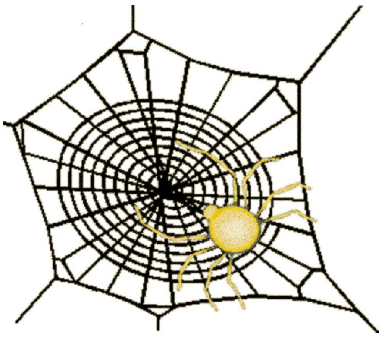
Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl

2003

MSc THESIS

Designing TCP/IP Functions In FPGAs

Weidong Lu



CE-MS-2003-09

Abstract

The increasing popularity of the Internet stimulates an explosive growth of the data transmitted on the Internet as well as the dramatic increase of the transmission speeds. As a result, the TCP/IP processing has become a bottleneck. Traditional software-based TCP/IP processing on general-purpose processors (GPPs) is no longer able to keep pace with network wire speeds. Consequently, there is an urgent need to design performance-critical TCP/IP functions as special functional units to accelerate the processing speeds and to offload the processing tasks from GPPs. Such functional units performing micro-level functions can be implemented on field-programmable gate arrays (FPGAs). FPGAs as programmable hardware devices are particularly suitable to encompass both high processing speeds and flexibility to meet the quickly changing Internet. In this thesis, an in-depth survey of the micro-level TCP/IP functions is first carried out and subsequently, some typical network services built upon these micro-level functions are identified. Based on profiling results, two micro-level functions, namely checksum and cyclic redundancy check (CRC), are selected as computational intensive functions to be implemented in FPGAs. In addition, the table lookup, which is a critical factor for the network address translation (NAT) service is implemented. The checksum calculation is implemented based on 16-bit one's complement adders. The 32-bit parallel calculation of CRC is implemented based on the Linear Feedback Shift Registers (LFSRs). An important contribution is a novel CRC update scheme to further improve the performance of CRC calculation. The novel scheme is based on the observation that only a small portion of the forwarded packet residing in the beginning of the frame is changed and the remaining frame is unchanged. Therefore, the CRC update only calculates the changed part, and afterwards, performs a single step update to obtain the new CRC code. Accordingly, the number of cycles required to calculate the CRC code is dramatically reduced and the frame transmission throughput can be improved. Finally, the implementation of NAT table lookup using block SelectRAMs as Content Addressable Memories (CAMs) on Xilinx FPGAs is described.

Faculty of Electrical Engineering, Mathematics and Computer Science

Designing TCP/IP Functions In FPGAs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Weidong Lu
born in Nanjing, China

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Designing TCP/IP Functions In FPGAs

by Weidong Lu

Abstract

The increasing popularity of the Internet stimulates an explosive growth of the data transmitted on the Internet as well as the dramatic increase of the transmission speeds. As a result, the TCP/IP processing has become a bottleneck. Traditional software-based TCP/IP processing on general-purpose processors (GPPs) is no longer able to keep pace with network wire speeds. Consequently, there is an urgent need to design performance-critical TCP/IP functions as special functional units to accelerate the processing speeds and to offload the processing tasks from GPPs. Such functional units performing micro-level functions can be implemented on field-programmable gate arrays (FPGAs). FPGAs as programmable hardware devices are particularly suitable to encompass both high processing speeds and flexibility to meet the quickly changing Internet. In this thesis, an in-depth survey of the micro-level TCP/IP functions is first carried out and subsequently, some typical network services built upon these micro-level functions are identified. Based on profiling results, two micro-level functions, namely checksum and cyclic redundancy check (CRC), are selected as computational intensive functions to be implemented in FPGAs. In addition, the table lookup, which is a critical factor for the network address translation (NAT) service is implemented. The checksum calculation is implemented based on 16-bit one's complement adders. The 32-bit parallel calculation of CRC is implemented based on the Linear Feedback Shift Registers (LFSRs). An important contribution is a novel CRC update scheme to further improve the performance of CRC calculation. The novel scheme is based on the observation that only a small portion of the forwarded packet residing in the beginning of the frame is changed and the remaining frame is unchanged. Therefore, the CRC update only calculates the changed part, and afterwards, performs a single step update to obtain the new CRC code. Accordingly, the number of cycles required to calculate the CRC code is dramatically reduced and the frame transmission throughput can be improved. Finally, the implementation of NAT table lookup using block SelectRAMs as Content Addressable Memories (CAMs) on Xilinx FPGAs is described.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2003-09

Committee Members :

Advisor:	Stephan Wong, CE, TUDelft
Member:	Reinder Nouta, CAS, TUDelft
Member:	Sorin Cotofana, CE, TUDelft
Member:	Louis Muller, CE, TUDelft

To my parents

Contents

List of Figures viii

List of Tables ix

Acknowledgments xi

1 Introduction **1**
- 1.1 The Internet . . . 1
- 1.2 Challenges . . . 2
- 1.3 Research Questions and Methodology . . . 4
- 1.4 Thesis Overview . . . 5

2 Background **7**
- 2.1 TCP/IP Protocol Suite . . . 7
  - 2.1.1 TCP/IP Model . . . 7
  - 2.1.2 Addressing . . . 10
  - 2.1.3 Communication Between Layers . . . 11
- 2.2 TCP/IP Functions . . . 13
- 2.3 Design Methodology . . . 17
- 2.4 Conclusions . . . 18

3 TCP/IP Functions **19**
- 3.1 Selection of TCP/IP Functions . . . 19
- 3.2 Checksum Calculation . . . 23
- 3.3 Cyclic Redundancy Check . . . 28
  - 3.3.1 Introduction . . . 28
  - 3.3.2 A Serial Implementation of CRC . . . 31
  - 3.3.3 The Parallel Implementation of CRC . . . 34
- 3.4 Fast CRC Update . . . 39
- 3.5 Network Address Translation . . . 45
- 3.6 Conclusions . . . 48

4 Implementations and Results **51**
- 4.1 Checksum . . . 51
- 4.2 Cyclic Redundancy Check . . . 54
- 4.3 Fast CRC Update . . . 59
- 4.4 Network Address Translation . . . 63
- 4.5 Conclusions . . . 66

5 Conclusions **67**
5.1 Summary 68
5.2 Main Contributions 68
5.3 Future Research Directions 69

Bibliography **72**

A F^{32} matrix for CRC-32 **73**

B Proof of Equation (3.23) **75**

C 16×56 CAM Waveform **77**

List of Figures

1.1 A generalized view of a part of the Internet[9] 2
1.2 Data communication on the Internet 3
1.3 Transmission speed vs. Moore's law[23] 4

2.1 OSI and TCP/IP model. 8
2.2 The specific protocols on each layer[24]. 8
2.3 IP address classes. 11
2.4 Data communications between layers[3]. 12
2.5 Data communications, from TCP/IP model's point of view. 13
2.6 Design flow. 17

3.1 Breakdown of operations processing times[13]. 20
3.2 Profile for IPv4 packet forwarding on the IXP1200 [5]. 21
3.3 Profile for packet forwarding on Pentium III cycle counters [14]. 22
3.4 Maximum bandwidth comparison on a GPP and an Intel IXP1200 [6]. . . 22
3.5 Clock rate and throughput comparison between four CRC implementations. 23
3.6 Checksum calculation for IP packet header. 24
3.7 4-bit lookahead carry generator[18]. 26
3.8 4-bit lookahead carry generator, a detailed diagram [18]. 26
3.9 16-bit lookahead carry generator [18]. 27
3.10 16-bit carry-lookahead adder. 27
3.11 Block diagram of checksum calculation. 28
3.12 CRC binary division. 30
3.13 Linear Feedback Shift Register (LFSR). 32
3.14 CRC binary division example, the number of parallel bits w = 4. 35
3.15 Block diagram for parallel calculation of CRC-32. 38
3.16 Parallel CRC for messages with the length not equal to the multiple of 32. 39
3.17 CRC verification and recalculation. 40
3.18 Detailed ethernet frame structure. 41
3.19 Block diagram of fast CRC update circuit. 42
3.20 Number of cycles: comparison. 43
3.21 NAT example. 46
3.22 NAT table. 47
3.23 NAT table lookup. 48

4.1 16-bit carry-lookahead adder. 52
4.2 Block diagram of checksum calculation. 52
4.3 Waveform of checksum calculation. 53
4.4 LFSR for parallel calculation of CRC32. 55
4.5 Testbench waveform for CRC32 parallel calculation. 55
4.6 Area utilization comparison for different number of parallel input bits. . . 57
4.7 Clock rate comparison for different number of parallel input bits. 57

4.8 Throughput comparison for different number of parallel input bits. 57
4.9 The block diagram of 64-bit parallel calculation of CRC-32. 58
4.10 The block diagram of CRC fast update. 59
4.11 The testbench waveform for the parallel CRC calculation of the new frame. 60
4.12 The testbench waveform for CRC fast update. 60
4.13 The block diagram for the transmission of the frame with CRC code. . . . 61
4.14 Throughput comparison between parallel and CRC fast update. 62
4.15 NAT table 64
4.16 CAM 16 × 8 Macro. 64
4.17 Cascaded CAM 16 × 56 Block. 65

C.1 Area utilization comparison for different number of parallel input bits. . . 78

List of Tables

2.1 Addresses for private networks[9]. 11

3.1 CRC generator polynomials in TCP/IP protocol suite. 31
3.2 Three most frequently occurred frame in the real Internet[17]. 43

4.1 Checksum calculation, utilization and performance report. 53
4.2 The VHDL code to perform XOR operation. 55
4.3 Area and static timing reports for different number of parallel input bits. 56
4.4 Fast CRC update area and performance report for different processing unit. 61
4.5 Clock cycles and calculation time comparisons on different frame sizes. . . 61
4.6 CRC update throughput for 64 bytes, 594 bytes and 1518 bytes. 62
4.7 16 × 56 CAM block, utilization and performance report. 65

Acknowledgments

F**irst** and foremost, I would like to thank my advisor, Prof. Stephan Wong, for his guidance, inspiring discussions and revisions of my thesis. He was so patient in reading my thesis, correcting my writing errors, and verifying my proofs. He taught me a lot in how to perform a thesis work. He gave me many suggestions in writing the thesis, too. My thesis can not be completed without his invaluable help.

Next, I would like to acknowledge all the members of my thesis defense committee: Prof. Sorin Cotofana, Prof. Reinder Nouta and Prof. Louis Muller for their helpful comments and suggestions.

I am grateful to Prof. Stamatis Vassiliadis for offering me the opportunity to do my MSc thesis in the Computer Engineering Group, also for his encouragement and kind help during my study here. I would also want to thank all the other members in the Computer Engineering group, especially Bert Meijs for his continuous effort to provide us with stable computer environments.

I want to thank all my colleagues in the CE students room and my Chinese and international classmates for their help and company during my MSc study. I had a very good time with them. I would like to mention Yunfei Wu and Gerard Fossung who are working within the same project with me for their kind help and suggestions.

Finally, I want to thank my parents for their endless love and support during these years. This thesis is dedicated to them. Special thanks are given to my uncle and aunt for their kindly support during my study in the Netherlands.

Delft, The Netherlands
July, 2003

Weidong Lu

Introduction

1

Nowadays, *with the increasing popularity of the Internet, users demand massive bandwidth to transmit increasingly more data and to provide advanced services with high quality. At the same time, the wide deployment of optical fibers enables data transmission at wire speeds. As the transmission speeds are reaching 10 Gbps (OC-192) and heading towards 40 Gbps (OC-768), the bottleneck for the high speed transmissions has become the network data processing speed. Special instances for the network processing tasks are the processing functions in the TCP/IP domain. TCP/IP functions are generally referred to as the data processing functions existing in the lower four layers of the TCP/IP model. They are working together to guarantee robust and effective data communications over the Internet. Traditionally, most of the TCP/IP processing functions are performed by software running on general-purpose processors (GPPs). As the network speeds increased drastically, GPPs become burdened with the large amount of TCP/IP processing. Moreover, the processing speeds of some of these functions, especially those computational intensive functions or those functions with high processing overheads, has lagged behind the network speeds. Accordingly, there is an urgent need to identify those performance-critical TCP/IP functions and accelerate them in order to keep pace with the transmission speeds. A challenge in designing the TCP/IP functions is that the demand for advanced services requires the network devices to support a wide range of applications and protocols, however, these applications and protocols are constantly evolving. Therefore, the designed TCP/IP functions must be flexible and adaptable to possible changes. Several new design techniques are developed that may facilitate the TCP/IP processing, one of which is the utilization of field-programmable gate arrays (FPGAs) in building TCP/IP processing functions. FPGAs can provide high processing speed as well as meet the challenge that to be flexible to possible changes for its gate level reconfigurability. In this thesis, we categorize the TCP/IP processing functions, and implement them in FPGAs.*

This chapter is organized as follows. Section 1.1 presents a brief introduction to the Internet. Section 1.2 discusses the emerging challenges posed by the modern Internet. Section 1.3 poses the research questions and addresses the methodology followed in designing TCP/IP functions. Section 1.4 concludes this chapter with an overview of the thesis.

1.1 The Internet

The Internet has dramatically changed our daily lives in many aspects. We exchange emails to communicate with our friends and colleagues, read the latest news, and search useful information on the Internet. Furthermore, we can make telephone calls, watch

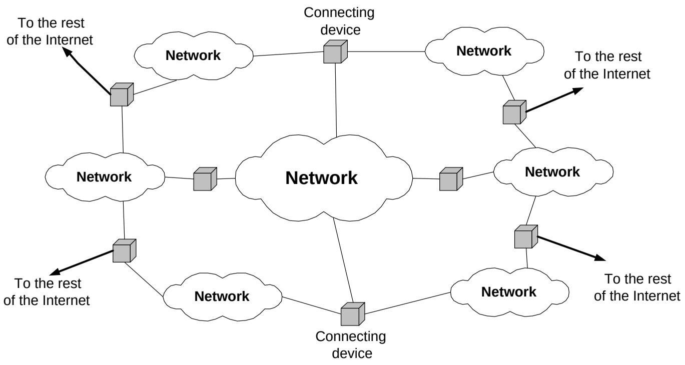


Figure 1.1: A generalized view of a part of the Internet[9]

TV programs or movies, pay bills through the Internet. The Internet brings people, who are geographically apart, closer and enhances our lives.

The Internet is a collection of thousands of networks connected by a common set of protocols which enable communication or/and allow the use of the services located on any of the other networks[16]. Figure 1.1 depicts a generalized view of a part of the Internet. The clouds represent networks and the square boxes denote connecting devices such as switches and routers. The common set of protocols running on these networks is referred to as the TCP/IP protocol suite, which is discussed in more detail in Section 2.1. These networks are connected and thus can communicate with each other on the Internet. Figure 1.2 demonstrates an example of data communication through the Internet. Each computer on the Internet must have a unique IP address to communicate with other computers. When one computer (called source) in network A wants to connect to another computer (called destination) in network B through the Internet, the source computer will send out the data in the form of packet. The packet mainly contains the destination IP address (D), the source IP address (S), the data that the source want to send (Data), and some control information to insure successful communication (omitted in the figure). After being generated and sent by the source computer, the packet will be forwarded by interconnecting devices using its destination address, and finally reach the destination computer. The destination computer will reply to the source according to the its request. Subsequently, the communication between these two computers thus have been established. The Internet services are built upon millions of communications like this.

1.2 Challenges

The drastic growth of the Internet causes an increased demand for network speeds that has been largely solved by optical networks. This is explained in the following. On one hand, with the popularity of the Internet, people demand advanced services such as multimedia delivery, voice over IP, quality of service and etc., which invoke an explosive increase of the data transmitted on the Internet. Adequate network bandwidths are required to guarantee the high quality of transmissions. On the other hand, the wide

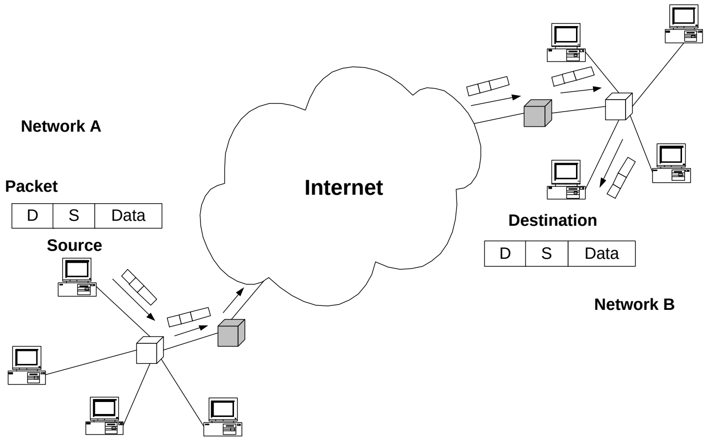


Figure 1.2: Data communication on the Internet

deployment of optical fibers over the Internet provides sufficient bandwidth for transmissions. Recently, the network transmission speeds are reaching 10 Gbps (OC-192) and heading towards 40 Gbps (OC-768). However, the bottleneck of the network speed has shifted to the network processing, especially the data processing within TCP/IP domain. Traditionally, the TCP/IP processing tasks were performed by software running on the general-purpose processors. As the speed of network interface reaches 10 Gbps, the processing speed of the general-purpose processor will no longer keep pace with the wire speeds and become the bottleneck. For instance, a 20 GHz general purpose processor running at full utilization would be needed to drive a 10 Gbps Ethernet network link[4]. This processing speed obviously cannot be achieved within a few years with the current growth trend of general-purpose processors. The situation can be best interpreted by the chart in Figure 1.3. The development speed of network transmission rates dramatically outpaces the development speed of the general-purpose processors. Moreover, the processing speeds of those computational intensive functions or functions with high processing overheads may lag behind the network speeds. Therefore, there is an urgent need to design these TCP/IP functions in special functional units to accelerate the processing speeds. This allows the general-purpose processors to focus on the control and management tasks and the tasks running on the application layer. Furthermore, the demand for advanced services and high performance transmission promotes the introductions of new applications and the changing of existing functions. This requires the designed functions to be flexible and adaptable for future requirements.

In summary, two **key challenges** exist for the design of TCP/IP processing functions:

1. The designed TCP/IP functions must be fast enough to keep pace with the development of network speeds. This could be characterized by the “Need for Speed”.

2. The request for advanced services leads to the need of introducing new or modifying existing applications, protocols and standards, and adding new functionalities. The design must be flexible to adapt to these changes. This could be characterized by

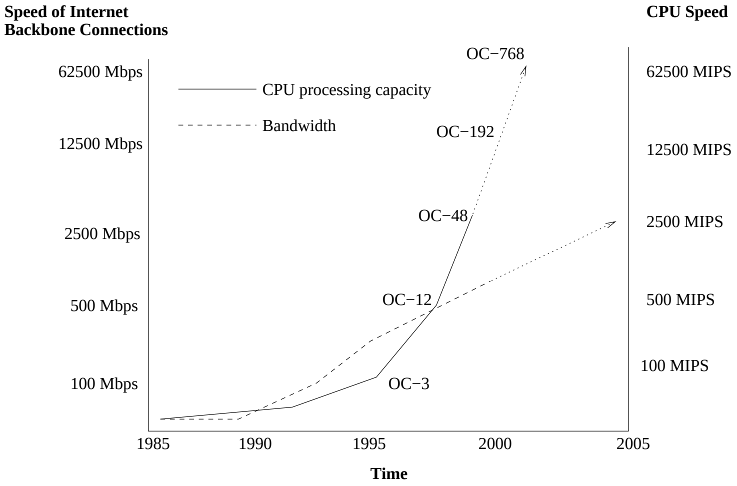


Figure 1.3: Transmission speed vs. Moore's law[23]

the "Need for Flexibility".

A traditional solution for the first challenge may be the application-specific integrated circuits (ASICs) since they can provide high performance processing at wire rates. However, ASICs are not flexible in order to meet the second challenge posed by the modern Internet. The static feature of an ASIC circuit limits the introductions of new functionalities. If there is any change on the functions or protocols, the ASICs have to be replaced by the newly designed devices. This is obviously expensive and unpractical.

A Field Programmable Gate Array (FPGA) is a reconfigurable hardware device that is programmable at gate level. It encompasses both the performance close to that of ASICs and flexibility of general-purpose processors, which makes it ideal for high performance processing and providing the flexibility to meet the possible changes of protocols or applications. Therefore, FPGA is a good solution for the design of the TCP/IP functions.

1.3 Research Questions and Methodology

FPGAs have been considered as an effective technology to meet the processing speeds at wire rates as well as to provide flexibility for the possible changes of applications and protocols. Two challenges raised in Section 1.2 are reviewed to examine how could FPGAs meet these challenges. The challenge, the "Need for Speed", requires to design the time-critical TCP/IP processing functions in FPGAs, and to achieve the processing speed-up as much as possible. The second challenge, the "Need for Flexibility", leads to the design of flexible functions in FPGAs that can be updated whenever the protocol or standard changes or new features are introduced. Therefore, in this thesis, the focus is on the design of the TCP/IP functions in a way that can meet the discussed challenges. The research questions are the following:

TCP/IP protocol suites consists of a number of protocols spanning on different layers of the TCP/IP model. In this thesis, we will only focus on the lower four layers of the TCP/IP model. In these layers, many TCP/IP functions can be identified. The first research question becomes:

1. *What are good candidate performance-critical functions for acceleration in FPGAs?*

If the candidate functions are identified, further efforts can be put to investigate these functions, and the second question arises:

2. *How can the selected performance-critical functions be designed in FPGAs to meet the challenges posed in Section 1.2?*

Finally, the performance of the designed TCP/IP functions have to be evaluated, therefore, the third question is:

3. *If the TCP/IP functions are designed, what are the speedup gains?*

In order to investigate the mentioned open questions, the methodology of the research is given as follows:

- The functions within the TCP/IP domain are surveyed and categorized. The profiling of the computational intensive functions and the functions with high processing overheads are presented, and the performance-critical functions are selected for further investigation.
- The detailed theory of the selected performance-critical functions is studied, the implementations and possible speedup methods are investigated.
- With the selected functions and their speedup designs, VHDL codes are written to describe their functionalities. The designs are further verified and synthesized to evaluate the performance. The clock rates and the utilizations are compared.

In the work described in this thesis, we do not intend to achieve actual hardware. The goal is only to investigate TCP/IP functions. Synthesis is performed to gain insight into the performance potential of our design. In addition, placement and routing of the design on several FPGAs has been performed in order to achieve a more accurate performance evaluation and ensure that the designs fit on respective FPGAs.

1.4 Thesis Overview

This section gives an overview of the remainder of this thesis.

Chapter 2 introduces the background technologies employed by the TCP/IP protocol suite, which are widely discussed in this thesis. The TCP/IP model, which is the basic architecture of network devices, is introduced. Other issues like addressing, communication between layers are presented. Furthermore, the TCP/IP functions are summarized and categorized into micro-level functions and network services And finally, the design platform, the design tools, and the design methodology utilized in this thesis are highlighted.

Chapter 3 discusses the selection of TCP/IP functions from the profiling result. Checksum and Cyclic Redundancy Check are selected as computational intensive functions to be implemented. Table lookup of the network address translation service is selected as a possible critical factor in FPGAs. The theory of the selected functions are further discussed in details and some novel ideas are proposed for accelerating the processing speeds.

Chapter 4 describes the implementations and results of the selected TCP/IP functions. The designs are first verified for their correctness, and next synthesized to get the estimated clock rates, area utilizations, etc. Furthermore, the throughput of the design can be evaluated from the clock rates.

Chapter 5 presents the conclusions. First, the summary of the conclusions in this thesis is given. Subsequently, the main contributions of the work described this thesis are highlighted. Finally, future research directions are presented.

Background

2

As discussed in Chapter 1, the Internet is an enormous network consisting of millions of computers and a variety of interconnecting devices, such as switches and routers. It is built upon a hierarchical architecture and utilizes a common set of protocols to provide effective, robust, and high speed communications between the Internet users. The most common set of protocols running on the Internet is the TCP/IP protocol suite and the hierarchical architecture in which the protocols are placed is known as TCP/IP protocol stack. In this chapter, the TCP/IP protocol suite is first discussed starting from the introduction of the TCP/IP model. The discussion of our thesis is based on the TCP/IP model. Protocols running at different layers in the TCP/IP model are highlighted. Addressing, as a key issue in networking, is presented in detail. Furthermore, the communication processes between two computers are discussed, the operations at each layer are briefly introduced. Subsequently, the TCP/IP processing functions which are executed during the communication processes are investigated and the functions on the data plane are further categorized into micro-level functions. Finally, the targeted platform and three design tools which are utilized in different phases of this thesis are presented. And the design flow and methodology are highlighted.

This chapter is organized as follows. Section 2.1 introduces the general topics in TCP/IP protocol suite: the TCP/IP model, three different kinds of addresses and their purposes, and how communication is established between two users. In Section 2.2, TCP/IP functions involved in the data communication is investigated and categorized in detail. Section 2.3 describes the design flow and tools we utilized in this thesis.

2.1 TCP/IP Protocol Suite

The TCP/IP is the most widely utilized protocol suite over the Internet. It has a hierarchical architecture which is made up of interactive models, and each module provides a specific functionality. There are also independent protocols running on each of these modules.

2.1.1 TCP/IP Model

The standard model for understanding and designing a network architecture is the well-known OSI (Open Systems Interconnection) model, which consists of 7 layers. However, our work mostly focuses on the lower four layers of the model, therefore, the discussion in this thesis is mainly based on a more practical TCP/IP model. TCP/IP Model is a hybrid model derived from the OSI Model. It combines the top three layers of the OSI model to a single application layer, accordingly, the TCP/IP model contains five layers: the physical layer, the data link layer, the network layer, the transport layer and the application layer. The TCP/IP model is depicted and compared with OSI model in Figure 2.1.

<table>
<tr><th>OSI Model</th><th>TCP/IP Model</th></tr>
<tr><td>Application</td><td rowspan="3">Application</td></tr>
<tr><td>Presentation</td></tr>
<tr><td>Session</td></tr>
<tr><td>Transport</td><td>Transport</td></tr>
<tr><td>Network</td><td>Network</td></tr>
<tr><td>Data link</td><td>Data link</td></tr>
<tr><td>Physical</td><td>Physical</td></tr>
</table>

Figure 2.1: OSI and TCP/IP model.

In the TCP/IP model, each layer communicates with its neighboring layers through standardized interfaces and at the same time provides services to its neighboring layers. Each layer contains a particular set of protocols. Figure 2.2 illustrates the specific protocols running at each of these layers and their relationships[24][9].

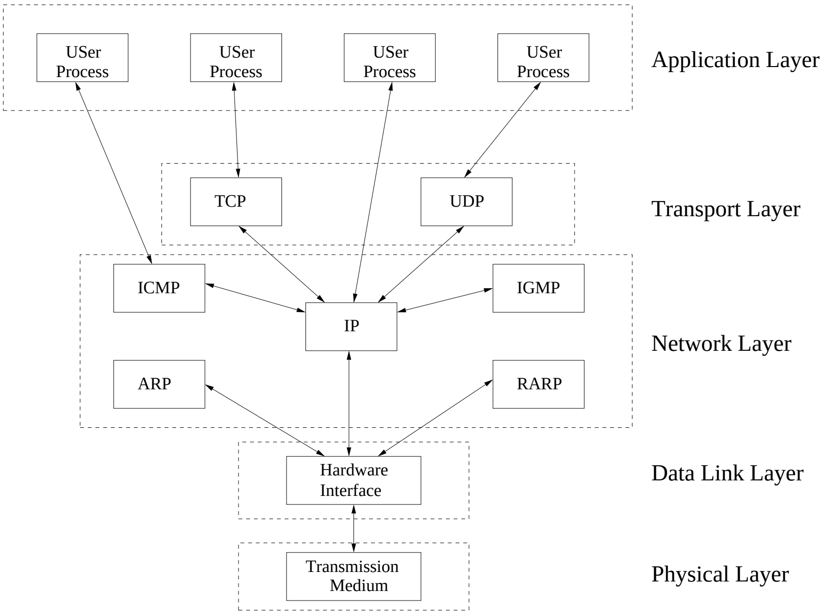


Figure 2.2: The specific protocols on each layer[24].

