execution of computationally demanding tasks of the application. Historically, FPGAs have been restricted to a narrow set of HPC applications due to their relatively high cost. However, advancements in the process technology have enabled vendors to manufacture chips containing millions of transistors. Thus in the past decade, the logic compute performance (clock frequency increase $\times$ logic cell count increase) has improved by 92 times, while the cost decreased for 90% [53]. Users of Accelerator Based Computing range from medical imaging, financial trading, oil and gas expiration, to bioinformatics and computational biology, data warehousing, data security, and others. Such applications usually consist of thousand or millions lines of code, thus it is common that they demand increasing amount of processing capabilities. The need for acceleration is growing exponentially, as does the need to develop more efficient HPC systems.

The process of porting a large application to an accelerator is highly complex and time consuming task. Not every application is suitable to map on an accelerator. A good initial candidate should follow the well-known *90/10 rule*, where a large fraction of the computation (i.e. 90%) occurs in a small fraction of the source code (i.e. 10%). This small region in the program that account for most of the program's execution is called program's *kernel* or *a core*. The region is

Table 3.1: Acceleration benefits on Virtex FPGAs

Algorithm/Application	FPGA	CPU	Speed-up Over Processor (times)
Cryptography: DES	Virtex-6	Intel Quad Core i7 at 2.67 GHZ	101
Cryptography Key Recovery: NTLM	Virtex-6	Intel Quad Core i7 920 at 2.67 GHZ	20
Seismic Imaging: Convolution	Virtex-5	8-core Xeon at 2.66 GHz	240
Proteomics: InsPecT/MS-alignment	Virtex-5	Xeon 2-core at 2.13 GHz	100
Financial options valution: Quadrature methods	Virtex-4	Pentium-4 at 3.6 GHz	33
Dense linear equations: LU factorization	Virtex-5	Xeon Woodcrest at 3 GHz	140
Sparse iterative equations: Conjugate gradient	Virtex-5	Xeon Woodcrest at 3 GHz	82

usually identified by using some profiling tools, which should exhibit some characteristics in order to be accelerated effectively. These include (a) some inherent parallelism or the ability to be transformed as such to benefit from parallel implementations, (b) high computation to communication ratio, (c) the use of bit-level or low-precision arithmetic, (d) simple computation kernel, and (e) uniform, non-divergent computations. The last three properties ensure small computational resource requirements, allowing for a large number of replications of the computation core and hence higher parallelism. Often the kernel has to be re-structured in order to be efficiently executed on an accelerator and afterwards integrated into the rest of the system.

Using FPGAs in the development of HPC systems can potentially deliver enormous performance gains. This has already been confirmed in many applications as shown in Table 3.1. Two main features made them attractive to the HPC community: (a) the ready availability and (b) high-power efficiency of high-density FPGAs.

3.2 Classes of data processing architectures

Digital systems consist of elements for data processing and storage, which are connected in different manners to form a design-specific architectures. There are three classes of data processing architectures: *programmable*, *reconfigurable*, and *application-specific* architectures [54]. In Figure 3.1 we illustrate the trade-off between flexibility, efficiency and performance for various

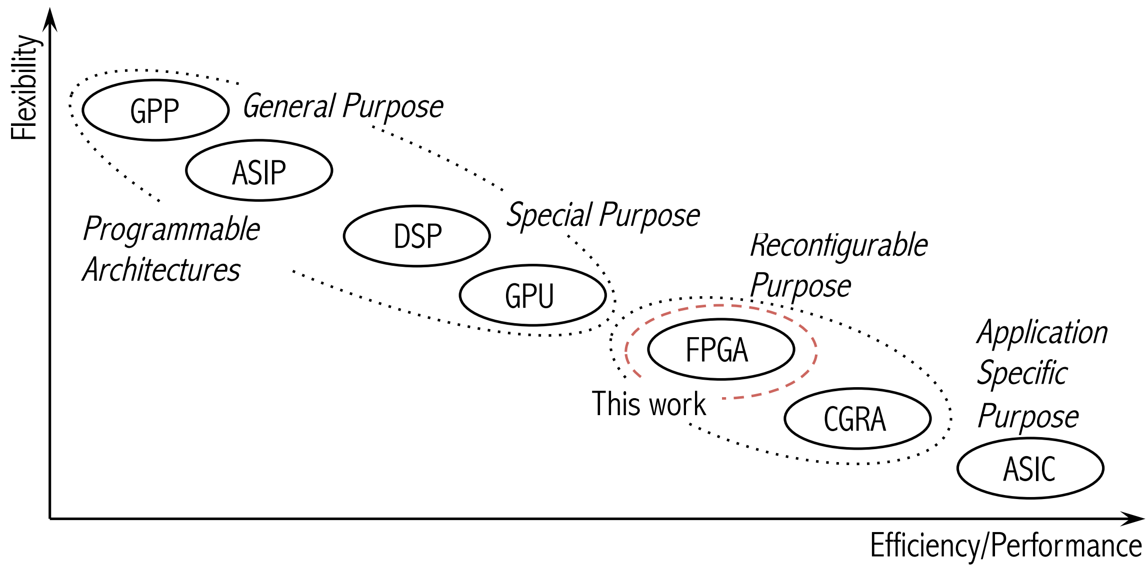

Figure 3.1: An illustration of architecture domains as a function of efficiency/performance and flexibility.

architectures. The term *flexibility* includes programmability and versatility (adaptability), while *efficiency* relates to processing performance and energy efficiency. For example, the general-purpose processor (GPP) provides high level of flexibility, while the Application-specific Integrated Circuit (ASIC) provides high level of efficiency and performance.

3.2.1 Programmable Architectures

An architecture whose programmability is defined by a different set of instructions is called programmable architecture. Examples of such architectures are the general-purpose processor, the application-specific instruction processor (ASIP), and the digital signal processor (DSP). They can be divided into three groups: *general-purpose* processors, *configurable instruction-set* processors, and *special-purpose* processors.

General-Purpose Processors have a general instruction set designed to serve a variety of applications. The instruction set is not optimized or tailored for any specific application domain. The instructions are processed in sequential order one after the other. The advantage of processing instructions in sequential order is high instruction locality, but one obvious bottleneck is high instruction dependency which prevents execution of instructions before they are needed (or while waiting for some resources to become available).

All general-purpose processors rely on the *von Neumann* instruction fetch-and-execute model. This model enables high level of flexibility, but it comes with a drawback of high energy consumption and overhead of fetching, decoding and executing a stream of instructions. The underlying

architecture cannot disable components that are not used at a time, hence it has high energy consumption. Another concern is that the clock speed of processors has grown much faster than the speed of memory. Hence, a processor is in idle state while waiting for instructions to be fetched from the memory. This gap between memory access speed and processor speed is known as the *von Neumann bottleneck*. Some techniques have been introduced to relieve this bottleneck, such as using caches or separating instruction and data memories. Software programmers always relied on a higher clock speeds for running their application faster. Increasing the processing performance usually requires a higher clock frequency f_{clk}, which consequently increases *the power consumption* and therefore also *the heat dissipation*. The supply voldage V_{DD} has a quadratic effect on the dynamic power as

$$P_{dyn} = \alpha C_L V_{DD}^2 f_{clk}, \tag{3.1}$$

where α is the switching activity and C_L is the load capacitance. Lowering the supply voltage also effects the propagation time t_p, which means that the system becomes slower. Hence, the processing power cannot depend only on increasing the system clock frequency.

Recently, a new *multicore* paradigm is introduced to address the limitation of pure sequential processing. Multiple processing units or cores are utilized to processes independent applications or parts of the same application. Theoretically, processing speed may potentially increase with a factor equal to number of processing units. However, the speed-up of a single thread is limited by the instruction level parallelism (ILP) in the program code. This means that the multicore model will not be able to provide any noticeable speed-up for an application with limited ILP. The speed-up S from using N parallel processing units is calculated using *Amdahl's law*:

$$S = \frac{1}{(1-p) + \frac{p}{N}} \tag{3.2}$$

where p is the fraction of the sequential program that can parallelized. In the case when 50% of a program can be executed in parallel, the speedup is limited to a modest factor of 2, no matter how many processors are used.

Another technique for exploiting processing power of multiple processing units is called thread level parallelism (TLP). The programmer divides application into multiple parallel threads of execution, which are executed on different processing units.

Configurable Instruction-Set Processors have ability to extend original instruction set for a given application. The process of extending the instruction set begins with identifying frequently used computational kernels which constitute the main part of the execution time. The *kernels* are groups of basic instructions that often occur in conjunction, hence it is possible to merge them

into a specialized instruction [55]. In addition to extending the processor with new instruction, the compiler tools have to be modified to support the extended instruction set.

Special-Purpose Processors contain additional features and usually include more than one computation unit and register bank in addition to common elements of a general-purpose microprocessor. Examples of a special-purpose processor are the digital signal processor and the graphic processing units (GPUs). The DSP is a microprocessor specialized for digital signal processing. In a DSP, a set of frequently used operations can be executed in a single clock cycle using a dedicated multiply-accumulate (MAC*) hardware. Some of the hardware features of a DSP include: multiple addressing modes such as modulo, ring-buffer and bit-reversed addressing, a memory architecture designed for streaming data, saturation logic for integer arithmetic, etc. The GPU resembles the stream processor with a more narrow application domain. Traditionally, GPUs provided none or limited programmability. However, in recent years the trend is changing toward general-purpose GPUs (GPGPU), which widens the application domain. Applications which can benefit from this concept exhibit real-time requirements or have long execution time.

3.2.2 Reconfigurable Architectures

Reconfigurable or *adaptive* architecture indicates that the logic functionality and interconnect of a computing system or device can be customized to suit a specific application through post-fabrication, user-defined programming [56]. Microprocessors change their functionality through *instructions*, while reconfigurable architectures change their functionality through *configuration bits*. Thus, a single reconfigurable architecture can be re-used to implement many potential applications, rather than requiring a separate custom circuit for each. This feature makes programmability of reconfigurable architectures lower than that of a GPP though. Applications are accelerated by allocating a set of required processing, memory and routing resources.

Recofigurable devices contain an array of computational elements whose functionality is determined through multiple programmable configuration bits. These elements, known as *logic blocks*, are connected using a set of routing resources that are also programmable. The size of the reconfigurable elements is referred to as the *granularity* of the device. In terms of granularity, there are two type of reconfigurable architectures: *fine-grained* and *coarse-grained* architectures. A comparison between fine- and coarse-grained architectures is provided in Table 3.2.

The **fine-grained reconfigurable architecture** allows bit-level manipulation by using small look-up tables (LUT). An example of a fine-grained reconfigurable architecture is illustrated in Figure 2.2a). A LUT has a limited number of inputs (typically less than six) which generate a single boolean output. Each logic block usually contains one or more flip-flops for fine-grained

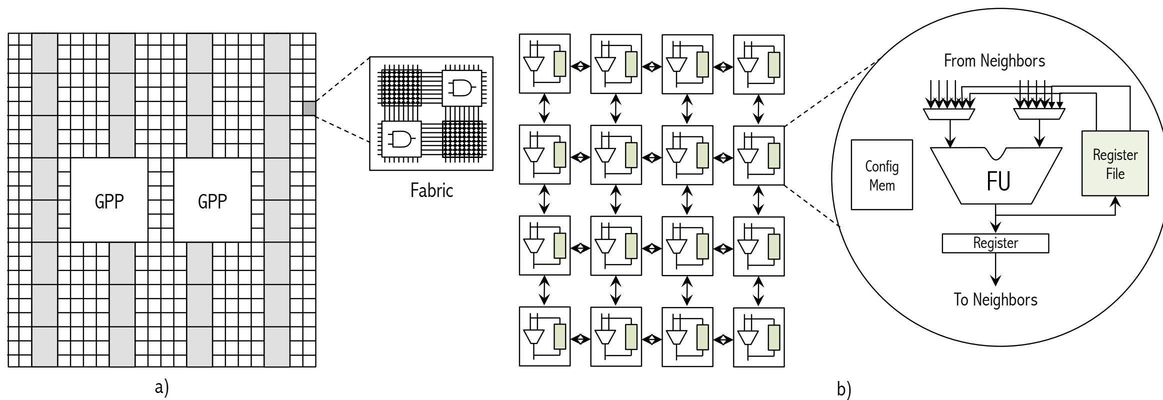

Figure 3.2: An example of a) fine-grained reconfigurable architecture with embedded memory (gray) and GPP macro-blocks. The FPGA fabric with logic blocks and switch boxes is magnified. On b) is an example of coarse-grained reconfigurable architecture with an array of processing elements.

storage. Today's reconfigurable devices typically have tens of thousands of lookup tables containing millions of gates of logic. The fine-grained architectures are very versatile and can be used to map any type of an algorithm. Since the computational requirements are either not known in advance or vary considerably among applications, the reconfigurable architecture has to be fine-grained in order to achieve high degree of flexibility. The fine-grained devices include programmable array logic (PAL), complex programmable logic devices (CPLD), and field-programmable gate arrays (FPGA). Reconfigurable processors have been widely associated with fine-grained architecture, which is used as a reference for an FPGA. However, this degree of flexibility may result in significant overheads of area, delay and power consumption. Examples of fine-grained architectures are discussed in great details in [57].

The **coarse-grained reconfigurable architecture**, also known as *CGRA*, consists of complex functional units (FU) ranging from ALUs and multipliers to full-scale processors. An example of a Coarse-Grained Reconfigurable Architecture is illustrated in Figure 3.2b). They can perform word-size operations such as addition, subtraction, and multiplication. Register files hold temporary values which are accessible only by a subset of FUs. Logic blocks are optimized for large computations which will perform operations much more quickly than a set of smaller cells connected to form the same type of structure. They will also consume less chip area. However, they are not efficient when bit-size operations are performed, since there is unnecessary area and speed overhead, as all of the bits in the full word size are computed. Examples of the coarse-grained architectures are RaPiD [58], Chameleon [59], Pleiades [60], MorphoSys [61], PACT's extreme processor platform [62], Montium [63], etc. Many DSP applications benefit from modular arithmetic operations, which have led to faster development of the coarse-grained architectures. The CGRAs have short reconfiguration times, low delay characteristics, and low power

Table 3.2: A comparison between fine- and coarse-grained architectures.

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

Table 3.3: A comparison summary of selected architecture domains.

Technology	**Performance/ Cost**	**Time to market**	**Time to change code functionality**	**Power Consumption**
ASIC	Very High	Very Long	Impossible	Low
FPGA	Medium/High	Long	Long	Low/Medium
ASIP	Medium	Medium	Medium	Medium/High
GPP	Low/Medium	Very Short	Very Short	High

consumption as they are constructed from standard cell implementations. Thus, gate-level reconfigurability is sacrificed, but the result is a large increase in hardware efficiency.

3.2.2.1 Comparison summary

An example of the 256-tap[3] *Finite Impulse Response* (FIR) implementation is shown in Figure 3.3: a) on a conventional DSP device with von Neumann architecture and b) on an FPGA. FPGAs can instantiate as many MAC* units as possible, while GPPs/ASIPs execute instructions in the sequential manner. Thus, FPGAs can execute all 256 MAC* operations in one clock cycle, while GPPs/ASIPs have to process 256 loops. This allows FPGAs to achieve much higher performance in selected algorithms than GPPs/ASIPs. FPGAs run on lower frequency (e.g. 400 MHz) than GPPs/ASIPs (e.g. 2 - 3 GHz), thus not every algorithm can be accelerated to a noticeable degree. The advantage of FPGAs over ASICs is that they have additional reprogrammability, while having ASIC-like performance and power consumption. They also have

[3]The number of taps is an indication of memory required to implement the filter, the number of calculation required and the amount of filtering.

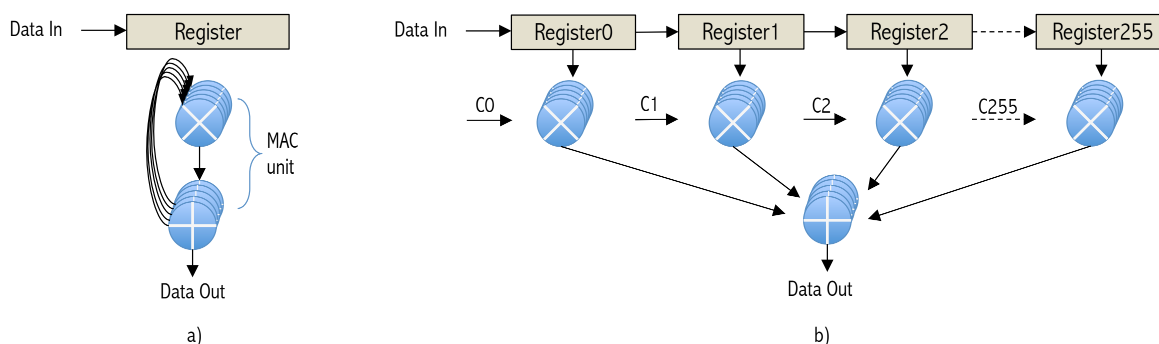

Figure 3.3: An implementation of 256-tap FIR filter in a) GPPs/ASIPs and b) FPGAs.

shorter development cycle and lower Non-Recurring Engineering (NRE) costs [64].

One drawback of FPGAs is that they have long time to change code functionality. This is due to a different style of programming than conventional software programming, which executes instructions sequentially. Even though there are many new environments to aid with hardware programming, FPGA programming is still reserved to specialists.

3.2.3 Application-Specific Architectures

An application-specific architecture presents a fully customized hardware implementation of a particular application or set of applications that share many common characteristics. Hence, such architecture contains only capabilities necessary to execute its targeted workloads. They exhibit the best performance and power consumption figures possible. They are non-reprogrammable which limits their application to high volume, relatively low cost and low power applications.

3.3 Overview of Field Programmable Gate Arrays

FPGAs have been invented in mid-1980 by one of Xilinx founders, Ross Freeman. FPGAs are commodity integrated circuits (IC) whose logic can be determined, or programmed, in the field. This is opposite to Application Specific Integrated Circuits (ASICs), where logic is fixed at fabrication time. FPGAs are less dense and slower than ASICs, but their main advantage is flexibility. They were first introduced as glue logic which made them popular in embedded systems.

The original concept of *reconfigurable* computing is credited to Gerald Estrin. In 1960's, he described a hybrid computer structure consisting of an array of reconfigurable processing elements. His aim was to attempt to combine the flexibility of software with the speed of hardware, with a main processor controlling the behavior of the reconfigurable hardware. However, this concept could not be realized at the time, because technology didn't exist. By the late 1980s silicon technology had advanced to a point that allowed complex designs to be implemented on a single chip âĂŞ large and very large-scale integration which paved the way for today's system-on-a-chip designs and FPGA programmable logic devices.

The first commercial reconfigurable computer, the Algotronix CHS2x4 [65], has been released in the beginning of 1990s. The Algotronix CHS2x4 had an array of CAL1024 processors in ISA format with up to 8 FPGAs, each with 1024 programmable cells. From that point many reconfigurable architectures became available, such as Garp from UC Berkley [66], PACT-XPP [62], FIPSOC [67], etc. Today, there are many FPGAs vendors, which beside Xilinx [68] include Altera [69], Lattice Semiconductor [70] and Atmel [71].

There are three similar circuit families to FPGAs: a) *Programmable Array Logic* (PAL), *Programmable Logic Array* (PLA) and *Programmable Logic Device* (PLD). In principal, they differ in two ways. First, FPGAs are *more flexible* in the types of functions that can be implemented. Second, FPGAs have significantly *more embedded components*, such as memory blocks, built-in multipliers and processor cores. FPGAs are more suitable for wide variety of applications, and this is why they are more popular.

3.3.1 FPGA Programming Technologies

There are a number of programming technologies that have been used for reconfigurable architectures. The well known technologies include static memory cells [72], flash [73] and anti-fuse [74]. Static memory programming technology has become the dominant approach for FPGAs because of its re-programmability and the use of standard CMOS process technology. It is expected that this technology will continue to dominate the other two programming technologies. In this section, an overview of three commonly used programming technologies is given.