Each layer depicted in Figure 2.2 is described briefly in the following:

1. **Physical Layer**
   Physical layer provides functions to transmit bits streams over transmission

medium. It defines the mechanical and electrical standards of the interface between the devices and transmission medium. The transmission medium could be the twisted-pair cable, coaxial cable and fiber-optic cable, etc. The physical layer also deals with attributes such as the data rate, transmission mode, synchronization of bits, and the other characters related to transmission.

2. **Data Link Layer**
   Data link layer transforms the physical layer, a raw transmission facility, to a reliable link. It makes the physical layer appear error free to the upper layer (network layer)[9]. It accomplishes this task by breaking the bits streams into data frames, adding some control information to the frames and protecting the frames by a certain error detection technique. The physical layer only takes care of the transmission of the bits streams and only at the data link layer, the frames could be recognized.

3. **Network Layer**
   Network layer is responsible for the delivery of a packet from the source host to the destination host across one or multiple networks. Network layer supports the following protocols:

   - **IP** (Internet Protocol), is an unreliable and connectionless protocol, which provides no error checking during the transmission and does its best to get data transmitted through to its destination, but without guarantee. It should be paired with TCP to perform reliable transmissions.
   - **ICMP** (Internet Control Message Protocol), is used by IP protocol to exchange error messages and other vital information with its peer network layers on the other hosts. The ICMP messages are first encapsulated into IP packets before sending to lower layers.
   - **IGMP** (Internet Group Management Protocol), is usually used with multicast to send a UDP datagram to multiple hosts. It is a companion to the IP protocol.
   - **ARP** and **RARP** (Address Resolution Protocol and Reverse Address Resolution Protocol), are used to convert between the IP addresses used by the network layer and the Ethernet addresses used by the data link layer.

4. **Transport Layer**
   As can be observed in Figure 2.2, TCP and UDP are two predominant protocols running on the transport layer.

   - **TCP** offers connection-based reliable data transmission services by utilizing data acknowledgments and retransmissions.
   - **UDP** is an connectionless, unreliable protocol. UDP does not provide error recovery services when sending and receiving packets.

5. **Application Layer**
   The application layer deals with particular user processes. Multiple processes could

run at the same time considering the characteristics of modern multi-process computers. There are numerous application protocols running at the application layer such as FTP, HTTP, SMTP, and etc.

2.1.2 Addressing

Addressing is a key issue in networking. Addresses could be regarded as identifications, and different addresses identify different things. There are three different kinds of addresses in the TCP/IP protocol suite: Ethernet Address (or MAC Address), IP Address (or Network Address), and Port Number (or Port Address). Ethernet address is utilized by the data link layer to determine a unique host within a network. IP address is employed by the network layer to identify a unique host on the Internet. And port number is adopted by transport layer to recognize a certain user process running on a multi-process host. We discuss them in detail as follow:

Ethernet Address, which has 48 bits, is the lowest level address, and works on the data link layer. The Ethernet address is imprinted by the manufacturer on the network interface card (NIC) when the NIC is produced. Each NIC has a unique Ethernet address. A computer must be equipped by at least one NIC to connect to the network, therefore, a computer will at least have one Ethernet Address. Each Ethernet address identifies one network interface between the computer and the network the computer is connected to.

IP Address is introduced at the network layer. It is a 32-bit address which can uniquely identify a host connected to the Internet. Only the packets with the accurate source and destination IP addresses can be correctly forwarded to their destination hosts and establish the communications. IP addresses consist of two parts: netID and hostID. NetID identifies a network and hostID defines a host on that network. Therefore, when a packet is sent onto the Internet, the interconnecting devices will find its destination network according to the NetID of its destination IP address, and then forward the packet to that network. When the packet reaches the destination network, the network will allocate its final destination host according to the hostID of the destination address.

The IP address is divided into five classes: A, B, C, D, and E, the first few bits of the IP address determine the class of the address. As we can see from Figure 2.3, different class has a different netID length. The binary numbers in the beginning of netID field define the address classes.

Furthermore, the Internet authorities have reserved three blocks of addresses for private networks. Table 2.1 demonstrates these addresses. Any organization can use these addresses for its private internal networks. Packets with the private addresses cannot be recognized by the routers and thus will be deleted by the on the Internet. In contrary to the private address, we call the IP address, which can be generally recognized and forwarded by the routers, the public address.

Port Number is defined at transport layer. At network layer, IP address uniquely defines a host on the Internet and is responsible to deliver packets at the host level, namely,

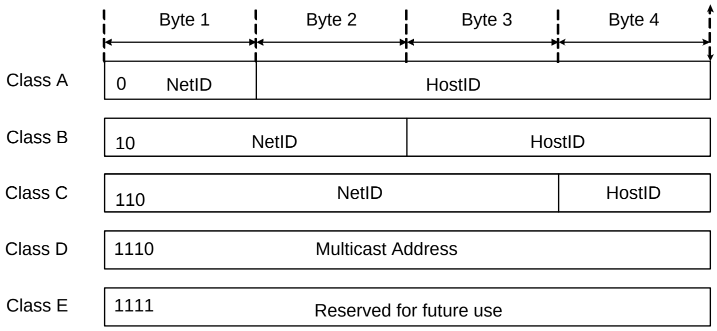


Figure 2.3: IP address classes.

Class	NetIDs	Number of Networks
A	10.0.0	1
B	172.16 to 172.31	16
C	192.168.0 to 192.168.255	256

Table 2.1: Addresses for private networks[9].

host-to-host communication. However, that is not enough to complete the communication between today's multi-process computers. The port number at transport layer is introduced to identify the processes running at the application layer of the multi-process computers. The port number is 16 bits long and ranges from 65535 to 0. There are some well know port numbers for specific application processes, for example, 23 for TELNET, 80 for HTTP, and 21, 23 for FTP and etc.

2.1.3 Communication Between Layers

In order to communicate with other computers, the data originating from the application layer must be transmitted down through different layers at the source, onto the transmission medium. At the destination, the data is transferred up through the same layers, and finally reaches the application layer. During this procedure, the data is processed at each of these layers. In addition, protocol control information is appended to (at the source) or stripped off (at the destination) at each layer to ensure that the data is transmitted to its final destination effectively and safely. Figure 2.4 depicts the whole process of such a communication. We mainly discuss the half process at the source computer part, for the operations at the destination are reversed to the operations at the source.

At the source part, data generated by the source is transmitted from the application layer down to the physical layer and then to the transmission medium. Data is transmitted in the form of PDUs (Packet Data Unit) through the layers. Each layer defines its own PDU, namely, Segment at the transport layer, Packet at the network layer, and Frame at the data link layer. PDU contains the data unit received from the upper layer and protocol control information (usually a header or trailer). The process of adding a header/trailer to the data unit received from upper layer is called **encapsulation**.

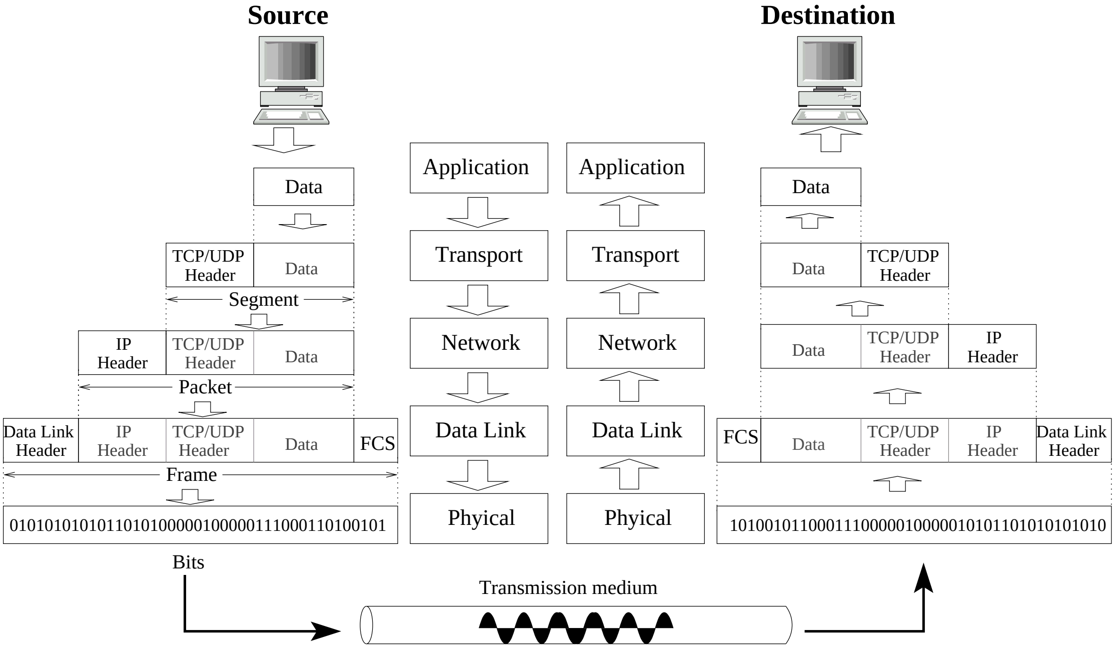


Figure 2.4: Data communications between layers[3].

Encapsulation takes place at every layer.

At the transport layer, a TCP/UDP header is appended to the data. The header contains the port number, which identifies a process running at the application layer, and some other control information specified by the transmission control protocol, such as the sequence number, acknowledgment number and windows size, etc.

At the network layer, an IP header is added to the data unit received from the transport layer. The header contains the source and the destination IP addresses, which will help to locate the destination on the Internet.

At the data link layer, a data link header and a trailer is attached to the data unit received from the network layer. The header contains the source and destination Ethernet address. And the trailer is an error detecting code called Cyclic Redundancy Check (CRC). The packet data unit at data link layer is called frame. Frame is the lowest packet data unit.

At physical layer, the frame is transmitted in a series of bits and traverses through the network and finally, reaches the destination. The opposite operations at the source computer will be performed, for example, the frame will be decapsulated, CRC code will be checked to ensure no corruption during transmission and etc. Finally the data will reach the application layer of the destination computer.

If the destination is not in the same network as the source computer, the frame will be first transmitted to the interconnecting devices. At the interconnecting devices, the frame is passed through the data link layer and network layer in order to determine which network the packet is going to be forwarded. Figure 2.5 demonstrates how the data traverses through the layers of the source, destination and interconnecting devices in order to establish the communication between two computers.

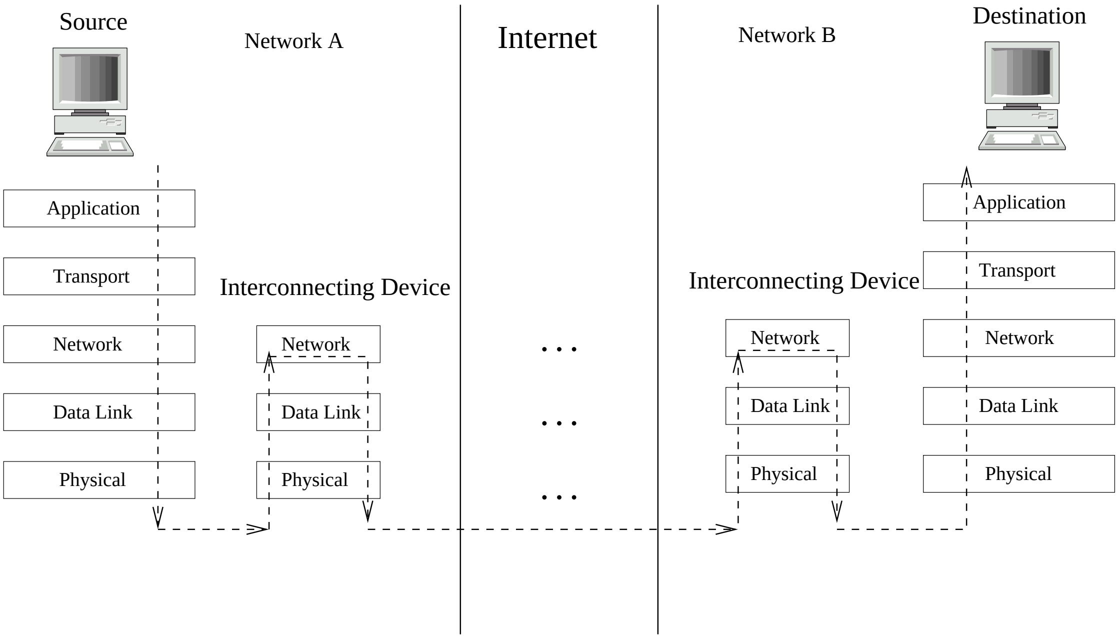


Figure 2.5: Data communications, from TCP/IP model's point of view.

In summary, while communicating with other computers, data originated from the source will traverse across different layers on the source and destination computers or/and interconnecting devices. At each layer, the data need to be processed and at the same time, many other functions are running in order for a robust and effective transmission. Consequentially, TCP/IP functions that perform the data processing tasks and those functions running in auxiliary can be identified and categorized in order to find the performance-critical operations. This work is going to be further investigated in the following section.

2.2 TCP/IP Functions

As discussed in Section 2.1.3 and depicted in Figure 2.4 and 2.5, data need to be processed at every layer it goes through while it is transmitted over the Internet. Different functions running on different layers may take charge of different kind of processing tasks. Generally, the TCP/IP functions can be divided into two parts:

- **Data plane** refer to the plane where the network data pass through. Data plane functions, such as classification, table lookup, are performed over every packets passing through the systems. Therefore, data plane has a large amount of data processing tasks. Since data are transmitted at wire speeds, these processing tasks are also required to perform at wire speeds. Most computational intensive operations are also performed on data plane.

- **Control plane** takes charge of control and management tasks that may coordinate the functions between data plane and control plane within the system and with the outside systems. It updates the tables on the data plane, performs signalling, interface management, and other complex actions that can not be executed on data plane.

Since the performance-critical operations are located in data plane and are required to perform at wire speeds, in our thesis, we will mainly focus on the functions on the data plane and refer to them as **micro-level functions**[1]. These functions may be found in more than one network services running on the data plane. Most of the micro-level functions are required to perform at the wire rates. However, not all these micro-level functions are performance-critical, therefore, we need to further investigate according to the profiling results in order to determine which functions are timing-critical for acceleration. The followings are typical micro-level functions:

1. **Encapsulation:** As depicted in Figure 2.4, when data move through the layers, some information must be added to the data. They are always in the forms of header or trailer. The process of adding information to the data is called the encapsulation. Encapsulation is performed by most of the layers, from the transport layer to the data link layer.

2. **Checksum:** Checksum is an error detecting technique adopted by the network layer and transport layer to protect the data against corruption during the transmission. At the network layer, checksum calculation is performed over the IP header. At the transport layer, checksum is calculated over the TCP/UDP header as well as the data coming from the application layer.

3. **Data parsing:** Data parsing is to obtain some information from the packets, usually from the header of the packets for further investigations. For example, the destination address need to be obtained to decide the next hop of the packet. Port number, or other information such as the source address, protocol field may be acquired for classification.

4. **Data modification:** Data modification is to modify a certain or several fields in a packet, for example, to decrease the TTL field, and accordingly, to change updated checksum field in the IP header. There are some other situations may need data modification, for example, the network address translation (NAT) will change the source IP address in the IP header.

5. **Bitwise comparison:** Bitwise comparison is widely used by different layers for different purposes. The inputs of the function could be two sequence of bits with the same length, the output could be a Boolean value reflecting whether the two inputs match or not. For example, when the data link layer receives a frame, it need to check whether the destination Ethernet address of the frame matches its Ethernet address. If yes, the frame is accepted, otherwise, the frame is abandoned. At the network layer, there are many uses of bitwise comparison. For example, it is used for routing table lookups, also for IP address matching, and version checking.

---

[1] We do this to indicate that these functions constitute the basic functions of many network services and protocols. Since they are basic functions and can not be further subdivided into smaller functions, we call them micro-level functions.

6. **Queuing/Scheduling:** Internet data must be stored when it is waiting for processing. Queue is a kind of buffer used to store the ingress data for processing and the outgress data after processing. At the same time, several scheduling mechanisms can be employed to identify the order that a packet is dequeued from the buffer.

7. **Cyclic redundancy check:** The Cyclic Redundancy Check (CRC) is an error detection technology that is widely utilized in networking. For example, CRC-32 is employed to generate the Frame Check Sequence (FCS) of a frame. CRC is calculated over the frame and appended in the trailer of that frame.

8. **Fragmentation:** When a frame is transmitted over the physical layer, the maximum size of the frame is restricted by the MTU (Maximum Transfer Unit) according to the physical network. For example, the MTU of the Ethernet protocol is 1500 bytes. The MTU is defined at the data layer according to the physical networks at the physical layer. Therefore, at the network layer, if the packet size is larger than the MTU, the packet need to be fragmented to a smaller size in order to fit the MTU of the frame. This process is thus called the fragmentation. The packets after fragmentation are encapsulated by new IP headers and transmitted to the data link layer.

9. **Table lookup:** Table lookup is to search a table according to certain criteria. For example, in routing table lookup, the destination address, as the searching criteria, will be examined and find the corresponding next hop for the packet with the destination address. There are many other situations where need table lookups. For example, in Address Resolution Protocol (ARP), the IP address must be replaced by the corresponding Ethernet address get from ARP table lookup. Several techniques can be employed to perform table lookup, for example, the longest prefix matching, the content addressable memory and etc.

10. **Classification:** Packet classification is to classify the packets based on the packet headers into equivalent classes called flows. A flow is defined by users, for example, the packets which have the source address start with prefix A.B and to the destination address with prefix C.D and with the destination port number E, and etc. Packet classification is need for functions like firewalls, Quality of Service, and other services that require the capability to isolate the traffic into different flows for further processing. The packet in the same flow will receive the same treatment. A typical example of classification is the demultiplexing at the transport layer. When data comes to the transport layer, it is categorized to different flows according to the port number at the IP header.

Furthermore, there are many network services running on the data plane which may consist of several micro-level functions. The followings are some typical network services.

1. **Address Resolution Protocol (ARP):** As discussed in Section 2.1.3, the IP packet received from the upper Network Layer should be encapsulated in a frame to pass through the Physical Layer. The address utilized by the data link layer

is Ethernet address. Therefore, the IP addresses in the packet header must be translated into Ethernet addresses. The best solution for mapping logical addresses to physical addresses is to be done dynamically. That is, the sender asks the receiver announce its physical address when needed. That is exact what ARP protocol does. In order to improve the efficiency of ARP protocol, a cache table is designed to hold the recent outgoing frames' Ethernet addresses. Since there is a high possibility that the recent used Ethernet addresses will be reused, the outgoing messages will firstly check the cache table to find the corresponding Ethernet address. If the corresponding address is not in the table, an ARP request is sent to ask for that Ethernet address. The important micro-level function involved is the cache table lookup.

2. **IPSec:** Internet Protocol Security is a suite of protocols providing security for IP traffic on the Network Layer. It accomplishes its goal by tunnelling, encryption and authentication.

    - Encryption is utilized to ensure that the transmitted data content are concealed to public. The most frequently used encryption algorithm is 56-bit DES. A possible implementation of DES in FPGAs is presented in [12].
    - Authentication Authentication is to verify the identification of a sender. An authentication method tries to test that the messages are from an authentic sender and not from any impostor. The digital signature based on the public key encryption/decryption is commonly used.

    The possible micro-level functions involved may be data parsing, data modification and etc..

3. **Firewall:** Firewall has become a critical component for the Internet for security reasons. Firewall will examine both the incoming and the outgoing traffic using certain criteria defined by the users. The traffic will be rejected to pass if it meets a certain rule. The critical micro-level functions involved are classification, table lookup, bitwise comparison, and etc.

4. **Network Address Translation:** Network Address Translation (NAT) is designed as a short term solution for the shortage of the IPv4 addresses. As discussed in Section 2.1.2, any host that want to access the Internet must have a unique IP address to identify itself. However, within the network equipped by a NAT server, the host could use a private address to communicate with the other computers on the Internet. The host with private address will send its packets first to the NAT server, and then the NAT server will map its private source address to a public address, and store the mapping information at the same time in a table called NAT table. The packets with the public address will be forwarded to the Internet. When the responding packet comes back, the mapping information will be retrieved from NAT table. And the NAT server will change the address back to the corresponding private address, and forward the packet to the host on that network. The critical micro-level function is the NAT table lookup. Other functions includes data parsing, data modification, checksum calculation, CRC calculation and etc.

In summary, the micro-level functions are required to run at wire speeds and can be combined to perform the functionalities of network services and protocols. A reasonable profiling may be established to find the most performance-critical functions and implement them in FPGAs. Furthermore, several micro-level functions can be integrated to perform the processing tasks of a network service so that the processing tasks can be offloaded from general-purpose processors.

2.3 Design Methodology

In Section 1.2, FPGAs are decided to be the target platform for the design of TCP/IP functions. FPGAs are digital devices that can implement logic circuits by programming the required functions[21]. FPGAs consist of two dimensional array of configurable logic blocks (CLBs) surrounded by input/output blocks (IOBs) and the routing matrix which interconnects the configurable logic blocks (CLBs). FPGAs can be reconfigured by downloading a bitstream file created from the design by synthesis and implementation tools.

Our design will start with the functional description of the selected TCP/IP functions in Very High Speed Integrated Circuit Hardware Description Language (VHDL), and end with running of the design in the implementation tools and get the bitstream file. The whole process can be accomplished by the design flow depicted in Figure 2.6.

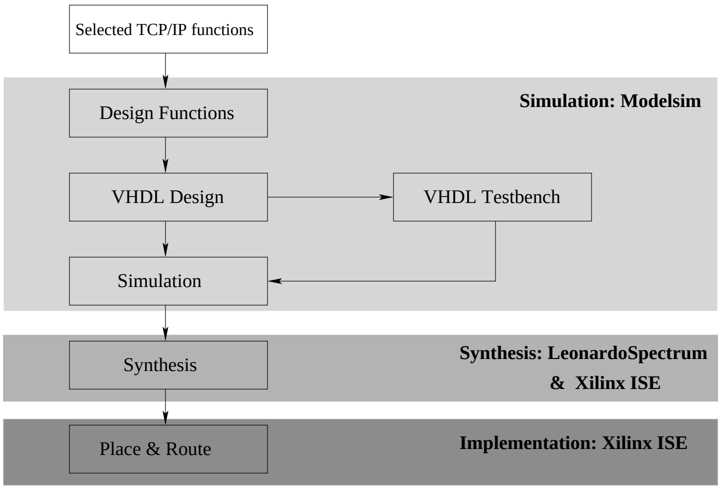


Figure 2.6: Design flow.

The design flow can further be separated into three steps: simulation, synthesis, and implementation. The design tool which is utilized at each of these three steps is specified and the detailed design methods are highlighted as follow.

- **Simulation: ModelSim SE PLUS 5.5e** from Model Technology is utilized to write VHDL code and run simulations. According to the design flow, we first write the VHDL code for the functions. After correctly compile the VHDL code, we verify the design to ensure its functionality is as intended. Subsequently, a testbench is generated to connect to the designed functions and compiled to run simulation. The testbench will test the key functionality of the design during simulation.

- **Synthesis: LeonardoSpectrum** from Mentor Graphics is employed to run synthesis. Synthesis uses user specified synthesis constraint (timing, power, area, etc.) to implement and optimize the RTL design into equivalent unit-delay primitive functional blocks (Flip-flops, logical gates, and etc.)[25]. From synthesis, we can also get, for example, the clock rate, the area information. However, synthesis only gives estimated values. In order to obtain more accurate results, place and route for a target FPGA can be performed in the implementation phase.

- **Implementation: Xilinx ISE 5.1.03i** from Xilinx Inc. is utilized to run placement and routing. After synthesis, the synthesis tool exports an EDIF netlist file that represents the functionality of the design. Xilinx ISE will take this netlist and then translate the input design netlists and write result to a single merged NGD netlist[11]. The design is mapped into CLBs and IOBs. After that, the design can be placed and routed. More accurate figures about the utilization and performance can be found in the place and route report and post place and route static timing reports, respectively. Xilinx ISE itself can also run synthesis. Therefore, we will list the performance and utilization results from both LeonardoSpectrum and Xilinx ISE. Finally, a configuration bitstream file can be created for the design to be downloaded to a targeted device.

In this thesis, we will use the performance results form the static timing report to evaluate the designed TCP/IP functions. For example, the throughput of the TCP/IP functions can be calculated and compared.

2.4 Conclusions

In this chapter, the general topics of TCP/IP protocol suite was first presented. The TCP/IP model, as a practical model for our thesis was illustrated and its five layers and protocols on each layer were discussed. Subsequently, the addressing, as a key issue in networking, was introduced. There are three different addresses: Ethernet address, IP address and port number and they span on different layers and take care different functionalities. Subsequently, we investigated the processing functions in the TCP/IP protocol stack and focused on the data plane, which requires processing at wire speeds. Micro-level functions were summarized, and network services built upon these micro-level functions were identified. Finally, the design tools and methodology were further described.

TCP/IP Functions

3

The *TCP/IP processing functions involved when computers are communicating with each other are highlighted in Chapter 2. In this Chapter, the selection of these TCP/IP functions is described in detail. Profiling results from literatures are illustrated and compared. Two functions, checksum and CRC are selected from micro-level functions as the candidate functions to be implemented. And the table lookup operation in network address translation service will be designed. The theory of the selected functions are explained. We start with the checksum function which is mainly 16-bit one's complement additions. A 16-bit carry-lookahead adder is then designed and cascaded to build a 32-bit one's complement adder to execute the checksum calculation. The theory of the cyclic redundancy check (CRC) is presented. Linear Forward Shift Register (LFSR) is a common approach to implement the serial calculation of CRC in hardware. The parallel design, which is derived from the state equations of serial LFSR implementation is further investigated. The 32-bit parallel input processing of the CRC-32 algorithm, which is most frequently utilized in TCP/IP protocol suite, is instantiated. Furthermore, based on the observation that the possible changed fields in an Ethernet frame are located only in the header part, and a large amount of data in the remaining frame is unchanged, a novel fast update method of CRC calculation is proposed. This will dramatically improve the throughput of the CRC calculation. Finally, the implementation of the table lookup function in network address translation using the on-chip block SelectRAMs as content addressable memories (CAMs) is presented.*

This chapter is organized as follows. In Section 3.1, the TCP/IP functions with high processing overheads or computational intensive functions are selected as candidate functions. In Section 3.2, one of the selected functions, the checksum algorithm is described and the solution to calculate the checksum by 16-bit one's complement addition is presented. The theory of calculating CRC in both serial and parallel implementation is explained in Section 3.3. A novel method to calculate CRC in a fast way is proposed in Section 3.4. Section 3.5 presents the network address translation and it main operation, NAT table lookup.

3.1 Selection of TCP/IP Functions

We had an in-depth survey of the TCP/IP functions in Section 2.2 and further categorized them into micro-level functions. Some typical network services are also discussed which may consist of serval micro-level functions. Since the micro-level functions are performed over every incoming packets, they are most possible performance-critical or computing intensive functions that may become the bottleneck and affect the overall processing speeds. For these functions, we select those which have high processing overheads and are possible to be accelerated and implemented in FPGAs. Furthermore, network services which consist of several micro-level functions can be designed to offload the processing burdens of the GPPs.