Static memory cells are the basic cells used for SRAM-based FPGAs (Figure 3.4.a)). The

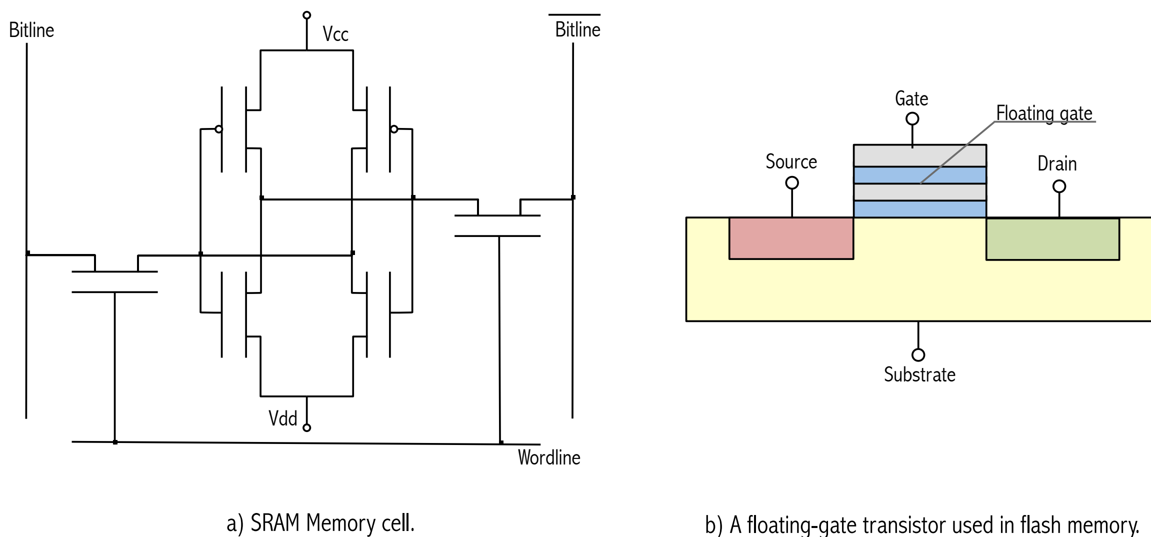

Figure 3.4: Two basic elements used in FPGA implementation technologies

cells are divided throughout the FPGA to provide configurability. They are used to program the routing interconnect of FPGAs and CLBs[4] that are used to implement logic functions. Since SRAM is volatile and can't keep data without power source, such FPGAs must be programmed (configured) upon start. SRAM-based FPGAs include most chips of Xilinx Virtex and Spartan families and Altera Stratix and Cyclone. The drawbacks associated with SRAM-based programming technology are (a) cost effectiveness in terms of area (every SRAM cell requires 6 transistors), (b) SRAM cells are volatile in nature and (c) they require external devices to permanently store the configuration data which adds the cost and area overhead.

There are two basic modes of programming SRAM-based FPGAs:

- Master mode, when FPGA reads configuration data from an external source, such as an external Flash memory chip.
- Slave mode, when FPGA is configured by an external master device, such as a processor. This can be usually done via a dedicated configuration interface or via a boundary-scan (JTAG) interface.

Flash-based programming technology uses flash as a primary resource for configuration storage, and does not require SRAM. It is not volatile in nature and is more area efficient than SRAM-based programming technology. However, it cannot be reprogrammed/reconfigured an infinite number of times as SRAM-based programming technology. This technology has an advantage of being less power consumptive and it is more tolerant to radiation effects. Using flash-based FPGAs can be a solution to prevent unauthorized bitstream copying. Examples of

[4]The CLB is the basic logic unit in an FPGA. It is discussed in details in section 3.3.3.

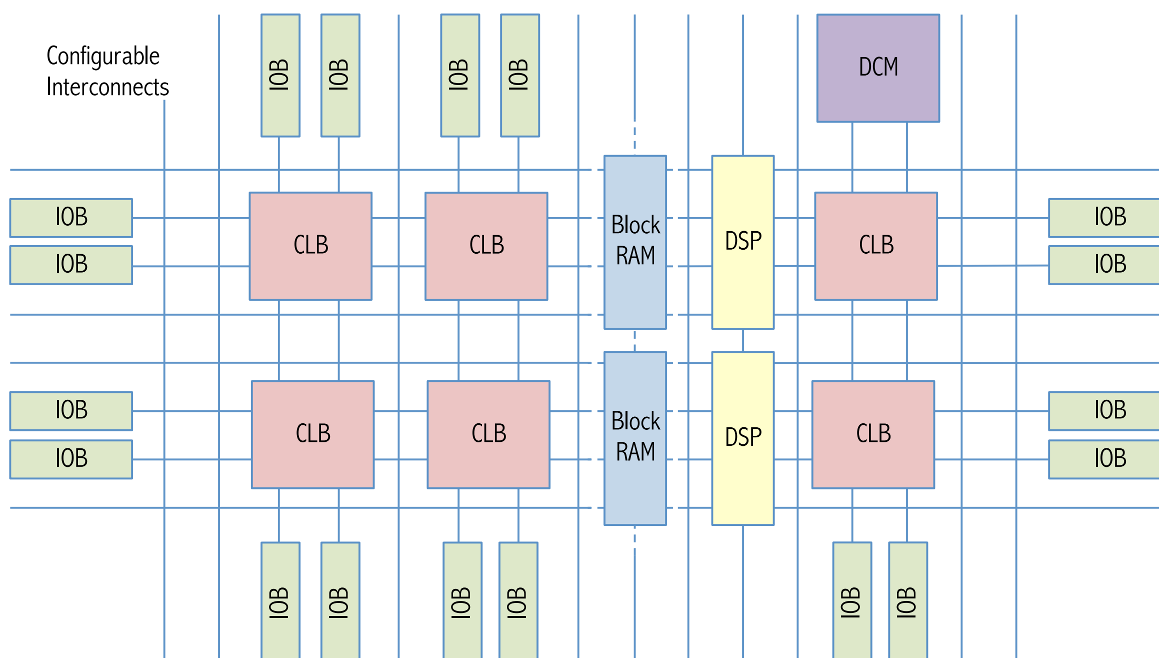

Figure 3.5: FPGA Block Structure

flash-based FPGA families include *Igloo* and *ProASIC3* that are manufactured by Actel. Figure 3.4.b) illustrates a floating-gate transistor used in flash memory.

Antifuse-based programming technology is different from the previous ones in that it can be programmed only once. The primary advantage of anti-fuse programming technology is its low area usage. There are however significant disadvantages associated with this programming technology. It does not make use of standard CMOS process and can not be reprogrammed. An example of antifuse-based device families include *Axcelerator* by Actel.

3.3.2 FPGA Architecture

Field Programmable Gate Arrays (FPGAs) are pre-fabricated silicon devices that can be electrically programmed in the field to the desired application or functionality requirements. They are based around a matrix of Configurable Logic Blocks (CLBs) connected through programmable interconnects (Figure 3.5). FPGAs incorporate hard (ASIC type) blocks of commonly used functionality such as RAM, clock management, and DSP. The CLB is the basic logic unit in an FPGA. Exact numbers and features vary from device to device, but every CLB consists of a configurable switch matrix with 4 or 6 inputs, some selection circuitry (MUX, etc), and flip-flops. The switch matrix is highly flexible and can be configured to handle combinatorial logic, shift registers or RAM. The programmable routing interconnect of FPGAs comprises of almost 90% of total area of FPGAs.

3.3.3 Configurable Logic Block

A configurable logic block (CLB) is a basic component of an FPGA that provides the basic logic and storage functionality for a target application design. The CLBs are organized in a grid array to implement different type of logic designs. Every new generation of FPGAs brings a range of technical advances ranging from power reduction and device density to small changes such as new placement of flipflops in the CLB. These small changes can be exploited by an experienced FPGA engineer to the fullest.

The CLB consists of a number of logic cells (LC) which include a lookup table (LUT), a flip flop, and connection to adjacent cells. The LUT uses combinatorial logic to implement a 4-, 5- or 6-input expression (AND, OR, NAND, addition, etc.) Multiple logic cells are grouped together to create a single unit, called a slice. In Figure 3.6 and Table 3.4 we show advancement in configuration of a CLB and a slice, and resource comparison of latest Xilinx FPGA families, respectively. The number of logic cells in a slice changes as the architecture of Virtex FPGAs changes. For example, the number of flip-flops has been doubled when compared with Virtex-4 FPGA slices.

Vendors such as Xilinx and Altera use LUT-based CLBs, which provides a good trade-off between too fine-grained and too coarse-grained logic blocks. Single CLB consists of a single basic logic element (BLE), or a cluster of locally interconnected BLEs. A BLE consists of a LUT and a flip-flop. A LUT can have k inputs and it contains $2k$ configuration bits. Single LUT can be used to implement any k-input boolean function. Figure 3.7 illustrates 4-input LUT and a D-type flip-flop. The output of LUT is connected to an optional flip-flop.

In Figure 3.6. we illustrate basic differences between two widely known Xilinx families *Spartan* (3 and 6) and *Virtex* (4, 5 and 6). **The Spartan-3 CLB** consists of four slices grouped in pairs. Each pair is organized as a column with an independent carry chain. Each slice is equivalent and contains two LUTs, two storage elements, wide-function multiplexers, carry logic and arithmetic gates. The carry chain supports fast and efficient implementations of mathematical operations. In Figure 3.7. we illustrate an example of basic Xilinx Spartan-3 CLB. Slices X0Y0 and X0Y1 make up the column-pair on the left, and slices X1Y0 and X1Y1 make up the column-pair on the right. Left-hand pair or *SLICEM* is labeled with an even "X" number, and right-hand pair or *SLICEL* designates the pair with an odd "X" number (e.g. X1). Switch Matrices are programmable interconnects of wire segments.

The Spartan-6 CLB consists of two slices, arranged side-by-side as part of two vertical columns. There are three types of CLB slices in the Spartan-6 architecture: SLICEM, SLICEL, and SLICEX. Each slice contains four LUTs, eight flip-flops, and miscellaneous logic. The LUTs are for general-purpose combinatorial and sequential logic support. 25% of Spartan-6 FPGA slices are SLICEMs. Each of the four SLICEM LUTs can be configured as either a 6-input LUT with one output, or as dual 5-input LUTs with identical 5-bit addresses and two independent

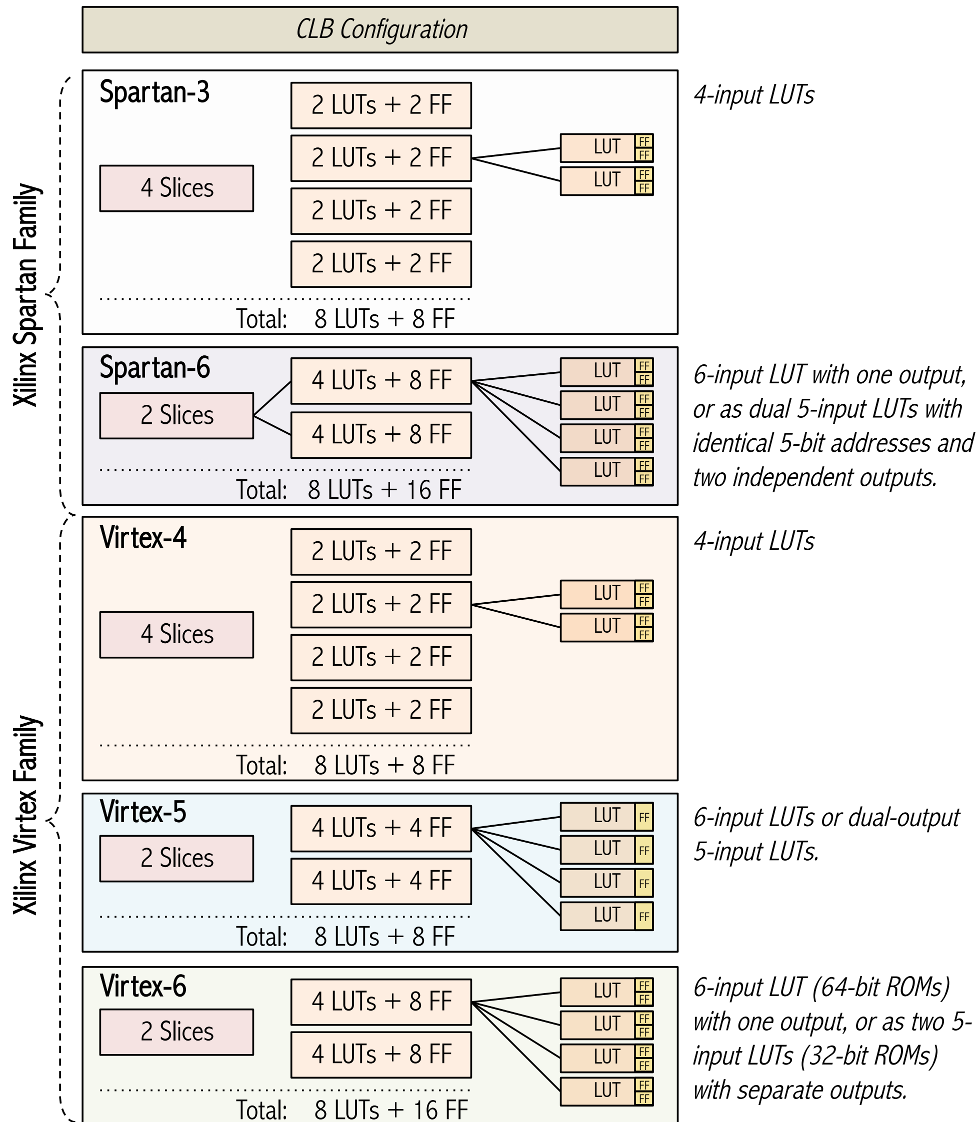

Figure 3.6: Advancement in configuration of a CLB and a slice in latest Xilinx families of FPGAs.

outputs. The SLICEL contain all the features of the SLICEM except the memory/shift register function and they occupy 20% of Spartan-6 FPGA slices. The SLICEX have the same structure as SLICELs except the arithmetic carry option and the wide multiplexers. They occupy 50% of Spartan-6 FPGA slices.

The Virtex-4 CLB has four slices (maximum of 64 bits). Each slice is equivalent and contains two configurable 4-input LUTs, two storage elements (edge-triggered D-type flip-flops or level sensitive latches), arithmetic logic gates, large multiplexers and fast carry look-ahead chain. Each CLB has internal fast interconnect and connects to a switch matrix to access general routing resources.

Virtex-5 FPGA CLB has two slices which are organized differently from previous generations. The slices are based on real 6-input look-up table technology. Each slice has four 6-input

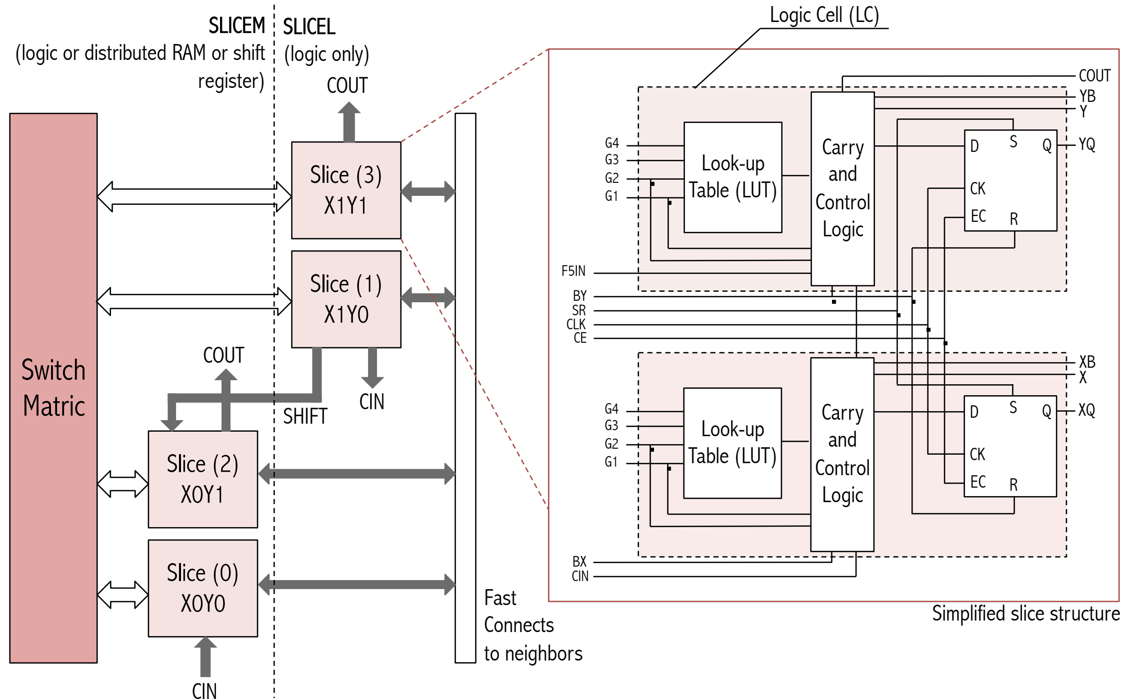

Figure 3.7: An example of basic Xilinx Spartan-3 Configurable Logic Block.

LUTs or dual-output 5-input LUTs, four storage elements (edge-triggered D-type flip-flops or level sensitive latches), arithmetic logic gates, large multiplexers and fast carry look-ahead chain.

Virtex-6 FPGA CLB has two slices, each having four LUTs, eight flip-flops, multiplexers and arithmetic carry logic. The LUTs can be configured as either one 6-input LUT (64-bit ROMs) with one output, or as two 5-input LUTs (32-bit ROMs) with separate outputs but common addresses or logic inputs.

Table 3.4: Resource comparison between five Xilinx generations of FPGAs.

Resources	Spartan-3 [75]	Spartan-6 [76]	Virtex-4 [77]	Virtex-5 [78]	Virtex-6 [79]
Logic Cells	1.6k – 73k	4k – 144k	12k – 200k	19k – 324k	72k – 741k
Slices	0.8k – 33k	600 –23k	5k – 87k	3k – 50k	11k – 115k
CLBs	192 – 8320	300 – 11519	1368 – 22272	1560 – 25920	5820 – 59280
Block RAM (kb)	72 – 1872	216 – 4824	648 – 9936	936 – 18576	5616 – 38304
DSP Slices	–	8 – 180	32 – 512	24 – 1056	228 – 2016
Serial Transceivers	–	0 – 8	0 – 24	0 – 48	0 – 36
SelectIO	124 – 300	132 – 576	320 – 960	172 – 1200	360 – 1200

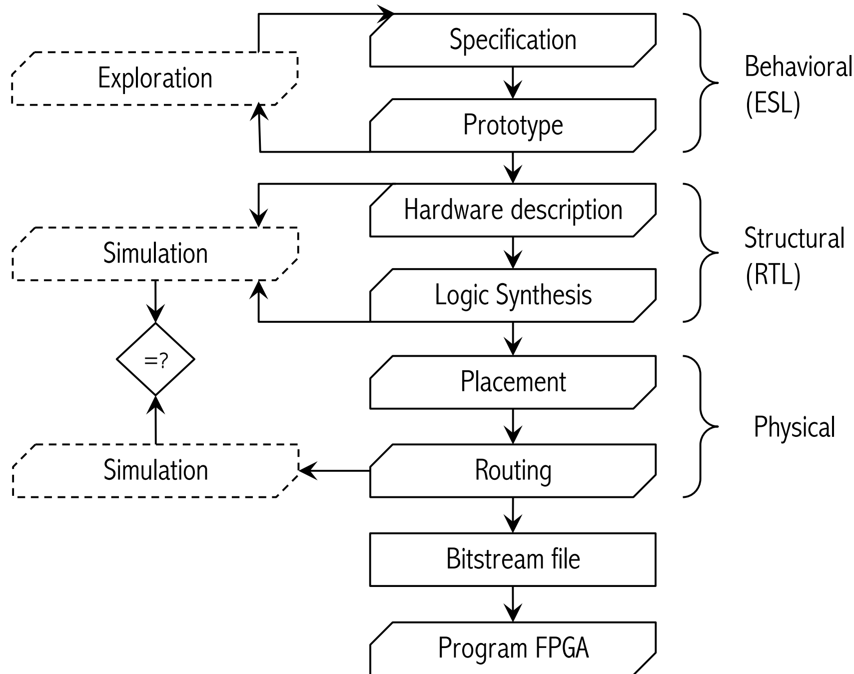

Figure 3.8: Design flow of system hardware development.

3.4 Hardware Design Flow

Hardware developers follow a design flow to systematically handle the design steps from an abstract specification (a model) to a functional hardware platform [54]. The most common flow used in the design of FPGAs involves the steps shown in Figure 3.8. Some of the steps imply use of tools supplied by the different FPGA vendors, however these tools do not help designer in initial steps of specification and prototyping.

The first step is to make a *specification* or a model of a system, which implies block diagrams and hierarchical schematics, design constraints and design requirements. *Prototyping* is required to verify the specification and allow hardware and software co-design. The prototype is usually programmed using high-level programming languages such as C, C++, Matlab or a high-level hardware description language such as System C. The specification is often modified after prototyping phase, which is called *Exploration* phase. *Hardware description* consists of transforming the design ideas into some Hardware Description Language (HDL). This can be accomplished automatically or manually. The code which is used to make a prototype can be automatically translated into HDL with, for example, C-to-VHDL compiler. The other approach is to manually describe a system at the register-transfer level (RTL). This approach might be time consuming, but often results in optimized solution. The two most popular HDLs are VHDL (Very High Speed Integrated Circuit HDL) and Verilog. The HDLs differ from conventional software programming language since they additionally support concurrent execution of statements in the

code.

Logic Synthesis is used to compile the hardware description into gate level. It generates a *netlist* which uses vendor specific primitives to implement the logic behavior specified in the HDL. The gate primitives are provided in form of a cell library which describes the functional and physical properties of each cell. Vendor specific synthesis tools provide options such as logic optimization, register load balancing and other techniques to enhance timing performance. After this stage it is possible to verify *netlist* against RTL description.

Placement takes the synthesized *netlist* and chooses a place for each of the primitives inside the chip. Then, *Routing* interconnects all primitives together satisfying the timing constraints, such as the frequency of the system clock. The resulting choices for the configuration of each programmable element in the FPGA chip is stored in a *bitstream* file which is later used to program an FPGA.

The design flow is an iterative process, where changes to one step may require the designer to repeat previous step in the flow. After some phases, it is possible to make a simulation and check functionality of a circuit.

Chapter 4

High-Speed Fully-Adaptable CRC Accelerators

Cyclic Redundancy Check is used in numerous applications as a method to detect errors in sequences of bits. Since it has good detecting properties it is used in data transmission and in data storage. However, it is often substituted with less efficient error detection schemes, since it is a computationally intensive process that adversely affects performance. It plays an important role in implementation of iSCSI protocol in Storage Area Networks (SANs) for detecting errors which occur between protocol transitions. In such a case, it is common practice to disable iSCSI digests in order to decrease latency, thus the network must rely on other mechanisms to detect corrupted data, such as TCP and Ethernet error detection mechanisms. Unfortunately, these mechanisms cannot detect errors which occur between upper layer protocol transitions. This can result in undetected data corruption, thus it can lead to various problems such as failed integrity check of a database. Therefore, it is desired to (1) reduce the computational burden, (2) make architecture generic enough to support a variety of applications, (3) make architecture scalable so it can process arbitrary number of data input (4) achieve significant improvements in throughput and (5) make it area efficient.

There are a couple of methods to implement CRC, and they are introduced in section 2.2. The simplest method imitates the standard hand calculations of polynomial division, and it is predominately used in hardware implementations. The simplest implementation of this method is when an input message is fed serially into a circuit, but there are other methods which introduce some level of parallelism. In related work in section 2.2.3, we introduced the most recent methods highlighting achievable throughput, the ability to adapt to different applications and the amount of processed bits at a time. It is important to note that the goal of traditional methods for designing CRC accelerators is acceleration of a specific application. In such accelerators, the resulting CRC value is determined by the CRC standard deployed by an application, which is usually fixed at the design time. We call these accelerators *non-adaptable*. Their usability is limited to only one CRC standard, thus they can be used by limited numer of applications. On the other hand, *adaptable* CRC accelerator has ability to generate CRC for a variety of CRC standards and thus support a wide range of applications. They eliminate the need for many

non-adaptable CRC accelerators. The fully-adaptable CRC accelerator has ability to process arbitrary number of input data and generates CRC for all currently defined CRC standards during run-time. In related work, we found only one method, based on parallel implementation of LFSR, which exhibits throughput of up to 25 Gbps. However, it has no mechanism to adapt to different standards. Other methods exhibit maximum of 6 Gbps and have very little to no adaptability.

We propose new CRC architecture based on FPGAs, which can potentially achieve high throughput, high adaptability in terms of support for CRC standards and process arbitrary amount of data at a time at minimum area utilization. FPGAs are chosen because of their potential to implement many functional units in parallel, thus many bits at a time might be processed, which in the end affects throughput. In this section, we describe design and implementation of non-adaptable and fully-adaptable CRC accelerators based on a table-based algorithm, which is suited for the flexible implementation. Although the table-based algorithm has been used in software, it has never been implemented in hardware as its performance is believed to be lower than traditional implementation. We prove that this approach can be successfully implemented on an FPGA and achieve significant performance improvements over related work.

Our contributions are as follows:

1. A design of non-adaptable CRC accelerator with sufficient performance and reasonable resource utilization using a table-based algorithm is proposed.

2. Based on the above design, a *fully-adaptable* CRC accelerator is proposed by integrating algorithm for generating CRCs and algorithm for generating contents of tables. Resulting architecture generates CRC for any known CRC standard during run-time. It achieves throughput of up to 418.8 Gbps, when $M = 1024$ (M is number of bits of input data).

3. We modify table generation algorithm in order to decrease its space complexity from $O(nm)$ to $O(n)$, where n is a number of tables, and m is a number of bits in a slice.

4. Design of our architectures guarantees scalability/expandability by processing arbitrary number of input data M at minimal area cost. In order to show efficiency of our architecture in terms of area utilization and throughput, we design five implementations, where $M \in 64, 128, 256, 512, 1024$.

4.1 Design of a CRC Accelerator

In order to support variable number of CRC Standards, we propose the structure consisting of a non-adaptable CRC accelerator core and Table Generation Module (TGM) as shown in Fig 4.1. From the viewpoint of an application, the total system is treated as a CRC IP* Core. The accelerator core consists of CRC Generation Module (CGM) and tables with pre-computed

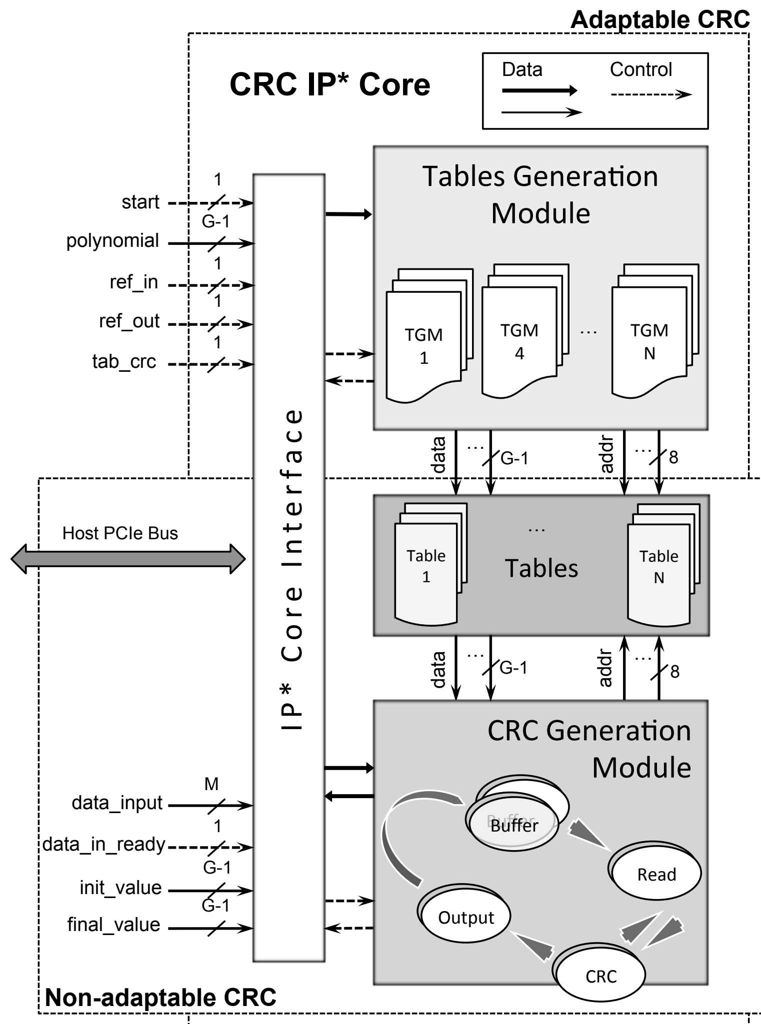

Figure 4.1: Design overview of the non-adaptable (accelerator core) and fully-adaptable CRC accelerators.

values. The CGM calculates CRC for provided data input every clock cycle by accessing pre-computed remainders stored in tables in parallel. The TGM generates pre-computed remainders for a specified generator polynomial G, and stores them into tables. The IP* Core Interface is responsible for managing generation of remainders, accessing and storing them from/into tables, and managing input/output buffers. Two main modules are not executed in parallel in order to maintain data consistency of tables. The fully-adaptable accelerator accepts variable width generator polynomial G up to 65 bits, while different input stream widths M require different FPGA implementation for each. The most significant bit in a polynomial G is always considered to be 1, thus the input polynomial is always $G-1$. The number of tables N depends on an input data M and it is equal to $M/8$.

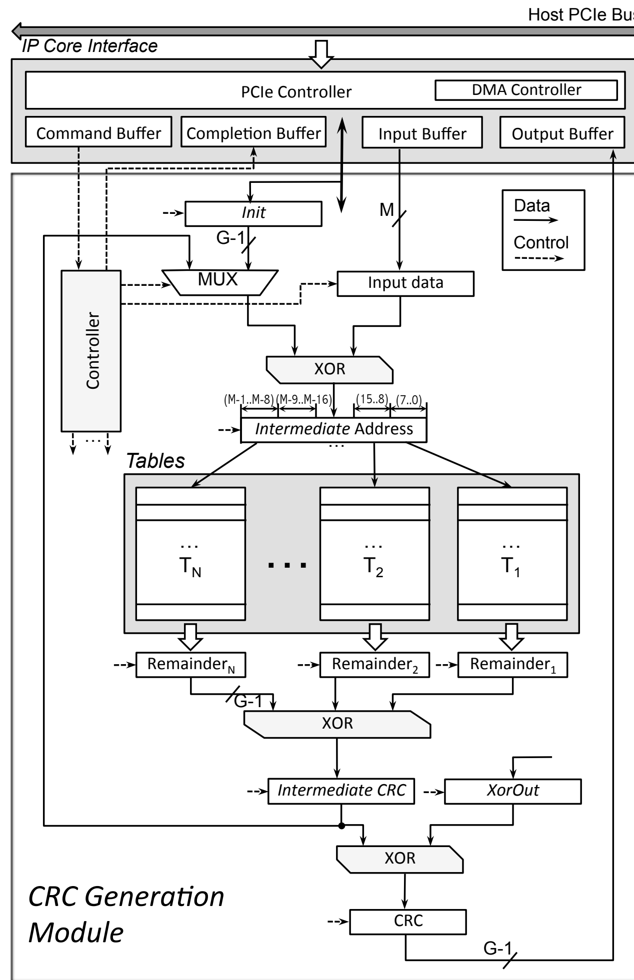

Figure 4.2: The generic architecture of CRC Generator Module - CGM and accompanying IP* Core Interface.

4.1.1 CRC Generation Module

Fig 4.2 shows the architecture of CRC Generator Module and IP* Core Interface. In the first iteration, *Intermediate Address* is formed by XORing input data with initial value (*Init*), while in other iterations *Intermediate CRC* is used instead of *Init*. *Intermediate Address* is then sliced into N eight bit slices, which are used as addresses to N Tables. The number N dependents on the number of input data M and it is equal to $M/8$. Then, Tables provide N Remainders, which are XORed to form *Intermediate CRC*. When the end of the Input Buffer is reached, *Intermediate CRC* is XORed with the final value (*XorOut*) and then stored in the Output Buffer.

For fully-adaptable implementation, the CGM must support variable number of CRC Standards. There are four parameters defined by a CRC Standard which affect execution of CGM: *Width*, *Poly*, *Init* and *XorOut*. In order to make this architecture fully-adaptable, we fix data-path's width to maximum number of bits in *Width* in order to support variable width generator polyno-

mial. CRCs with higher degree polynomials are still not used in practice, thus there is no need to implement them yet. The CGM indirectly depends on the *Poly* through TGM which generates reminders in tables (which will be discussed in the next section). In the fully-adaptable architecture, the IP* Core Interface permits usage of the CGM only until tables are ready, thus the CGM doesn't have to check if tables are ready or not.

The CGM is implemented in a pipeline with three stages as following. In the first stage, M bits are read from the Input Buffer and stored into a temporary register. This value is then XORed with either Init or previous *Intermediate CRC*, depending on the iteration. In the second stage, *Intermediate Address* is sliced into N slices and used to access N remainders from N tables. These remainders are then XORed to form *Intermediate CRC*. In the third stage, *Intermediate CRC* is XORed with XorOut and stored into the Output Buffer. The latency of CGM module is three cycles, but CRC is generated every cycle (the throughput is one cycle).

Fully-adaptable CGM's architecture has the following aspects to provide adaptability:

- data-path is extended to maximum number of bits in $Width$,
- structure of tables $T_N..T_1$ is extended to maximum $2^{Slice} \times Width$, and tables are *read-write*, instead of previously read-only,
- values in tables *and* resulting CRC are aligned to right, while in the case when $G < 65$ bit positions $64 - (G - 1)$ are filled with zeros, since XORing a value with zeros will result with the value itself,
- Init and XorOut values are programmable, instead of hard-coded in the circuit,
- controller is modified in order to support integration with TGM and IP* Core Interface.

4.1.2 Tables Generation Module

For all table-based CRC algorithms tables are generated by a separate algorithm. The results are used to form a data-set in the program for generating CRCs. In software implementations, tables are generated sequentially and independently from each other. In our design, we integrate these two algorithms and in this section we discuss the architecture of Tables Generation Module - TGM. This architecture enables support for a variable number of CRC Standards, as well as processing arbitrary number of bits at a time. We provide a pseudo-code for generating remainders in Fig. 4.3 based on the description in [27, 80].

We present a general block diagram of our single TGM in Fig. 4.4a). Every clock cycle, the counter generates a message ranging from 0 to 2^{Slice} (line 2 in Fig. 4.3). This message then passes through the input reflection unit (line 4) or directly to the remainder generator (line 9 - 15), depending on a CRC standard. The input reflection unit reflects message bits by swapping

```
1:  for Offset = 1 → N do
2:    for i = 0 → 2^Slice do
3:      if RefIn = true then
4:        input ← reflect(i, Slice)
5:      else
6:        input ← i
7:      end if
8:      r ← input << (Width − Slice)
9:      for j = 0 → Offset ∗ Slice do
10:       if r ∧ input then
11:         r ← (r << 1) ⊕ Poly
12:       else
13:         r ← r << 1
14:       end if
15:     end for
16:     if RefOut = true then
17:       r ← reflect(r, Width)
18:     end if
19:   end for
20: end for
```

Figure 4.3: Pseudocode for generating contents of tables based on the description in [27, 80]. The $Offset$ is the position of a byte in an input message M that is being processed.

them around its center. Prior to forwarding a message to the remainder generator, the message is shifted to left by $Width - Slice$ bits (line 8), depending on the width of a data-path and a slice.

The remainder generator unit performs long division operation, consisting of series of sequential operations. Operations are inter-dependent from the results of a previous operation, thus it is impossible to execute them simultaneously. We described a single operation and defined it as $R\ Module$ (Fig. 4.4d). Number of cycles required to generate one remainder depends on the offset of a byte in input message M. At the end, the remainder is reflected or forwarded to output. This unit determines the speed and the area of the circuit.

The TGM is designed to be independent of the width of a generator polynomial, thus the TGM in Fig 4.1. does not require additional input for the width of a generator polynomial. The polynomial is aligned to left, while in the case when $G < 65$ bit positions from $(64 - (G - 1))$ to 0 are filled with zeros. This feature significantly simplified design of TGM.

In the next two sections we will describe two possible architectures for TGM and analyze them.

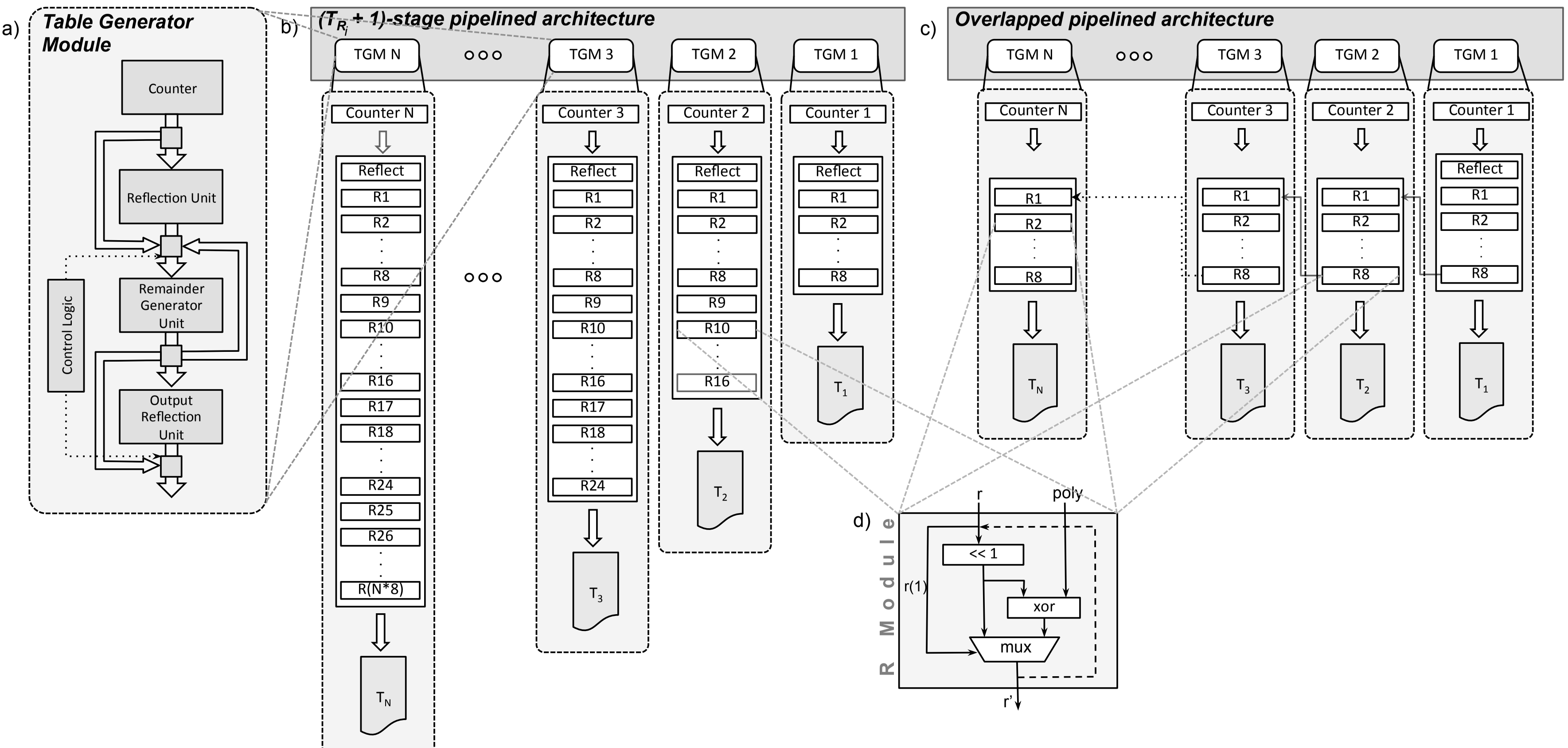

Figure 4.4: a) A general block diagram of a single Table Generation Module - TGM; b) A schematic of $(T_{R_i} + 1)$-*stage* pipelined architecture; c) A schematic of *Overlapped* pipelined architecture (preferred design); d) Architecture of a single operation of Remainder Generator Unit called *R Module*. The pipeline registers are placed after all *Reflect* and *R Modules* in both architectures.

4.1.2.1 ($T_{R_i} + 1$)-stage pipelined architecture

Straight-forward implementation of TGM in hardware is to design N circuits, and generate contents for each table in parallel. In order to generate one remainder each cycle we consider multiplying R Modules and using pipelining to exploit the intrinsic parallelism. The schematic of this architecture is presented in Fig. 4.4b).

The number of R Modules depends on the width of input data M. We explored possibility of processing $M \in 64, 128, 256, 512, 1024$. Thus, the total number of R Modules *per every consecutive table* $i \in 1, 2, 3, ... N$ is calculated with the following formula:

$$T_{R_i} = Offset_i \times Slice_length, \tag{4.1}$$

where $Offset$ is the position of a byte in an input message M that is being processed. However, the space complexity of this approach is $O(nm)$ when we consider the total number of R Modules *per algorithm*:

$$T_{R_N} = \sum_{i=1}^{N} T_{R_i} = Slice_length \times \sum_{i=1}^{N} Offset_i. \tag{4.2}$$

Consequently, the minimum required number of stages in the pipeline for the first table is 9 stages, and every consecutive table requires 8 additional stages (shown in Fig. 4.4b). The first remainder is generated after T_{R_i} clock cycles, followed by other remainders generated in every clock cycle.

By our estimations (detailed in Section 4.2.), the architecture with 64 and 128 tables couldn't fit in a moderate size FPGA, while the architecture for smaller number of tables would occupy substantial amount of area on a modern FPGA. In next section we introduce the method to reduce space complexity by overlapping specific operations.

4.1.2.2 Overlapped pipelined architecture

We noticed that we can reduce number of R Module *per table* to only 8 modules, by forwarding last non-reflected value from the R8 Module as an input to the R1 Module in the next TGM. The concept is illustrated in Fig. 4.4c). Doing so, we significantly reduced number of R Modules, and we simplified architecture of TGMs from 2 to N. Architecture of the first TGM was modified to output non-reflected value from R8 Module, and corresponding logic was added to connect this value to the following TGM. First non-reflected value can be forwarded in T_{R_i} clock cycle, just before output reflection. This also means that the second TGM can start processing one clock cycle before the first table outputs its first remainder. Initial reflection is not necessary for tables 2 to N, since the counter value was reflected in the first table. We keep counters in other generators to generate addresses for corresponding remainders. Thus, *TGM 1* has nine pipeline stages, while other TGMs have only eight. The input reflection is not needed due to the

forwarding of non-reflected values from *TGM 1*. Thus, the latency of *TGM 1* is nine cycles, while the latency of *TGM 2* to *8* is eight cycles.