After some literature studies, the checksum calculation and the CRC calculation are believed to be two most computing intensive and time-critical functions that may impede the processing speeds.

- **Checksum:** There are various occasions where need the checksum calculation, for example, at the IP header, ICMP's entire message (include header and data), TCP and UDP's entire message , and etc.

  In paper [13] by Jonathan Kay and Joseph Pasquale, they categorized the processing overheads into several operations, including checksum, data move, data structure, errorcheck, mbuf, opsys, protspec, and others. They showed that checksum calculation is the main processing overhead, and the overhead grows as the size of the message increases. Figure 3.1 depicts the cumulative percentages of the processing time for TCP and UDP according to the operations categorized by the authors. Checksum, as one of the operations, has the maximum processing overheads, as depicted in the black region at the bottom of these two charts.

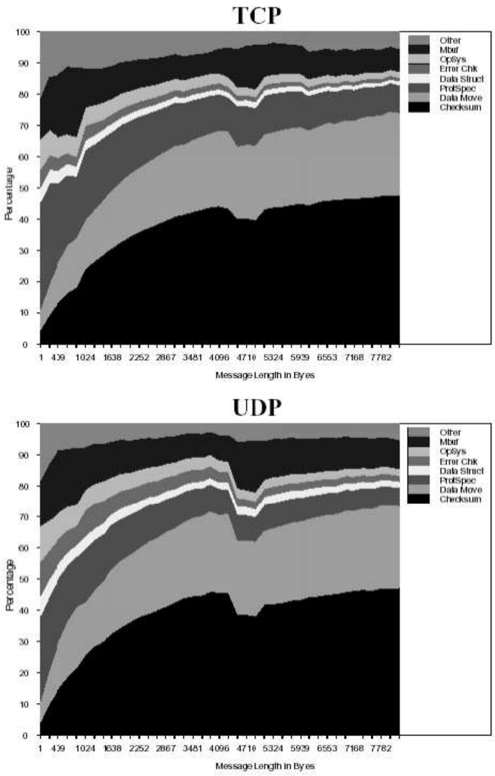


Figure 3.1: Breakdown of operations processing times[13].

In paper [5] by Mel Tsai et al, a profile of the IPv4 forwarding on the Intel IXP1200 network processor with the 64-byte packets is illustrated in Figure 3.2. From the figure, we can observe that the Header Validation occupies the highest processing time. The header validation mainly performs the following operations:

- Check version in the version field
- Check the header length field
- Calculate the header checksum

In these three steps, the checksum calculation is the most time consuming operation, and need to be considered as a function to accelerate.

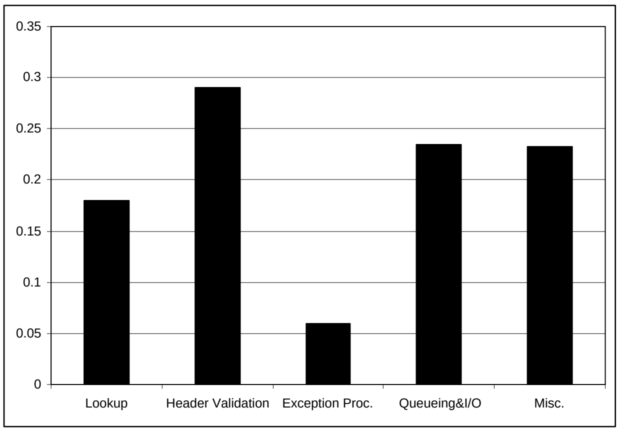


Figure 3.2: Profile for IPv4 packet forwarding on the IXP1200 [5].

The other profiling of IPv4 packet forwarding shows the same result as the checksum has the largest processing overheads. They do their experiments on the click router they build, and classify the IP forwarding functions into several elements such as the CheckIPHeader, which calculates the checksum, the LookupIPRoute, which performs table lookup and etc. The costs were measured by Pentium III cycle counters. The profiling result is demonstrated in Figure 3.3. From the figure, we observe that the CheckIPHeader dramatically outranges over the other elements. The Checksum calculation is performed on the data of TCP and UDP datagram, and on the header of IP packet. When calculating checksum, the packet is divided into n-bit sections (n is usually 16). Then, these sections are added using one's compliment arithmetic so that the result is also 16 bit long. The sum is finally complimented to get the checksum. We will further discuss the checksum calculation in Section 3.2.

- **Cyclic Redundancy Checking:** Cyclic Redundancy Checking is one of the most frequently used techniques for detecting transmission errors. One of the CRC techniques utilized in networking is the CRC-32 algorithm employed by Ethernet. The CRC code is calculated over the Ethernet frame and appended to the trailer of that frame. Figure 3.4 illustrates the maximum CRC calculation bandwidth on general-purpose processor and Intel IXP1200 network processor, and compares the bandwidth with the other two functions: MD5 and Route table lookup. From

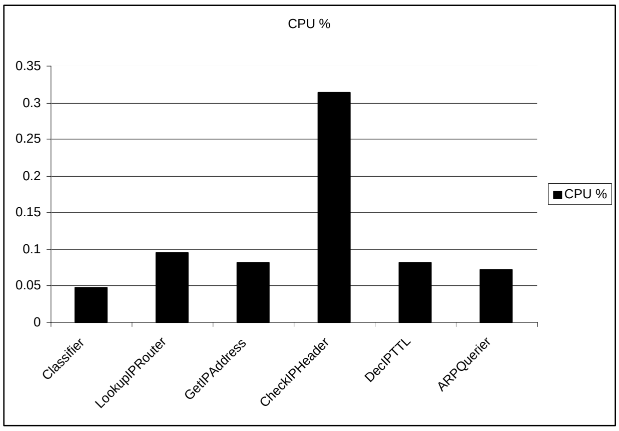


Figure 3.3: Profile for packet forwarding on Pentium III cycle counters [14].

the figure, we notice that the CRC has the lowest bandwidth at 300Mbps for IXP1200 and 200Mbps for general-purpose processor. Therefore, CRC calculation could be a bottleneck for packet processing to reach wire speed. Another Paper [26] compares serval different CRC implementation including software implementations, hardware ASIC implementations, parallel implementations and serial ones over Ethernet CRC-32 algorithm. Their results are demonstrated in Figure 3.5. Software implementation on common RISC machine has the lowest throughput. The maximum throughput of the design is 1.663Gpbs, which still can not meet the latest throughput requirement for 10Gigabit Ethernet. Therefore, CRC calculation is selected as the candidate accelerate function to be implemented in FPGA.

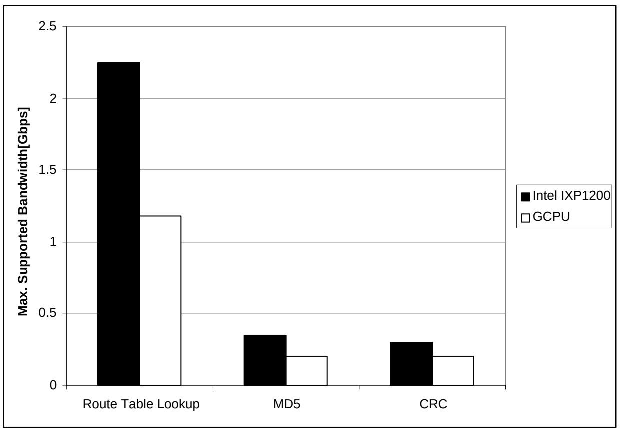


Figure 3.4: Maximum bandwidth comparison on a GPP and an Intel IXP1200 [6].

- **Network Address Translation:** Network address translation is one of the network services that is required to run at wire speeds, since every packet passing through must be modified. The source addresses or/and port number are changed. The TTL field in IP header are decreased. The micro-level functions that are in-

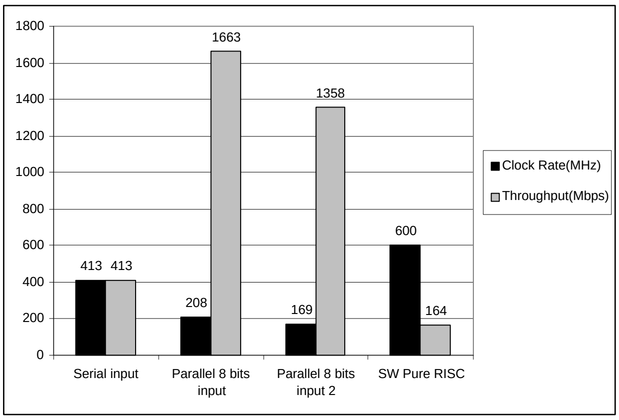


Figure 3.5: Clock rate and throughput comparison between four CRC implementations.

volved in the NAT operation is data parsing, which obtains the source IP address, source port number and etc, the NAT table lookup, which find the corresponding mapping for a certain destination address or/and port number. Checksum and CRC recalculation are also required by the NAT service.

To sum up, three TCP/IP functions are selected to be implemented in FPGAs. The first two are micro-level functions that are computational intensive, and possible to become the bottleneck of the network speeds. The other one is the table lookup function in the NAT service. In the following sections, we are going to discuss in detail the in-depth theories of these three functions.

3.2 Checksum Calculation

As discussed in Section 3.1, Checksum is a critical factor affecting the packet processing speed. The checksum is performed and verified at both the network layer and the transport layer. At the network layer, checksum is calculated over the IP header while at the transport layer, checksum is calculated over the entire TCP/UDP message (header+data). The main steps of the checksum algorithm are as follows[7].

1. The adjacent octets of the data (to be protected by checksum) are paired to form 16-bit fields.

2. The one's complement addition is performed over these fields.

3. The one's complement of the sum result in Step 2 is placed in the checksum field.

4. To check a checksum, the 16-bit one's complement addition is performed over the data as well as the checksum field. If the result consists of all 1's. the check succeeds and thus indicates that there is no error.

Therefore, the calculation of the checksum is a sequence of 16-bit one's complement additions. We take the checksum calculation of the IP packet header as an example.

The IP packet header is depicted in Figure 3.6, each row of the header contains 32 bits and is divided into two 16-bit fields. The 16-bit one's complement addition is performed over every half of the row, and the final result is put in the Header Checksum field.

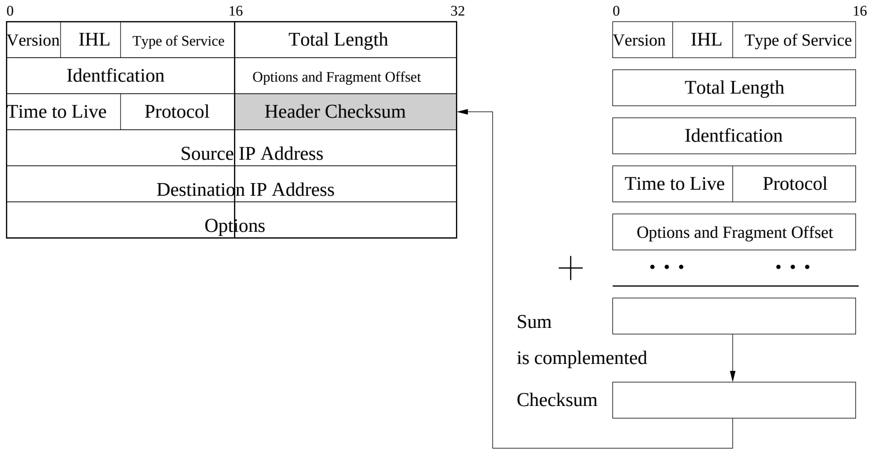


Figure 3.6: Checksum calculation for IP packet header.

As explained above, we notice that the implementation of the checksum function can be reduced to the implementation of a 16-bit one's complement adder. Furthermore, one's complement addition can be performed by a two's complement adder by propagating the carry-out signal to the carry-in. Therefore, our first target is to design a fast 16-bit two's complement adder [18]. There are many basic implementations of adders such as bit-serial and ripple-carries adders. The bit serial adders perform the addition bit by bit, and in the ripple carry adders, the adder wait for the carry-in bit of the previous full adder. However, the propagation of carries is the key factor which impede the high-speed performance of the adder. The latency of these basic adders grows linearly as the number of bits of the adder increases. Therefore, it is not suitable to build the 16-bit adder utilizing these basic adder schemes.

The carry-lookahead adder (CLA) is a commonly used scheme for multiple bits two's complement additions. The carry-lookahead adder does not wait for the carry signals as the ripple-carry adders do. Instead, the CLA predicts its carry by its inputs and two other signals called the generate and propagate signals [18]. The complete theory of the carry look-ahead adder is discussed in [18]. Here, we mainly discuss the design of the two's complement 16-bit carry look-ahead adder for checksum calculation.

For two operands additions, we assume that the operands are x and y respectively, and the sum is s. We use x_i, y_i, s_i to denote the digits at the ith position of the number x, y and s. Therefore, the sum digit can be determined from the operand digit x_i and y_i and the carry-in bit c_i: (In this section, $x \oplus y$ denotes an XOR operation and $x \cdot y$ or simply xy denotes an AND operation)

$$s_i = x_i \oplus y_i \oplus c_i \tag{3.1}$$

The generate and propagate signals at ith position are calculated as follows:

$$g_i = x_i y_i \tag{3.2}$$

$$p_i = x_i \oplus y_i \tag{3.3}$$

With these two signals, we can predict the carry-in at the (*i+1*)th position as follows:

$$c_{i+1} = g_i + c_i p_i \tag{3.4}$$

Equation (3.4) indicates that there is a carry-in at the (*i+1*)th position ($c_{i+1} = 1$) if a carry is generated in position i ($g_i = 1$) or there is a carry-in at position i and that carry is propagated at position i ($c_i p_i = 1$). Equation (3.4) is then called the carry recurrence equation. And we can easily unroll the recurrence and finally get the c_i as a logic function of the generate and propagate signals and c_{in}. For example, if $i = 4$, we have:

$$\begin{aligned}
c_1 &= g_0 + c_0 p_0 && (3.5)\\
c_2 &= g_1 + c_1 p_1 \\
&= g_1 + (g_0 + c_0 p_0) p_1 \\
&= g_1 + g_0 p_1 + c_0 p_0 p_1 && (3.6)\\
c_3 &= g_2 + c_2 p_2 \\
&= g_2 + (g_1 + g_0 p_1 + c_0 p_0 p_1) p_2 \\
&= g_2 + g_1 p_2 + g_0 p_1 p_2 + c_0 p_0 p_1 p_2 && (3.7)\\
c_4 &= g_3 + c_3 p_3 \\
&= g_3 + (g_2 + g_1 p_2 + g_0 p_1 p_2 + c_0 p_0 p_1 p_2) p_3 \\
&= g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 + c_0 p_0 p_1 p_2 p_3 && (3.8)
\end{aligned}$$

Here, c_0 is the carry-in (c_{in}) and c_4 is the carry-out (c_{out}) of a 4-bit adder, respectively. With the carry bit at each position, we can calculate the sum bit at every position through Equation (3.1). The adder is called the full carry-lookahead adder. In the design, we use AND logic circuits to produce g_i, use XOR gates to produce p_i and the sum bit s_i.

Furthermore, we can cascade five 4-bit carry-lookahead adders to build the 16-bit carry-lookahead adder. Before that, we need to know the carry-in bits of each 4-bit carry lookahead sub-blocks, which are c_0, c_4, c_8, c_{12}, respectively. In order to get these bits, we introduce the block generate and block propagate signals as follows:

$$\begin{aligned}
g_{[i,i+3]} &= g_{i+3} + g_{i+2} p_{i+3} + g_{i+1} p_{i+2} p_{i+3} + g_i p_{i+1} p_{i+2} p_{i+3} && (3.9)\\
p_{[i,i+3]} &= p_i p_{i+1} p_{i+2} p_{i+3} && (3.10)
\end{aligned}$$

Equation (3.9) illustrates that there will be a carry-in at position $(i+4)$, if a carry is generated at position $(i+3)$, or a carry is generated at position $(i+2)$ and propagated at position $(i+3)$ or a carry is generated at position $(i+1)$ and propagated at position $(i+2)$ and $(i+3)$, and etc. Equation (3.10) indicates that a carry-in at the ith position (c_i) is propagated to position $(i+4)$, if and only if each of the four positions from i to $(i+3)$ propagates. Therefore, given the incoming carry c_i, and the block generate and propagate signals $g_{[i,i+3]}$ and $p_{[i,i+3]}$, we can easily predict the carry-in bit at position c_{i+4} simply by rewriting Equation (3.4) in the following form:

$$c_{i+4} = g_{[i,i+3]} + c_i p_{[i,i+3]} \tag{3.11}$$

A *lookahead carry generator*, which is developed to generate the carry bits as well as the block generate and propagate signals, is depicted in Figure 3.7. And it detailed block diagram is illustrated in Figure 3.8.

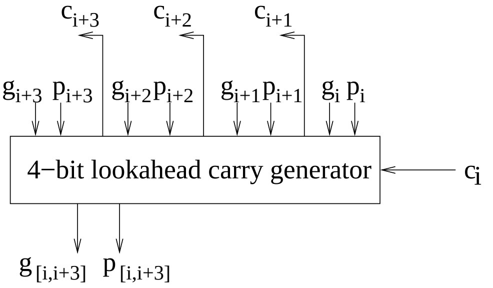


Figure 3.7: 4-bit lookahead carry generator[18].

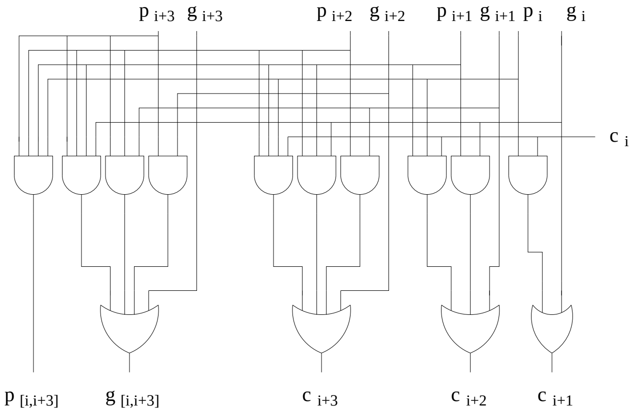


Figure 3.8: 4-bit lookahead carry generator, a detailed diagram [18].

It is clear that two gate levels are required to generate the carry bits and the block generate and propagate signals. If g_i and p_i at each position are available, four 4-bit lookahead carry generator running in parallel can be employed to produce block generate and propagate signals from $g_{[0,3]}, p_{[0,3]}$ to $g_{[12,15]}, p_{[12,15]}$ as depicted in Figure 3.9. Notice that at this moment, although we already have the generate and propagate signals at each position, the corresponding carry bits can not be calculated since the carry-in bits (c_4, c_8, c_{12}) for the upper left three sub-blocks are unknown. Recursive equation Equation (3.11) can be utilized to generate carry bits at position $i = 4, 8, 12$, respectively:

$$\begin{aligned}
c_4 &= g_{[0,3]} + p_{[0,3]}c_0 \\
c_8 &= g_{[4,7]} + p_{[4,7]}c_4 \\
&= g_{[4,7]} + p_{[4,7]}(g_{[0,3]} + p_{[0,3]}c_0) \\
&= g_{[4,7]} + p_{[4,7]}g_{[0,3]} + p_{[4,7]}p_{[0,3]}c_0 \\
c_{12} &= g_{[8,11]} + p_{[8,11]}c_8 \\
&= g_{[8,11]} + p_{[8,11]}(g_{[4,7]} + p_{[4,7]}g_{[0,3]} + p_{[4,7]}p_{[0,3]}c_0) \\
&= g_{[8,11]} + p_{[8,11]}g_{[4,7]} + p_{[8,11]}p_{[4,7]}g_{[0,3]} + p_{[8,11]}p_{[4,7]}p_{[0,3]}c_0
\end{aligned}$$

The 4-bit lookahead carry generator at the bottom of the Figure 3.9 is utilized to perform above equations. And finally, these three carry bits and a larger block generate signal and block propagate signal $g_{[0,15]}, p_{0,15}$ are produced. c_4, c_8, c_{12} are further used as the carry-in bits for the upper four lookahead carry generators demonstrated in Figure 3.9. Accordingly, the carry bits at each position can be generated.

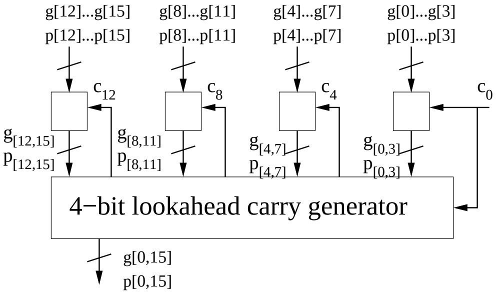


Figure 3.9: 16-bit lookahead carry generator [18].

Therefore, from x_i, y_i, we get g_i, p_i and from g_i, p_i we get $g_{[i,i+3]}, p_{[i,i+3]}$. With these block generate and propagate signals and c_0 we can predict c_4, c_8, c_{12}, respectively. These three carry bits are send to the carry-in bits of the upper left three lookahead carry generators so that the rest of the carry bits $c_{i+1}, c_{i+2}, c_{i+3}, i = 4, 8, 12$, are produced as depicted in Figure 3.8. With those 16 carry bits generated by the 16-bit lookahead carry generator, the 16-bit carry lookahead adder is illustrated in Figure 3.10. The generate and propagate signals are first produced by Equations (3.2) and (3.3). And then, these signals are send into the 16-bit lookahead carry generator, of which the detailed block diagram is depicted in Figure 3.9. Finally, the carry bit at each position as well as the carry-out bit can be achieved and using Equation (3.1) the final sum can be calculated. The 16-bit two's complement adder is then accomplished. However, the checksum calculation is based on the one's complement addition, where the carry-out of 16-bit two's complement addition must be propagated to the carry-in of the adder. In this case, the carry-out of the 16-bit two's complement adder is called the "end around carry". As depicted in Figure 3.10 and 3.9, the carry-out signal is generated prior to the addition (XOR block: $s_i = x_i \oplus y_i \oplus c_i$), and the real carry-in (not the feedback carry-out) for the addition is always zero. We can directly put the carry-out bit of the 16-bit adder to the carry-in so that the one's complement addition is accomplished.

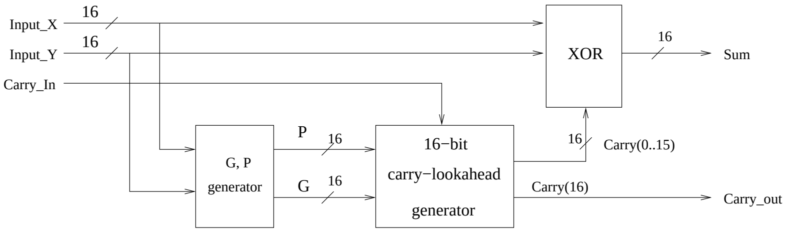


Figure 3.10: 16-bit carry-lookahead adder.

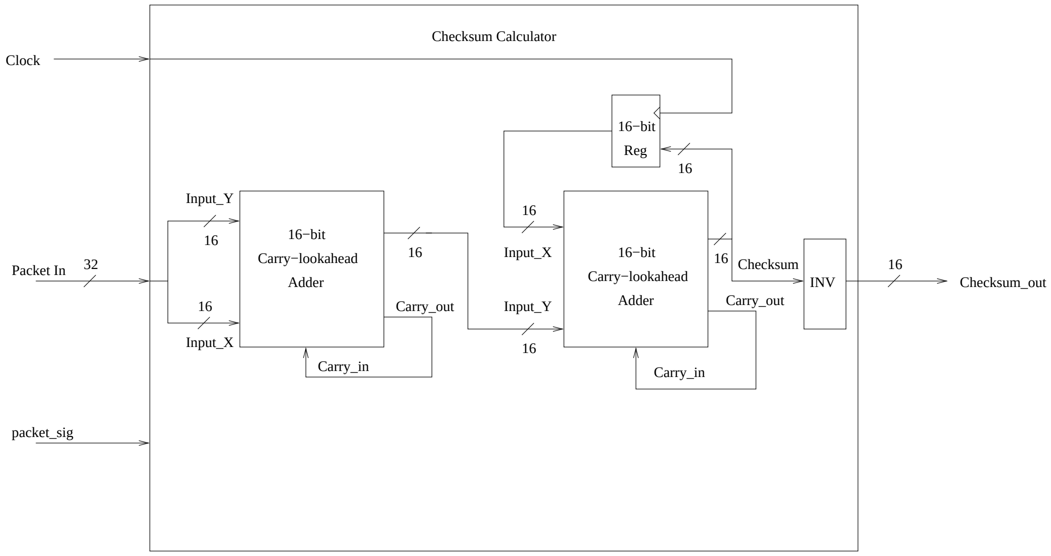


Figure 3.11: Block diagram of checksum calculation.

Moreover, for one's complement addition, the order of the operand inputs is independent. For example, in the right part of Figure 3.6, these 16-bit fields can be added together in any mixed order. Therefore, we can put 32 bits per clock cycle and accumulate the addition result. Two one's complement adders can be implemented to perform the 32-bit calculations. The block diagram of the checksum calculation circuit can be depicted in Figure 3.11. The 32-bit input is separated into two 16-bit half word and perform the addition. And then the addition result is forwarded to another 16-bit one's complement adder to perform accumulation. When the input data is finished, the output in the second adder is complemented and outputed as checksum.

3.3 Cyclic Redundancy Check

In this section, the Cyclic Redundancy Check (CRC) [10] [22] error detection technique is presented. First of all, we describe the background and theory of the CRC technique. CRC-32, which is utilized to generate the Frame Checking Sequence (FCS) in the Ethernet frame is explained as an example. Subsequently, we illustrate the theory of serial and parallel implementation of the CRC-32 algorithm and finally a novel method to update the CRC code in a fast way is proposed.

3.3.1 Introduction

The Cyclic Redundancy Check (CRC) is an error detection technique that is widely utilized in digital data communication and other fields such as data storage, data compression, and etc. There are many CRC algorithms, each of which has a predetermined *generator polynomial* $G(x)$ that is utilized to generate the CRC code. For example, in TCP/IP protocol suite, the most frequently utilized CRC algorithm is the CRC-32 algorithm employed by Ethernet, which has the following generator polynomial:

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x^{1} + x^{0},$$

where m = 32 is the highest order or called the degree of the generator polynomial and also the length of the CRC code. We can extract the coefficient of $G(x)$ and represent it in binary form as:

$$P = \{p_{32}, p_{31}, \ldots, p_1, p_0\} = \{100000100110000010001110110110111\},$$

which has m + 1 = 33 bits. The most significant bit of P, p_{32}, corresponds to the coefficient of x^{32}, the highest order of $G(x)$. Similarly, p_{31} corresponds to the coefficient of x^{31}, which is 0 in this case, and the other bits follow the coefficients at their corresponding positions. P is called the *generator*, and uniquely coincides with the generator polynomial.

When CRC technique is applied, a CRC code is appended to the end of the data message during transmission. Assume that the data message is represented by D, which may have hundreds of bits and the CRC code is denoted by C with the length m, the degree of the generator polynomial. Accordingly, the transmitted data unit with CRC code can be denoted by $T = \{DC\} = D \times 2^m + C$. The CRC code C is generated so that T is an exact multiple of generator P. Therefore, if T is transmitted and there is no error during transmission, the received message $\hat{T}$ must also be an exact multiple of the same P. Otherwise, a transmission error must have occurred. The process of generating CRC codes and detecting transmission errors can be described as follows.

1. m 0s are appended to the data message D, the new data unit can be denoted by $D \times 2^m$.
2. $D \times 2^m$ is divided by the generator P using modulo 2 arithmetic.
3. The remainder of the division is obtained as the CRC code C.
4. The CRC code is appended to the data message D, therefore, the data unit to be transmitted is $T = \{DC\} = D \times 2^m + C$, and is an exact multiple of P. m is the length of the CRC code.
5. The received message $\hat{T}$ is divided by the same generator P. If there is no error introduced during transmission, $\hat{T} = T = D \times 2^m + C$ is still an exact multiple of P. If errors are introduced, $\hat{T} = T + E$ will no longer be the exact multiple of P, resulting in a nonzero remainder after division.

The binary division generally can be performed by a sequence of shifts and subtractions. Furthermore, the modulo 2 division makes addition and subtraction equal to bitwise XOR. Therefore, in modulo 2 arithmetic, binary division can be accomplished by shifts and bitwise XORs. A simple example is illustrated in Figure 3.12 to demonstrate how CRC code is calculated from generator P.

In this example, the data message is $D = \{11011010\}$. The generator polynomial is $G(x) = x^4 + x^1 + x^0$, or in binary form, generator $P = \{10011\}$. And $m = 4$ is the

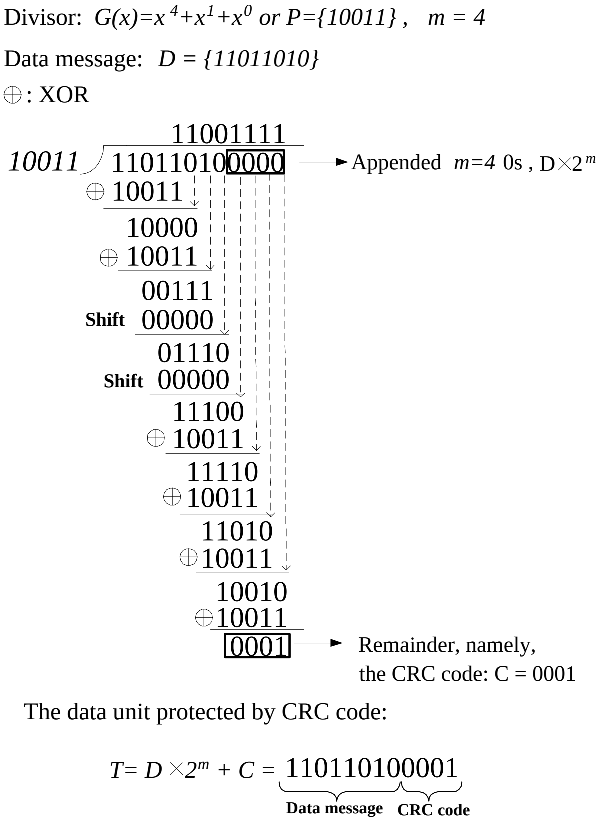

Divisor: G(x)=x^4+x^1+x^0 or P={10011} , m = 4
Data message: D = {11011010}
⊕: XOR
11001111
10011
110110100000
Appended m=4 0s , D×2^m
⊕ 10011
10000
⊕ 10011
00111
Shift 00000
01110
Shift 00000
11100
⊕ 10011
11110
⊕ 10011
11010
⊕ 10011
10010
⊕ 10011
0001
Remainder, namely, the CRC code: C = 0001
The data unit protected by CRC code:
T= D ×2^m + C = 110110100001
Data message
CRC code

Figure 3.12: CRC binary division.

degree of the generator polynomial. In the beginning, 4 0s are appended to the data unit. Subsequently, the data unit with appended zeros is divided by the generator $P = 10011$. Bitwise XORs are first performed over 11011 and the divisor 10011, the remainder 1000 is achieved. The next bit in the data message is concatenated to the remainder, and another bitwise XOR is executed over the remainder and the divisor. When the left most bit of the remainder is '0', a shift operation is performed. The shift/bitwise XOR goes on when finally, all the bits of the dividend are calculated and a remainder of the division is obtained, which is the CRC code, $C = 0001$. The CRC code is appended to the data message to form the new data unit $T = D \times 2^m + C = 110110100001$ that is said to be protected by the CRC code. Consequently, message T is an exact multiple of the divisor P. Message T can be transmitted and if the received message $\hat{T} = T + E$ (E is the possible errors introduced during transmission) is found not to be a multiple of P, then the receiver knows that a transmission error must have occurred ($E \neq 0$).

However, there is some possibility that the introduced error E cannot be detected if E is also a multiple of P. Therefore, the generator P or the generator polynomial $G(x)$ should be carefully chosen so that the likelihood that the error E is a multiple of P is minimized. The probability of the undetected error of CRC code is discussed in detail in [2]. In general, it is possible to prove that the following types of errors can be detected by standardized CRC polynomials[15]:

- All single-bit errors;
- All double-bit errors;
- Any odd number of errors;
- Any burst error, for which the length of the burst error is less than m, the degree of $G(x)$;
- Most burst errors with the length larger than m can also be detected.

Six standardized CRC generator polynomials are widely utilized in TCP/IP protocols as illustrated in Table 3.1. Ethernet and 802.5 use CRC-32, while HDLC uses CRC-CCITT. ATM uses CRC-8, CRC-10, and CRC-32.

CRC	m	$G(x)$
CRC-8	8	$x^8 + x^2 + x + 1$
CRC-10	10	$x^{10} + x^9 + x^5 + x^4 + x + 1$
CRC-12	12	$x^{12} + x^{11} + x^3 + x^2 + 1$
CRC-16	16	$x^{16} + x^{15} + x^2 + 1$
CRC-32	32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11}$
		$+x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$

Table 3.1: CRC generator polynomials in TCP/IP protocol suite.

From above discussion, it is clear that Cyclic Redundancy Check (CRC) is a robust error detecting technique that is widely employed by the TCP/IP protocol suite. The main operation in CRC calculation is the binary division which is performed by a sequence of shift and subtract operations. Modulo 2 arithmetic is adopted, in which multiplication is performed by AND and addition and subtraction are equivalent to bitwise XOR. In the next sections, several CRC implementation methods are going to be introduced, starting from a serial implementation of the CRC calculation.

3.3.2 A Serial Implementation of CRC

As mentioned in Section 3.3.1, the binary division to accomplish CRC can be performed by shifts and subtractions. Modulo 2 arithmetic is employed where substraction is performed by bitwise XOR. It is possible to calculate CRC by either software or hardware. Software method calculates the CRC code through table lookup, and is limited by the lower throughput as discussed in Section 3.1. In hardware implementations, the CRC calculation (modulo 2 division) can be easily performed by logical combinations of shift registers and XOR gates. The Linear Feedback Shift Register (LFSR) [10] [22] is a common approach designed to accomplish the serial calculation of CRC in hardware.

Figure 3.13 illustrates the basic architecture of LFSR for serial calculation of CRC. The flip-flops in the figure are shift registers which store the remainder after every subtraction. The number of shift registers equals m, the degree of the generator polynomial

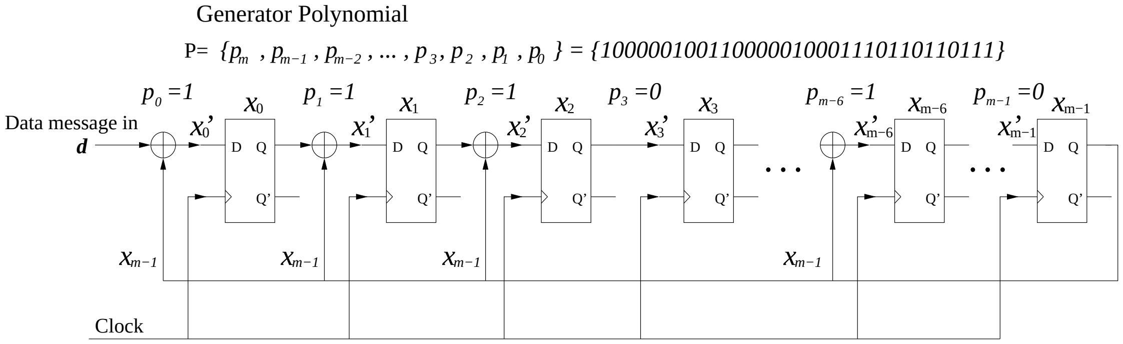


Figure 3.13: Linear Feedback Shift Register (LFSR).

$G(x)$, which is also the length of the CRC code. The data message shifts in from the left, beginning with the most significant bit and ending with the m 0s that is attached to the data message. When all the messages have been shifted in, the final value in the shift registers is the remainder of the division, namely, CRC code. Let $X = \{x_{m-1}, ..., x_1, x_0\}$ denote the current state of the shift registers. d is the serial input of the data unit. $X' = \{x'_{m-1}, ..., x'_1, x'_0\}$ is the input of the shift registers at the next clock cycle. From Figure 3.13, the states of the registers can be written as:

$$x'_0 = (p_0 \cdot x_{m-1}) \oplus d \tag{3.12}$$

$$x'_i = (p_i \cdot x_{m-1}) \oplus x_{i-1}, \ i = \{1, 2, ..., m-1\}. \tag{3.13}$$

If $p_i = 1$, there is an XOR operation between two registers and if $p_i = 0$, $x'_i = x_{i-1}$, which is purely a shift operation.

The LFSR can be regarded as a discrete-time time-invariant linear system. In linear system theory, the state equation for LFSRs can then be expressed as follows (see [10] [19]):

$$X(i+1) = F \cdot X(i) + G \cdot U(i) \tag{3.14}$$

where, $X = [x_{m-1} \ldots x_1 \ x_0]^T$ denotes the state of the shift registers;
$X(i)$ represents the ith state of the registers, namely, the remainder after ith subtraction;
$X(i+1)$ denotes the $(i+1)$th state of the registers, the remainder after $(i+1)$th subtraction;
$U(i)$ denotes the ith serial input bit;
F is a $m \times m$ matrix and G is a $1 \times m$ matrix.

If the generator polynomial is written as $P = \{p_m, p_{m-1}, ..., p_0\}$, F and G matrices can be derived from Equations (3.12) and (3.13) as follows:

$$F = \begin{bmatrix} p_{m-1} & 1 & 0 & \cdots & 0 \\ p_{m-2} & 0 & 1 & \cdots & 0 \\ \ldots & \ldots & \ldots & \ddots & \ldots \\ p_1 & 0 & 0 & \cdots & 1 \\ p_0 & 0 & 0 & \cdots & 0 \end{bmatrix} \tag{3.15}$$

$$G = [0\,0 \cdots 0\,1]^T \tag{3.16}$$

$$U = d \tag{3.17}$$

If $X = [x_{m-1} \ldots x_1 \; x_0]^T$ is used to denote the current state: ith state, and $X' = [x'_{m-1} \ldots x'_1 \; x'_0]^T$ is employed to denote the next state: $(i+1)$th state, the system state equation can thus be written as:

$$X' = F \cdot X + G \cdot d \tag{3.18}$$

Furthermore, if F and G are substituted by Equations (3.15) and (3.16), we can write Equation (3.18) in matrix form as:

$$\begin{bmatrix} x'_{m-1} \\ x'_{m-2} \\ \cdots \\ x'_1 \\ x'_0 \end{bmatrix} = \begin{bmatrix} p_{m-1} & 1 & 0 & \cdots & 0 \\ p_{m-2} & 0 & 1 & \cdots & 0 \\ \cdots & \cdots & \cdots & \ddots & \cdots \\ p_1 & 0 & 0 & \cdots & 1 \\ p_0 & 0 & 0 & \cdots & 0 \end{bmatrix} \cdot \begin{bmatrix} x_{m-1} \\ x_{m-2} \\ \cdots \\ x_1 \\ x_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ \cdots \\ 0 \\ 1 \end{bmatrix} \cdot d \tag{3.19}$$

Expand Equation (3.19), the equations are expressed as follows[1]:

$$\begin{cases} x'_{m-1} & = & (p_{m-1} \cdot x_{m-1}) & \oplus\, x_{m-2} \\ x'_{m-2} & = & (p_{m-2} \cdot x_{m-1}) & \oplus\, x_{m-3} \\ & \vdots & & \\ x'_1 & = & (p_1 \cdot x_{m-1}) & \oplus\, x_0 \\ x'_0 & = & (p_0 \cdot x_{m-1}) & \oplus\, d \end{cases} \tag{3.20}$$

Equation (3.20) exactly coincides with the LFSR in Figure 3.13 and Equations (3.12),(3.13). If $p_i = 1$, we have $x'_i = x_{m-1} \oplus x_{i-1}$ $(i = 1, 2, ..., m-1)$, which means there is an XOR operation between register x_i and register x_{i-1}. If $p_i = 0$, we have $x'_i = x_{i-1}$, which means there is no XOR operation between these two registers, only shift operation is performed. If $x_{m-1} = 0$, we have $x_0 = d$ and $x'_i = x_{i-1}$ for $i = 1, 2, ..., m-1$, which corresponds to the shift operation in Figure 3.12 when the left most bit is '0'.

The performance of the LFSR for serial CRC calculation is evaluated as follows. As observed from Figure 3.13, there is only one XOR gate between two flip-flops. The latency for the LFSR is estimated as:

$$T_{LFSR} = T_{reg} + T_{XOR}$$

where T_{reg} represents the total flip-flop delay, and T_{XOR} is the delay of an XOR gate. Assume that the data message has k bits and the CRC code is m-bit long. For serial implementation of CRC calculation, the data message is shifted in one bit per clock cycle. Therefore, $(k+m)$ cycles are required to perform the calculation of a CRC code.

---

[1] Note: the multiplication of matrix also follows the modulo 2 arithmetic, that is, multiplication is performed by AND and addition is performed by XOR.

3.3.3 The Parallel Implementation of CRC

In the serial implementation of CRC, the data is processed one bit per clock cycle. In total, $(k{+}m)$ cycles are needed to obtain a CRC code for a message with k bits. Although the serial implementation is simple and can run at a high clock rate, it suffers from the low data throughput for its serial input. The parallel CRC implementation [10] [19] is designed to increase the throughput of CRC calculation by processing multiple input bits in parallel every clock cycle.

In this section, we extend the notation introduced in Section 3.3.2 in order to describe the parallel CRC implementation. The linear system state equation Equation (3.14) is rewritten as follows. The parallel CRC calculation equation is derived from this equation.

$$X(i{+}1) = FX(i) + GU(i) \tag{3.21}$$

Remember $X(i)$ is the ith state of the shift registers, $X(i+1)$ is the $(i+1)$th state of the shift registers. $U(i)$ is the serial input bit at state $X(i)$, F is a $m \times m$ matrix and G is a $1 \times m$ matrix.

The solution for the system state equation can be written as (see [10]):

$$X(i) = F^i X(0) + [F^{i-1}G,\ \ldots,\ FG,\ G][U(0), \ldots,\ U(i-1))]^T \tag{3.22}$$

where $X(i)$ denotes the ith state of the shift registers, F^i is the F matrix to the power of i, which is still a $m \times m$ matrix. $X(0)$ is the initial state of the shift registers,

$$[F^{i-1}G,\ \ldots,\ FG,\ G][U(0),\ \ldots,\ U(i-2),\ U(i-1)]^T = [0,\ \ldots,\ 0, U(0),\ \ldots,\ U(i-1)]^{T2} \tag{3.23}$$

Input vector $U = [U(0),\ \ldots,\ U(i-2),\ U(i-1)]$ is a collection of the serial input bits from the initial state input bit $U(0)$ to the $(i-1)$th state input $U(i-1)$. Equation (3.22) can be simplified as:

$$X(i) = F^i X(0) + [0,\ \ldots,\ 0, U(0),\ \ldots,\ U(i-1)]^T \tag{3.24}$$

Equation (3.24) demonstrates how the ith state of the shift registers $X(i)$ is related to the initial states $X(0)$ and the input vector $U = [U(0), U(1), ..., U(i-1)]$. The idea of parallel CRC calculation is originated from this equation, that is, to calculate $X(i)$ directly from $X(0)$ and the input vector $U = [U(0), U(1), ..., U(i-1)]$ in one clock cycle. The calculation of the intermediate states $X(1), \ldots, X(i-1)$ are eliminated. If $i = 1$, namely, only one bit is processed in one clock cycle, we have $X(1) = FX(0) + [0 \ldots 0, U(0)]$, which is the same as the serial implementation. If $i = 2$, namely, 2 bits are processed in parallel in one clock cycle, we have $X(2) = F^2 X(0) + [0, \ldots 0, U(0)\ U(1)]$, which calculates the 2^{nd} states of the registers from the initial states $X(0)$, and the input vector $[U(0)U(1)]$ in one clock cycle.

Now, assuming that w multiple bits of the data unit are processed in parallel in one clock cycle ($w \leq m$, where m is the length of the CRC code). The w-bit input vector is $U = [U(0),\ U(1),\ \ldots,\ U(w-1)]$. Since the calculation starts from

---

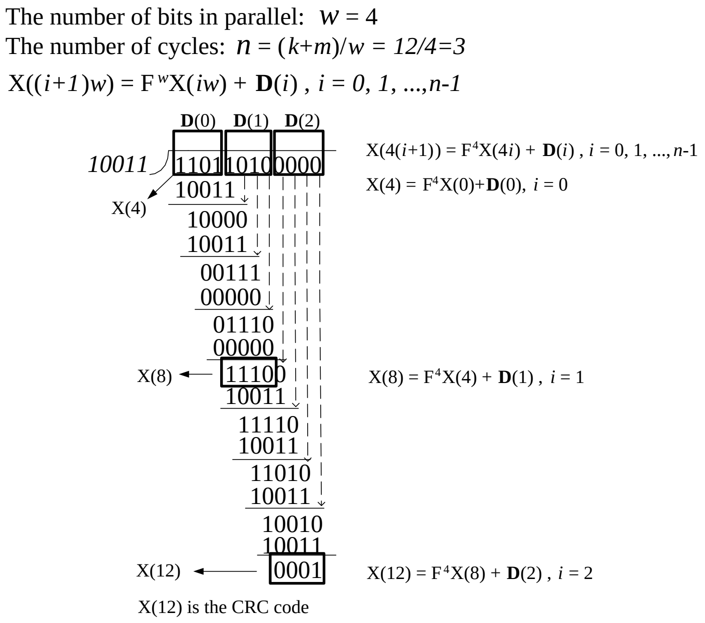
[2] The proof will be given in Appendix. B

the most significant bit, $U(0)$, the first shift-in bit, can be represented by d_{w-1}. Similarly, $U(1) = d_{w-2}$, $\ldots$, $U(w-1) = d_0$. Furthermore, if $\mathbf{D}(0)$ is utilized to denote the m-bit parallel input vector $[0, \ldots, 0, U(0), \ldots, U(i-1)]^T$ in Equation 3.24, we have $\mathbf{D}(0) = [0, \ldots 0, d_{w-1}, d_{w-2}, \ldots, d_1, d_0]^T$. Now, we have $i = w$ and $\mathbf{D}(0) = [0, \ldots, 0, d_{w-1}, d_{w-2}, \ldots, d_1, d_0]^T$, Equation (3.24) becomes:

$$X(w) = F^w X(0) + \mathbf{D}(0) \tag{3.25}$$

From Equation 3.25, we can get the register state after the first w-bit parallel calculation, $X(w)$, from initial state $X(0)$ and the first parallel input $\mathbf{D}(0)$. Furthermore, since the system is time-invariant, we can calculate the register state after $(i{+}1)$th w-bit parallel calculation, $X((i+1)w)$, by the ith register state $X(iw)$ (state after the ith w-bit parallel calculation) and the input vector at the state $X(iw)$. If $\mathbf{D}(i)$ is utilized to denote the input vector at the state $X(iw)$, we have $\mathbf{D}(i) = [0, \ldots, 0, d_{iw+w-1}, d_{iw+w-2}, \ldots, d_{iw}]^T$. Therefore, Equation 3.25 can be extended as follows.

$$X((i+1)w) = F^w X(iw) + \mathbf{D}(i) \tag{3.26}$$



The number of bits in parallel: w = 4
The number of cycles: n = (k+m)/w = 12/4=3
X((i+1)w) = F^wX(iw) + D(i) , i = 0, 1, ...,n-1
D(0) D(1) D(2)
10011
1101 1010 0000
X(4)
10011
10000
10011
00111
00000
01110
00000
X(8)
1110
10011
11110
10011
11010
10011
10010
10011
X(12)
0001
X(12) is the CRC code
X(4(i+1)) = F^4X(4i) + D(i) , i = 0, 1, ..., n-1
X(4) = F^4X(0)+D(0), i = 0
X(8) = F^4X(4) + D(1) , i = 1
X(12) = F^4X(8) + D(2) , i = 2

Figure 3.14: CRC binary division example, the number of parallel bits w = 4.

Figure 3.14 demonstrates an example of parallel CRC calculation with multiple input bits $w = m = 4$. The example in Figure 3.12 is re-listed here, but now parallel calculation is employed. The dividend is divided into three 4-bit fields, acting as the parallel input vectors $\mathbf{D}(0), \mathbf{D}(1), \mathbf{D}(2)$, respectively. The initial state is $X(0) = [0\,0\,0\,0]^T$. From Equation (3.26), we have,

$$\begin{aligned} X(4) &= F^4 \cdot X(0) + \mathbf{D}(0) \\ X(8) &= F^4 \cdot X(4) + \mathbf{D}(1) \\ X(12) &= F^4 \cdot X(8) + \mathbf{D}(2) \end{aligned}$$

From above equations, it is clear that the parallel approach is to merge a number of substraction or shift operations into one single clock cycle. From $X(0)$ and $\mathbf{D}(0)$, $X(4)$ is obtained in one clock cycle. From $X(4)$ and $\mathbf{D}(1)$, $X(8)$ is achieved and so on. And finally, the state $X(12)$ is the CRC code. From this example, it is obvious that we need only 3 clock cycles to get the CRC code in parallel CRC calculation while we need 12 cycles in serial CRC calculation.

Therefore, for parallel calculation, if the next states $X((i+1)w)$ is denoted by X', the current states $X(iw)$ is denoted by X and the parallel input vector is denoted by $\mathbf{D}$, we have,

$$X' = F^w \cdot X \oplus \mathbf{D} \tag{3.27}$$

This is the recursive equation developed to calculate CRC code in parallel. F^w is the wth power of F matrix, which is still a $m \times m$ matrix. $X = [x_{m-1},\ x_{m-2},\ \ldots, x_1,\ x_0]^T$ is the state of the m shift registers. $\mathbf{D}$ is the input vector with the size m. It is notable that the parallel calculation of CRC code is a recursive calculation of the next state of the registers X' by the current states X, predetermined matrix F^w and the parallel input $\mathbf{D}$ using Equation (3.27). For example, in Figure 3.14, $X(8)$ is calculated from $X(4)$, F^4 and $\mathbf{D}(1)$. The recursive calculation is terminated when the input bits are finished.

Equation (3.27) is a generic equation that can apply to any generator polynomial $G(x)$ and any number of parallel bits w ($w \leq m$, where m is the degree of $G(x)$). In Section 3.3.1, several standardized CRC generator polynomials were introduced as demonstrated in Table 3.1. All these generator polynomials can fit in Equation (3.27). We choose the most complex and the most frequently utilized generator polynomial in TCP/IP, CRC-32, as an example. The generator polynomial of CRC-32 is:

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + x^0,$$

and generator can be written as:

$$P = \{p_{32}, p_{31}, \ldots, p_1, p_0\} = \{100000100110000010001110110110111\},$$

We have $m = 32$, the degree of $G(x)$, therefore, there are 32 shift registers and $X = [x_{31}, x_{30}, \ldots, x_1, x_0]^T$. From Equation (3.15), the 32×32 matrix F_{32} goes as follows:

$$F_{32} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 & 0 \\ \ldots & \ldots & \ldots & \ddots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & \cdots & 1 & 0 & 0 \\ 1 & 0 & 0 & \cdots & 0 & 1 & 0 \\ 1 & 0 & 0 & \cdots & 0 & 0 & 1 \\ 1 & 0 & 0 & \cdots & 0 & 0 & 0 \end{bmatrix}$$

[Note: from now on, our discussion will be based on the CRC-32 algorithm, therefore, F_{32} is denoted by F for simplicity. In the following text, if it is not mentioned, F will be used to represent the F matrix for CRC-32, F_{32}.]

The number of parallel input bits w can be any number between 1 and 32. For the reason that most network protocols are byte-based, the parallel input bits are generally selected as a multiple of 8, for example, $w = 8, 16, 32$. Since the parallel input vector can be denoted by $\mathbf{D} = [0, \ldots, 0,\ d_{w-1}, d_{w-2}, \ldots, d_1, d_0]^T$, different $\mathbf{D}$ for different w can be obtained. If $w = m = 32$, $\mathbf{D} = [d_{31}, d_{30}, \ldots, d_1, d_0]^T$. Otherwise, if $w < m = 32$, $\mathbf{D} = [0\ \ldots\ 0\ d_{w-1}\ d_{w-2}\ \ldots\ d_1\ d_0]$, $(m-w)$ zeros are put in front. In this thesis, $w = 32$ is selected as the parallel input in order to increase the throughput of the calculation as high as possible. Therefore, the 32×32 matrix F^{32} need to be pre-calculated. Accordingly, Equation (3.27) becomes

$$X' = F^{32} \cdot X \oplus \mathbf{D} \tag{3.28}$$

If Equation (3.28) is represented in matrix form, we have,

$$\begin{bmatrix} x'_{31} \\ x'_{30} \\ \vdots \\ x'_1 \\ x'_0 \end{bmatrix} = \begin{bmatrix} F^{32}(31)(31) & F^{32}(31)(30) & \cdots & F^{32}(31)(0) \\ F^{32}(30)(31) & F^{32}(30)(30) & \cdots & F^{32}(30)(0) \\ \cdots & \cdots & \ddots & \cdots \\ F^{32}(1)(31) & F^{32}(1)(30) & \cdots & F^{32}(1)(0) \\ F^{32}(0)(31) & F^{32}(0)(30) & \cdots & F^{32}(0)(0) \end{bmatrix} \cdot \begin{bmatrix} x_{31} \\ x_{30} \\ \vdots \\ x_1 \\ x_0 \end{bmatrix} \oplus \begin{bmatrix} d_{31} \\ d_{30} \\ \vdots \\ d_1 \\ d_0 \end{bmatrix} \tag{3.29}$$

where $F^{32}(i)(j), i = 0 \ldots 31, j = 0 \ldots 31$ is the value at ith row and jth column of F^{32} matrix. Expand above equation, we get,

$$\begin{aligned} x'_{31} &= \Big(F^{32}(31)(31) \cdot x_{31}\Big) \oplus \Big(F^{32}(31)(30) \cdot x_{30}\Big) \ldots \oplus \Big(F^{32}(31)(0) \cdot x_0\Big) \oplus d_{31} \\ x'_{30} &= \Big(F^{32}(30)(31) \cdot x_{31}\Big) \oplus \Big(F^{32}(30)(30) \cdot x_{30}\Big) \ldots \oplus \Big(F^{32}(30)(0) \cdot x_0\Big) \oplus d_{30} \\ \vdots & \quad \vdots \\ x'_1 &= \Big(F^{32}(1)(31) \cdot x_{31}\Big) \oplus \Big(F^{32}(1)(30) \cdot x_{30}\Big) \ldots \oplus \Big(F^{32}(1)(0) \cdot x_0\Big) \oplus d_1 \\ x'_0 &= \Big(F^{32}(0)(31) \cdot x_{31}\Big) \oplus \Big(F^{32}(1)(30) \cdot x_{30}\Big) \ldots \oplus \Big(F^{32}(0)(0) \cdot x_0\Big) \oplus d_0 \end{aligned}$$

Consequently, the block diagram of parallel calculation of CRC-32 can be drawn based on above equations. The block diagram is depicted in Figure 3.15. Only the inputs x'_0 and x'_{31} have been illustrated. They coincide with the equation above. Other inputs are omitted by the figure, but they all follow the corresponding equations expanded from Equation (3.29).