Any additional table will add the latency of 8 cycles to a number of cycles from the start of calculation. The total latency is the same as in $(T_{R_i} + 1)$-stage pipelined architecture, but the amount of resources used is significantly reduced.

In this architecture, the total number of R Modules *per algorithm* is calculated with:

$$T_{R_N} = N \times 8, \tag{4.3}$$

where $N \in 8, 16, 32, 64, 128$. Thus, the space complexity is reduced from $O(nm)$ to $O(n)$.

4.1.3 Effects of architecture's scalability

As we mentioned earlier, we explored possibility of processing arbitrary number of input data $M \in 64, 128, 256, 512, 1024$ bits and in this section we discuss effects on our architecture. The first series of table generations read $M = 64$ bits (*Slicing–by–8*) and require eight TGMs. Every other extension in the number of processed bits doubles the number of required TGMs. In order to keep resource utilization at minimum, we decided to re-use these eight modules for other implementations. The basic idea is shown in Fig 4.5. Clock cycles are presented as numbers on the left side from a table, while numbers inside the tables represent position of the remainder in a table. For the second series of table generations (*Slicing–by–16*), which read 128 bits, we modified *TGM 1* to accept non-reflected values from *TGM 8*. This is essentially forwarding of non-reflected values again to *TGM 1*. The only problem is that *TGM 1* can start generating remainders for table 9 after 264 clock cycles, while *TGM 8* generates first non-reflected value in cycle 64. This means that non-reflected values cannot be forwarded directly to *TGM 1*. Thus, we decided to introduce a temporary table for *TGM 8*'s non-reflected values - *Table X*. *TGM 1* will start generating remainders for Table 9 just after it finishes generation of remainders for Table 1. In order to generate remainders for Table 9, *TGM 1* will read input values from Table X (shown with line 1 in Fig 4.5.) instead from the counter shown in TGMs basic architecture. All the following tables will start executing with 8 clock cycles latency compared to a previous table.

For the third series of Table generations, which read 256 bits, we had to introduce another temporary table - *Table Y*. *TGM 8* generates new non-reflected values for Table 9. before *TGM 1* reads all the values from the first temporary table, thus these values will be temporarily stored in Table Y. Then, Table Y will be used for generation of Table's 17 remainders (line 2), while Table 24 will store its non-reflected values in Table X (line 3). Input into Table 25 will be non-reflected values stored in Table X (line 4), and so forth. Table Y will be lastly re-used by Table 120, in order to calculate remainders for total number of 128 tables for *Slicing–by–128*. This will be 7th usage of this table during re-generation of tables for *Slicing–by–128* implementation (line 7).

Table X and *Table Y* are implemented as dual-port BlockRAMs, thus there are no pipeline stalls in any *TGM*. Due to its pipelined design, *TGM 1* can start reading first non-reflected value

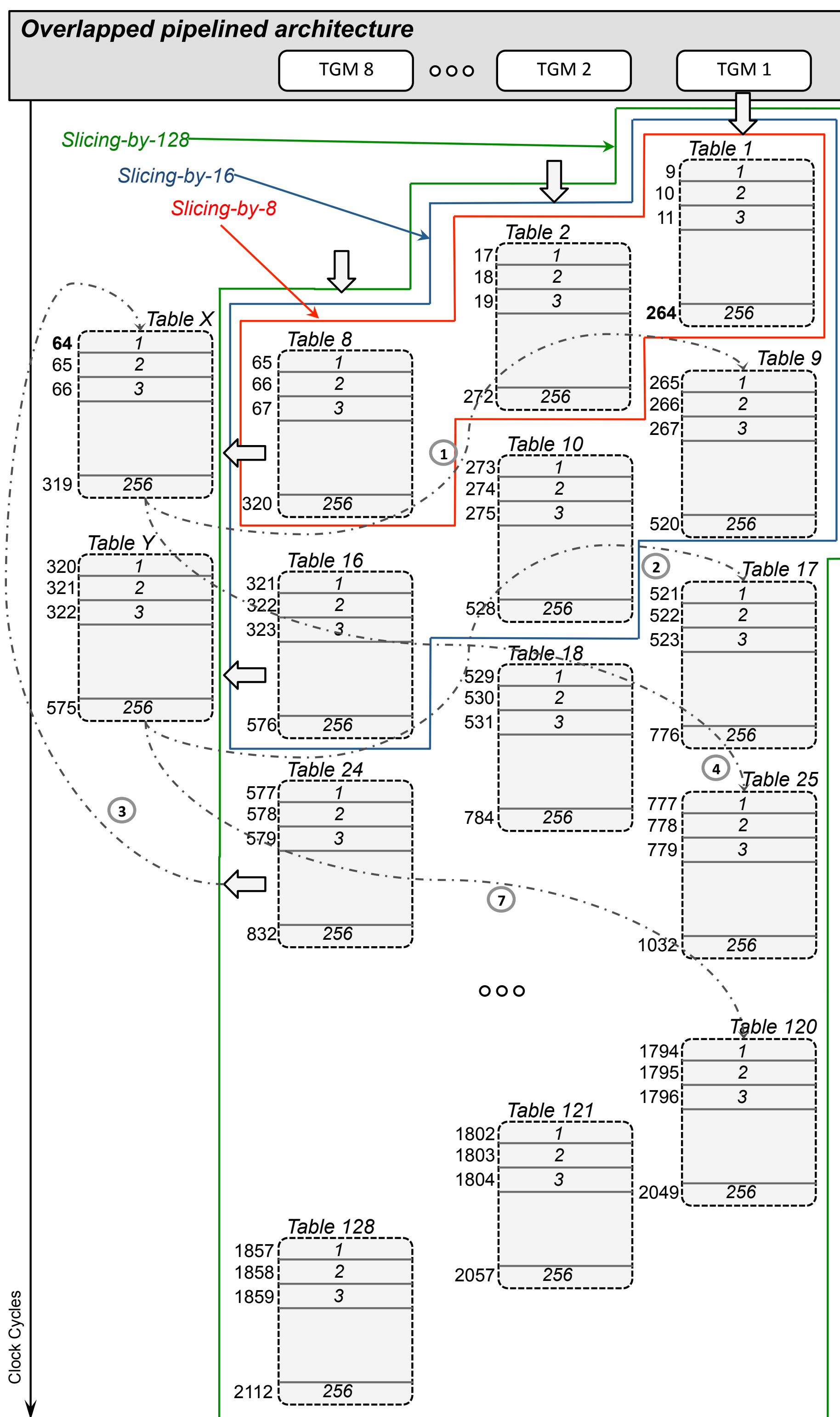

Figure 4.5: All five implementations use only eight TGMs for generating contents of 1 - 128 tables.

from *Table X* in 257 cycle, instead of 319 cycle. Thus, the first value in Table 9 is generated in cycle 265. This technique also applies to remaining tables.

In order to support this idea, data-path and controller of each of these series of table generations is significantly different from each other. Resource utilization is therefore kept minimal, as will be discussed in Section 4.2. This architecture enables further expansion of throughput with minimal resource utilization.

4.1.4 The IP* Core Interface

When an application requests CRC IP* Core service, the request is first delivered to the IP* Core Interface (as seen in Fig 4.2). The Interface is connected with external processing systems through the PCI Express bus. It consists of a PCI Express Controller, command buffer, completion buffer, input buffer and output buffer. A command request is delivered to the CRC IP* Core through the command buffer. There are two types of command request: Table re-generation and CRC generation. After the command is processed, a completion result is delivered to the host CPU through the completion buffer. The input buffer is used to store data fetched from the main memory by DMA. Similarly, output buffer is used to store data that will be transferred into the main memory by DMA.

When a user invokes a routine in the CRC IP* Core device driver, it creates a command request corresponding to the user's request and stores it into the command buffer. The IP* Core Interface then reads the command request and performs operations corresponding to the request. After the request is processed, the IP* Core Interface creates the completion result and stores it in the completion buffer. The IP* Core Interface then interrupts the host CPU to report completion of a request. The CPU reports the result to the user program.

4.2 FPGA Implementation

We designed a prototype implementation using the Xilinx Virtex 6 LX550T device (xc6vlx550t-2ff1760). The design was written in VHDL and Xilinx's ISE 12.4 design environment was used for all parts of a design flow including synthesis, mapping and place and route. The behavioral correctness of each circuit has been manually checked through a series of simulatios in Xilinx ISim 12.4. The advantage of implementing table-based CRC architecture on an FPGA comes from FPGA's ability to access look-up tables in parallel, thus by increasing number of processed bits at a time we can maintain increase in throughput.

After the implementation of TGMs on an FPGA, we came to the conclusion that $(T_{R_i} + 1)$-*stage pipelined architecture* is area consuming, when we consider re-generating remainders for 32, 64 or 128 tables in parallel. We measured number of resources for a single *R Module* which is 8 slices or 32 LUTs. Then, we implemented two versions of this architecture with $M = 64$

Table 4.1: Resource utilization of R Modules in $(T_{R_i} + 1)$-stage pipelined architecture for $M \in 64, 128, 256, 512, 1024$. Column three represents roughly estimated resources utilization based on the resources required by a single R Module, while column four represents actual resource utilization on the Xilinx Virtex 6 LX550T.

Xilinx Virtex 6 LX550T	TGMs (Tables)	LUTs **est.**	LUTs
$M = 64$ bits	8	9.2k (2%)	10.5k (3%)
$M = 128$ bits	16	34.8k (10%)	37.6k (10%)
$M = 256$ bits	32	**135k (39%)**	-
$M = 512$ bits	64	**532k (154%)**	-
$M = 1024$ bits	128	**2113k (614%)**	-

and $M = 128$, with 8 and 16 TGMs/Tables, respectively. We also roughly estimated number of resources used for R Modules in $M \in 64, 128, 256, 512, 1024$ architectures. The results are presented in Table 4.1. The estimated resources were very close to the measured resource utilization, thus we concluded that the majority of resources were used for R Modules. Most likely, architectures with $M = 512$ and $M = 1024$ would not fit on this fairly large FPGA chip, while architecture with $M = 256$ would occupy substantial amount of resources - 39% on the Xilinx Virtex 6 LX550T.

Overlapped pipelined architecture is used for implementation of TGM. Maximum of eight TGM's are used in *all* five implementations. This feature ensured minimal resource usage for a number of input bits at a time, and also enabled further expansion of the architecture.

Number of tables grow proportionally with the number of bits processed at a time, thus it is important to choose storage elements with fastest access time. To measure performance of this architecture, we implemented tables in a) BRAM and in b) logic. Tables are read-write, but writing and reading operations are not overlapped in order to maintain consistency of tables. For implementation *a)* tables are implemented as RAM modules in Block RAM.

4.3 Evaluation

4.3.1 Non-adaptable CRC accelerator core

First of all, we evaluated the performance and resource usage of the CRC accelerator core when it is used as a fixed non-adaptable one. In Table 4.2, Slicing–by–N_{32} and Slicing–by–N_{64} show the case when the resulting CRC value is 32 and 64, respectively.

As shown later, the throughput achieved is superior or comparable to the traditional LFSR im-

plementation. Thus, it appears that the accelerator with table-based algorithm achieved enough performance with reasonable cost.

Table 4.2: Resource utilization of *non-adaptable* Slicing–by–N_{32} and Slicing–by–N_{64} algorithms on the Xilinx Virtex 6 LX550T, where $N = M/8$, $M \in 64, 128, 256, 512, 1024$, and "32" and "64" represent the width of a resulting CRC value (related to 33 and 65 generator polynomial). Tables are implemented in BRAM.

	Slicing–by–N$_{32}$				
Xilinx Virtex 6 LX550T	$M = 64$	$M = 128$	$M = 256$	$M = 512$	$M = 1024$
LUTs (max 343680)	205	404	676	916	1180
Fully used LUT-FF pairs	159	359	493	714	979
BRAM (max 632)	8	16	32	64	128
Max. operating freq. (MHz)	469.14	431.86	332.92	332.92	347.98
Throughput (Gpbs)	29.32	53.98	83.23	166.46	347.98
	Slicing–by–N$_{64}$				
Xilinx Virtex 6 LX550T	$M = 64$	$M = 128$	$M = 256$	$M = 512$	$M = 1024$
LUTs (max 343680)	540	780	1186	1328	2230
Fully used LUT-FF pairs	458	698	849	946	1824
BRAM (max 632)	8	16	32	64	128
Max. operating freq. (MHz)	468.02	430.91	332.36	332.36	347.37
Throughput (Gpbs)	29.25	53.86	83.09	166.18	347.37

4.3.2 Fully-adaptable CRC accelerator

4.3.2.1 Throughput and resource usage

In Table 4.3, the performance and resource usage of fully-adaptable CRC accelerators are shown. In this implementation, *Overlapped pipelined architecture* is used in TGM, and tables are implemented in: a) BRAM and b) logic. Note that the *fully-adaptable* architecture is capable of generating remainders for any known CRC standard, up to 65 bits of generator polynomial, during run-time. Its usability is much broader than non-adaptable architecture.

The throughputs of four implementations of non-adaptable and fully-adaptable CRC accelerators are shown in Fig. 4.6. The maximum throughput was achieved with fully-adaptable architecture with tables implemented in logic (Table 4.3b). It is 418.8 Gbps when M = 1024, and it is up to 31% higher than adaptable architecture with tables implemented in BRAM. Compared to non-adaptable CRCs, the configuration of BRAM was changed from read-only to read-write, thus

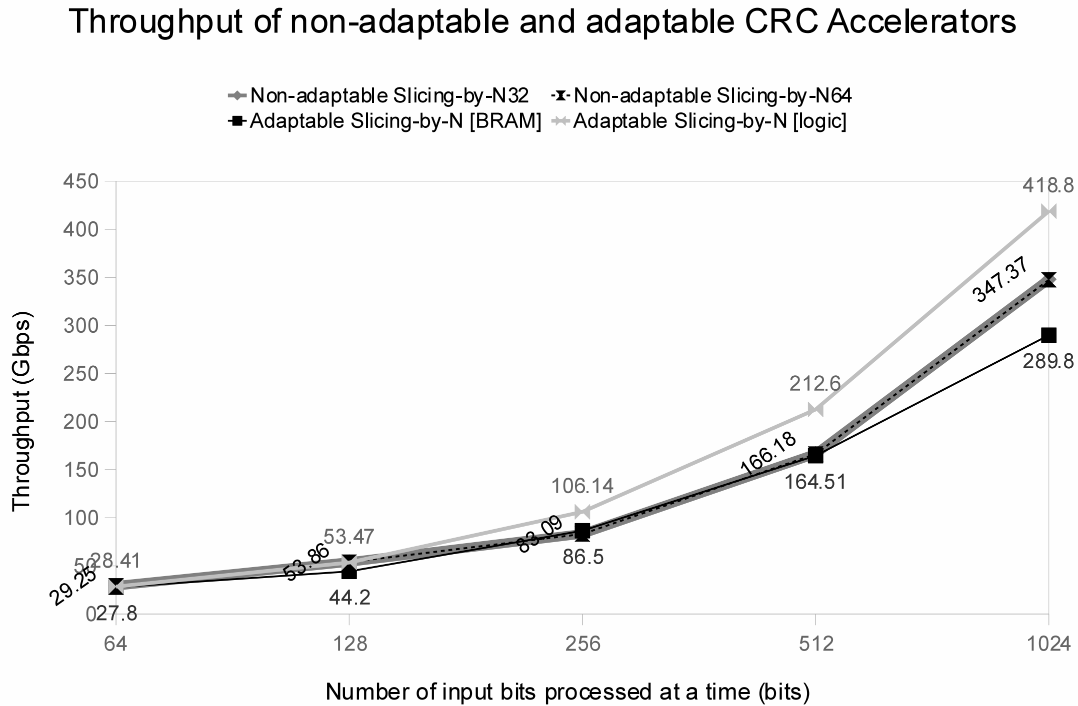

Figure 4.6: Throughputs of four implementations of non-adaptable and fully-adaptable CRC accelerators.

the total critical path was increased to approximately 1.78 ns, and between 0.512 to 0.696 ns of routing delay. This is why fully-adaptable CRCs with BRAM show decrease in throughput, but they can still support most demanding applications. *Slicing–by–*N_{32} and *Slicing–by–*N_{64} exhibit almost the same trend in throughput, because their architecture is not noticeable different. We show that each time we double number of processed bit at a time, architecture's throughput also doubles in all four implementations. Thus, when choosing a type of accelerator, the trade off is between flexibility/adaptability, throughput and resource utilization.

Even though fully-adaptable architecture a) exhibits highest throughput among all implementations, it also exhibits highest resource utilization on Xilinx Virtex 6 LX550T board. It occupies between 1.6% and 14.2% of LUTs resources, while the architecture b) occupies only 1-2% of LUTs resources. LUTs resource utilization of non-adaptable accelerators is around 1% on the same device. Resource utilization is kept at minimum in each implementation with different M's, as explained in Section 4.1.3. Every consecutive implementation in a) adds insignificantly more resources, while in b) most resources are occupied by contents of tables. We think that resource utilization is still acceptably low, especially for fully-adaptable architecture a).

Table 4.3: Resource utilization of *fully-adaptable* CRC accelerator, where $N = M/8$ and $M \in 64, 128, 256, 512, 1024$. We present two implementations with tables implemented in a) BRAM and b) in logic on the Xilinx Virtex 6 LX550T.

	a) Slicing–by–N [BRAM]				
Xilinx Virtex 6 LX550T	$M = 64$	$M = 128$	$M = 256$	$M = 512$	$M = 1024$
LUTs (max 343680)	3398	3405	3949	5119	6774
Fully used LUT-FF pairs	3082	2977	3481	3604	3617
BRAM (max 632)	8	16+1	32+2	64+2	128+2
Max. operating freq. (MHz)	352.37	345.31	337.93	321.31	283.1
Throughput (Gpbs)	27.8	44.2	86.50	164.51	289.8
	b) Slicing–by–N [logic]				
Xilinx Virtex 6 LX550T	$M = 64$	$M = 128$	$M = 256$	$M = 512$	$M = 1024$
LUTs (max 343680)	5571	9151	14861	26114	48756
Fully used LUT-FF pairs	3306	5084	6185	10839	17218
BRAM (max 632)	–	–	–	–	–
Max. operating freq. (MHz)	443.9	417.8	414.61	415.14	408.9
Throughput (Gpbs)	28.41	53.47	106.14	212.6	418.8

4.3.2.2 Time for re-generation of tables

It’s important to consider time required for regeneration of content in tables. This happens only when CRC standard changes. In Table 4.4. we present number of cycles and time required for each algorithm we implemented. We consider this to be reasonably quick and not noticeable by

Table 4.4: Number of clock cycles and time required for re-generation of tables when generator polynomial changes.

Input data	Tables	Clock Cycles	BRAM (μs)	logic (μs)
$M = 64$ bits	8	320	0.91	0.72
$M = 128$ bits	16	576	1.67	1.38
$M = 256$ bits	32	1088	3.22	2.62
$M = 512$ bits	64	1600	4.98	3.85
$M = 1024$ bits	128	2112	7.46	5.17

a user.

4.3.3 Comparison to Related Work

Table 4.5. provides a summary of related works for generating CRC implemented on a different technologies. Although it is difficult to compare performance and area parameters for different technologies, some valid comparisons can be made.

Compared to all non-adaptable hardware designs, [28] achieves the best throughput, but our circuit is more than 2 times faster (M=128). Unfortunately, the area used in this design is not provided. When compared with fastest software solution [27], our design is 8 times faster with further ability to extend number of bits processed at a time. In [26, 33, 35] the throughput is significantly lower than our implementations and the circuits have to be taken off-line in order to support other CRC Standards.

There are only two adaptable hardware implementations with limited support for a number of generator polynomials [25, 36]. They differ from our implementation in that they require separate implementation for every generator polynomial, while our circuits support variable number of CRC standards with only one implementation. Our fully-adaptable *Slicing–by–16* (M=128) implementation is 41 times faster than [36] soft-coded design with 32 bit CRC and 14.5 times faster than hard-coded design. Unfortunately, the reconfiguration time and area utilization are not provided. [25] is generic in its design, thus it can be scaled to process 64, 128 or 256 bits, with maximal theoretical throughput of 40 Gbps at 256 bits. It is 6 times slower compared with our adaptable *Slicing–by–8* implementation, and 5 times slower compared with adaptable *Slicing–by–16*. The reconfiguration time is not very specific - under 1μs, just as our adaptable *Slicing–by–8*. It uses different implementation technology, thus it is very difficult to compare area used.

As can be seen from Table 4.5., there is no much research about CRC circuits that support 64 generator polynomial, so we cannot compare our implementation to any other in that terms.