Furthermore, it is proved in paper [10] that other matrices F^w with the number of parallel bits $w < 32$, such as F^8, F^{16}, can be derived from matrix F^{32} as:

$$F^w = [\text{the last } w \text{ columns of } F^{32} \mid \frac{I_{32-w}}{0}]$$

The first w columns of F^w are the last w columns of F^{32}, the upper part of the rest $(32-w)$ columns are identity matrix I_{32-w} with the size $(32-w) \times (32-w)$, and the lower part are all zeros. This property of the F^w matrix and the previously mentioned fact that Equation (3.27) can be regarded as a recursive calculation of the next state X' by matrix F^w, current state X and parallel input $\mathbf{D}$, make the 32-bit parallel input vector suitable for any length of messages besides the multiple of 32 bits. Remember that the length of the message is byte-based. If the length of message is not the multiple of 32, after a sequence of 32-bit parallel calculation, the final remaining number of bits of the message could be $8, 16$,or 24. For all these situations, an additional parallel calculation

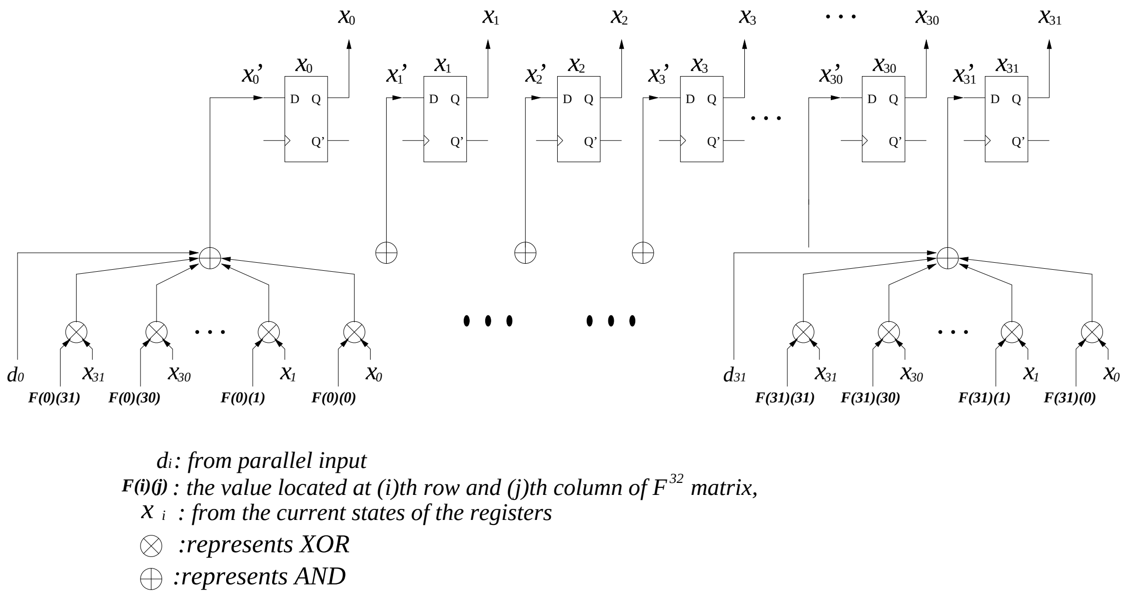


Figure 3.15: Block diagram for parallel calculation of CRC-32.

$w = 8, 16, 24$ is needed by choosing the corresponding F^w. Since F^w can be easily derived from F^{32}, the calculation can be performed using Equation 3.27 within the same circuit as 32-bit parallel calculation, the only difference is the F^w matrix. An simple example is depicted in Figure 3.16. This example differs with the example given in Figure 3.14 only in that two more bits "01" are added to the data message resulting in that the length of the message is not the multiple of the number of parallel processing bits $w = 4$. From Figure 3.16, two more bits ($\mathbf{D}(3)$) need to be calculated after getting $X(12)$. Therefore, F^2 must be obtained from matrix F^4, and the extra two bits are stored at the lower significant bits of the input vector $\mathbf{D}$. Equation (3.27) can then be applied to calculate the final state $X(14)$, which is the CRC code. Therefore, only an extra cycle is needed for calculating the extra bits if the data message length is not the multiple of w, the number of parallel processing bits.

It is worth to notice that in CRC-32 algorithm, the initial state of the shift registers is preset to all '1's. Therefore, $X(0) = 0xFFFF$. However, the initial state X(0) does not affect the correctness of the design. In order for better understanding, the initial state $X(0)$ is still set to $0x0000$ when the circuit is implemented in Chapter 4.

The performance of parallel CRC calculation is evaluated as follows. The increased latency of the parallel calculation of CRC code, compared with the serial LFSR implementation, is the level of XOR gates introduced. As can be observed from Figure 3.15, the 32 registers are running in parallel, the maximum latency of the whole system depends on the maximum number of XORs at the inputs of the registers. The number of XORs can be estimated by the number of '1's in a row of the F^{32} matrix. For example, the input of Register x_{31} in Figure 3.15, the number of XORs is equal to the number of '1's in the row $F(31)(0) \dots F(31)(31)$ and plus 1 for the one-bit input d_{31}. From Appendix A, we can read that there are 13 '1's in the first row of the F^{32}

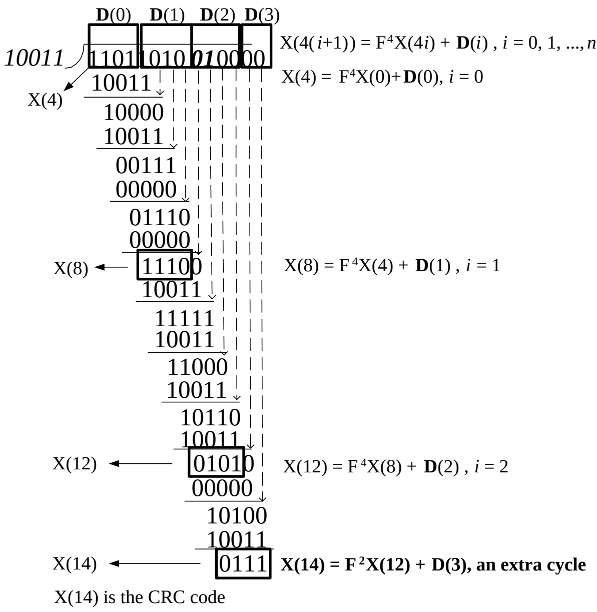


Figure 3.16: Parallel CRC for messages with the length not equal to the multiple of 32.

matrix, consequently, the number of XORs for the input of Register x_{31} is 14 (13 plus one data message input d_{31}). Therefore, 4 XOR gate levels are needed to accomplish the 14 XORs using 2-input XORs without optimization. Finally, a maximum of 17 '1's found in matrix F^{32} leads to 5 XOR gate levels. And there is one AND gate level as can be observed from the figure. Accordingly, the total latency of the parallel calculation of CRC is estimated as:

$$T_{Parallel} = T_{reg} + 5T_{XOR} + T_{AND}$$

where T_{reg} is the total flip-flop delay, and T_{XOR} is a delay of an XOR gate, and T_{AND} is the delay of an AND gate. If we assume that the length of the data is k bits and the degree of generator polynomial is m bits, the data used to generate CRC code has the length of k+m bits. In parallel calculation of CRC, we divide the m+k bits into n w-bit units ($n = \frac{k+m}{w}$). Using equation (3.27), n clock cycles are required so as to get the final state $X(n)$, which is the CRC code.

3.4 Fast CRC Update

In parallel CRC calculation, $n = \frac{k+m}{w}$ cycles are needed to obtain the CRC code, where k is the length of the data message, m is the length of the CRC code and w is the number of bits processed in parallel. In calculating the Frame Check Sequence (FCS) of an Ethernet frame, the CRC code is calculated over a frame with length ranging from 64 bytes to 1518 bytes. Assuming a parallel CRC calculation with $w = 32$, the number of cycles to calculate the FCS ranging from 16 cycles ($\frac{64bytes \times 8}{32} = 16$) to 380 cycles

[3]$(\frac{1518bytes \times 8}{32} = 380)$.

If we investigate the situations that how CRC is calculated and verified during the whole process of data transmission over the Internet, we can find that when a frame reaches an interconnecting device, the following steps are performed in sequence. The process is also demonstrated in Figure 3.17:

1. CRC verification is performed in order to verify the correctness of the frame. If it is failed to verify, the frame is discarded.
2. The frame is passed through upper layers. Upper layer operations, such as finding the next hop according to the destination address, updating the TTL (Time To Live) field and checksum field in the IP packet header and etc. are performed.
3. The reassembled frame without FCS is forwarded to perform CRC32 calculation.
4. The calculated FCS will be appended to the frame and the frame will be forwarded through physical layer to its next hop.

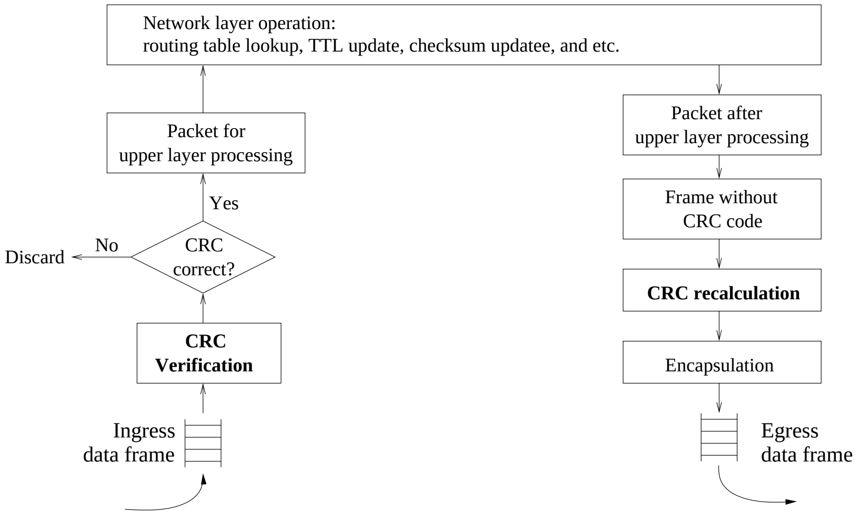


Figure 3.17: CRC verification and recalculation.

Furthermore, the difference between an incoming frame and an outgoing frame (after being updated by higher layer operations) is small and the differences usually occur at fixed bit-positions [7] [8]. Figure 3.18 depicts a detailed Ethernet frame format. The fields in gray are the most likely to be changed fields. The TTL field in the IP header is decreased by 1 every time when a packet passes through a router. The Header Checksum in the IP header also changes accordingly. The source Ethernet address and the destination Ethernet address are different as well. All of them will be modified when the frame is sent out. The changed fields, 15 bytes in total, are only a small portion of the whole frame, which may range from 64 byte to 1518 bytes. Furthermore, we observe

---

[3] The number of cycles are ceiled to the smallest integer not less than the result, if it is not the multiple of 32. Please refer to the situation depicted in Figure 3.16

that the changed fields lies in the first 26 bytes of the frame and the rest of the frame remains intact. Therefore, it is not necessary to recalculate the CRC code over the entire frame, instead, a fast CRC update is more preferable.

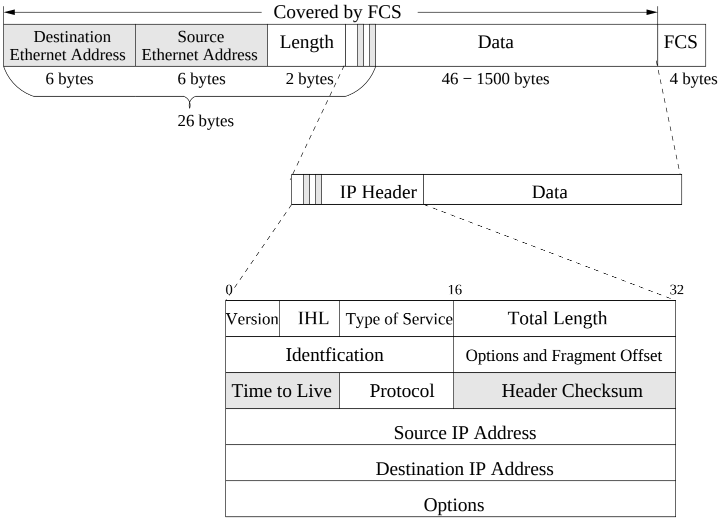


Figure 3.18: Detailed ethernet frame structure.

Our fast CRC update design is based on above observations and the state equation discussed in Section 3.3.3:

$$X' = F^{32} \cdot X \oplus \mathbf{D} \tag{3.30}$$

where F^{32} is a 32×32 matrix for CRC32, X' and X is the next and current states of the register, respectively and $\mathbf{D}$ is the parallel input vector. Furthermore, the associative and distributive laws apply for the modulo 2 arithmetic, that is,

$$\begin{array}{rcl} a \cdot (b \oplus c) & = & (a \cdot b) \oplus (a \cdot c) \\ a \oplus (b \oplus c) & = & (a \oplus b) \oplus c \end{array}$$

Now, assuming that the states of the shift register follow the sequence $X(0), X(1), X(2), \ldots, X(n)$ and the parallel inputs follow the sequence $\mathbf{D}(0), \mathbf{D}(1), \mathbf{D}(2), \ldots, \mathbf{D}(n-1)$, where $n = \frac{k+m}{w}$ is the number of cycles to get the final CRC code. From the recursive equation Equation (3.30), we have:

$$\begin{array}{rcl} X(1) & = & F^{32} \cdot X(0) \ \oplus \mathbf{D}(0) \\ X(2) & = & F^{32} \cdot X(1) \ \oplus \mathbf{D}(1) \\ & \cdots & \\ X(7) & = & F^{32} \cdot X(6) \oplus \mathbf{D}(6) \\ \hline X(8) & = & F^{32} \cdot X(7) \oplus \mathbf{D}(7) \\ X(9) & = & F^{32} \cdot X(8) \oplus \mathbf{D}(8) \\ & = & F^{32} \cdot \Big(F^{32} \cdot X(7) \oplus \mathbf{D}(7) \Big) \oplus \mathbf{D}(8) \\ & = & \Big((F^{32})^2 \cdot X(7) \Big) \ \oplus \ \Big(F^{32} \cdot \mathbf{D}(7) \Big) \ \oplus \ \mathbf{D}(8) \\ & \cdots & \end{array}$$

The equations above the line, are calculated form the changed fields, from $\mathbf{D}(0)$ to $\mathbf{D}(6)$. And the equations below the line are calculated from the unchanged fields and they are all denoted by the intermediate state $X(7)$. Finally, the final register states $X(n)$, $n = \frac{k+m}{w}$, namely the CRC code, can be denoted by the intermediate states $X(7)$ and the unchanged fields.

$$CRC = X(n) = \Big((F^{32})^{n-7} \cdot X(7)\Big) \;\oplus\; \Big((F^{32})^{n-8} \cdot \mathbf{D}(7)\Big) \;\oplus \ldots \oplus \mathbf{D}(n-1) \qquad (3.31)$$

For the old CRC code before frame update and the new CRC code after the update, Equation (3.31) can be described as follows:

$$CRC_{old} = \Big((F^{32})^{n-7} \cdot X(7)_{old}\Big) \;\oplus\; \Big((F^{32})^{n-8} \cdot \mathbf{D}(7)\Big) \oplus \ldots \oplus \mathbf{D}(n-1)$$

and

$$CRC_{new} = \Big((F^{32})^{n-7} \cdot X(7)_{new}\Big) \;\oplus\; \Big((F^{32})^{n-8} \cdot \mathbf{D}(7)\Big) \oplus \ldots \oplus \mathbf{D}(n-1)$$

As discussed before, all differences between the old and new frame located in the first 28 bytes of the frame. Therefore, for $i > 6$, the parallel input $\mathbf{D}(i)$ remains unchanged. If we XOR two CRC codes, the two set of unchanged fields are eliminated:

$$\begin{aligned} CRC_{new} \oplus CRC_{old} &= \Big((F^{32})^{n-7} \cdot X(7)_{new}\Big) \oplus \Big((F^{32})^{n-7} \cdot X(7)_{old}\Big) \\ &= (F^{32})^{n-7} \cdot \Big(X(7)_{new} \oplus X(7)_{old}\Big) \end{aligned}$$

therefore, the new CRC code can be represented in the following form:

$$CRC_{new} = (F^{32})^{n-7} \cdot \Big(X(7)_{new} \oplus X(7)_{old}\Big) \;\oplus\; CRC_{old} \qquad (3.32)$$

Consequently, the block diagram of the CRC update circuit can be depicted in Figure 3.19. The CRC recalculation is only performed over the first 28 bytes of the frame, the intermediate state $X(7)$ is collected, and compared with the old one which can be obtained through CRC verification, and perform the update. Finally, the new CRC is obtained by adding the update result to the old CRC code.

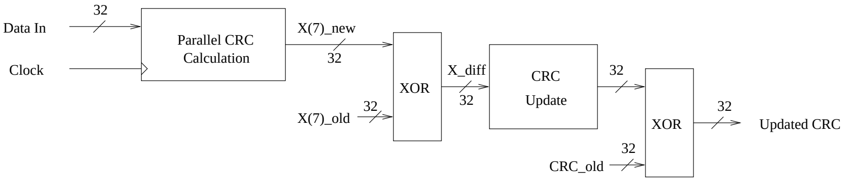


Figure 3.19: Block diagram of fast CRC update circuit.

In Equation (3.32), for a fixed frame length, the matrix $(F^{32})^{n-7}$ ($n = \frac{k+m}{w}$) can be precalculated, $X(7)_{old}$ can be obtained from the CRC verification, $X(7)_{new}$ is calculated through parallel CRC circuit. For the smallest frame size, 64 bytes, we have $n = \frac{64\text{bytes} \times 8\text{bits}}{32} = 16$. Equation (3.32) can be written as

$$CRC_{new} = (F^{32})^{9} \cdot \Big(X(7)_{new} \oplus X(7)_{old}\Big) \;\oplus\; CRC_{old} \qquad (3.33)$$

where $(F^{32})^9$ is a fixed 32×32 matrix, $X(7)_{new}$ and $X(7)_{old}$ can be calculated from parallel CRC circuit. CRC_{old} is the original CRC code and CRC_{new} is the updated CRC code. Therefore, for CRC fast update of a 64-byte frame, we only need to calculate the first 26 bytes of the frame, namely get intermediate state X(7) and then perform the CRC update to get the new CRC code.

For the calculation of the first 26 bytes CRC, 7 cycles are needed (32-bit parallel calculation) and for update, only one cycle is needed. In total only 8 cycles are needed for the fast update of the 64-byte frame with $(F^{32})^9$ precalculated.

Furthermore, as can be observed on the real Internet, there are three predominant frame sizes: 64 bytes, 596 bytes and 1518 bytes. They occupy $35\%, 11.5\%$ and 10% of the total, respectively [17]. Similar results can also be found in [6] and [27]. The descriptions of the predominant frame are listed in Table 3.2.

Frame size	Distribution	Frame type description
64 bytes	35%	The minimum-size Ethernet frame, consisting of 14 bytes Ethernet header, 20 bytes IP header, 26 bytes payload and 4 bytes Ethernet trailer (FCS).
594 bytes	11.5%	Ethernet frame including 14 bytes Ethernet header, 20 bytes IP header, 556 bytes payload and 4 bytes Ethernet trailer (FCS).
1518 bytes	10%	The maximum-size Ethernet frame, consisting of 14 bytes Ethernet header, 20 bytes IP header, 1480 bytes payload and 4 bytes Ethernet trailer (FCS).

Table 3.2: Three most frequently occurred frame in the real Internet[17].

It is wise to find the corresponding $(F^{32})^{n-7}$ for these frame sizes and implement them as CRC update units as well so that the improvement is maximized. Accordingly, for the frames of these sizes, only 8 cycles are need to calculate CRC code compared with the $n = \frac{k+m}{w}$ cycles in the parallel CRC calculation circuit. Figure 3.20 depicts the comparison, the performance improvement grows linearly as the increase of the frame size.

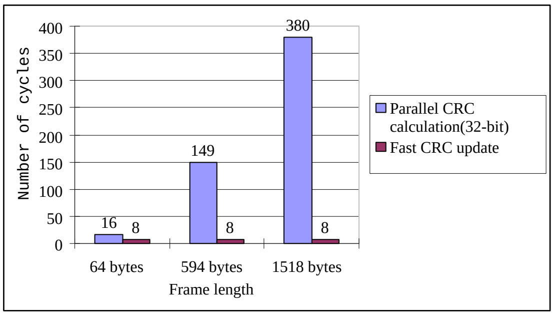


Figure 3.20: Number of cycles: comparison.

Once the $(F^{32})^{n-7}$ is determined, for example, we have $(F^{32})^9$ in Equation (3.33), the

CRC update scheme can be extended to update the CRC code for the frames with any length. Suppose h cycles are needed to calculate parallel 32-bit CRC over a frame with random length (obviously $h \geq 16$, which is the cycles for the smallest 64-byte frame). From Equation (3.31), we have $n = h$, the final CRC code is

$$X(h) = \Big((F^{32})^9 \cdot X(h-9)\Big) \oplus \Big((F^{32})^8 \cdot \mathbf{D}(h-9)\Big) \oplus \Big((F^{32})^7 \cdot \mathbf{D}(h-8)\Big) \oplus \cdots \oplus \mathbf{D}(h-1) \tag{3.34}$$

Since $h \geq 16$, $h - 9 \geq 7$, from $\mathbf{D}(h-9)$ on, the inputs are surely those intact fields in the frame. Equation 3.34 is equivalent to Equation 3.31 except the state is changed to $X(h-9)$. Therefore, Equation (3.33) can be rewritten as:

$$CRC_{new} = (F^{32})^9 \cdot \Big(X(h-9)_{new} \oplus X(h-9)_{old}\Big) \ \oplus \ CRC_{old} \tag{3.35}$$

Equation (3.35) differs with Equation (3.33) only in $X(h-9)$, which depends on the size of the frame. We stop the parallel CRC calculation at the stage $X(h-9)$, and perform one step CRC update. Therefore, for any frame size, in fast CRC update scheme, at least 8 cycles is reduced compared with the parallel CRC calculation. If we can build more circuits for $(F^{32})^{n-7}$, for example, $(F^{32})^{n-7}$ for 64 bytes, 128 bytes, 256 bytes etc, much more cycles will be reduced. Equation 3.35 then becomes:

$$CRC_{new} = (F^{32})^{n-7} \cdot \Big(X(h-(n-7))_{new} \oplus X(h-(n-7))_{old}\Big) \ \oplus \ CRC_{old} \tag{3.36}$$

In our thesis, we only build and test the circuits for 64 bytes, 594 bytes and 1518 bytes. Their corresponding $(F^{32})^{n-7}$ matrices are precalculated in Matlab.

The performance of the CRC update circuit is as follows. As can be seen from Figure 3.19, the block "parallel CRC calculation" is the same as the one discussed in Section 3.3.3. If we compare Equation 3.33 and Equation (3.30), the CRC update equation is the same as the parallel state recursive equation except the $(F^{32})^{n-7}$ matrix is different. Therefore, CRC update can be applied to the same scheme depicted in Figure 3.15. We calculate these three different matrices in MATLAB, and find that for $(F^{32})^9$, 486 '1's, namely, 486 XOR inputs exist and a maximum 22 '1's in one row. For $(F^{32})^{142}$, 537 XOR inputs exist, and a maximum 22 '1's in one row. For $(F^{32})^{373}$, 507 XOR inputs exist and a maximum 22 '1's in one row. Therefore, for all the three matrix, a maximum 5 XOR gate levels are required without optimization. If count in the 32-bit XOR at the input and the output, a total 7 levels XOR are required without optimization. Consequentially, the total latency for the CRC update circuit is estimated:

$$T_{CRCupdate} = T_{reg} + 7T_{XOR} + T_{AND}$$

where T_{reg} is the total flip-flop delay, and T_{XOR} is a delay of an XOR gate, and T_{AND} is the delay of an AND gate.

The performance of the three implementation of CRC calculation can be compared as follows. For serial LFSR implementation, the latency for the system can be evaluated as:

$$T_{LFSR} = T_{reg} + T_{XOR}$$

For 32-bit parallel implementation of CRC-32, the latency for the system is:

$$T_{Parallel} = T_{reg} + 5T_{XOR} + T_{AND}$$

For Fast update of CRC-32, the latency is:

$$T_{CRCupdate} = T_{reg} + 7T_{XOR} + T_{AND}$$

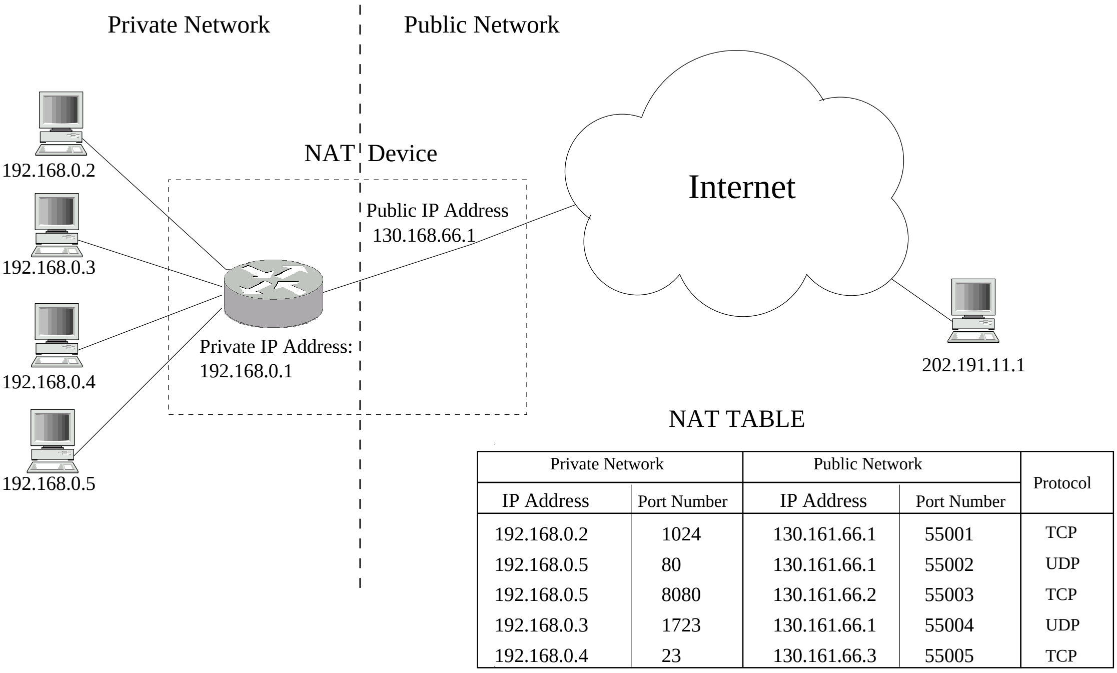
Obviously, the serial LFSR implementation is the fastest implementation, but it suffers from the low data rate, because of the serial implementation. The parallel implementation performs slower than the serial one, but it processes 32 bits at one clock cycle, the throughput is very high. The total cycles needed for a $(k + m)$ bits frame, is calculated as $n = \frac{k+m}{32}$. The fast CRC update, for a fixed length of frame, for example, 64 bytes, 594 bytes or 1518 bytes, only calculate 7 cycles in parallel calculation and perform 1 clock cycle's update. Therefore, if we use the slowest path is counted the total cycles would be only 8.

3.5 Network Address Translation

As discussed in Section 2.1.2, each device on the Internet must have a unique IP address to communicate with other devices. The IP address is a 32-bit long binary number in IPv4. In recent year, as the size of the Internet growing exponentially, here comes the problem that the lack of the IP address on the Internet dramatically impedes the development of the Internet. IP version 6 has been designed as a long term solution for shortage of IP addresses, for the IPv6 address has 128 bits and can provide sufficient IP addresses. Unfortunately, the deployment of IP version 6 is too expensive to be accomplished in short term, since it requires to change most of the routers on the Internet and needs huge investments. Network address translation (NAT) is designed as a short term solution for IPv4 addresses shortage. It can be installed incrementally and requires minimal changes to either routers or hosts.

The basic idea of NAT is that all the hosts within a network communicate with each other using private IP addresses, while communicate with the other computers outside the network through a NAT server/router. The inner network is called the private network and the outer network is called the public network. However, as discussed in Section 2.1.2, the private IP addresses can not be recognized by the routers on the Internet. Therefore, every packet intending to access the Internet must pass through the NAT devices. When an internal packet with private IP address comes to the network address translation devices, the NAT server/router will translate the private IP addresses into a public IP address and then forward it to the Internet. At the same time, NAT devices will store the translation in a table called NAT table. Consequently, the new packet with public IP address can be identified by routers and other computers on the Internet and reach its final destination. When a reply returns from the public network, the NAT devices retrieve the translation information from the NAT table, translate the public IP address back into that private IP address and send the packet to its destination in the private network. Only packets from the private network and their replies can pass through the NAT devices. Nothing originating outside the private network can pass through the NAT, since the NAT table will not contain an entry of a unknown IP address.

One or several public IP addresses are utilized by the NAT server to perform such translations. Therefore, the utilization of NAT permits an almost unlimited number of private network users to access the Internet and only one or serval public IP addresses are required for that network to communicate with other computers on the Internet.



NAT TABLE

Private Network		Public Network		Protocol
IP Address	Port Number	IP Address	Port Number	
192.168.0.2	1024	130.161.66.1	55001	TCP
192.168.0.5	80	130.161.66.1	55002	UDP
192.168.0.5	8080	130.161.66.2	55003	TCP
192.168.0.3	1723	130.161.66.1	55004	UDP
192.168.0.4	23	130.161.66.3	55005	TCP

Figure 3.21: NAT example.

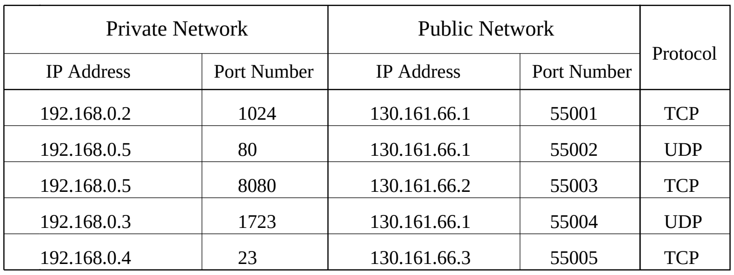
Figure 3.21 demonstrates an example NAT services. Computers at the left side of the dotted line are hosts in the private network, they hold the private addresses. The private network is connected by a NAT device to the public network, namely, the Internet, as depicted at the right side of the dotted line. Assume that a host in the private network, say 192.168.0.2, want to communicate with a host on the Internet, say host with the IP address 202.191.11.1. The communication between these two computers is established through the following steps:

1. The private host 192.168.0.2 send out a packet with the destination address 202.191.11.1
2. The packet first arrives at the NAT router. The router obtains the source IP address and port number of the packet, namely, the private address: 192.168.0.2 and a port number 1024. The NAT table is searched to find the existing entry that corresponds to the source IP address. If the entry is found, the NAT will change the original IP address and the port number to the public address and port number stored in that entry. If the entry is not found, NAT will allocate a public IP address and port number to the packet, and change the address field in the

packet accordingly. At the same time the entry is added to the NAT table. (Note that, the NAT router must be configured as a default gateway for all the private hosts willing to access the public network. In other words, all packets from the private network destined the public network must pass through the NAT router in order for address translations.)

3. In this example, the NAT device find the entry in the NAT table. Therefore, it replaces the private source IP address 192.168.0.2 with the public IP address (130.161.66.1) and forwards the packet onto the public network to its destination(202.191.11.1). The port number 1024 is also replaced by 55001. This port translation will be used later to help to find the entry for the responding packet back to its originator.
4. The destination 202.191.11.1 receives the packet and replies using the NAT's public IP address(130.161.66.1) as its destination.
5. The NAT receives the responding packet from the public host 202.191.11.1. NAT table lookup is performed to determine the destination of the responding packet inside the private network. The table lookup is based on the port number associated with the public IP address in the NAT table. If there is a matching port number, the NAT will forward the packet to the hosts corresponding to that port number. Otherwise, the packet will be abandoned.
6. The private host 192.168.0.2 receives the packet and continues the communication. The router performs Steps 2 through 5 for each packet.

There are two situations where a NAT table lookup is needed: one is address mapping for outgoing packets, and the other is the address restore for the incoming reply packet. An example of NAT table is illustrated in Figure 3.22.



Private Network		Public Network		Protocol
IP Address	Port Number	IP Address	Port Number	
192.168.0.2	1024	130.161.66.1	55001	TCP
192.168.0.5	80	130.161.66.1	55002	UDP
192.168.0.5	8080	130.161.66.2	55003	TCP
192.168.0.3	1723	130.161.66.1	55004	UDP
192.168.0.4	23	130.161.66.3	55005	TCP

Figure 3.22: NAT table.

Figure 3.23 depicts NAT table lookup operations. For outgoing packets, the table lookup is performed at the private network part over the private IP address, port number and the protocol. If there is a match found in the NAT table, the private IP address and port number fields in the packet header will be mapped to that matched values accordingly. And if there is no match, an entry (a one to one mappings of private address and port number to public address and port number) will be generated, and the packet will be modified according to that entry. Consequently, the newly updated

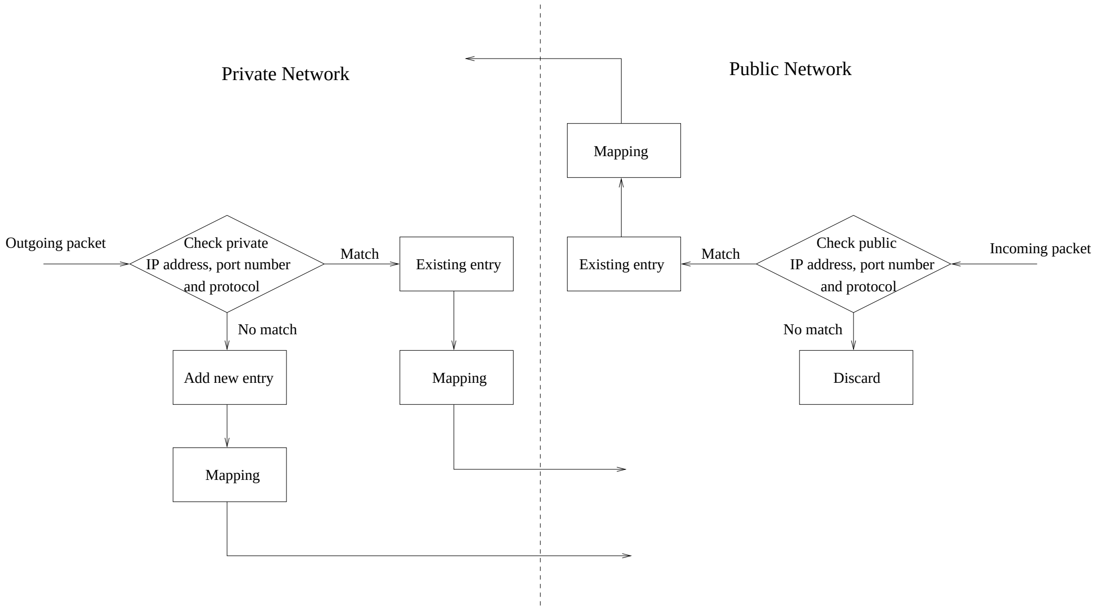


Figure 3.23: NAT table lookup.

packet will be forwarded to the Internet and reach its destination. For incoming packets, a similar table lookup of the entries at the public network part is performed. Public IP address, port number and protocol are matched. If there is a match, these fields in the packets header will be changed, and the newly updated packet will be forwarded. If there is no match, the packet will be abandoned.

As discussed above, the table lookup is one of the main operations for network address translation (NAT). The other operations include data parsing for getting the IP address, port number from the packets, data modification for modify the packet according to the found entry, checksum and CRC calculation for the updated packets, and etc. Other control plane operations may include NAT table maintenance and update, public IP address allocation and etc. As described in Section 3.1, all these functions except the control plane tasks can be implemented in FPGAs so that the NAT operation can be fully offloaded from general-purpose processors.

3.6 Conclusions

In this chapter, the selection of the performance-critical functions was described. Three functions have been selected to implement in FPGAs. The theory of selected TCP/IP functions was discussed in detail. For checksum calculation, the theory is based on the one's complement addition, and thus the 16-bit one's complement carry lookahead adder is designed to perform the calculation of checksum. The theory of CRC calculation is based on the modulo 2 arithmetic. The LFSR (Linear Forward Shift Register) was designed as the fundamental blocks in both serial and parallel CRC calculation. The parallel calculation of CRC was based on a recursive equation Equation (3.27) that is derived from LFSR. Furthermore, a novel fast CRC update method was introduced and its performance was evaluated. Finally, we presented the network address translation (NAT) service, and its main operation, the NAT table lookup.

Implementations and Results

4

The theory of the selected TCP/IP functions are discussed in details in Chapter 3. In this chapter, the implementation of the selected TCP/IP functions are discussed and the design results are presented. We start with the checksum calculation function. As discussed in Section 3.2, we cascade two 16-bit one's complement adders to perform the 32-bit checksum calculations. The design is verified and the synthesis, placement and routing (PAR) are performed targeting different FPGAs. Subsequently, we implement the parallel CRC-32 calculation circuit. The number of bits processed in parallel are increased from 8 *to* 32 *and the different utilizations and performances are compared and the reasons for the difference are investigated. The 32-bit parallel calculation of CRC-32 targeting Xilinx Virtex II FPGA are selected to estimate the throughput of the design. Furthermore, based on the parallel calculation circuit, a novel fast CRC update scheme is proposed and designed based on the observation that the possible changes when an Ethernet frame is forwarded, located at the first 28 bytes of that frame. Therefore, there is no need to recalculate the CRC code over the entire frame, a fast single step update can be achieved using the designed circuit. The PAR and static timing results are given to demonstrate the performance improvements. Finally, the implementation of the table lookup function in the network address translation (NAT) is described in detail. Content Addressable Memory is implemented using block SelectRAMs of the Virtex FPGA for the NAT table lookup. Consequently, the table lookup operations can be performed in one clock cycle.*

The organization of this chapter is as follows. In Section 4.1, the implementation of a 16-bit carry lookahead adder is first presented, the 32-bit checksum calculation circuit is build based on the 16-bit carry lookahead adders. In Section 4.2, the implementation of parallel CRC-32 calculation circuit is discussed. In Section 4.3 the implementation of a novel fast CRC update design is proposed. In section 4.4, the implementation of the table lookup function in network address translation is presented.

4.1 Checksum

The theory of calculating checksum is discussed in Section 3.2. We simplify the checksum calculation to the 16-bit one's complement addition. And a 16-bit carry-lookahead adder is designed to perform the calculation. We recall the block diagram of 16-bit carry-lookahead adder as follows.

With the implemented 16-bit carry-lookahead adder, we can build the checksum calculation circuit as depicted in Figure 4.2. Two inputs of the 16-bit one's complement adders are added together and each of them contains half of the 32-bit data field. We use a packet_sig signal to notify the system that a new packet is coming, and these two

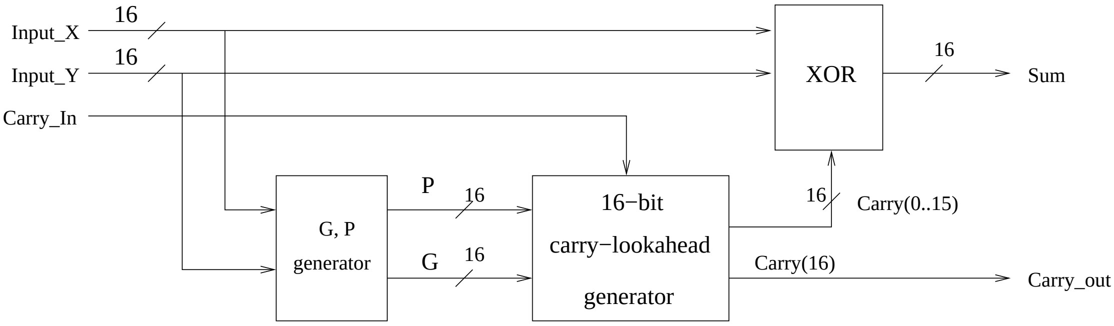


Figure 4.1: 16-bit carry-lookahead adder.

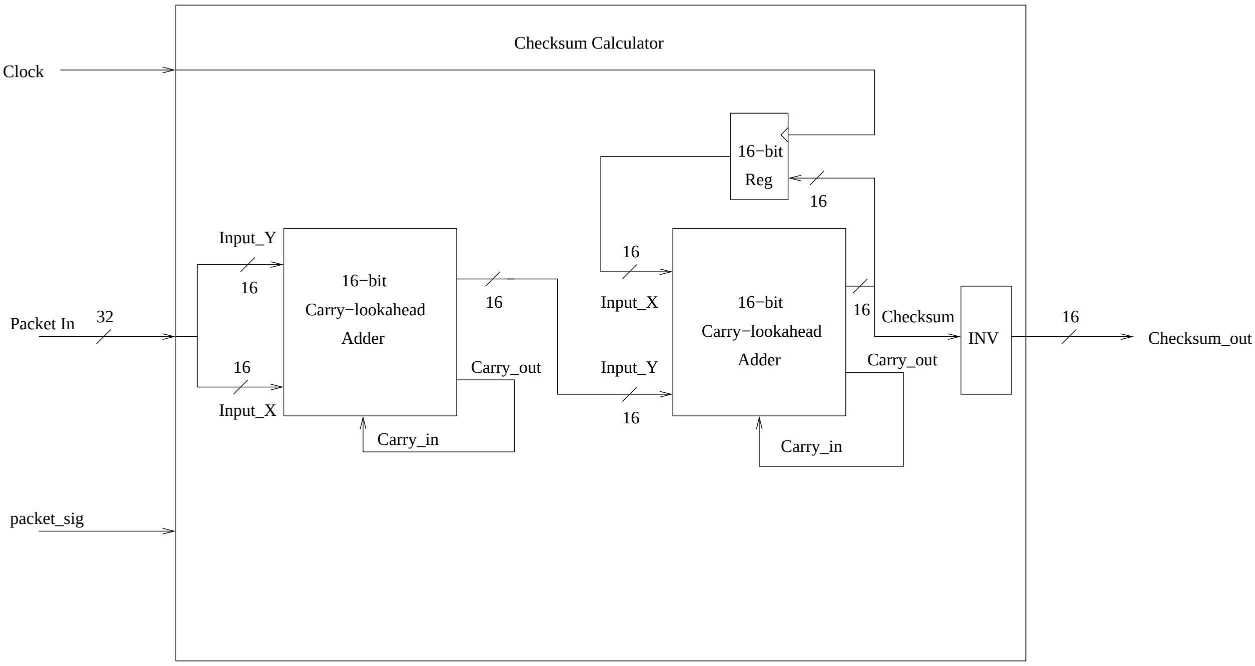


Figure 4.2: Block diagram of checksum calculation.

adders will automatically be reset and start a new addition. At the same time, the addition results are inverted and sent to the output, which is the final checksum value.

A testbench is built to verify the design and the waveform from the simulation is depicted in Figure 4.3. We use the checksum calculation in the IP header as an example. IP header contains 20 bytes, accordingly, there are five 32-bit inputs. The checksum field in the IP header is set to zeros as depicted at the lower 16 bits in the third packet_input. Finally, an extra cycle is needed to calculate the two values in the upper and lower 16-bit CLA and get the checksum.

The design is synthesized in leonardo spectrum and targeted two FPGA platforms: Xilinx Virtex and Virtex II FPGAs, which are widely utilized in FPGA designs. We use the device v50bg256 for Virtex and 2v40fg256 for Virtex II, respectively. And the speed grade -6 is chosen in order to achieve the best performance. The output of the synthesis,

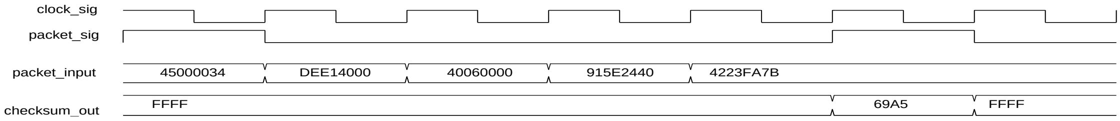


Figure 4.3: Waveform of checksum calculation.

the EDIF file, is forwarded to the implementation tool, Xilinx ISE. The placement and routing are performed and the PAR and static timing reports are achieved. As mentioned in Section 2.3, we can run synthesis from both Leonardo Spectrum and Xilinx ISE. Both of the results are listed in Table 4.1.

Technology	Device	Leonardo Spectrum (EDIF)		Xilinx ISE (XST VHDL)	
		Utilization	Performance	Utilization	Performance
Virtex	v50bg256-6	81 slices	63 MHz	86 slices	62 MHz
Virtex II	2v40fg256-6	81 slices	124 MHz	86 slices	116 MHZ

Table 4.1: Checksum calculation, utilization and performance report.

We choose the result of Xilinx Virtex II and the lower maximum frequency of the post PAR static timing report is selected for performance evaluation. Furthermore, in our design, 32 bits are processing per clock cycle, the throughput can be calculated as follows:

$$\text{Throughput} = 32 \text{ bits} \times 116 \text{ MHz} = 3.712 \text{ Gbps}$$

Although the throughput is less than the requirement, further work can be investigated to improve the performance. We can increase the number of 16-bit adders running in parallel. Pipeline can be built to perform additions. Furthermore, a incremental checksum calculation is proposed in [20], which recognize the changed fields in the data to be calculated, and perform checksum update using the following equation:

$$HC' =\sim (\sim HC+ \sim m + m')$$

where HC' denotes the new checksum, HC denotes the old checksum, m represents the old value of a 16-bit field and m' represents the updated value of a 16-bit field. Therefore, in some situations that only a small portion of the data is changed, such as the updating of the IP header, the checksum calculation could be replaced by checksum incremental update. For each changed field, there are only two 16-bit one's complement additions. The total number of additions is reduced. For example, as can be observed in Figure 3.18, IP header has 20 bytes, if checksum calculation is performed over the whole IP header, 9 additions needs to be performed. But since there are only two changed fields, which are in gray in Figure 3.18, only 4 16-bits additions are needed. Consequently, the throughput of checksum calculation speeds will be more than doubled.

4.2 Cyclic Redundancy Check

This section describes the implementation of the parallel calculation of cyclic redundancy check (CRC). CRC-32 is selected as an example. As discussed in Section 3.3, the parallel calculation of CRC-32 is based on the recursive formula, Equation (3.27):

$$X' = F^w \cdot X \oplus \mathbf{D},$$

where X denotes the current state of the CRC shift registers and X' denotes the next state of the CRC shift registers after one clock cycle. F^w is a 32×32 matrix, F matrix to the power of w, where w is the number of bits processed in parallel. $\mathbf{D}$ is the 32-bit input. If w is less than 32, the input bits are stored at the lower significant bits of $\mathbf{D}$, and the other bits of $\mathbf{D}$ remain '0', namely, $\mathbf{D} = \{0, ..., 0, d_{w-1}, d_{w-2}, ..., d_0\}$.

The F matrix is derived from Equation 3.15. For CRC-32, we have $m = 32$, thus

$$F = \begin{bmatrix} p_{31} & 1 & 0 & \cdots & 0 \\ p_{30} & 0 & 1 & \cdots & 0 \\ \cdots & \cdots & \cdots & \ddots & \cdots \\ p_1 & 0 & 0 & \cdots & 1 \\ p_0 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

where $P = \{p_m, p_{m-1}, \cdots, p_0\}$=$\{100000100110000010001110110110111\}$. If we take w = 32, namely, the number of bits processed in parallel is 32, the F^{32} matrix can be precalculated in MATLAB as follows:

$$F^{32} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 0 \\ 0 & 1 & 1 & \cdots & 0 \\ 1 & 0 & 1 & \cdots & 0 \\ \cdots & \cdots & \cdots & \ddots & \cdots \\ 1 & 1 & 1 & \cdots & 1 \end{bmatrix}$$

The complete matrix F^{32} is listed in Appendix A. As mentioned in Section 3.3.3, other matrices such as F^8, F^{16}, F^{24} can be easily derived from matrix F^{32}. F^{32} matrix is stored as an array of constant bit-vector in the design. As depicted in Table 4.2, we find the '1's in the matrix F^w and a '1' denotes an XOR operation. Xtemp stores the temporal result of $(F^{32} \cdot X)$, and $X2$ stores the current states of the shift register, namely, is identical to X.

The block diagram of the parallel CRC circuit is depicted in Figure 4.4. There are 32 shift registers. For each register $i, 0 \leq i \leq 31$, the input of the register is the XOR of the parallel input bit d_i and the multiplication result of the ith row in the F^w matrix and $X = [x_{31}, x_{30}, \ldots, x_0]$. For example, the input of shift register x_{31}, we write its input as follows (from F^{32} matrix, we have $F^{32}(31)(31)$ = '1', $F^{32}(31)(30)$ = '1' …, and $F^{32}(31)(0)$ = '1', we omit positions which may have the value '1' between $F^{32}(31)(30)$ and $F^{32}(31)(0)$.):

$$x'_{31} = \left(F^{32}(31)(31) \cdot x_{31}\right) \oplus \left(F^{32}(31)(30) \cdot x_{30}\right) \oplus \cdots \oplus \left(F^{32}(31)(0) \cdot x_0\right) \oplus d_{31}$$

```
for i in 31 downto 0 loop
    for j in 31 downto 0 loop
        if F(i)(j) =' 1' then
            Xtemp(i) := Xtemp(i) xor X2(j);
        end if;
    end loop;
end loop;
```

Table 4.2: The VHDL code to perform XOR operation.

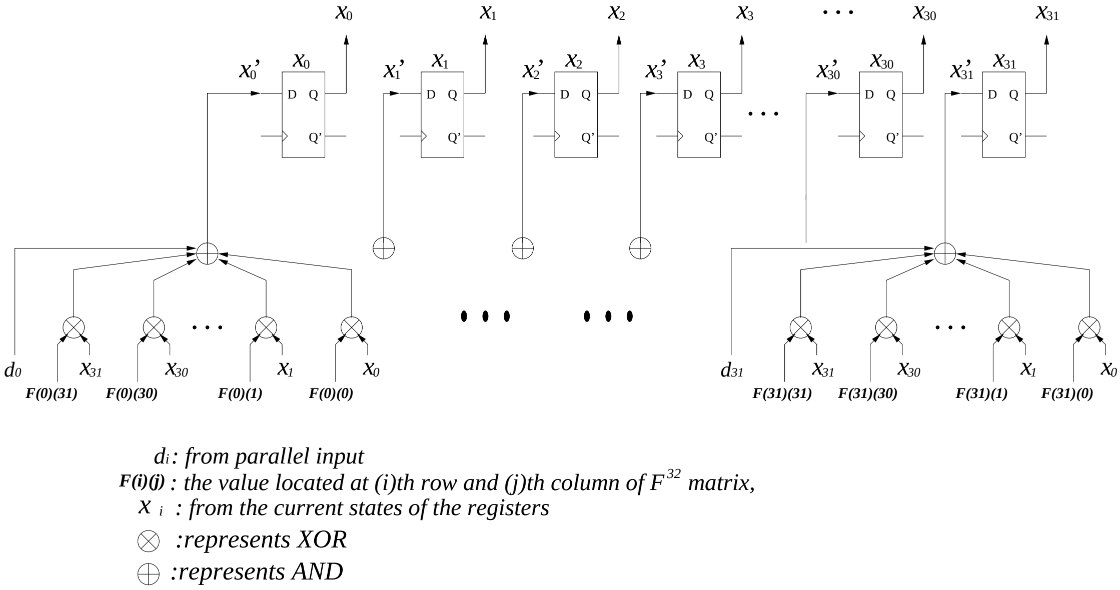


Figure 4.4: LFSR for parallel calculation of CRC32.

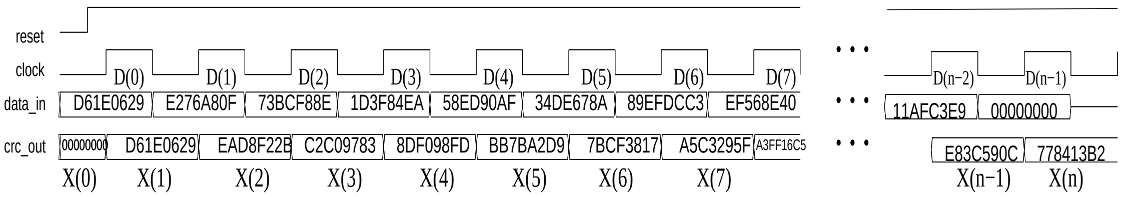


Figure 4.5: Testbench waveform for CRC32 parallel calculation.

This equation coincides with the recursive equation Equation (3.27).

A testbench is generated to verify the functionality of the design. A 64-byte data message is utilized as the input. And the testbench waveform is illustrated in Figure 4.5. As observed in this figure, the final 32-bit input is all zeros, which is the appended 32 zeros to the data message. A sequence of the input vector $\mathbf{D}$ (from $\mathbf{D}(0)$ to $\mathbf{D}(n-1)$ as depicted in Figure 4.5) and the register state X (from $X(0)$ to $X(n-1)$) are highlighted. Every rising edge of the clock, $F^{32} \cdot X \oplus \mathbf{D}$ is performed. And finally, the CRC code is obtained as

$$CRC = X(n) = 0x778413B2$$

$n = \frac{64 \text{ bytes} \times 8 \text{ bits}}{32} = 16$ cycles are needed to obtain the CRC code.

The design is synthesized by LeonardoSpectrum and ISE. And both the synthesis outputs are forwarded to Xilinx ISE to run placement and routing. The place and route report and post place and route static timing report can be achieved. From the reports, the performance and area estimations are presented as follows. We choose different number of bits processed in parallel and target different Xilinx FPGAs and compare the results.

Technology	Number of parallel bits	Leonardo Spectrum (EDIF) Utilization	Leonardo Spectrum (EDIF) Performance	Xilinx ISE (XST VHDL) Utilization	Xilinx ISE (XST VHDL) Performance
Xilinx Virtex v50bg256-6	1 bit(serial)	16slices	211 MHZ	18 slices	167 MHz
	8 bits	22 slices	147 MHZ	22 slices	199 MHz
	16 bits	38 slices	131 MHZ	49 slcies	186 MHz
	32 bits	63 slices	103 MHZ	82 slcies	173 MHz
Xilinx Virtex II 2v40cs144-6	1 bit(serial)	16 slices	409 MHZ	18 slices	337 MHz
	8 bits	22 slices	379 MHZ	22 slices	266 MHz
	16 bits	38 slices	290 MHZ	49 slices	294 MHz
	32 bits	63 slices	200 MHZ	81 slices	246 MHz

Table 4.3: Area and static timing reports for different number of parallel input bits.