Table 4.5: A summary of different CRC designs from Related Work and our implementations on the Xilinx Virtex 6 LX550T: **a) fully-adaptable CRC with *Overlapped architecture* with tables in BRAM** and **b) tables in logic**; **c) non-adaptable Slicing–by–N$_{32}$ CRC** and **d) non-adaptable Slicing–by–N$_{64}$ CRC**.

Design	Polynomial	M	Adaptable	Re-generation time	Technology	Area	Throughput
[27]	32	32, 64	√	N/A	Pentium 1.7 GHz (90nm)	-	1.4, 3.6
[33]	32	32	–	N/A	350nm (~137 MHz)	162 LUTs	4.38
[35]	32	32, 64	–	N/A	350nm AMS (180 MHz)	7.73 mm^2	5.76
[26]	16, 32	8, 16, 32	–	N/A	FLEX10KE ALTERA family	149 - 1849 LC	1.1 (8b) - 4.6 (32b)
[28]	32	128	–	N/A	90nm ST CMOS (200 MHz)	N/A	~25
[36]	8, 32	128	–/√	N/A	180nm (200 MHz)	N/A	1.3 - 3.7
[25]	32	32 (64)	√	$< 1\mu$s	130nm UMC standard cell	0.15 mm^2	4.92 (9.84)
Our a)	up to 64	64-1024	√	0.91-7.46μs	Virtex 6 LX550T	3398 - 6774 LUTs	27.8 - 289.8
Our b)	up to 64	64-1024	√	0.72-5.17μs	Virtex 6 LX550T	5571 - 48756 LUTs	28.41 - 418.8
Our c)	32	64-1024	–	N/A	Virtex 6 LX550T	205 - 1180 LUTs	29.32 - 347.98
Our d)	64	64-1024	–	N/A	Virtex 6 LX550T	540 - 2230 LUTs	29.25 - 347.37

4.4 Summary

Cyclic Redundancy Check is a well known error detection scheme used to detect corruption of digital content in digital networks and storage devices. Since it is a computationally intensive process which adversely affects performance, hardware acceleration using FPGAs has been tried and satisfactory performance has been achieved. However, recent extended usage of

networks and storage systems require various correction capabilities for various CRC standards. Traditional hardware designs based on the LFSR (Linear Feedback Shift Register) tend to have fixed structure without such flexibility.

Here, non-adaptable and fully-adaptable CRC accelerators based on a table-based algorithm are proposed. The table-based algorithm is a flexible method commonly used in software implementations. It has never been implemented with the hardware, since it is believed that the operational speed is not enough. However, by using pipelined structure and efficient use of memory modules in FPGAs, it appeared that the table-based fixed CRC accelerators achieved better performance than traditional implementation. Based on the implementation, the *fully-adaptable* CRC accelerator which eliminate the need for many non-adaptable CRC implementations is proposed. The accelerator has ability to process arbitrary number of input data and generates CRC for any known CRC standard, up to 65 bits of generator polynomial, during run-time. Further, we modify table generation algorithm in order to decrease its space complexity from $O(nm)$ to $O(n)$. On Xilinx Virtex 6 LX550T board, the fully-adaptable accelerators occupy between 1 to 2% area to produce maximum of 289.8 Gbps at 283.1 MHz if BRAM is deployed, or between 1.6 - 14% of area for 418 Gbps at 408.9 MHz if tables are implemented in logic. Proposed architecture enables further expansion of throughput by increasing a number of input bits M processed at a time.

Chapter 5

Design and implementation of IP-based iSCSI Offload Engine on an FPGA

The Internet Protocol (IP) based storage systems provide a flexible and high-performance block data access for storage applications. Their unique contribution is the ability to integrate storage networking into mainstream data communications. Hence, they have expanded the boundaries of traditional data storage by employing standard IP networks such as Gigabit Ethernet. Since Ethernet-based LANs have long been the industry standard, it is expected that overall performance of storage systems will improve with expected increase of Ethernet's data rate. The iSCSI protocol [17] defines one approach for accessing and transporting data over commonly utilized TCP/IP infrastructure. It is developed by the Internet Engineering Task Force (IETF) with the goal to map the SCSI protocol over TCP/IP. This approach enables storage devices to be attached to IP-based networks. The protocol ensures high data integrity through header and data digests in the specific iSCSI Protocol Data Units (PDUs). However, the processing of iSCSI digests (CRC) is considered to be the most computationally intensive part of the iSCSI protocol processing [18]. This is especially evident for multi-Gbps networks where the impact of computational overhead is so pronounced that the current state of the art processors cannot take advantage of the capacity of the network. Thus, it is common practice to disable data digests [19]. In such cases, data integrity is ensured with only TCP and/or Ethernet error detection mechanisms, which cannot detect errors which occur between upper layer protocol transitions.

This problem has been addressed by offloading computationally intensive parts to a special purpose hardware. Thus far, commercial hardware iSCSI solutions have been implemented by using TCP/IP Offload Engines (TOE) or iSCSI host bus adapters (HBA). These systems offload either TCP/IP protocol stack or both TCP/IP and iSCSI protocol onto a specialized hardware. Offloading TCP/IP protocol stack shows significant decrease of CPU utilization, but digest processing still requires significant time on a CPU. Even though iSCSI digests have been identified as the most computationally intensive part of iSCSI protocol processing, it is not enough to offload only iSCSI digests. In such a case, the communication overhead between software and

hardware parts might undermine all the performance gain. On the other hand, it is challenging to offload all iSCSI processes onto an FPGA. There are three primary reasons. First, the scope of iSCSI code is too large and complex, and requires a lot of programming effort and time. Second, some functions such as authentication, authorization and security are challenging to implement in hardware. Third, it is thought that operating frequency of FPGAs is not enough to accomplish required throughput for high-speed networks.

The performance of software initiators is limited by the processing power of a general purpose processor, especially for the multi-Gbps networks [20]. The biggest concern is *high level of CPU utilization*, which affects execution of other applications. This has led to extensive research of offloading protocol processing to hardware. One example is iSCSI HBA, which guarantees performance for computationally intensive applications, but due to its underlining technology it prevents the addition of new functions. Thus, it is considered inflexible solution. Flexibility is an added value that enables easy adaptation to future changes in a protocol or an application. There has been only one attempt to offload iSCSI protocol to an FPGA [14]. However, the maximum reported throughput is only 86 Mbps without processing digests, and 31.84 Mbps with digests. This is not adequate for multi-Gbps networks.

Thus, we address this problem by partitioning Open-iSCSI code into two parts, one executed on general purpose processor and other offloaded to an FPGA-based adapter. Less frequently executed parts, such as processes belonging to Login Phase, are executed on general purpose processor, while frequently executed parts are implemented on an FPGA. In section 2.3.4 we performed performance analysis of Open-iSCSI and identified instructions which are executed most frequently. Not surprisingly, these are *Data-In* and *Data-Out* instructions, which are also called *data transfer functions*. However, it is not enough to offload only these instructions since communication between software and hardware parts might become overwhelming due to the exchange of control information. Thus, we also offload non-data transfer functions which are closely related to processing of data transfer functions. In the new architecture, we also integrate our previous work from Chapter 4, but employing several CRC generation units. In our research, we target mission-critical applications which require high data integrity, such as those of financial and banking transactions where database integrity failures might lead to lost funds, inaccurate stock exchange or credit card transactions. In these systems it is required to enable header and data digests, which adversely affects overall performance.

Precisely, our contributions are:

1. We analyze iSCSI traffic and identify the most commonly used functions. We measure and analyze CPU utilization and throughput of Open-iSCSI [21], which is an open source software based iSCSI initiator (section 2.3.4).

2. Based on (1), we offload data transfer and related non-data functions to an FPGA based

adapter. Data transfer functions are the most computationally intensive and the most executed functions in a common case scenario. Other functions which do not affect performance are implemented in software on a general purpose processor. The resulting architecture relieves the host CPU from computational burden imposed by the software implementation.

3. It is proved that the new architecture can overcome the performance limitations imposed by a single processor which operates on 15 times higher frequency than our FPGA implementation. The iSCSI Offload Engine allows very low utilization on the host CPU of approximately 3%.

4. Our architecture guarantees flexibility, since many functions are implemented on a general purpose processor. Any new feature, such as security functions, specification updates, CRC standards, etc., can be easily implemented.

5.1 Design and implementation of iSCSI Offload Engine

The iSCSI adapter consists of an iSCSI Offload Engine, a TCP/IP Offload Engine, an iSCSI Offload Engine Interface, a memory controller, a 10-Gigabit Ethernet Media Access Controller (MAC) and an eXtended Attachment Unit Interface (XAUI) Core. The 10-Gigabit Ethernet MAC is used to interface to Physical Layer devices in a 10-Gigabit Ethernet (10GE) system. The XAUI Core allows physical separation between the data link layer and physical layer devices in a 10GE system. Fig. 5.1 illustrates the structure of iSCSI adapter based on the architecture of the iSCSI Offload Engine. The design overview is based on Xilinx ML605 Evaluation Board. More implementation details are provided in Section 5.2.1. Our architecture is relying on the existing TCP/IP Offload Engine, which is well researched subject [10, 12, 13, 43].

In a typical iSCSI session, an initiator initiates series of read and/or write SCSI commands, after which appropriate responses follow, as illustrated in Fig. 5.2. Several read and/or write commands, as well as their data and responses usually intertwine, depending on the readiness to transmit data on initiator and target side. Data transmitted from a target to an initiator is regarded as *reading part* of the session (reception). Similarly, the transfer from an initiator to a target is regarded as *writing part* of the session (transmission). Thus, *the iSCSI Offload Engine* consists of two modules which divide processing work into reception work - *the Reception Module* (Rx) and transmission work - *the Transmission Module* (Tx). The architecture of two modules is discussed in Sections 5.1.1 and 5.1.2, respectively.

The Control Module enables sharing of data between Reception and Transmission Modules, TCP/IP Offload Engine, and modified Open-iSCSI initiator. The memory controller handles buffer memory which holds the packet buffers. The packet buffers consist of a header, data, header digest and data digest areas.

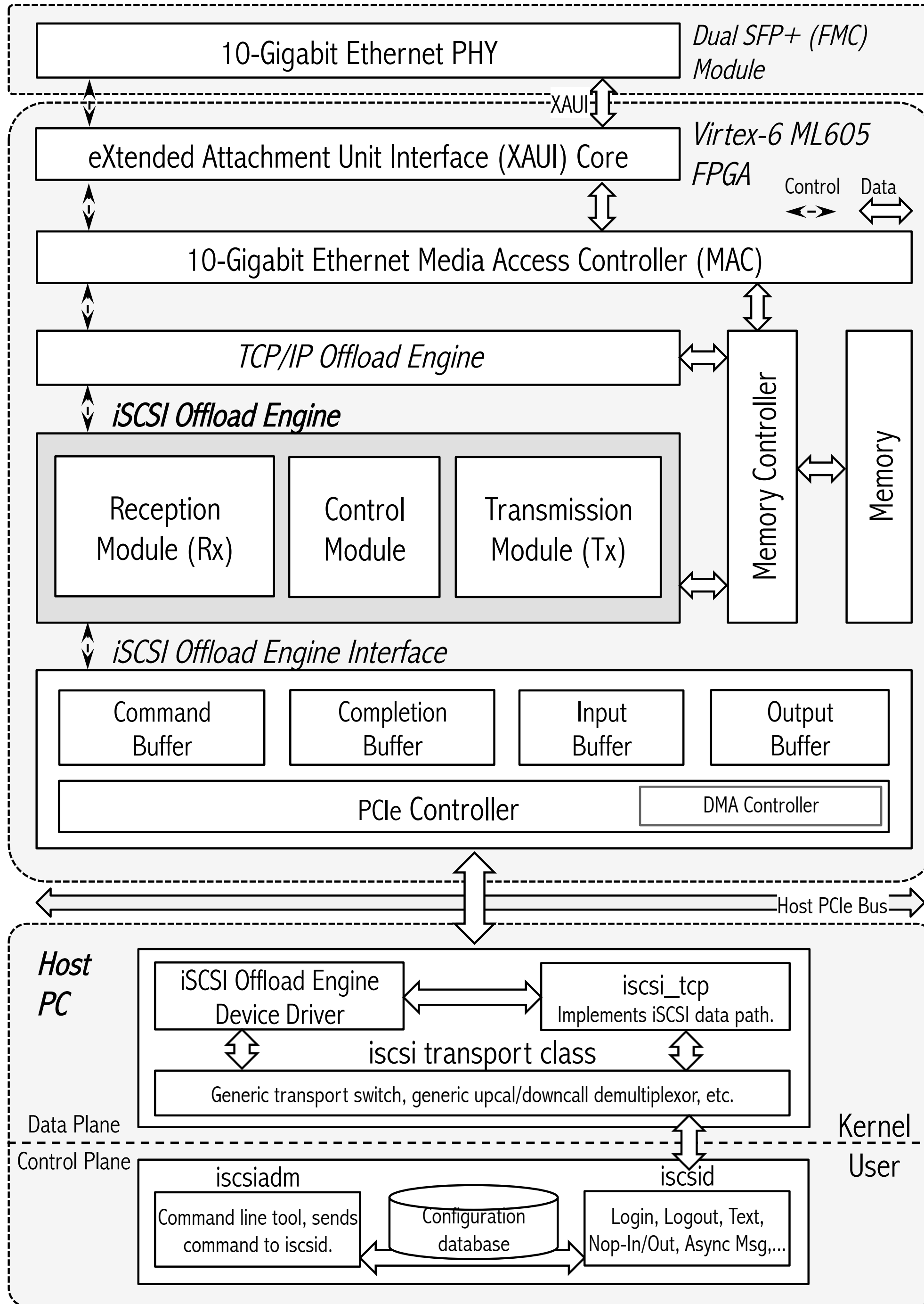

Figure 5.1: Design overview of the proposed iSCSI Offload Engine and modified Open-iSCSI implementation. The design is based on Xilinx ML605 Evaluation Board, which has only a 1000-BASE Ethernet interface. Hence, additional Dual SFP+ FMC [81, 82] and 10GbE SFP+ transceiver are required to achieve throughput of over 1 Gbps. More implementation details are provided in Section 5.2.1.

Operations implemented in the the Reception Module are: detection of iSCSI Frames, generation and validation of header digest, identification of a PDU, calculation and validation of data payload digest, and storing data in the host memory. Operations implemented in the Transmission Module are: fetching data from the host memory, creation of a header, and generation of header and data digests.

Table 5.1 displays the minimum set of opcodes defined on an initiator and a target. Based on our analysis of Open-iSCSI (Section 2.3.4), we offload processing of PDUs marked in bold to an FPGA. These PDUs are the most computationally intensive and the most frequently executed. Except R2T and SNACK Request, they all require data digests. The operations such as Asynchronous Message, Text Request and Text Response, Nop-In and Nop-out, and Reject also require data digest, but they are executed far less frequently. Thus, these functions are implemented on a general purpose processor.

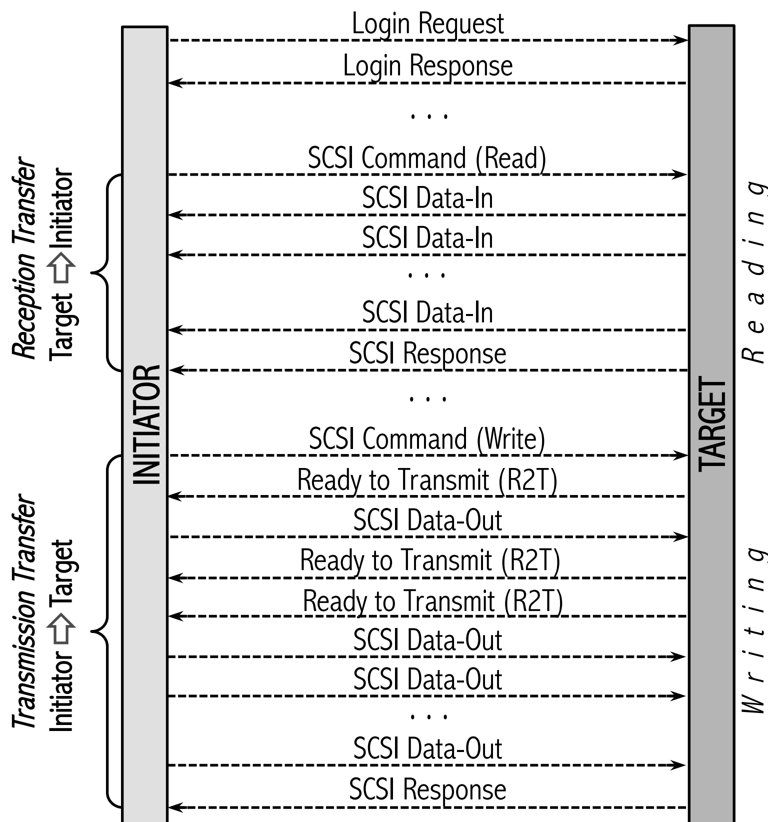

Figure 5.2: An overview of two transfer directions and two common sets of operations executed during reading and writing processes.

On the host CPU, we modified Open-iSCSI and Linux kernel to bypass certain iSCSI functions and TCP/IP layers. The Open-iSCSI is partitioned into kernel and user parts (Fig. 5.1), which implement iSCSI data plane and the control plane, respectively. The interface between these two parts is implemented using Netlink sockets. The socket library functions are handled

in a single system call (sys_socketcall). Depending on the type of a function, the sys_socketcall calls either the iSCSI Offload Engine Device Driver or the TOE Device Driver. The iSCSI Offload Engine Device Driver consists of a set of routines which control the iSCSI Offload Engine. The modifications are discussed in details in Section 5.1.5.

Table 5.1: A minimum set of opcodes defined on an initiator and a target. The iSCSI Offload Engine processes the most computationally intensive data-transfer and related non-transfer operations in both directions, marked in bold.

Initiator to Target (Tx)		
No.	Opcode	Name
T1	0x00	NOP-Out (H&D)
T2	0x01	**SCSI Command (H&D)**
T3	0x02	SCSI Task Management function request (H)
T4	0x03	Login Request
T5	0x04	Text Request (H&D)
T6	0x05	**SCSI Data-Out (H&D)**
T7	0x06	Logout Request (H)
T8	0x10	**SNACK Request (H)**
T9	0x1c-1e	Vendor specific codes
Target to Initiator (Rx)		
No.	Opcode	Name
R1	0x20	NOP-In (H&D)
R2	0x21	**SCSI Response (H&D)**
R3	0x22	SCSI Task Management function response (H)
R4	0x23	Login Response
R5	0x24	Text Response (H&D)
R6	0x25	**SCSI Data-In (H&D)**
R7	0x26	Logout Response (H)
R8	0x31	**Ready To Transfer (R2T) (H)**
R9	0x32	Asynchronous Message (H&D)
R10	0x3c-0x3e	Vendor specific codes
R11	0x3f	Reject (H&D)

(H&D) : Header and data digest (H) : Header digest

When a user requests iSCSI Offload Engine service, this request is first delivered to the iSCSI Offload Engine Interface. The modules then read the request from the Command Buffer and perform required operations. The data is copied to/from main memory of the host CPU into

Input/Output buffers by using DMA. The host CPU then fetches results from the Completion and Output Buffers and delivers them to the user program.

5.1.1 The Reception Module (Rx)

Fig. 5.3a) illustrates the structure of Reception Module (Rx). It consists of the Packet Controller, the SNACK Controller and the Rx Buffer Controller. After the incoming packet is processed by the TCP/IP Offload Engine, the TCP payload is transferred to the Rx Buffer Controller. The Packet Controller parses the header, identifies a PDU and validates header and data digest. If a PDU represents a SCSI Data-In, a SCSI Response or a R2T PDU, it is processed by the Packet Controller, else it is forwarded to the main memory to be processed by the software initiator or to a different processing engine.

The operations performed in the Reception Module are:

1. Header Parser reads the first 64 bits from the buffer, which contains information about PDU's opcode, the total length of additional header segment and the length of data segment. If the opcode belongs to one of the following PDUs: a) SCSI Response, b) SCSI Data-In or c) R2T, the PDU is processed by our offload engine. In any other case, the PDU is forwarded to the main memory to be processed by the software or to a different processing engine.

2. Calculation of header digest begins in parallel with parsing header. The header digest is calculated by the CRC Generation Unit (detailed in Section 5.1.3).

3. The newly calculated header digest is validated by comparing it to the received header digest. If equal, the CRC Generation Unit begins calculation of data digest.

4. When data digest is validated, the data is copied from Rx Buffer directly to the host memory via DMA.

5. When header or data digests are not validated, the packet in Rx Buffer is dropped. Then, an appropriate SNACK request is generated and sent by the SNACK Controller. In a session that supports error recovery, the target may request positive acknowledgment of input data. In this case, the SNACK Controller generates SNACK DataACK PDU and sends them through Tx Buffer.

Some operations are executed in parallel in order to improve the performance. Parsing of a header and calculation of header's digest are executed in parallel, as well as calculation of data digest and validation of the header digest. When data digest is validated, the data is copied from Rx Buffer directly to the host memory via DMA without a copy (direct data placement). The header and data digests are calculated with CRC Generation Unit, and validated by the Parser. The architecture of CRC Generation Unit is detailed in Section 5.1.3.