The clock rate, area, and throughput of the serial, and parallel implementation of CRC calculation for 8-bit, 16-bit and 32-bit are compared in Figure 4.6, 4.7, 4.8, respectively.

We take the 32 bits parallel implementation for Xilinx Virtex II FPGA for performance evaluation. The clock rate for the design is 246 MHz and the throughput can be calculated as follow.

$$\text{Thoughput} = 32 \text{ bits} \times 246 \text{ MHz} = 7.88 \text{ Gbps}$$

Therefore, the throughput of the 32-bit parallel calculation of CRC-32, although very high, still can not meet the processing requirement for 10 Gigabit Ethernet. From Figure 4.7 and Figure 4.8, it is obvious that although the clock rates decrease as the number of bits processed in parallel increases, the throughput of the design still increase. Therefore, it is possible to increase the number of parallel bits processed per clock cycle, for example, 64 bits, to achieve a higher throughput. Recall that the parallel calculation of CRC is a recursive calculation of the next register state X' from X and the parallel input $\mathbf{D}$, and

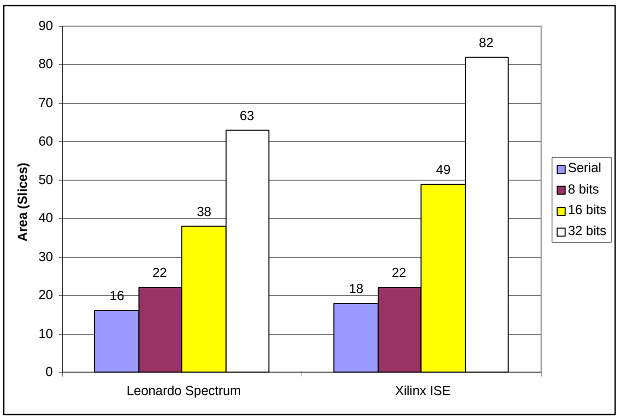


Figure 4.6: Area utilization comparison for different number of parallel input bits.

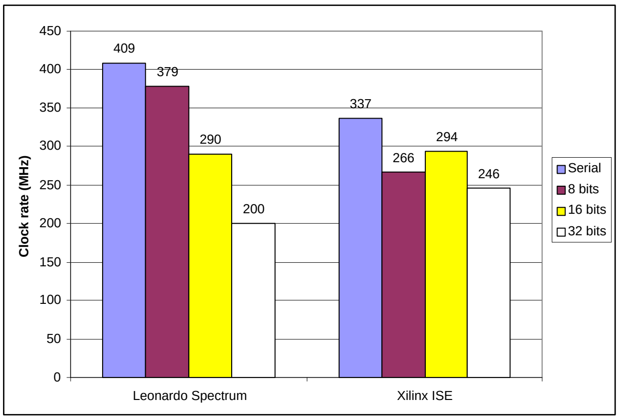


Figure 4.7: Clock rate comparison for different number of parallel input bits.

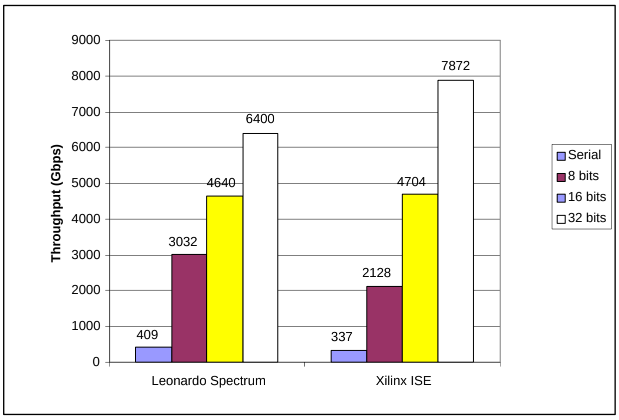


Figure 4.8: Throughput comparison for different number of parallel input bits.

it is based on the recursive equation, Equation (3.27). We expand the recursive equation as follows:

$$
\begin{aligned}
X(1) &= F^{32}\cdot X(0)\ \oplus \mathbf{D}(0)\\
X(2) &= F^{32}\cdot X(1)\ \oplus \mathbf{D}(1)\\
&= \Big((F^{32})^2\cdot X(0)\Big)\ \oplus \underline{\Big(F^{32}\cdot \mathbf{D}(0)\oplus \mathbf{D}(1)\Big)}\\
X(3) &= F^{32}\cdot X(2)\oplus \mathbf{D}(2)\\
X(4) &= F^{32}\cdot X(3)\oplus \mathbf{D}(3)\\
&= \Big((F^{32})^2\cdot X(2)\Big)\ \oplus \underline{\Big(F^{32}\cdot \mathbf{D}(2)\oplus \mathbf{D}(3)\Big)}\\
&\cdots\\
X(i) &= \Big((F^{32})^2\cdot X(i-2)\Big)\ \oplus \underline{\Big(F^{32}\cdot \mathbf{D}(i-2))\ \oplus \mathbf{D}(i-1)\Big)}
\end{aligned}
$$

From above derivations, it is clear that we can calculate $X(i)$ based on $X(i-2)$, $\mathbf{D}(i-2)$ and $\mathbf{D}(i-1)$. That is exactly the solution for 64-bit parallel calculation of CRC. The block diagram of 64-bit parallel calculation of CRC as depicted in Figure 4.9. It is of the same structure as Figure 4.4.

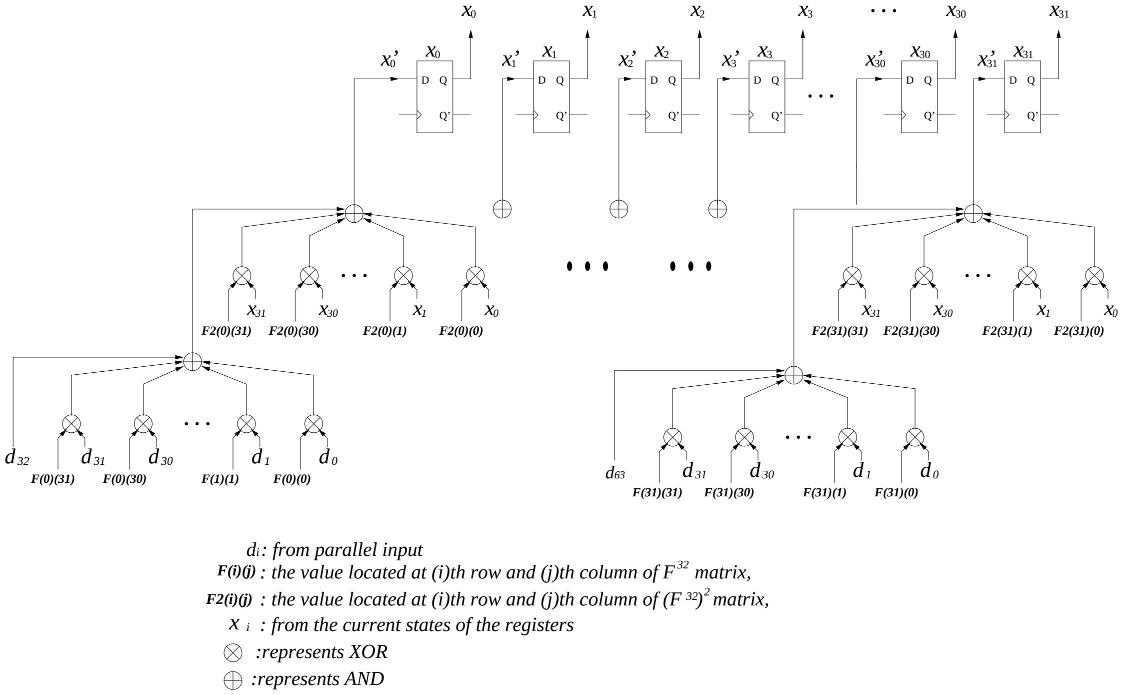


Figure 4.9: The block diagram of 64-bit parallel calculation of CRC-32.

Although Figure 4.9 is depicted in hierarchy, there is no actual hierarchy in the circuit. Those two XORs at every input of the register could be regarded as a single XOR. The only increase is the number of XORs per input of the register, since F matrix has a maximum 17 '1's, plus the maximum 19 '1's in F^2 and one input bit, the maximum number of XORs is 36, resulting in 6 XOR gate levels without optimization. The clock rate of the circuit will be lower than the others, but the throughput is higher, since only one extra gate level is introduced. The area utilization will be doubled since the total number of XORs is about 2 times more than the 32-bit parallel calculation circuit.

The throughput of 64-bit parallel calculation of CRC may meet current trends of network speeds. However, it is not the efficient way. In parallel calculation of CRC, the CRC is verified when receiving the frame and recalculated when sending it out. Section 4.3 describe an implementation of the CRC update circuit so that we can update the CRC codes instead of recalculating them.

4.3 Fast CRC Update

In Section 3.4, we discussed a novel fast CRC update algorithm. The theory relies on the fact that the modified bits in an Ethernet frame are mainly located at the beginning of that frame, and a large portion of data remain unchanged. Therefore, there is no need to recalculate the CRC over the entire frame every time. What we need is just to update the CRC code according to the changed fields. First of all, we briefly describe the parallel calculation of CRC-32. Every clock cycle, a 32-bit input $\mathbf{D}$ is sent to the input of the registers, the calculation ($X' = F^w \cdot X \oplus \mathbf{D}$) is performed and the result is stored in the shift registers. For a 64-byte frame, we need $\frac{64\times 8 \text{ bits}}{32} = 16$ clock cycles to produce the CRC code. However, from Figure 3.18, we observed that the changes only occur in the first 28 bytes of the frame. Therefore, after calculation of these 28 bytes, we can develop a single step update of the CRC code instead of calculating the following $\frac{(64-28)\times 8 \text{ bits}}{32} = 9$ cycles. 9 cycles have been saved, furthermore, 64-byte frame is the smallest frame, for larger size frames, more cycles can be saved.

The equation utilized to update the CRC code is as follows.

$$CRC_{new} = (F^{32})^{n-7} \cdot \Big(X(7)_{new} \oplus X(7)_{old}\Big) \oplus CRC_{old} \tag{4.1}$$

And the block diagram of the fast CRC update is illustrated in Figure 4.10.

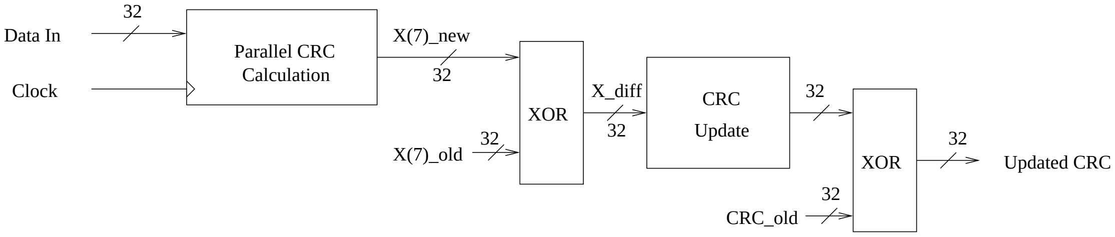


Figure 4.10: The block diagram of CRC fast update.

A testbench with a 64-byte input is generated to verify the design. We use the same example as depicted in Figure 4.5. It is regarded as the old frame. The 32-bit parallel calculation of the updated frame is illustrated in Figure 4.11. The changed fields are $\mathbf{D}(0), \mathbf{D}(1), \mathbf{D}(2), \mathbf{D}(5), \mathbf{D}(6)$ compared with the old frame depicted in Figure 4.5. The final result, namely, the new CRC code is:

$$CRC_{new} = X(n) = 0x555BB661$$

The testbench waveform of the fast CRC update is depicted in Figure 4.12. First of all, the first 28 bytes of the frame (from $\mathbf{D}(0)$ to $\mathbf{D}(6)$) are calculated through parallel

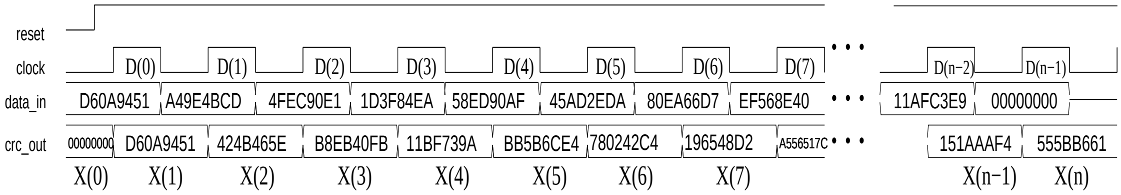


Figure 4.11: The testbench waveform for the parallel CRC calculation of the new frame.

calculation of CRC, subsequently, $X(7)$ is obtained and applied to Equation (4.1) and the new CRC code is obtained in one step update as demonstrated in Figure 4.12. It shows the same result as the parallel CRC calculation of the new frame depicted in Figure 4.11:

$$CRC_{new} = 0x555BB61$$

the same as the $X(n)$ in Figure 4.11.

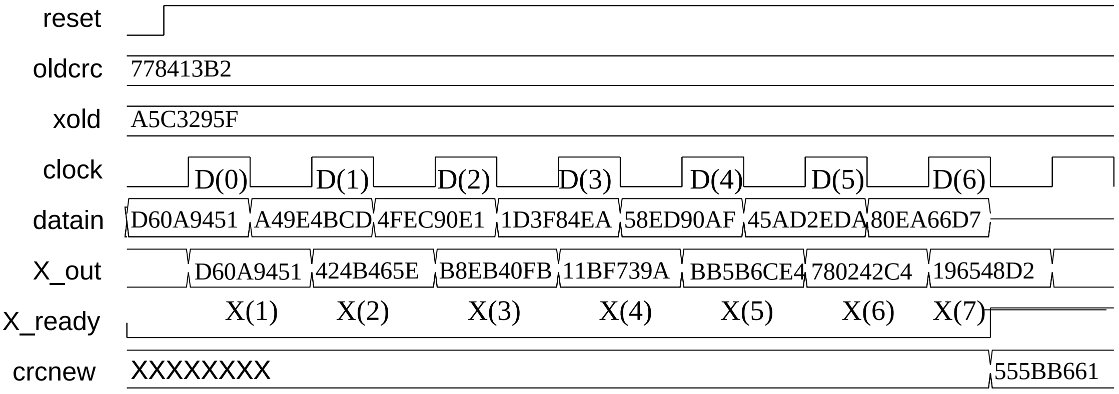


Figure 4.12: The testbench waveform for CRC fast update.

Synthesis, placement and routing are performed on the design for three building blocks: 64 bytes, 594 bytes and 1518 bytes, respectively. The area utilization and the performance from post place and route static timing report are demonstrated as follows: With these results, the performance of the fast CRC update is evaluated as follows. For the frame with the length 64 bytes, 594 bytes and 1518 bytes, the time utilized to obtain the CRC code is compared as demonstrated in Table 4.3. There is a great reduce of the calculating time of the CRC. For larger frame size, the reduced ratio is very high.

In order to evaluate the throughput of CRC update, the way how frames with CRC codes are transmitted is first analyzed. As depicted in Figure 4.13, the CRC code is calculated while the frame is being transmitted. Serval delay elements (depicted in grey and named T) are inserted in the data path in order to synchronize with the CRC calculation so that the CRC code is transmitted exactly after the frame has been transmitted. Therefore, the throughput of this block is highly depended on the time consumed in CRC calculations. The shorter the CRC calculation time, the faster the frame is transmitted.

Technology	Device	Speed Grade	Processing Units	Utilization	Performance
Xilinx Virtex	v200bg256	-6	$(F^{32})^9$	291 slices	121 MHZ
			$(F^{32})^{142}$	241 slices	124 MHZ
			$(F^{32})^{373}$	299 slices	110 MHZ
Xilinx Virtex II	2v250fg256	-6	$(F^{32})^9$	283 slices	196 MHZ
			$(F^{32})^{142}$	239 slices	236 MHZ
			$(F^{32})^{373}$	290 slices	217 MHZ

Table 4.4: Fast CRC update area and performance report for different processing unit.

Frame	Clock cycle (clock rate)		CRC calculation time comparisons		
(bytes)	Parallel CRC	CRC update	Parallel CRC	Fast update	Time reduced
64	16 (246 MHz)	8 (196 MHz)	65 ns	41 ns	36.9%
594	149 (246 MHz)	8 (236 MHz)	606 ns	34 ns	94.4%
1518	380 (246 MHz)	8 (217 MHz)	1545 ns	37 ns	97.6%

Table 4.5: Clock cycles and calculation time comparisons on different frame sizes.

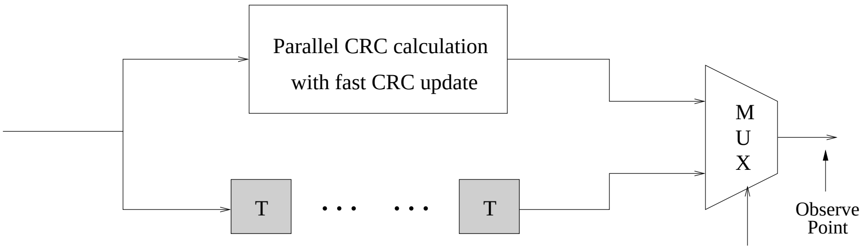


Figure 4.13: The block diagram for the transmission of the frame with CRC code.

If we assume that CRC is the only bottleneck of the whole system, and the transmission speed is unlimited, we can estimate the maximum throughput that the parallel calculation of CRC with CRC update can support. The observation of the throughput is performed from the observe point depicted in Figure 4.13. The total throughput of the CRC calculation with CRC update is estimated as follows. Assume that the three processing units are only applied to their corresponding frame length, that is, 64 bytes, 594 bytes, 1518 bytes, respectively and the clock rates for these three circuits are obtained from Table 4.3. For the 35% 64 bytes frame on the Internet, 8 cycles running at 196.85 MHz are needed. Therefore, the throughput for 64 bytes frame is:

$$(64 \times 8) \text{ bits}/(8 \text{ cycles}/196 \text{ MHz}) = 12.48 \text{ Gbps}$$

Similarly, the throughput for 594-byte frame and 1518-byte frame can be calculated and the results are listed in Table 4.3.

frame length (bytes)	frame distribution	clock cycles	clock rate (MHz)	calculation time (ns)	throughput (Gbps)
64	35%	8	196	41 ns	12.48
594	11.5%	8	236	34 ns	139.79
1518	10%	8	217	37 ns	328.21

Table 4.6: CRC update throughput for 64 bytes, 594 bytes and 1518 bytes.

Therefore, the total throughput of the CRC parallel calculation with fast update could be estimated as follows. For simplicity, assume that the rest of the frames other than 64 bytes, 594 bytes, 1518 bytes still use parallel CRC calculation instead of using fast update.

$$
\begin{array}{rrcl}
 & 12.48 \text{ Gbps} \times 35\% & = & 4.37 \\
+ & 139.79 \text{ Gbps} \times 11.5\% & = & 16.07 \\
+ & 328.28 \text{ Gbps} \times 10\% & = & 32.83 \\
+ & 7.88 \text{ Gbps} \times 43.5\% & = & 3.43 \\
 & & \mathit{Total} & 56.7 \text{ Gbps}
\end{array}
$$

This throughput is a rough estimation of the throughput that the parallel calculation of CRC with fast CRC update can support. If the CRC fast update scheme is also utilized to calculate CRC code of the frames other than 64 bytes, 594 bytes and 1518 bytes, the throughput could be much higher. Furthermore, if a simple pipeline is built on parallel CRC and CRC update processing unit, one extra cycle can be saved, and the throughput can be further improved. The estimated throughput of parallel CRC with fast CRC update is compared with the other implementations, the result is depicted in Figure 4.14.

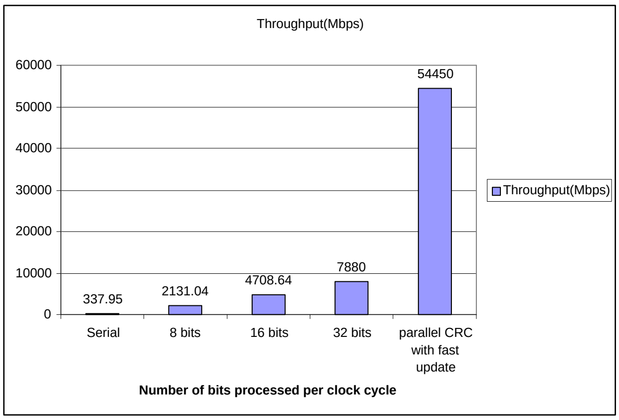


Figure 4.14: Throughput comparison between parallel and CRC fast update.

Therefore, if the other bottlenecks are excluded and the transmission speeds are assumed to be unlimited, the parallel calculation of CRC with CRC update can reach a throughput of 56.7 Gbps, which is ample to meet the requirement of the transmission speeds, for example, the trends for 40 Gbps Ethernet. Actually, the throughput is only a theoretical estimation, the real throughput still can not reach that high considering the other bottlenecks. However, with this design, it is obvious that CRC calculation will no longer be considered as a bottleneck for the TCP/IP processing.

The CRC update implementation offers several additional advantages. Since only 28 bytes data plus 8 bytes original CRC code and intermediate states are needed to be copied and sent to calculate the new CRC instead of the total data message which may have hundreds of bits. The time used for copying data, and the buffer used to store the to be calculated data are reduced. Furthermore, only 8 cycles are required to obtain the new CRC, the corresponding power consumption is also reduced compared with the CRC recalculation of the overall frame. A disadvantage is that some extra buffer may be needed to store the intermediate state $X(7)$, or X(h-(n-7)), and the old CRC code. The entry also need to point to the frame body which may be stored in the main buffer of the system. The buffer can be located on the data link layer and do not need to pass on to the upper layers. And thanks to the modern techniques for memory, a buffer with 64-byte entries is not an expensive implementation. It is a small sacrifice compared with the great throughput.

In summary, the fast CRC update scheme can dramatically increase the throughput of the CRC recalculation. Furthermore, it is suggested in [8] that it is not necessary to check the CRC at every routing node. The low error probabilities of the modern high speed network make it possible to only update the CRC code during forwarding and verify the CRC code only at the destination. Another suggestion to reduce the complexity of CRC update is to calculate the CRC code from the least significant bit of the frame so that the position of the possible changed fields in the frame are independent of the frame length. Consequently, the F matrix in calculation is fixed and the number of ones in the F matrix will be smaller. Therefore, the total complexity of the design could be minimized and the circuit may be faster.

The advantage of implementing the CRC parallel calculation as well as the CRC fast update scheme in FPGAs is that these two design can be applied to any other CRC algorithm such as CRC-16, CRC-8, and for other protocols and applications. The Only change is to specify the different generator polynomial and the number of bits in parallel calculation, w. Consequently, the F^w matrix can be precalculated in order for further usage. And the FPGAs can be reconfigured to adapt to these changes. Furthermore, the fast CRC update scheme is not only suitable for Ethernet, but also hold true for other protocols such as ATM (ALL-5 frames). And the fast CRC update may have a broader usage because of the wide utilizations of the CRC error detecting techniques.

4.4 Network Address Translation

As discussed in Section 3.1 and Section 3.5, among the functions involved in network address translation (NAT), NAT table lookup is a main function that need to be performed at wire speeds. NAT table contains 5 fields, namely, the private IP address and port number, the public address and port number, and protocol. Totally, 104 bits are required for each entry. Furthermore, the total number of entries could be about several hundreds or one thousands. This number compared with the number of entries in the routing table is much smaller. Therefore, it is possible to store the NAT table in a Content Addressable Memory (NAT)to facilitate the lookups.

Private Network		Public Network		Protocol
IP Address	Port Number	IP Address	Port Number	
192.168.0.2	1024	130.161.66.1	55001	TCP
192.168.0.5	80	130.161.66.1	55002	UDP
192.168.0.5	8080	130.161.66.2	55003	TCP
192.168.0.3	1723	130.161.66.1	55004	UDP
192.168.0.4	23	130.161.66.3	55005	TCP

Figure 4.15: NAT table

CAMs are specialized memory devices that permit rapid and massively parallel searches. In a search, all data stored in the memory is simultaneously compared to the search key during one clock cycle. The output is the index of the matched entry. If there is no match, the output is all 0s. CAMs can be implemented using Xilinx Virtex Block RAM[1]. The block SelectRAM memory is built into the Virtex devices and can be utilized as a 16-word deep by 8-bit wide (16 × 8) CAM. Furthermore, we can extend the size of CAMs by cascading serval block SelectRAMs. The CAM16 × 8 Macro implemented by Xilinx Virtex Block RAM is depicted in Figure 4.16.

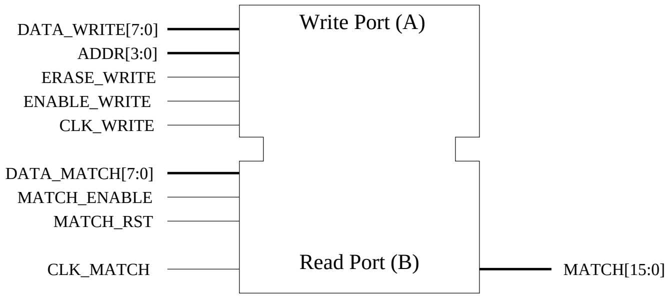


Figure 4.16: CAM 16 × 8 Macro.

The data width of the memory (DATA_WRITE[7:0]) and search key (DATA_MATCH[7:0]) are both 8 bits. The length of the address (ADDR[3:0]) is 4-bit, that is, there are $2^4 = 16$ entries in each CAM Macro. The output of the Macro is the decoded address of the matched entry. The decoded address is achieved by converting the 4-bit address to a 16-bit bit-vector. For instance, if there is a matched entry at the address “0010”, the decoded address output is “0000000000000100”. A decoded address output allows multiple matched entries. For example, the matched entries could be found in serval locations in the CAM. The output “0000000000001100” means that the matched entries are found at address “0010” and “0011”.

As can be observed in Figure 3.22 and Figure 3.23, the table lookup should be performed in two directions, one for outgoing packets and one for incoming packets. Each search entries has 56 bits (32 bits IP address + 16 bits port number + 8 bits protocol), and each entry is mutually exclusive. We cascade 7 CAM 16 × 8 Macros to build a CAM 16 × 56 Block. Furthermore, we use two CAM 16 × 56 blocks, one stores the IP address and port number at the Private network side, and the other stores the

IP address and port number at the public network side.