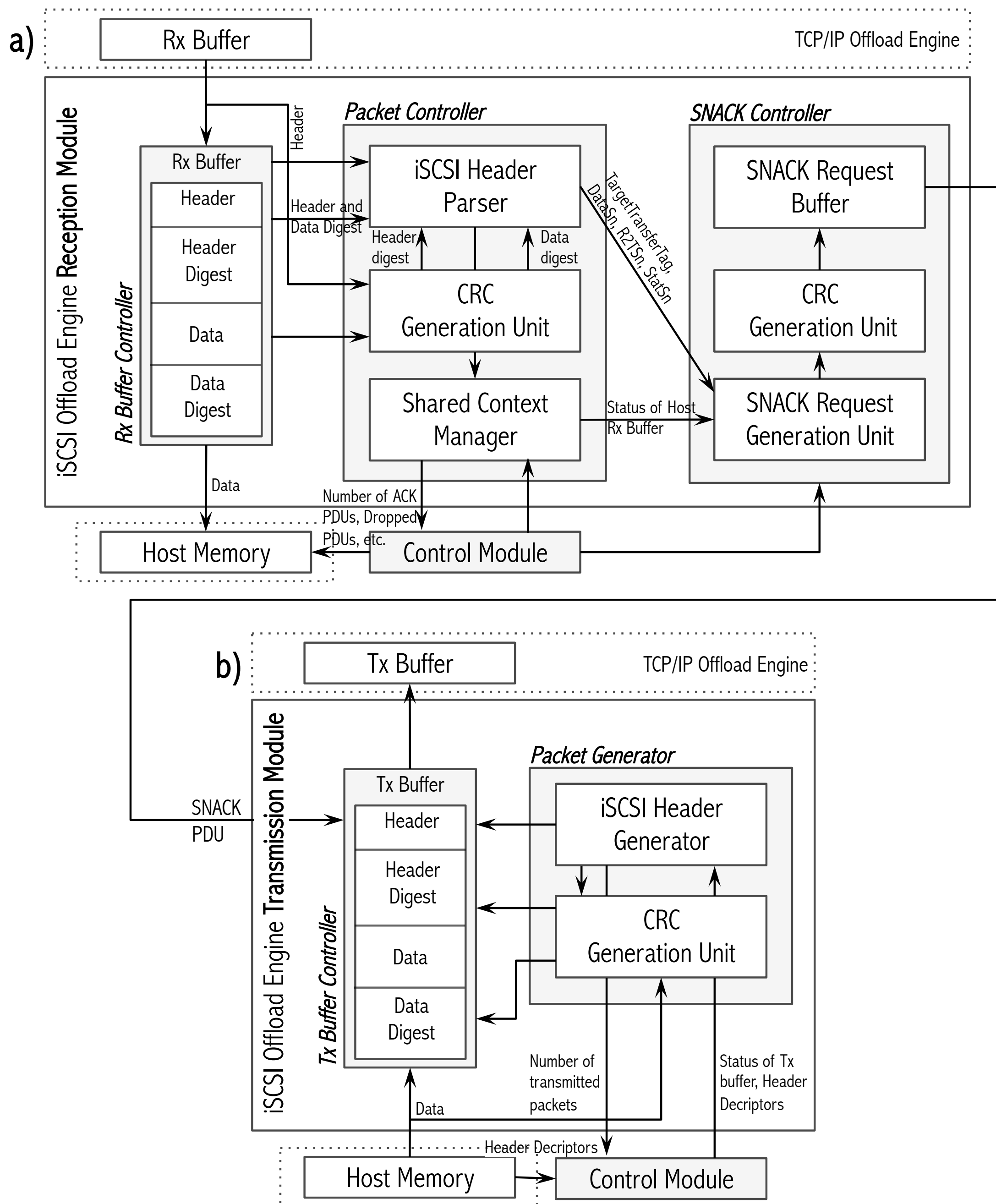

Figure 5.3: The structure of Reception and Transmission Modules in the iSCSI Offload Engine.

Even though *SNACK Request* is originally in the transmission data-path, we implement the SNACK controller in the Reception Module in order to shorten the time required to generate a request for re-transmission or acknowledgment of data. However, a SNACK PDU is sent through the Tx Buffer. The re-transmission request (SNACK) is generated when the header or data digests are not validated (the packet in the Rx Buffer is dropped in this case). In order to reduce the overhead of acknowledging each incoming packet, we design the SNACK Controller to generate a single delayed SNACK request for a group of missed status, data, or R2T PDUs within a task. This decreases the number of interrupts and improves the performance, since there are fewer number of requests.

The other example when the SNACK Controller is used is when a session supports error recovery. In this case, the target requests a positive acknowledgment in the form of SNACK DataACK PDU. This operation begins in parallel with the operation to store the validated packet from the Rx Buffer to the host memory. By implementing this operation in the hardware, the resources at the target are released faster, thus enabling more resources for other transactions. The SNACK Controller has independent CRC Generation Unit, thus it can generate header digest in parallel with the Packet Controller.

5.1.2 The Transmission Module (Tx)

Fig. 5.3b) illustrates the structure of Transmission Module (Tx). It consists of the Packet Generator and the Tx Buffer Controller. When the iSCSI Offload Engine device driver requests creation of an iSCSI PDU, the header descriptors are fetched from the main memory via DMA and forwarded to the Packet Generator. The iSCSI Header Generator creates a new header and forwards it to CRC Generation Unit to create a header digest.

When iSCSI Offload Engine device driver requests the Transmission Module (Tx) to create an iSCSI PDU, the operations performed are:

1. Header descriptors are fetched from the main memory and forwarded to the iSCSI Header Generator using the Control Module.
2. When header is generated, it is stored in the header area of the Tx Buffer, and then forwarded to the CRC Generation Unit to generate its digest. These operations are executed in parallel.
3. Once header digest is generated, it is stored in header digest area of the Tx Buffer.
4. The Tx Buffer Controller forwards the data payload to the CRC Generation Unit, while it is being copied from the host memory.
5. After it is generated, the data digest is stored in the Tx Buffer in the Data Digest area.

The following two sets of operations are executed in parallel. First, the generation of header digest is executed in parallel with transfer of a header from the iSCSI Header Generator to the Tx Buffer. Second, the generation of data digest is executed in parallel with transfer of data from the main memory to the Tx Buffer. The Tx Buffer Controller forwards a PDU to the TCP/IP offload engine for creation of TCP/IP/Eth header information. The TCP/IP Offload Engine then sends a request to the Gigabit Ethernet controller to transmit the packet.

5.1.3 The CRC Generation Unit

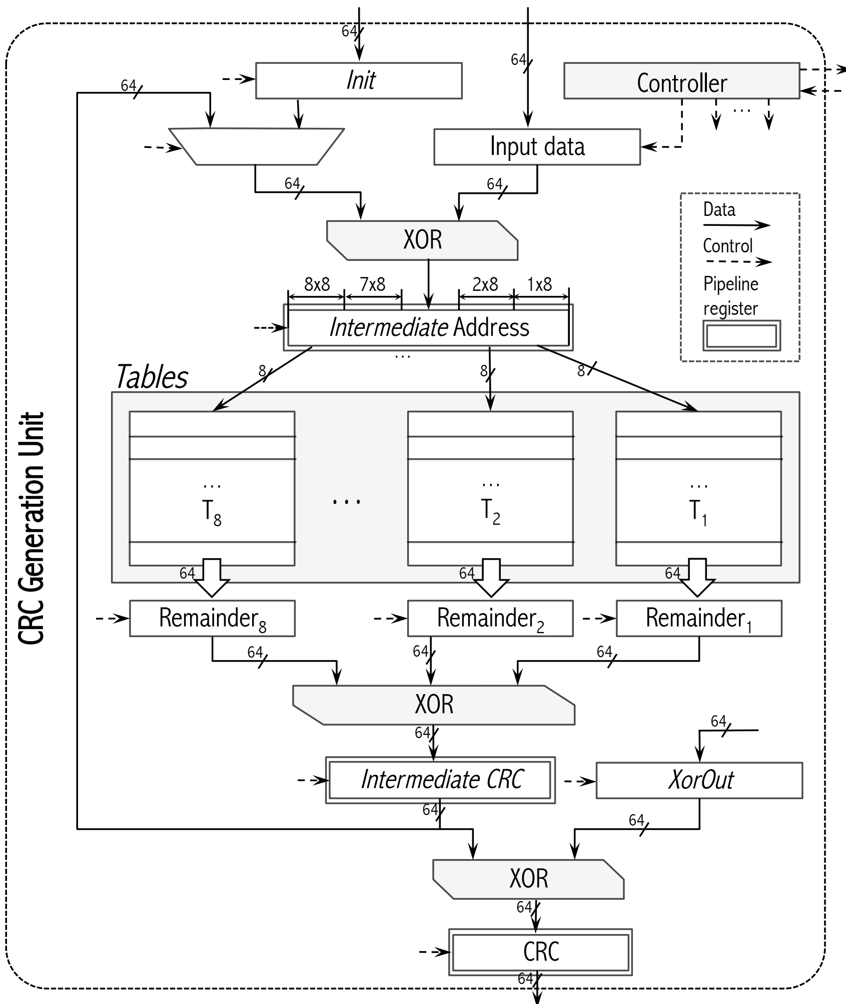

Figure 5.4: The architecture of CRC Generation Unit.

Fig. 5.4 illustrates the architecture of CRC Generator Unit based on our previous research with high-speed CRC accelerators [83–85]. The CRC algorithm deploys eight tables ($T8, ..., T2, T1$) with pre-computed remainders. The architecture is pipelined in three stages, and the throughput is 64 bit/cycle. The digest (CRC) of input data is formed with the following steps. In the first iteration, the *Intermediate Address* is formed by XORing input data with initial value (*Init*). In the

every other iteration, the *Intermediate CRC* is used instead of *Init*. The *Intermediate Address* is then sliced into eight 8-bit slices, which are used as addresses to access eight tables in parallel. Eight remainders are XORed to form the *Intermediate CRC*. The *Intermediate CRC* is XORed with the final value (*XorOut*) when the controller indicates the end of data. The digest is stored in the digest area of a buffer.

In order to achieve high degree of flexibility, we made some modifications to the original design. We enabled support for any given 32-bit CRC Standard defined in the iSCSI Specification [17]. The values of parameters *Init* and *XorOut*, and the contents of tables depend on a CRC standard. The values of parameters are changed on the request of the iSCSI Offload Engine device driver. However, the contents of tables is challenging to be replaced in the similar manner. Thus, we use difference-based partial reconfiguration. We minimize the dynamic part of the circuit to only tables, which allows us to generate a small bitstream containing only differences between two versions of the design. Then, we wrote a set of scripts to automatically assign new values corresponding to a CRC Standard, and stored them into the tables (BRAM components). The new values are stored into tables with the Xilinx FPGA Editor. We automatize the process of generating new bitstream which allows complete flexibility. The idea is to provide a number of pre-generated bitstreams (for a set of CRC standards) with the unit.

5.1.4 The Control Module

The Control Module shares information among four components: Reception and Transmission Modules, TCP/IP Offload Engine, and iSCSI software initiator. It supports fast and efficient data sharing by using quad-port memory [86]. In Fig. 5.5 we illustrate an exemplary exchange of information between an initiator and a target, where italic font displays direction of the new information coming from an initiator to a target, and inversely. The iSCSI initiator must verify consistency of the values used in all task-related PDUs. Thus, it stores important information in *five* look-up tables in a memory of a Control Module and forwards them to an appropriate unit for verification.

Fig. 5.6 illustrates the flow of information between modules in the iSCSI Offload Engine. Before sending a request to create a command PDU, the device driver first checks the status of iSCSI Offload Engine via the *session* table in the Control Module. Along with the request, it sends an address of the buffer with a set of information required to form a command PDU. This set is defined by the RFC 3720 [17]. In the case of "*SCSI Cmd PDU*" (Fig. 5.5), following kinds of information are being exchanged: Logical Unit Number (LUN), Initiator Task Tag (ITT), expected transfer length, command sequence (CmdSn), expected status number (ExpStatSn), etc. The LUN is used to identify a Logical Unit within a target, and ITT to identify a new task in the initiator. The *command* table is used to store these information. In the response to a command, the target sends a set of information such as a Target Transfer Tag (TTT), expected

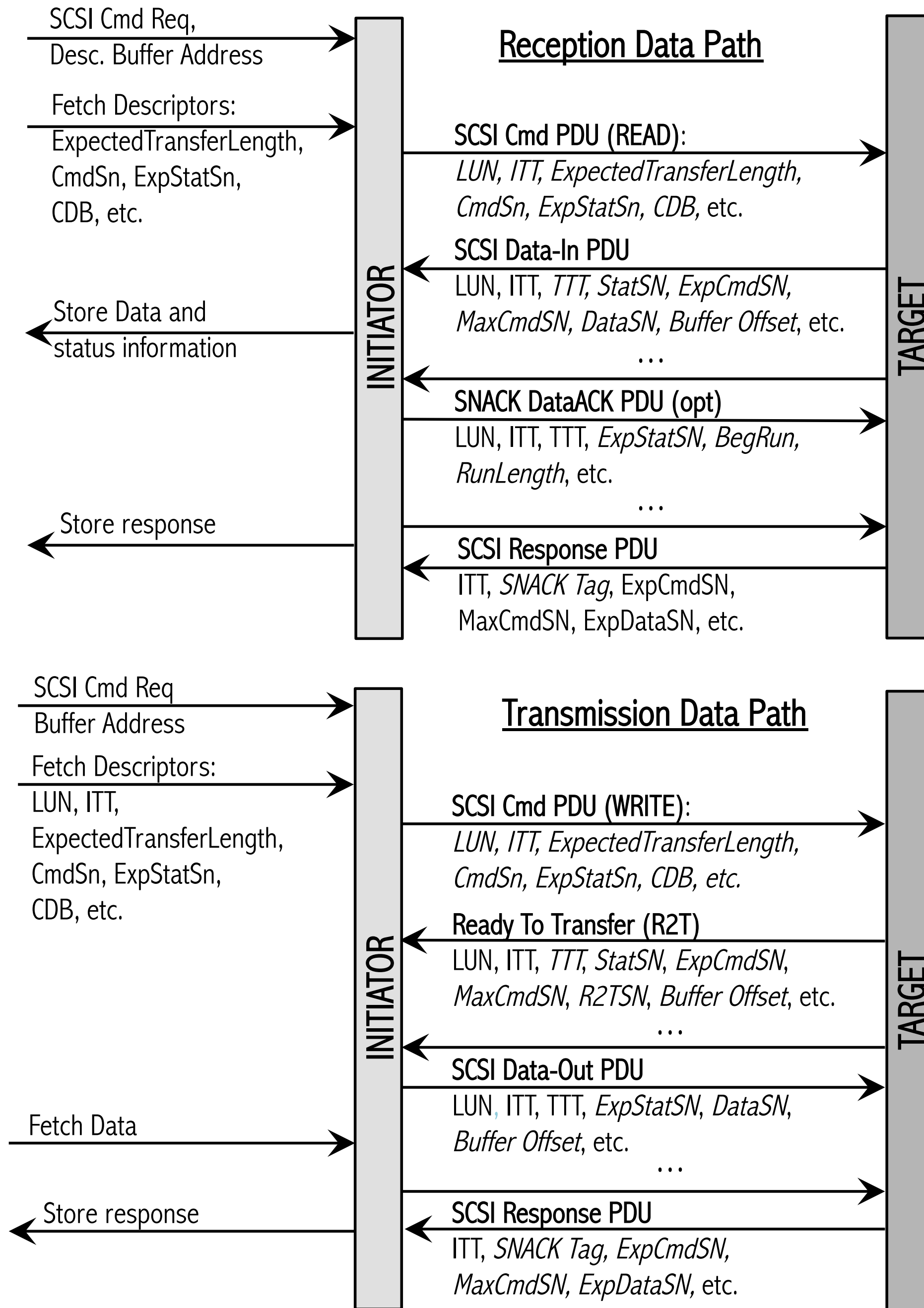

Figure 5.5: An exemplary exchange of information between the initiator and target. Italic font displays direction of the new information coming from an initiator to a target and inversely. The information is used for validation of a PDU.

command sequence number, sequence number of data PDU, etc. The Control Module also holds information regarding the status of transmit and receive buffers in the host memory, as well as in the TCP/IP Offload Engine. The *R2T* table holds information received from a target through R2T PDU, which are later used by the Transmission Module to create a SCSI Data-Out PDU. The *SNACK* table holds information required to generate SNACK requests, which can be requested from iSCSI Offload Engine device driver or generated directly by the Reception Module. When a PDU is acknowledged, it sends necessary information to modified Open-iSCSI. Lastly, the *data_address* tables holds the address where the data is directly copied from Rx Buffer to the host memory via DMA.

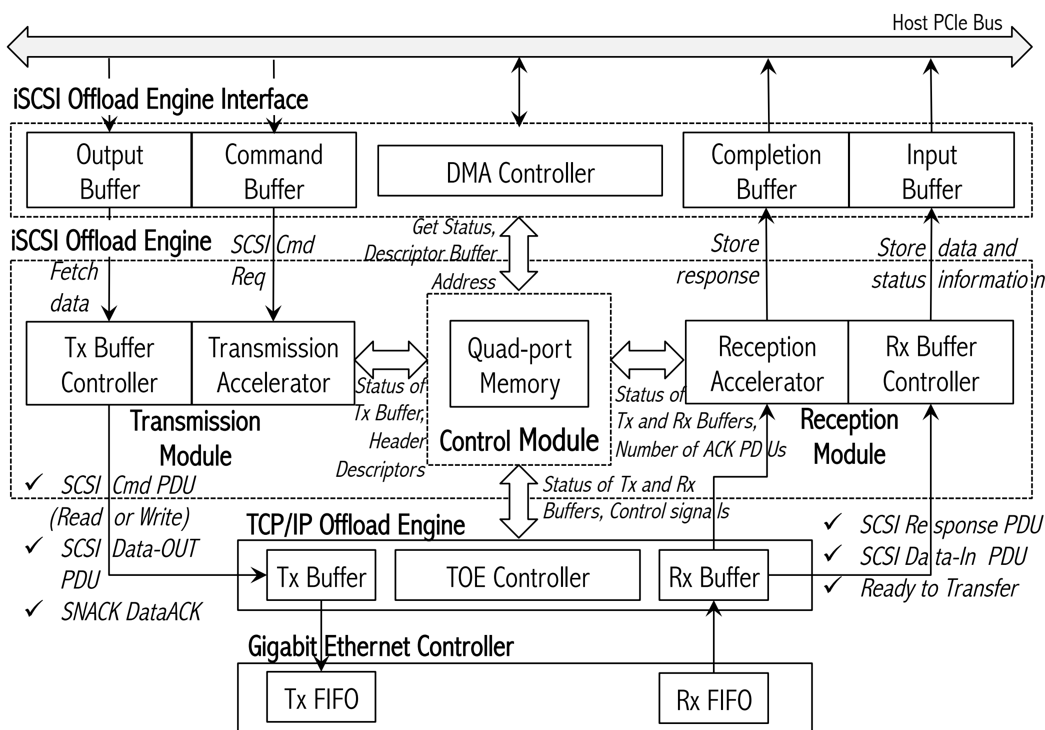

Figure 5.6: The flow of information between modules in the iSCSI Offload Engine.

5.1.5 Modification of the Open-iSCSI Initiator

Traditionally, Open-iSCSI is partitioned into kernel and user parts, which implements iSCSI data path (Read and Write), and the control plane, respectively. The interface between these two parts is implemented using Netlink sockets. The SCSI subsytem in the Linux kernel is divided into three levels: upper, middle and low-level drivers. The task of the upper level is to take requests from outside of the SCSI subsystem, and turn them into actual SCSI requests. The requests are passed down to the middle level (known as SCSI Mid-Level, SML), which handles support for file system, bus scanning, queuing of commands, error handling, etc. The low level drivers (LLD) transfer commands, data, status, messages etc. between initiator and the target.

We modified Open-iSCSI's data-path to bypass some of the SCSI functions and TCP/IP layers in the Linux kernel. The Open-iSCSI spawns two threads for every connection in a session: a transmit thread (tx_thread) and a receive thread (rx_thread). Fig. 5.7 shows an exemplary unmodified and modified data-paths for creating a SCSI Command by the tx_thread. First, the SCSI Mid-Level passes commands to the low level drivers through queuecommand() call. Then, the initiator generates unique Initiator Task Tag (ITT) and allocates memory for a new command initialized with it (a). The PDU fields are then prepared and stored in the memory (b). A new command is added to the linked list of all pending commands (c), and tx_thread is woken up to send the PDU to a target (d). Then, a TCP routine is called to send a SCSI Command PDU to a target (e).

The tx_thread and rx_thread data-paths are modified to bypass the processing of T2, T6 and T8 PDUs, and R2, R6 and R8 PDUs, respectively. In the transmission path, a command is first identified by its opcode and forwarded either to unmodified data-path (tx_thread) or to the new iSCSI agent - offload_engine_agent (f), which is responsible for performing communication with iSCSI Offload Engine. Then, a new request is forwarded to the sys_socketcall to be transmitted to a target through either (g) iSCSI_OE_socketcall (T2, T6, T8) or (h) TOE_socketcall (T1, 3-5, 7). In both cases, the Linux TCP/IP stack in the sys_socketcall is bypassed, by which we eliminated copying of user data to the socket buffer (a kernel copy). Instead, we translate the virtual address of the user's data into a physical address by using get_user_pages and kmap functions. The address is sent to iSCSI Offload Engine, which is followed by the DMA request. The requests are created and pushed into the Command Buffer.

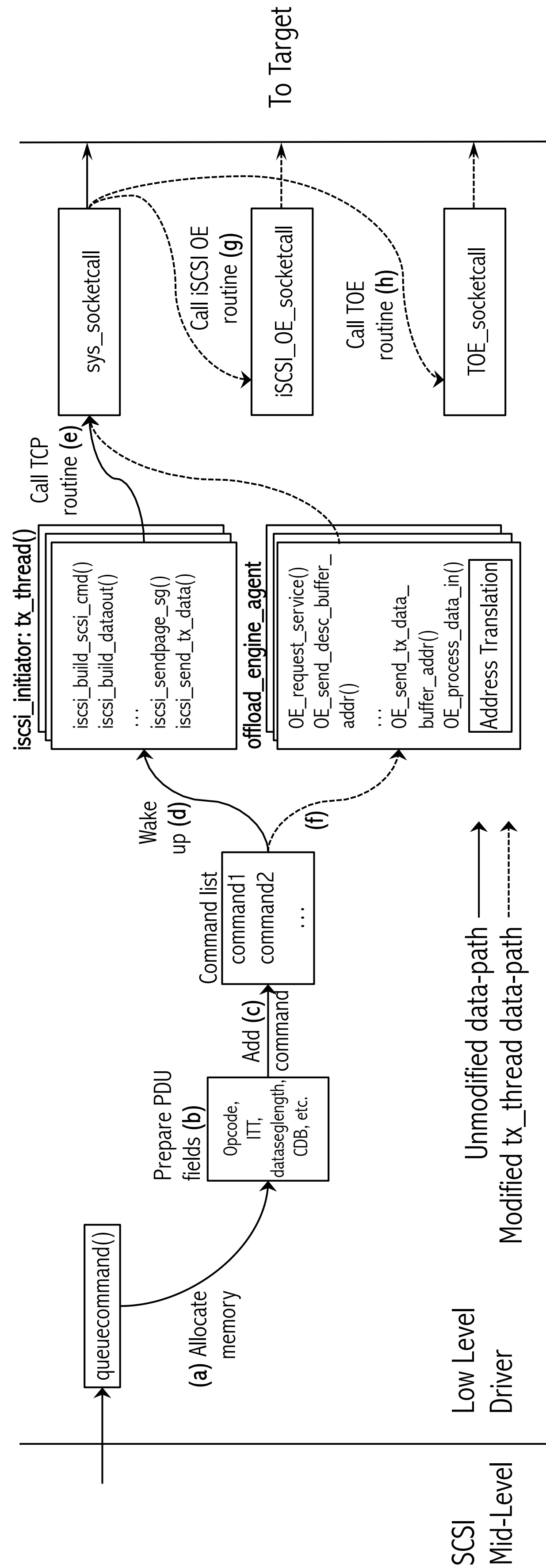

Figure 5.7: Unmodified and modified data-paths for creating a SCSI Command by tx_thread.

5.2 Implementation Results and Analysis

5.2.1 iSCSI Offload Engine Board

Design of our iSCSI Offload Engine is best suited for new Xilinx platforms, such as Virtex-6 HXT or Virtex-7 FPGAs, which contain all the necessary hardware for high-bandwidth and high-performance applications. However, the functionality of the proposed iSCSI Offload Engine is verified on the ML605 board, which is equipped with the Virtex-6 XC6VLX240T-1FFG1156 FPGA [87]. Our test platform includes the Multi-port memory controller for accessing the external DDR3 memory. It is connected with host PC with 64-bit PCI Express x4, with transfer rate of 16 Gbps in each direction. The synchronization between the CPU and the FPGA is performed by the PCIe Message Signaled Interrupts (MSI). This allows an FPGA task to wait for a data being produced by a software task and inversely.

The board has only a 1000-BASE Ethernet interface, hence additional Dual SFP+ FMC [81, 82] and 10GbE SFP+ transceiver are required to achieve throughput of over 1 Gbps. The Dual SFP+ FMC is an FPGA Mezzanine Connector [88] daughter card with two SFP+ connectors, two 10 Gbps physical layer transceivers which provide full PCS, PMA, and XGXS sub-layer functionality. We utilize only one transceiver. The daughter card is connected to the High Pin Count (HPC*) J64 connector of the ML605 board.

As illustrated in Fig. 5.1, the iSCSI PDUs are formed and encapsulated by iSCSI and TCP/IP Offload Engines and sent out through the LogiCORE IP 10-Gigabit Ethernet MAC [89]. The 10-Gigabit link is supported by the LogiCORE IP XAUI core [90] using a SFP+ cable. The iSCSI Offload Engine is clocked at the standard Ethernet interface frequency of 156.25 MHz, which allows fully synchronous and lowest latency data exchange with DINIGroup TCP/IP Offload Engine [13] and the MAC.

5.2.2 Elapsed time of main operations

In order for a network adapter to achieve the throughput of approximately 10 Gbps, it has to be able to process a 1500-byte packet in 1.2 μs. Table 5.2 shows elapsed time of main operations processed in iSCSI Offload Engine for a 1500-byte packet with data digests enabled. We design our iSCSI Offload Engine to interface DINIGroup's TCP/IP Offload Engine [13], which works at the full 10 GbE line rate. Input to output packet latency of the TOE is less than 1 μs, however elapsed time of some operations are already included in elapsed time of iSCSI Offload Engine, such as fetching and storing data from/to host memory and DMA Initialization. These operations require 58% of total time for transmission, and 53% for reception processing. Thus, elapsed time for TCP/IP processing is only 0.21 μs for transmission, and 0.23 μs for reception. The total elapsed time for transmitting a 1500-byte packet is 1.449 μs, and receiving 1.538 μs.

Table 5.2: Elapsed time of main operations processed in the iSCSI Offload Engine for a 1500-byte data packet.

	Hardware Operation	**Elapsed time (μs)**
Transmission Module	(1) DMA Initialization	.045
	(2) Fetching descriptors from host memory and Header Generation	.038
	(3) Header Digest Generation	.038
	(4) Fetching data from host memory and Data Digest Generation	1.162
DINIGrp TOE [13]	(5) TCP/IP Processing and storing a packet into network interface	.21
	Total:	**1.449**
DINIGrp TOE [13]	(1) Fetching a packet from network interface and TCP/IP Processing	.23
Reception Module	(2) Parsing header and Header Digest Generation and validation	.099
	(3) DMA Initialization	.045
	(4) Data Digest Generation and validation, storing data into host memory	1.162
	Total:	**1.538**

5.2.3 CPU Utilization and Throughput

Fig. 5.8 shows CPU utilization and throughput of write micro-benchmarks of three implementations: Open-iSCSI running on Intel Core2 CPU 2.40 GHz with 8 GB of RAM, Chelsio T110 iSCSI ASIC-based HBA with CRC enabled (the results are published in [45]), and our iSCSI Offload Engine. We ran the same set of micro-benchmarks (as discussed in Section 2.3.4) for several thousand times with I/O sizes ranging from 128 bytes to 128 KB. We used Ethernet standard Maximum Transmission Unit (MTU) of 1500 bytes. The processing cost of read micro-benchmarks is very similar to write micro-benchmarks, with slightly lower throughput for reading process.

The average CPU utilization of software-based Open-iSCSI varied from 33% to 55% according to a write size. The iSCSI Offload Engine exhibits very low utilization of approximately 3% on the host CPU, which is 10-15 times reduction compared with Open-iSCSI implementation on

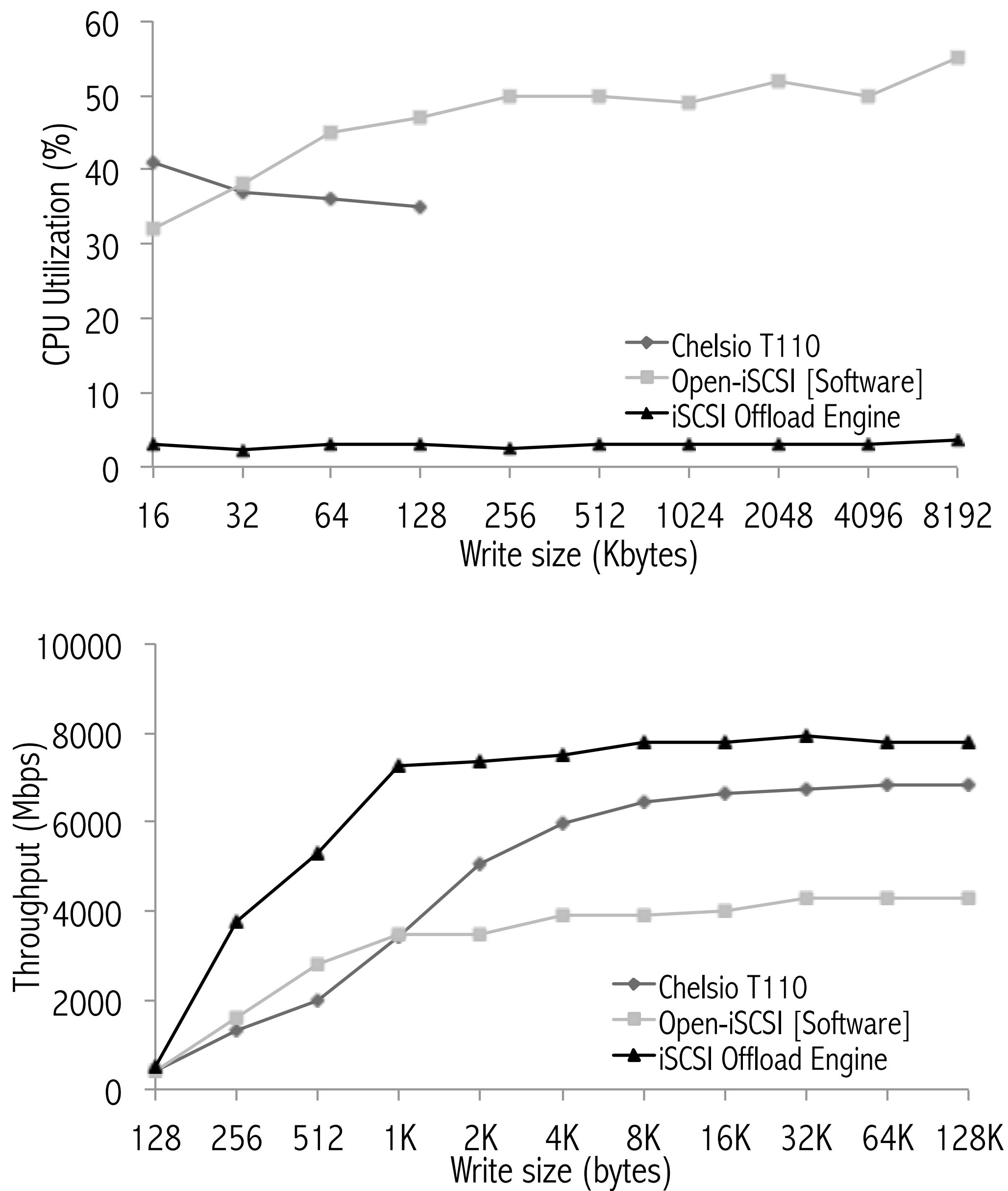

Figure 5.8: Comparison of throughput and CPU utilization of write micro-benchmarks for 1500 bytes MTU.

Intel Core2 CPU 2.40 GHz, and 10 times reduction compared with Chelsio T110. Unfortunately, we were unable to acquire the host CPU Utilization for the Chelsio T110 for I/O workload sizes higher than 128 KB. Also, from [45] it is not clear why Chelsio T110 exhibits such high CPU Utilization on the host. However, we think it is because Chelsio T110 only offloads expensive byte touching operations, such as header and data digests generation/checking, while all other related tasks are performed on the host CPU. Our iSCSI Offload Engine additionally processes related non-data transfer functions on an FPGA, which results in decreased number of instructions and interrupts on the host CPU. Additionally, delayed SNACKs and direct data placement in the host memory also decreased the number of interrupts.

There has been significant increase in throughput when iSCSI is offloaded to hardware. Fig. 5.8 shows the overall throughput for different values of write I/O workload size. The software-based Open-iSCSI is executed on 15 times higher clock frequency (2.40 GHz) than iSCSI Offload Engine. However, the maximum transmission throughput of iSCSI Offload Engine is 7.81 Gbps, while the reception throughput is 7.34 Gbps. The results show 2 times speedup over software-based Open-iSCSI. One of the principal reasons why iSCSI Offload Engine achieves higher throughput is because large number of operations are executed in parallel. Specifically, the Open-iSCSI requires 2.15 cycles *per byte* to generate a CRC value for 8KB of data, while our CRC Generation Unit requires 1 cycle for *eight bytes*. Thus, our CRC Generation Unit requires 17 times less cycles to generate a CRC value than Open-iSCSI.

5.2.4 Reconfiguration time

We measured the time required to upload full bitstream and bitstreams of Partial Reconfiguration (PR) modules for the CRC Generation Unit. We used JTAG to upload both bitstreams on the specified FPGA board. The configuration time depends on the size of a bitstream and Test Clock frequency (TCK) for boundary-scan operations. Fixed number of clock cycles required for pre- and post-processing while programming an FPGA is also included in configuration time as specified in [91], while minimum TCK frequency was 15 MHz. The size of the full configuration bitstream is 9 MB, and the size of PR bitstream is 43 KB. The time to upload these two bistreams is 6.15 s and 0.03 s, respectively. Thus, uploading the PR bitstream is 205 times faster than the time required to upload full bitstream. This feature ensures fast adaptability to new CRC Standards in the future of iSCSI protocol.

5.2.5 Resource Utilization

Table 5.3 shows resource utilization on Virtex-6 XC6VLX240T FPGA. The receive and transmit buffers are configurable 4KB and 64KB buffers mapped onto dedicated on-chip Block RAMs. The CRC Generation Unit requires only 540 LUTs, and 4 dual-port BRAM for holding contents of its pre-computed remainders. The small resource footprint indicates that multiple instances

Table 5.3: Resource utilization of two modules on Virtex-6 XC6VLX240T FPGA. The iSCSI Engine does not contain resource utilization of DINIGrp TOE.

Module	Buffer (KB)	Slices (%)	LUTs (%)	FFs (%)	BRAMs (%)
iSCSI Engine	4 64	5983 (16) 5983	15.7k (10) 15.7k	8.9k (3) 8.9k	24 (3) 52 (6)
DINIGrp TOE [13]	4 64	2742 (3) 2742	7k (4.6) 7k	4k (1.3) 4k	6 (.7) 34 (4)

can be used, thus increasing performance of the system.

5.2.6 Comparison to Related Work

We compare our work with two other attempts to offload iSCSI to hardware [14, 44], which are discussed in Related Work. Han-Chiang Chen et al. [14] has significantly lower throughput than our Offload Engine. Their method is to execute iSCSI on a Xilinx FPGA embedded platform, without any hardware accelerated parts. This method has very low CPU utilization, but throughput is limited by the low frequency of PowerPC 405 processor and does not pose technological challenge. The method of Chung-Ho Chen et al. [44] consists of offloading data-transfer iSCSI functions by using C-to-HDL translation process. The difference in design is that our iSCSI Offload Engine offloads not only data transfer functions, but also related non-data functions such as SCSI Command, SCSI Response, R2T and SNACK request. This has contributed to faster processing of requests and release of resources. Even though our iSCSI Offload Engine uses higher capacity host bus and higher clock frequency, it has higher level of parallelism since we used more CRC Generation Units. The architecture of CRC circuit in Chung-Ho Chen et al. [44] uses simple linear feedback shift register, which is less efficient than the architecture of our CRC circuit [83]. Thus, our initiator is able to achieve 7 times higher throughput. The CPU utilization is similar as in Han-Chiang Chen et al. [14] and Chung-Ho Chen et al. [44], since both offload iSCSI to hardware in some terms.

5.3 Summary

The IP-based storage systems often require bandwidth intensive access to storage devices, thus they exhibit high CPU utilization and low throughput when executed in a principally software implementation. This is especially evident for multi-Gbps networks where the impact of computational overhead is so pronounced that the current state of the art processors cannot take advantage of the capacity of the network. We address this problem by proposing new

iSCSI Offload Engine architecture for high data rate storage networking. Based on our analysis of open source Open-iSCSI initiator, we offload the most computationally intensive and the most executed functions in a common case scenario, while other functions are implemented in a modified Open-iSCSI initiator on a general purpose processor. Our architecture overcomes the performance limitations imposed by a single processor which runs on 15x higher operating frequency than our accelerator. It exhibits very low CPU utilization of approximately 3% on the host CPU, which is 10-15x reduction compared with software implementation. The maximum transmission throughput is 7.81 Gbps, while reception throughput is 7.34 Gbps, which is 2 times speedup over software. The new architecture also shows comparable performance with Chelsio T110 ASIC-based HBA, and has more flexibility.

Chapter 6

Conclusions

In order to enable higher levels of agility, programmability and flexibility of hardware-based accelerators we presented a methodology for designing non-adaptable and fully-adaptable cyclic redundancy checks accelerators, which are based on a multiple table-based algorithm. We also studied behavior of software-based Open-iSCSI initiator and analyzed its network traffic based on several cases. We proposed new iSCSI Offload Engine architecture for processing iSCSI data transfer and related non-data functions on an FPGA based adapter.

6.1 Concluding Remarks

In recent years, the growth of network traffic is increasing with inexorable certainty. It has been driven by a widespread adoption of smartphones, tables, video content and exchange of vast quantities of data over Internet in general. Today, the major concern is that this volume of network traffic on the Internet has begun outpacing server capacity to manage incoming data. Storage systems are crucial components which provide high levels of data integrity and availability of the critical data. The iSCSI protocol defines one approach for accessing and transporting data over commonly utilized TCP/IP infrastructure. The protocol ensures high data integrity through header and data digests in the specific iSCSI Protocol Data Units. However, the processing of iSCSI digests is considered to be the most computationally intensive part of the iSCSI protocol processing.

There are three main approaches to building IP-based storage systems. The first approach is by using software-based TCP/IP stack. This approach relies on the belief that the performance will scale with ever-increasing CPU speed. However, after a recent paradigm change to multicore architectures, where the CPU speed doesn't exceeds 3 GHz, it is not clear how it will effect performance of sequential processes. One such process is cyclic redundancy checks. The second approach is to *offload* the entire TCP/IP stack onto a specialized hardware, which is called TPC/IP Offload Engines (TOE). This approach reduces the TCP copy-and-checksum and interrupt overheads. The third approach is HBA approach, which have a specific storage

transport interface and it is aware of protocol semantics. All three approaches have been tried and an interesting results have been reported. The software approach has the highest flexibility, but it cannot guarantee significant increases in performance in the future. On the other hand, TOE and HBA approaches reported an interesting performance gain, however they have little to no flexibility to future changes in networking infrastructures. Thus, the trade-off is between flexibility and performance.

In this dissertation, we first addressed the problem of efficient implementation of one of major bottlenecks in an iSCSI protocol implementation, which is the generation of Cyclic Redundancy Checks. We propose a methodology for designing two types of CRC accelerators, which we call *non-adaptable* and *fully-adaptable* CRC accelerators. The accelerators are based on a table-based algorithm which has never been used in hardware implementations. The reason is because it is believed that operational speed of current reconfigurable technologies is not enough to obtain significant performance results. We prove that this approach can be successfully implemented on an FPGA and achieve significant performance improvements over related work. We focus on enhancing throughput, the amount of processed bits at the time and the flexibility to adapt to different applications. Secondly, we analyzed the implementation of open-source Open-iSCSI software and its network traffic in the most common scenarios. We identified the most commonly used functions, and we measured CPU utilization and throughput. Based on this analysis, we proposed new architecture called iSCSI Offload Engine, which offloads data transfer and related non-data functions to an FPGA based adapter. Functions which do not affect performance were implemented in a modified Open-iSCSI initiator on a general purpose processor.

Our results show that deploying more than one type of a *computing engine* can satisfy the current and future performance demands. By offloading certain tasks to an FPGA accelerator, we proved that it is possible to obtain high throughput and relieve a CPU of high computational burden. It appeared that combining two technologies, general purpose processor and an FPGA, enables dramatic benefits in the ever-present demand for a greater computational performance. The FPGAs have the computing performance which can be used with general purpose processors to enable greater level of flexibility and agility. The accelerators can be designed to adapt to different application domains, which extend their usability, decreases area utilization and eliminates the time required for re-design and re-programming. Our research is one step into a direction of integrating these two technologies, since it is very likely that their significance will increase as time progresses. The most important feature of this integration is the *flexibility* of FPGAs, which will most probably play important role in the future *programmable* networks. In our research, we have shown that it is feasible to deploy such an architecture on the example of CRC accelerators and iSCSI Initiator.