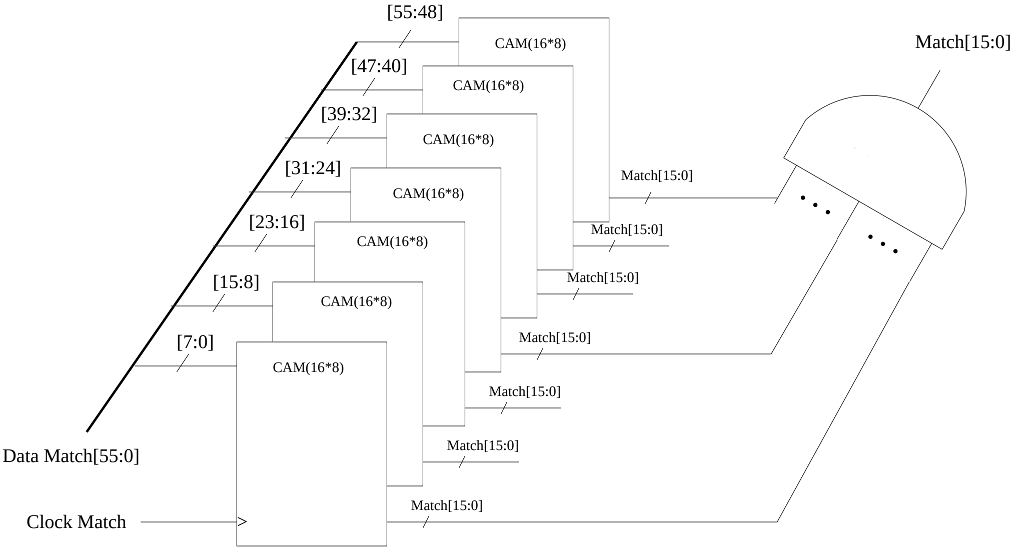


Figure 4.17: Cascaded CAM 16 × 56 Block.

Figure 4.17 depicts the block diagram of CAM 16 × 56, the 56-bit search entry is divided into seven 8-bit small search entries, and the matched entry will be searched in each CAM16 × 8 block according to the 8-bit small search entry. Finally, the seven outputs, namely, the match signals will be anded to determine the final decoded address of the matched entry. If the output is all 0s, there is no matched entry found, therefore, we need to add a new entry into the next free location in the CAM memory block.

A testbench has been generated to verify the functionality of the 56-bit CAM table lookup. The waveform is depicted in Appendix C. “data_in” is the 56-bit input data, either for data matching or for data writing. The“addr” signal specifies the address when write memory is enabled, otherwise, it remains zero. “write_enable” invokes a write operation, which will take for 2 cycles and write the data specified in “data_in” to the address specified in the signal “addr”. Signal “match_enable” invokes a data matching operation, which will find the data entry for the data specified in “data_in”. If there is a matching, the address of that entry is outputted at “match_addr”, and together with a ‘1’ at the “match_ok” signal output.

The synthesis, placement and routing are performed by Xilinx ISE on the implemented 16 CAM block. And the area and performance are reported as follows.

Technology	Device	Xilinx ISE	
		Utilization	Performance
Virtex	v300bg352-6	69 slices	53 MHz
Virtex II	2v250fg256-6	66 slices	101 MHz

Table 4.7: 16 × 56 CAM block, utilization and performance report.

The performance of CAM using Virtex block SelectRAM can be evaluated as follows. We take the clock rate for Virtex II as an example. The clock rate is 101.28 MHz. For a match operation only one clock cycle is needed, and a write operation takes two clock cycles. For only data matching, the throughput of the design is 101.28M packets per second, this searching speed is sufficient for the wire speeds provided, nowadays. With these 16×56 CAM blocks, we can cascade more CAM Macros to increase the total number of entries. With the development of Xilinx Virtex series, it is possible to implement the 56-bit CAM with hundreds of entries. The number of entries is sufficient for a middle-size networks with hundreds of or several thousands of computers considering that not all the computers need to access the Internet at the same time.

Network address translation is one of the network services that are required to be supported by interconnecting devices such as routers. But it is not necessarily the must. Therefore, it is very intriguing to implement NAT in FPGAs, normally, FPGAs can be configured to perform other operations. If the NAT operation is required, FPGAs can be reconfigured to fulfil the NAT operations. This is a good example that designing TCP/IP functions in FPGAs can encompass both the high processing speeds and the flexibility of FPGAs to support a wide range of protocols and network services.

The same scheme is also suitable to the implementation of Address Request Protocol (ARP). As described in Section 2.2, ARP can also maintain a lookup table for the IP address to the Ethernet address mapping. Compared with the NAT, the table is smaller, 80-bit (48-bit Ethernet address + 32-bit IP address) entries are required.

4.5 Conclusions

In this chapter, the implementation details of the selected TCP/IP functions were discussed. The simulation results were given in waveforms and the synthesis results for different FPGAs were demonstrated and compared in tables. The performance of fast CRC update was compared with the CRC calculation as well. The NAT table lookup functions based on the use of the Xilinx on-chip block SelectRAMs as content addressable memories (CAMs) was explained. The CAMs were designed to store the NAT tables.

Conclusions

5

In *this thesis, we described the design of several TCP/IP functions to eliminate the possible bottlenecks for network processing and to offload the processing tasks from the general-purpose processors (GPPs). Field-programmable gate arrays (FPGAs), as programmable hardware devices, have been selected as the target design platform. TCP/IP processing functions were surveyed and categorized into the micro-level functions. Furthermore, network services, such as network address translation, firewall, which may consist of several micro-level functions, were presented. Micro-level functions are those functions that need to be run over every packet, thus need to be performed at wire speeds. Since some of these functions are computationally intensive, there may be some processing bottlenecks. Therefore, the profiling information for these functions were studied, and the possible critical paths were identified as the candidates for acceleration. Checksum and cyclic redundancy check (CRC) were selected as the computational intensive micro-level functions to be designed. And the table lookup were selected as a critical factor in network address translation (NAT) to be implemented. In checksum calculation, two 16-bit one's complement carry lookahead adders were designed to calculate the checksum. The first adder calculates the addition of half of the 32-bit data fields. The addition results were added together using an additional one's complement adder and the checksum was obtained by inverting the final result. In the parallel calculation of CRC, calculations based on different parallel multiple input bits were implemented and compared. 32-bit parallel input was selected as the final design due to its high throughput. And it was proved that the implementation of 32-bit parallel input can still fit those data with the lengths which are not the multiple of* 32. *Furthermore, a novel CRC fast update scheme has been proposed based on the observation that when a packet is forwarded by the routers, only a number of bits residing in the first 28 bytes of the Ethernet frame are changed and the rest remain intact. CRC fast update is designed to only calculate the changed 28 bytes using parallel CRC calculation and perform a single update step afterwards. For some selected fixed length frames, the number of cycles needed to obtain the new CRC code is 8. For variable length frames, the number of cycles are also reduced to a smaller number. Accordingly, the throughput of the CRC circuit can be largely increased. The CRC update scheme also provides advantages such as less power consumption and the reduced time used for data moving, etc. Finally, the NAT table lookup functions are implemented using Xilinx block SelectRAMs as Content addressable Memories (CAMs).*

This chapter is organized as follows. Section 5.1 gives a summary for the thesis. Section 5.2 addresses the main contributions of this thesis. Section 5.3 provides future research directions.

5.1 Summary

In Chapter 2, the general topics of TCP/IP protocol suite was first presented. The TCP/IP model, as a practical model for our thesis was illustrated and its five layers and protocols on each layer were discussed. Subsequently, the addressing, as a key issue in networking, was introduced. There are three different addresses: Ethernet address, IP address and port number and they span on different layers and take care different functionalities. Subsequently, we investigated the processing functions in the TCP/IP protocol stack and focused on the data plane, which requires processing at wire speeds. Micro-level functions were summarized, and network services built upon these micro-level functions were identified. Finally, the design tools and methodology were further described.

In Chapter 3, the selection of the performance-critical functions was described. Three functions have been selected to implement in FPGAs. The theory of selected TCP/IP functions was discussed in detail. For checksum calculation, the theory is based on the one's complement addition, and thus the 16-bit one's complement carry lookahead adder is designed to perform the calculation of checksum. The theory of CRC calculation is based on the modulo 2 arithmetic. The LFSR (Linear Forward Shift Register) was designed as the fundamental blocks in both serial and parallel CRC calculation. The parallel calculation of CRC was based on a recursive equation Equation (3.27) that is derived from LFSR. Furthermore, a novel fast CRC update method was introduced and its performance was evaluated. Finally, we presented the network address translation (NAT) service, and its main operation, the NAT table lookup.

In Chapter 4, the implementation details of the selected TCP/IP functions were discussed. The simulation results were given in waveforms and the synthesis results for different FPGAs were demonstrated and compared in tables. The performance of fast CRC update was compared with the CRC calculation as well. The NAT table lookup functions based on the use of the Xilinx on-chip block SelectRAMs as content addressable memories (CAMs) was explained. The CAMs were designed to store the NAT tables.

5.2 Main Contributions

The main contributions of this thesis are highlighted in the following.

1. The TCP/IP processing functions have been highlighted. Micro-level functions, which are performed over most of the packets passing through, were identified. The profiling of the micro-level functions were further investigated and we determined that checksum and CRC calculation are most computationally intensive functions.

2. For checksum calculation, the processing speed can reach 3.712 Gbps, in order to meet current trend of network speeds, more checksum adders can be implemented in parallel to perform the calculations. Pipelined checksum calculation can be also implemented to improve the performance. Furthermore, incremental checksum calculation can be implemented based on the 16-bit one's complement adder.

3. The throughput of the parallel calculation of CRC can reach 6.4 Gbps. A more efficient CRC update scheme was proposed. With this scheme, a throughput up to 50 Gbps can be achieved. Furthermore, since the CRC update only need to calculate the first 28 bytes of the frame, if 4 bytes old CRC code as well as the 4 bytes intermediate register states are included, totally, only 36 bytes of data need to be send to the CRC calculation units to be processed. The power consumption of the proposed design is much smaller since the reduced number of bits that is to be processed. The time costed in data moving is also reduces.

4. A network service called network address translation (NAT) was presented. The main operation in NAT was the table lookup. A novel method that utilize the Xilinx Virtex block SelectRAMs as content addressable memories (CAMs) has been developed to facilitate the table lookup.

5. With above designed incremental checksum update, CRC fast update, and other non time-critical micro-level functions, NAT operations on the data plane can be fully offloaded from GPPs to FPGAs.

5.3 Future Research Directions

In this thesis, three TCP/IP functions have been designed and implemented in FPGAs. In order to integrate these functions into the whole network processing system within the interconnecting devices, future work could be done in the following fields.

1. **Real hardware implementations and measurements:** In this thesis, three TCP/IP functions are designed. Synthesis and place&route are performed. Finally, bitstream files of the designs are obtained by the implementation tools. Therefore, future work can be performed in downloading these bitstream files into real FPGAs to measure the actual delay and performance as well as to verify the real functionalities.

2. **Checksum:** Our checksum calculation is based on the 16-bit one's complement adders. The current design can only support throughput up to 3.7 Gbps. That is not fast enough to meet the future network speeds. Further efforts can be made to design a faster 16-bit one's complement adder, and a more complicated pipeline implementation can be designed to improve the performance.

3. **Cyclic redundancy check:** The parallel calculation of CRC-32 and its fast update are implemented. The performance can meet both the current and the future network speeds. The future work that need to be completed is as follows.

    - Queuing and Scheduling: The design of the fast CRC update introduced a special buffer for storing the old CRC and the intermediate register states. Therefore, a good queuing and scheduling mechanism must be designed to incorporate the special buffer with the queue which is used to store packets.
    - Control plane tasks: Control plane tasks should be updated to incorporate the queuing and scheduling functions in order to integrate the designed functions

into the whole system. Control plane tasks should also be able to recognize the changed fields in the packet in order to decide how many bytes need to be calculated in parallel calculation of CRC before performing fast update. Furthermore, some control plane tasks can be developed in cooperation with the designed functions to provide flexibility. For example, the parallel calculation of CRC-32 is implemented, the design can also applicable to the other CRC algorithms in case they are needed. Therefore, the F matrix can be pre-calculated and the corresponding designs can be configured into FPGAs, this task can be accomplished by some control plane tasks running at the general-purpose processors.

4. **Network address translation:** NAT table lookup using Virtex on-chip block SelectRAMs has been implemented. As discussed in Section 5.2, table lookup is a main operation of network address translation. The other functions like data parsing, checksum updating, CRC updating can be further implemented in FPGAs so that the total operations in data plane of NAT service can be integrated in an FPGA. As a result, the data processing workload of NAT can be fully offloaded from general-purpose processors (GPPs). GPPs can only take charge of the control tasks such as time counting to update NAT table, public IP address allocations, and etc.

5. **More data plane functions:** Functions running on the data plane can be further investigated and implemented. The function classification, for example, is widely utilized by modern network devices to provide advanced service like QoS.

Bibliography

[1] Jean-Louis Brelet and Lakshmi Gopalakrishnan, *Using Virtex-II Block RAM for High Performance Read/Write CAMs*, Xilinx Application Notes (2002).

[2] Dexter Chun and J.K.Wolf, *Special Hardware for computing the probability of undetected error for certain binary CRC codes and test results*, IEEE Transaction on Communications (1994).

[3] Cisco, *Interconnecting Cisco Network Devices*, CA: Cisco System, Inc., 1999.

[4] Eric Yeh et al., *Introduction to TCP/IP Offload Engine(TOE)*, 2002.

[5] Mei Tsai et al, *A Benchmarking Methodology for Network Processors*, 1st Workshop on Network Processors(NP-1), 8th Int. Symposium on High Performance Computing Architectures (HPCA-8) (2002).

[6] Memik et al., *Evaluating Network Processors using Netbench*, ACM Transactions on Embedded Computing System (2002).

[7] R. Braden et al., *Computing the Internet Checksum*, IETF RFC 1071 (1988).

[8] Marcel Waldvogel Florian Braun, *Fast Incremental CRC Updates for IP over ATM Networks*, IEEE Workshop on High Performance Switching and Routing (2001).

[9] Behrouz A. Forouzan, *TCP/IP Protocol Suite*, McGraw-Hill, 2003.

[10] G. Patane G. Campobello and M. Russo, *Parallel CRC Realization*, 2003.

[11] Xilinx Inc., *ISE 5 In-Depth Tutorial*, 2003.

[12] William H. Mangione-Smith Jason Leonard, *A Case Study of Partially Evaluated Hardware Circuits: Key-Specific DES*, Field-Programmable Logic and Applications. 7th International Workshop (1997).

[13] Jonathan Kay and Joseph Pasquale, *Profiling and Reducing Processing Overheads in TCP/IP*, IEEE/ACM Transactions on Networking (1992).

[14] Eddie Kohler, *The Click Modular Router*, Ph.D. thesis, Massachusetts Institute of Technology, 2001.

[15] Bruce S. Davie Larry L. Peterson, *Computer Networks, A System Approach*, Morgan Kaufmann Publishers, San Fransisco, California, 2000.

[16] Piet Van Mieghem, *Data Communication Networking*, Delft University of Technology, 2003.

[17] Mitsuhiro Miyazaki, *Workload Characterization and Performance for a Network Processor*, Master's thesis, Princeton University, Department of Electrical Engineering, 2002.

[18] Behrooz Parhami, *Computer Arithmetic, Algorithms and Hardware Designs*, Oxford University Press, 2000.

[19] Tong-Bi Pei and Charles Zukowski, *High-Speed Parallel CRC Circuits in VLSI*, IEEE Transactions on Communications (1992).

[20] A. Rijsinghani, *Computation of the Internet Checksum via Incremental Update* , IETF RFC 1624 (1994).

[21] J.Rose S.D. Brown, R.J. Francis, *Field Programmable Gate Arrays*, Kluwer Academic Publishers, 1992.

[22] S.Lin and Jr D.J.Costello, *Error Control Coding: Fundamentals and Applications*, Prentice-Hall, Inc., 1983.

[23] Sorin Cotofana Stamatis Vassiliadis, Stephan Wong, *Network Processors: Issues and Prospectives*, in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA-2001) (2001).

[24] Richard Stevens, *TCP/IP Illustrated, Volume 1*, Addison-Wesley, 1994.

[25] Model Technology Product Support, *ModelSim HDL Simulation FPGA Design Flow*, www.model.com (2001).

[26] Tomas Henrikson Ulf Nordqvist and Dake Liu, *CRC Generation for Protocol Processing*, In proceedings of NORCHIP 2000, Turku, Finland (2000).

[27] *Modeling Tomorrow's Internet*, www.lightreading.com.

A

F^{32} matrix for CRC-32

number of ones per row	F^{32} matrix
13	1 1 1 1 1 0 1 1 1 0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0 0 1 0 0 0 0 0
14	0 1 1 1 1 1 0 1 1 1 0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0 0 1 0 0 0 0
14	1 0 1 1 1 1 1 0 1 1 1 0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0 0 1 0 0 0
14	0 1 0 1 1 1 1 1 0 1 1 1 0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0 0 1 0 0
15	0 0 1 0 1 1 1 1 1 0 1 1 1 0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0 0 1 0
13	1 0 0 1 0 1 1 1 1 1 0 1 1 1 0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0 0 1
13	1 0 1 1 0 0 0 0 0 1 1 0 1 1 1 0 1 0 0 0 1 0 0 1 0 0 0 0 1 1 0 0
13	0 1 0 1 1 0 0 0 0 0 1 1 0 1 1 1 0 1 0 0 0 1 0 0 1 0 0 0 0 1 1 0
14	1 0 1 0 1 1 0 0 0 0 0 1 1 0 1 1 1 0 1 0 0 0 1 0 0 1 0 0 0 0 1 1
14	1 0 1 0 1 1 0 1 1 0 0 0 1 1 0 1 0 1 0 1 1 0 1 0 0 0 0 0 0 0 0 1
12	1 0 1 0 1 1 0 1 0 1 0 0 0 1 1 0 0 0 1 0 0 1 1 0 0 0 1 0 0 0 0 0
12	0 1 0 1 0 1 1 0 1 0 1 0 0 0 1 1 0 0 0 1 0 0 1 1 0 0 0 1 0 0 0 0
12	0 0 1 0 1 0 1 1 0 1 0 1 0 0 0 1 1 0 0 0 1 0 0 1 1 0 0 0 1 0 0 0
13	1 0 0 1 0 1 0 1 1 0 1 0 1 0 0 0 1 1 0 0 0 1 0 0 1 1 0 0 0 1 0 0
14	1 1 0 0 1 0 1 0 1 1 0 1 0 1 0 0 0 1 1 0 0 0 1 0 0 1 1 0 0 0 1 0
14	0 1 1 0 0 1 0 1 0 1 1 0 1 0 1 0 0 0 1 1 0 0 0 1 0 0 1 1 0 0 0 1
15	0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 1 1 0 0 1 0 0 1 1 1 0 1 1 1 0 0 0
15	0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 1 1 0 0 1 0 0 1 1 1 0 1 1 1 0 0
16	1 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 1 1 0 0 1 0 0 1 1 1 0 1 1 1 0
17	1 1 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 1 1 0 0 1 0 0 1 1 1 0 1 1 1
17	1 0 0 1 1 1 1 1 0 0 0 1 0 0 1 1 1 1 0 1 0 0 1 0 0 0 0 1 1 0 1 1
17	1 0 1 1 0 1 0 0 0 0 0 0 1 0 0 1 0 1 1 0 0 0 1 0 0 0 1 0 1 1 0 1
12	0 0 1 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 1 1 1 0 1 0 0 0 1 1 0 1 1 0
13	1 0 0 1 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 1 1 1 0 1 0 0 0 1 1 0 1 1
16	0 0 1 1 0 0 1 1 1 1 1 0 0 0 0 1 1 0 0 0 0 1 0 1 1 0 1 0 1 1 0 1
15	0 1 1 0 0 0 1 0 0 1 1 1 0 0 0 0 0 1 0 0 1 0 0 1 1 1 1 1 0 1 1 0
15	0 0 1 1 0 0 0 1 0 0 1 1 1 0 0 0 0 0 1 0 0 1 0 0 1 1 1 1 1 0 1 1
17	1 1 1 0 0 0 1 1 0 0 0 1 1 1 0 0 1 0 0 1 1 0 0 1 0 1 0 1 1 1 0 1
15	1 0 0 0 1 0 1 0 0 0 0 0 1 1 1 0 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 0
16	1 1 0 0 0 1 0 1 0 0 0 0 0 1 1 1 0 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1
13	0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 0 0 1 1 1 0 1 0 1 1 0 0 0 0 1 1
13	1 1 1 1 0 1 1 1 0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0 0 1 0 0 0 0 0 1
Total: 452	

B

Proof of Equation (3.23)

$[F^{i-1}G \; \cdots \; FG \;\; G][U(0) \ldots U(i-1)]^T = [0 \ldots 0, U(0) \ldots U(i-1)]^T$

Proof:

Given,

$$
\begin{aligned}
F &= \begin{bmatrix} p_{m-1} & 1 & 0 & \cdots & 0 \\ p_{m-2} & 0 & 1 & \cdots & 0 \\ \cdots & \cdots & \cdots & \ddots & \cdots \\ p_1 & 0 & 0 & \cdots & 1 \\ p_0 & 0 & 0 & \cdots & 0 \end{bmatrix} \\
G &= [0\,0 \cdots 0\,1]^T
\end{aligned}
$$

F is a 32×32 matrix and G is a 32×1 vector. We have,

$$
\begin{aligned}
G &= [0\,0 \cdots 0\,1]^T \\
FG &= [0\,0 \cdots 1\,0]^T \\
F^2G &= \begin{bmatrix} \mathrm{x} & p_{m-1} & 1 & 0 & 0 & \cdots & 0 \\ \mathrm{x} & p_{m-2} & 0 & 1 & 0 & \cdots & 0 \\ \mathrm{x} & p_{m-3} & 0 & 0 & 1 & \cdots & 0 \\ \mathrm{x} & \cdots & \cdots & \cdots & \cdots & \ddots & \cdots \\ \mathrm{x} & p_2 & 0 & 0 & \cdots & 0 & 1 \\ \mathrm{x} & p_1 & 0 & 0 & \cdots & 0 & 0 \\ \mathrm{x} & p_0 & 0 & 0 & \cdots & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ \cdots \\ 0 \\ 0 \\ 1 \end{bmatrix} \\
&= [0\,0\,0 \cdots 1\,0\,0]^T \\
&\;\vdots \\
F^{i-1}G &= \begin{bmatrix} \mathrm{x} & \cdots & \mathrm{x} & p_{m-1} & 1 & 0 & 0 & \cdots & 0 \\ \mathrm{x} & \cdots & \mathrm{x} & p_{m-2} & 0 & 1 & 0 & \cdots & 0 \\ \mathrm{x} & \cdots & \mathrm{x} & p_{m-3} & 0 & 0 & 1 & \cdots & 0 \\ \mathrm{x} & \cdots & \mathrm{x} & \cdots & \cdots & \cdots & \cdots & \ddots & \cdots \\ \mathrm{x} & \cdots & \mathrm{x} & p_i & 0 & 0 & 0 & \cdots & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \mathrm{x} & \cdots & \mathrm{x} & p_1 & 0 & 0 & 0 & \cdots & 0 \\ \mathrm{x} & \cdots & \mathrm{x} & p_0 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ \cdots \\ 0 \\ \cdots \\ 0 \\ 1 \end{bmatrix} \\
&= [0\,0\,0 \cdots 1 \cdots 0\,0]^T
\end{aligned}
$$

where x denotes either '1' or '0', the value is not important in the proof. And we get

$$[F^{i-1}G \;\cdots\; FG \;\; G] \;=\; \underbrace{\begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \cdots & \cdots & \cdots & \ddots & \cdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}}_{i\text{ columns}}$$

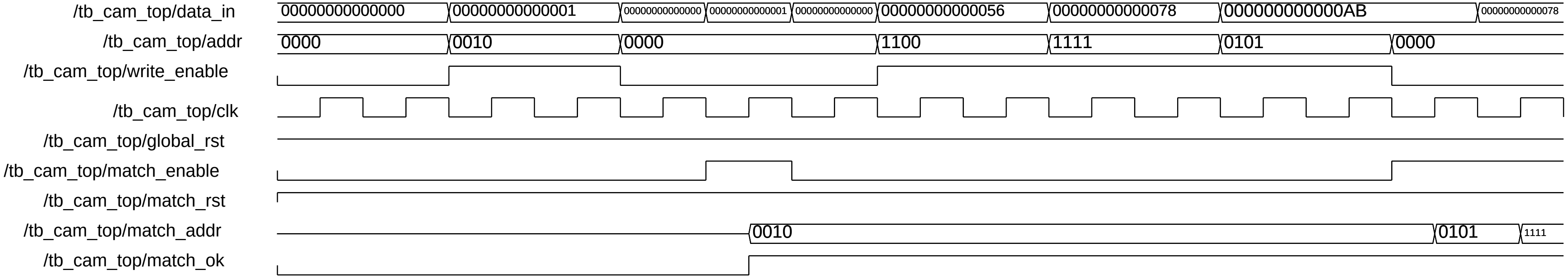
$[F^{i-1}G \;\cdots\; FG \;\; G]$ is a $m \times i$ matrix, and its lower part is a $i \times i$ identity matrix, therefore,

$$[F^{i-1}G \;\cdots\; FG \;\; G][U(0)\ldots U(i-1)]^T \;=\; [\underbrace{0\ldots 0}_{m\text{-}i},\;\; \underbrace{U(0)\ldots U(i-1)}_{i}]^T$$

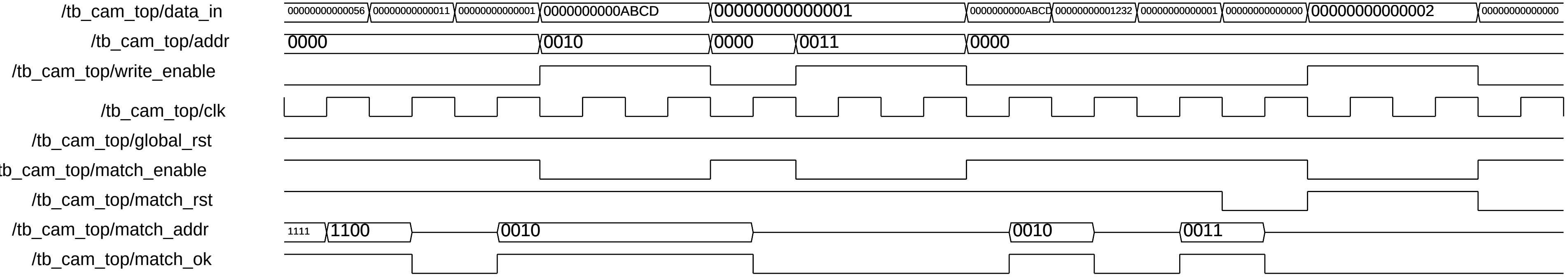
C

16 × 56 CAM Waveform

Please refer to the next page.

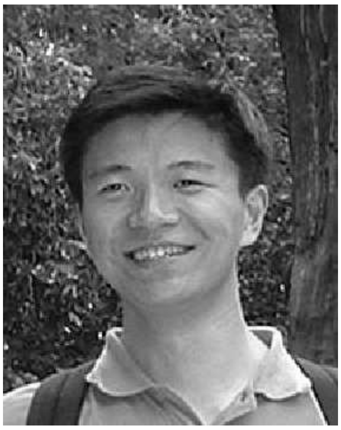


16× 56 CAM Waveform



16× 56 CAM Waveform Continued

Curriculum Vitae



Weidong Lu was born in Nanjing, China, on August 20th, 1979. In 1994, he entered Nanjing Jinling High School, where he spend three memorable years with his excellent teachers and classmates. In 1997, he started his undergraduate study in the Radio Engineering Department at Southeast University, Nanjing, China. Four years study led him into the intriguing world of electrical engineering. And in July 2001, he received his Bachelor's degree.

From August 2001, he started his Master of Science study in the Electrical Engineering Department at Delft University of Technology(TU Delft), Delft, The Netherlands. In September 2002, he entered the Computer Engineering Laboratory, where he started his M.Sc thesis under the supervision of dr.ir. Stephan Wong. His thesis topic is "Designing TCP/IP functions in FPGAs" and his research interests include: high performance computer networks, computer architecture, embedded system design and wireless communications and networks.