Abbreviations and Acronyms

10GE 10-Gigabit Ethernet
ALU Arithmetic Logic Unit
ASIC Application Specific Integrated Circuit
ASIP Application-Specific Instruction Processor
BLE Basic Logic Element
BRAM Block Random Access Memory
CGM CRC Generation Module
CGRA Coarse-Grained Reconfigurable Architecture
CLB Configurable Logic Blocks
CMOS Complementary MetalâĂŞOxideâĂŞSemiconductor
CPU Central Processing Unit
CRC Cyclic Redundancy Checks
DARPA Defense Advanced Research Projects Agency
DDR2 Double Data Rate
DMA Direct Memory Access
DRAM Dynamic Random Access Memory
DSP Digital Signal Processor
EC European Commission
EU European Union
FIFO First-in First-out
FIR Finite Impulse Response
FMC FPGA Mezzanine Connector
FPGA Field Programmable Gate Array
FU Functional Unit
GF Galois Field
GFMAC Galois Field Multiplication and Accumulation
GPCPU General Purpose Central Processing Unit
GPGPU General Purpose Graphic Processing Unit
GPP General-Purpose Processors
GPU Graphic Processor Unit

HBA Host Bus Adapters
HPC High Performance Computing
HPC* High Pin Count
HPRC High Performance Reconfigurable Computing
IC Integrated Circuits
IP Internet Protocol
IP* Intellectual Property
IPTO Information Processing Techniques Office
iSCSI Internet Small Computer System Interface
ITT Initiator Task Tag
JTAG Joint Test Action Group
LC Logic Cells
LFSR Linear Feedback Shift Register
LUN Logical Unit Number
LUT Lookup Table
MAC Media Access Controller
MAC* Multiply-Accumulate
MSI Message Signaled Interrupts
MTU Maximum Transmission Unit
NRE Non-Recurring Engineering
PAL Programmable Array Logic
PCI Peripheral Component Interconnect
PDU Protocol Data Unit
PLA Programmable Logic Array
PLD Programmable Logic Device
PMA Physical Medium Attachment Sublayer
PSC Physical Coding Sublayer
R2T Ready to Transmit
RAM Random Access Memory
RTL Register Transfer Level
RTR Real-Time Recovery
SAN Storage Area Networks
SNACK Selective Negative Acknowledgment
SRAM Static Random-Access Memory
TCP/IP Transmission Control Protocol/Internet Protocol
TGM Table Generation Module
TOE TCP/IP Offload Engines
TTT Target Transfer Tag

XAUI eXtended Attachment Unit Interface

XGXS 10 Gigabit Ethernet Extended Sublayer

Bibliography

[1] Cisco's VNI Forecast Projects the Internet Will Be Four Times as Large in Four Years. http://newsroom.cisco.com/press-release-content?type=webcontent&articleId=888280, May 2012.

[2] Padmasree Warrior. The Future of IT. http://blogs.cisco.com/news/the-future-of-it/, January 2013.

[3] Yen-Kuang Chen. Challenges and opportunities of internet of things. In *17th Asia and South Pacific Design Automation Conference (ASP-DAC), 2012*, pages 383–388, 2012.

[4] Dave Evans. The Internet of Everything: How More Relevant and Valuable Connections Will Change the World. http://www.cisco.com/web/about/ac79/docs/innov/IoE.pdf, 2012.

[5] David Storm. 100Gbps and beyond: What lies ahead in the world of networking. http://arstechnica.com/information-technology/2013/02/100gbps-and-beyond-what-lies-ahead-in-the-world-of-networking/, February 2013.

[6] Bob Wheeler. 10 Gigabit Ethernet In Servers: Benefits and Challenges. http://www.hp.com/products1/serverconnectivity/adapters/ethernet/10gbe/infolibrary/10GbE_White_Paper.pdf, 2005.

[7] Andrew Gallatin, Jeff Chase, and Ken Yocum. Trapeze/IP: TCP/IP at Near-Gigabit Speeds. *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference, Monterey, California, USA*, June 1999.

[8] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. Tcp onloading for data center servers. *Computer*, 37(11):48–58, 2004.

[9] Server Network I/O Acceleration, 2004.

[10] Zhong-Zhen Wu and Han-Chiang Chen. Design and Implementation of TCP/IP Offload Engine System over Gigabit Ethernet. In *Proceedings.15th International Conference on Computer Communications and Networks, 2006. ICCCN 2006.*, pages 245–250, 2006.

[11] The Chelsio Terminator 3 ASIC, Third-generation 10Gb Ethernet Unified Wire Engine for iSCSI, RDMA and TCP/IP Applications. http://www.chelsio.com/assetlibrary/products/T3_Unified_Wire_Eng_WP.pdf.

[12] Hankook Jang, Sang-Hwa Chung, and Dae-Hyun Yoo. Design and implementation of a protocol offload engine for TCP/IP and remote direct memory access based on hardware/software coprocessing. *Microprocessors and Microsystems*, 33(5-6):333 – 342, 2009.

[13] DINIGroup TCP Offload Engine IP: For Latency Critical, FPGA-based Embedded Networking Applications. http://www.applistar.com/wp-content/uploads/2012/06/TOE_Brief_v092.pdf, April 2012.

[14] Zheng-Ji Wu Han-Chiang Chen and Zhong-Zhen Wu. Implementation of Offloading the iSCSI and TCP/IP Protocol onto Host Bus Adapter. *Conference on Mass Storage Systems and Technologies*, 2006.

[15] Software Defined Networking: A new paradigm for virtual, dynamic, flexible networking, October 2012.

[16] Sanjeev Mervana. The Programmable Network: End-to-End Visualization and Control. http://blogs.cisco.com/news/the-programmable-network-end-to-end-visualization-and-control/, May 2013.

[17] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet Small Computer Systems Interface(iSCSI), RFC 3720.

[18] H.M. Khosravi, Abhijeet Joglekar, and R. Iyer. Performance characterization of iSCSI processing in a server platform. In *Performance, Computing, and Communications Conference, 2005. IPCCC 2005. 24th IEEE International*, pages 99–107, 2005.

[19] Jerry Daugherty. Understanding iSCSI Digests: Accurately Evaluating the Cost and Risk of Disabling Digests. http://www.jdsu.com/ProductLiterature/Understanding-iSCSI-Digests-white-paper-30162803.pdf, 2009.

[20] Prasenjit Sarkar, Sandeep Uttamchandani, and Kaladhar Voruganti. Storage Over IP: When Does Hardware Support Help? In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 231–244, Berkeley, CA, USA, 2003. USENIX Association.

[21] Open iSCSI, Open source iSCSI Initiator implementation. http://www.open-iscsi.org/.

[22] T.-B. Pei and C. Zukowski. High-speed parallel CRC circuits in VLSI. *IEEE Transactions on Communications*, 40(4):653–657, 1992.

[23] T.V. Ramabadran and S.S. Gaitonde. A tutorial on CRC computations. *Micro, IEEE*, 8(4):62–75, 1988.

[24] C. Borrelli. IEEE 802.3 Cyclic Redundancy Check. http://www.xilinx.com/support/documentation/application_notes/xapp209.pdf, 2001.

[25] C. Toal, K. McLaughlin, S. Sezer, and Xin Yang. Design and Implementation of a Field Programmable CRC Circuit Architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(8):1142–1147, 2009.

[26] F. Monteiro, A. Dandache, A. M"Sir, and B. Lepley. A fast CRC implementation on FPGA using a pipelined architecture for the polynomial division. In *the 8th IEEE International Conference on Electronics, Circuits and Systems, 2001. ICECS 2001*, volume 3, pages 1231–1234 vol.3, 2001.

[27] M.E. Kounavis and F.L. Berry. A Systematic Approach to Building High Performance Software-based CRC generators. In *Proceedings of the 10th IEEE Symposium on Computers and Communications*, pages 855–862, 2005.

[28] C. Mucci, L. Vanzolini, I. Mirimin, D. Gazzola, A. Deledda, S. Goller, J. Knaeblein, A. Schneider, L. Ciccarelli, and F. Campi. Implementation of Parallel LFSR-based Applications on an Adaptive

DSP featuring a Pipelined Configurable Gate Array. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 1444–1449, March.

[29] Allen Kent and James G. Williams Rosalind Kent. *Encyclopedia of Microcomputers: Volume 6 - Electronic Dictionaries in Machine Translation to Evaluation of Software: Microsoft Word Version 4.0*. CRC press, 1 edition, June 1990.

[30] J.M. Simmons. A strategy for designing error detection schemes for general data networks. *Network, IEEE*, 8(4):41–48, 1994.

[31] Richard E. Blahut. *Algebraic Codes for Data Transmission*. Cambridge University Press, 1 edition, March 2003.

[32] John Gill. EE 387 Algebraic Error Control Codes, lecture notes. `http://www.stanford.edu/class/ee387/`, Autumn 2012.

[33] G. Campobello, G. Patane, and M. Russo. Parallel CRC realization. *IEEE Trans. Comput.*, 52(10):1312–1319, October 2003.

[34] D. V. Sarwate. Computation of cyclic redundancy checks via table look-up. *Commun. ACM*, 31(8):1008–1013, August 1988.

[35] Tomas Henriksson and Dake Liu. Implementation of fast CRC calculation. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ASP-DAC '03, pages 563–564, New York, NY, USA, 2003. ACM.

[36] H. Michael Ji and E. Killian. Fast parallel CRC algorithm and implementation on a configurable processor. In *IEEE International Conference on Communications, 2002. ICC 2002.*, volume 3, pages 1813–1817 vol.3, 2002.

[37] A. Akagic and H. Amano. An FPGA Implementation of CRC Slicing-by-N algorithms. Technical Report 319, The Institute of Electronics, Information and Communication Engineers, 2010.

[38] Wireshark network packet analyzer. `http://www.wireshark.org/`.

[39] Linux SCSI target framework. `http://stgt.sourceforge.net/`.

[40] OProfile: system-wide profiler for Linux systems. `http://oprofile.sourceforge.net/`.

[41] The UNH-iSCSI project. `http://unh-iscsi.sourceforge.net/`.

[42] D. Dalessandro, P. Wyckoff, and G. Montry. Initial Performance Evaluation of the NetEffect 10 Gigabit iWARP Adapter. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–7, 2006.

[43] Intilop's 76-nanosecond TOE Based System Etablishes a Record 93% TCP/IP Bandwidth at a Major Customer's 10G Network Deployment. `http://www.sbwire.com/press-releases/10g-toe/tcp-performance/sbwire-156548.htm`, August 2012.

[44] Chung-Ho Chen, Yi-Cheng Chung, Chen-Hua Wang, and Han-Chiang Chen. Design of a Giga-bit Hardware Accelerator for the iSCSI Initiator. In *Proceedings 2006 31st IEEE Conference on Local Computer Networks*, pages 257–263, 2006.

[45] Chelsio Communications T110 10-gigabit HBA: iSCSI HBA Performance Testing by VeriTest, June 2004.

[46] High-Performance Computing: Europe's place in a Global Race, European Commission. `http://ec.europa.eu/information_society/newsroom/cf/document.cfm?action=display&doc_id=891`, 2012.

[47] The Human Brain Project. http://www.humanbrainproject.eu/.

[48] Exascale Computing Study Report. http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ECS_reports.htm, 2008.

[49] P. Kogge et. al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf, 2008.

[50] N. Eicker and Th. Lipper. An accelerated Cluster-Architecture for the Exascale, 2011.

[51] G.E. Moore. The future of integrated electronics. *Electronics Magazine*, 1965.

[52] Katherine Compton and Scott Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Comput. Surv.*, 34(2):171–210, June 2002.

[53] Prasanna Sundararajan. High Performance Computing Using FPGAs. http://www.xilinx.com/support/documentation/white_papers/wp375_HPC_Using_FPGAs.pdf, 2010.

[54] Thomas Lenart. *Design of Reconfigurable Hardware Architectures for Real-time Applications, Modeling and Implementation*. PhD thesis, Lund University, Sweden, 2008.

[55] A.C. Cheng. A Software-to-Hardware Self-Mapping Technique to Enhance Program Throughput for Portable Multimedia Workloads. In *4th IEEE International Symposium on Electronic Design, Test and Applications, DELTA 2008*, pages 356–361, 2008.

[56] Russell Tessier and Wayne Burleson. Reconfigurable Computing for Digital Signal Processing: A Survey. *J. VLSI Signal Process. Syst.*, 28(1/2):7–27, May 2001.

[57] K. Tatas, K. Siozios, and D. Soudris. A Survey of Existing Fine-Grain Reconfigurable Architectures and CAD tools. In Stamatis Vassiliadis and Dimitrios Soudris, editors, *Fine- and Coarse-Grain Reconfigurable Computing*, pages 3–87. Springer Netherlands, 2008.

[58] Carl Ebeling, DarrenC. Cronquist, and Paul Franklin. RaPiD Reconfigurable pipelined datapath. In ReinerW. Hartenstein and Manfred Glesner, editors, *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, volume 1142 of *Lecture Notes in Computer Science*, pages 126–135. Springer Berlin Heidelberg, 1996.

[59] Reiner Hartenstein. Coarse grain reconfigurable architecture (embedded tutorial). In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, ASP-DAC '01, pages 564–570, New York, NY, USA, 2001. ACM.

[60] Arthur Abnous, Hui Zhang, Marlene Wan, George Varghese, Vandana Prabhu, and Jan Rabaey. *The Pleiades Architecture*, pages 327–359. John Wiley and Sons, Ltd, 2002.

[61] H. Singh, Ming-Hau Lee, Guangming Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, 2000.

[62] V. Baumgarte, G. Ehlers, F. May, A. Nuckel, M. Vorbach, and M. Weinhardt. PACT XPP Self-Reconfigurable Data Processing Architecture. *The Journal of Supercomputing*, 26(2):167–184, 2003.

[63] Gerard J.M. Smit, Michel A.J. Rosien, Yuanqing Guo, and Paul M. Heysters. Overview of the Tool-Flow for the Montium Processor Tile. In *International Conference on Engineering of Reconfigurable Systems and Algorithms, ERSA 2004*, pages 45–51. CSREA Press, 2004.

[64] Khaled Benkrid. High Performance Reconfigurable Computing: From Applications to Hardware. *IAENG International Journal of Computer Science*, 35(1), 2004.

[65] Algotronix History. http://www.algotronix.com/people/tom/album.html.

[66] Garp: Combining a Processor with a Reconfigurable Computing Array. http://brass.cs.berkeley.edu/garp.html.

[67] J.M. Moreno, J. Cabestany, E. Cant √âĽě, J. Faura, P. Duong, M.A. Aguirre, and J.M. Insenser. Fipsoc. A Novel Mixed FPGA for System Prototyping. In Andrzej Napieralski, Zygmunt Ciota, Augustin Martinez, Gilbert Mey, and Joan Cabestany, editors, *Mixed Design of Integrated Circuits and Systems*, volume 434 of *The Springer International Series in Engineering and Computer Science*, pages 169–173. Springer US, 1998.

[68] Xilinx. http://www.xilinx.com/about/company-overview/.

[69] Altera. http://www.altera.com/corporate/about_us/.

[70] Lattice Semiconductor. http://www.latticesemi.com/corporate/.

[71] Atmel. http://www.atmel.com/about/corporate/.

[72] Freeman R.H. Carter W., Duong K. et al. A user programmable reconfiguration gate array. *IEEE Custom Integrated Circuits Conference*, pages 233âĂŞ–235, 1986.

[73] Chiu T.L. Guterman D.C., Rimawi I.H. et al. An electrically alterable nonvolatile memory cell using a floating-gate structure. *IEEE Trans. Electron Devices*, 26(4):576–586, 1979.

[74] Chua H.T. Birkner J., Chan A. et al. A very-high-speed field programmable gate array using metal-to-metal antifuse programmable elements. *Micro*, 23(7):561–568, 1992.

[75] Spartan-3 FPGA Family Data Sheet, DS099. http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf, October 29, 2012.

[76] Spartan-6 Family Overview, DS112 (v3.1). http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf, August 30, 2010.

[77] Virtex-4 Family Overview, DS160 (v2.0). http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf, October 25, 2011.

[78] Virtex-5 Family Overview, DS100 (v5.0). http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf, February 6, 2009.

[79] Virtex-6 Family Overview, DS150 (v2.4). http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf, January 19, 2012.

[80] M.E. Kounavis and F.L. Berry. Novel Table Lookup-Based Algorithms for High-Performance CRC Generation. *Computers, IEEE Transactions on*, 57(11):1550–1560, 2008.

[81] Dual SFP+ FMC Module. http://hitechglobal.com/FMCModules/FMC_SFP+.htm.

[82] Xilinx ML605 Board Accessories, List of FMC Modules. http://www.xilinx.com/products/boards_kits/board_accessories.htm.

[83] A. Akagic and H. Amano. Performance analysis of fully-adaptable CRC accelerators on an FPGA. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 575–578, 2012.

[84] A. Akagic and H. Amano. A study of adaptable co-processors for cyclic redundancy check on an fpga. In *International Conference on Field-Programmable Technology (FPT), 2012*, pages 119–124, 2012.

[85] Amila Akagic and Hideharu Amano. High speed CRC with 64-bit generator polynomial on an FPGA. *SIGARCH Comput. Archit. News*, 39(4):72–77, dec 2011.

[86] N. Sawyer and M. Defossez. Quad-Port Memories in Virtex Devices, Application Note XAPP228 (v1.0), September 24, 2002. http://www.xilinx.com/support/documentation/application_notes/xapp228.pdf.

[87] ML605 Hardware User Guide, v1.8. http://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf, October 2, 2012.

[88] Raj Seelam. I/O Design Flexibility with the FPGA Mezzanine Card (FMC), Xilinx WP315 (v1.0), August 19, 2009.

[89] Xilinx LogiCORE IP 10-Gigabit Ethernet MAC v11.2, Xilinx UG773, October 19, 2011.

[90] Xilinx LogiCORE IP XAUI v10.2, DS266 January 18, 2012.

[91] Virtex-6 FPGA Configuration, UG360 (v3.2), November 1, 2010.

Publications

Related Papers

Journal Papers

[1] Akagic Amila and Hideharu Amano, Design and implementation of IP-based iSCSI Offload Engine on an FPGA, IPSJ Transactions on System LSI Design Methodology (TSLDM11), Vol. 6, No. 1, pp. – , Aug 2013.

[2] Akagic Amila and Hideharu Amano, High-Speed Fully-Adaptable CRC Accelerators, IEICE TRANS. INF. & SYST., Vol. E96, No. 6, pp. – , Jun 2013.

International Conference Papers

[3] Akagic Amila and Hideharu Amano, A study of adaptable co-processors for Cyclic Redundancy Check on an FPGA, *International Conference on Field-Programmable Technology (FPT), 2012*, pp. 119 – 124, Dec 2012, Seoul, (South) Korea

[4] Akagic Amila and Hideharu Amano, Performance analysis of fully-adaptable CRC accelerators on an FPGA, *22nd International Conference on Field Programmable Logic and Applications (FPL), 2012*, pp. 575–578, Sep 2013, Oslo, Norway

[5] Akagic Amila and Hideharu Amano, High Speed CRC with 64-bit generator polynomial on an FPGA, *International Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies, HEART 2011*, Jun 2011, Imperial College London, London, UK

[6] Akagic Amila and Hideharu Amano, Performance Evaluation of Multiple Lookup Tables Algorithms for generating CRC on an FPGA, *International Symposium on Access Spacess, ISAS 2011*, Jun 2011, Yokohama, Japan

Domestic Conference Papers and Technical Reports

[7] Marijana Cosovic, Akagic Amila and Zdenka Babic, Performance Analysis of Modular Multipliers Implementations On FPGA, *Scientific Professional Symposium INFOTEH-JAHORINA*, Vol. 10, Ref. E-VI-6, pp. 869–873, March 2011. (In Bosnian)

[8] Akagic Amila and Hideharu Amano, Multiple Table Lookup Implementation of Error Correction on an FPGA, *Design, Automation and Test in Europe, DATE 2011*, Grenoble, France, March 2011.

[9] Akagic Amila and Hideharu Amano, An FPGA Implementation of CRC Slicing-by-N algorithms, *IEICE Technical Reports (RECONF)*, Vol. 110, No. 319, pp. 19–24, Nov 2010.

[10] Akagic Amila Dusanka Boskovic and Novica Nosovic, Implementation of MIC-MAC-1 Hypothetical Processor on an FPGA, *Scientific Professional Symposium INFOTEH-JAHORINA*, Vol. 9, Ref. E-V-12, pp. 748–751, March 2010. (In Bosnian)

[11] Akagic Amila and Novica Nosovic, GRID Security Architecture, *Scientific Professional Symposium INFOTEH-JAHORINA*, March 2007. (In Bosnian)