

Faculty of Computer Science & Information Technology



Perpustakaan SKTM

VHDL Description for IP Engine

WXES3182

Name: Zellal Bathich

Matrix: WEK010401

Supervisor : Mr. Noorzaily Mohd. Noor

Moderator : Mr. Yamani Mohd. Idna Idris

Table Of Contents

Abstract

Acknowledgement

Chapter 1: Introduction

1.1 Introduction.....	1
1.2 Problems Definitions.....	2
1.3 Scope.....	2
1.4 Objectives.....	4
1.5 Constraints.....	5
1.6 Scheduling.....	6

Chapter 2: Literature Review

2.1 Introduction.....	8
2.2 Network Protocol Layer.....	8
2.3 Layering Models.....	9
2.4 The TCP/IP Stack.....	10
2.5 TCP/IP Protocols.....	13
2.6 Internet Protocol.....	14
2.7 IP Address.....	16
2.8 IP Address Classes.....	17
2.9 Netmasks.....	18
2.10 Subnet Address.....	18
2.11 Direct Broadcast Address.....	19
2.12 limited Broadcast Address.....	19
2.13 IP Routing.....	19
2.14 ARP.....	20
2.15 IP Packet Structure.....	21
2.16 IP Packet Processing.....	24
2.17 IP Fragmentation Processing at a Router.....	26
2.18 IP Fragmentation Processing at the receiving End System.....	27
2.19 Reception of a Frame form The Ethernet.....	27
2.20 Comparison between Existing System And Proposed System Architecture.....	30
2.20.1 Existing System.....	30
2.20.2 System Proposed.....	31
2.20.3 Protocol Processor (Proposed System).....	33
2.20.4 IP Engine.....	36

Chapter 3: Methodology

3.1 Methodology.....	38
----------------------	----

Chapter 4: System Analysis

4.1 VHDL.....	40
4.2 What Is VHDL.....	40
4.3 VHDL Advantages.....	42
4.4 VHDL And Verilog Comparison.....	44

Chapter 5: System Design

5.1 System Design.....	48
5.2 IP Engine Block Diagram.....	49
5.3 Internal Block Diagram.....	51
5.4 Process Flow.....	56

Chapter 6: System Implementation

6.1 Introduction.....	57
6.2 Design Entry.....	58
6.3 Modeling Entity.....	59
6.4 Model Analysis.....	68
6.5 Synthesis.....	68

Chapter 7: System Testing And Evaluation

7.1 Simulation and Testing.....	70
7.2 Cycle Simulation.....	72
7.3 System Testing.....	73

Abstract

Chapter 8: Discussion.....75

Chapter 9: Conclusion.....76

References

Appendices A : VHDL Source Code

Appendices B : PeakFPGA User Manual

Appendices C : Main References

Abstract

Developing hardware support for network layer protocol processing is a very complex and demanding task. However, for optimal performance hardware acceleration can be required. To cope with the situation, this project presents a high-level design approach, which targets the development of configurable and reusable components. Therefore it obtains the integration of advanced tools for the development of the IP Engine into the design environment.

This process is illustrated based on a TCP/IP header analysis and validation component for which initial performance results are presented. The development of this Engine is embedded in an approach to develop flexible and configurable protocol engines that can be optimized for specific application.

By implementing the IP Engine in hardware it will help reducing communication bottlenecks replacing expensive software solutions, which are based on 32 bit processor cores. With its small footprint design it will improve low power-consumption, highly cost-effective solution to Perform all protocol functions of TCP/IP and UDP/IP connections for sustained bit rates of up to 100 Mbps independent of packet payload sizes and other connection parameters.

Acknowledgment

Utter most gratitude goes to the almighty Allah for all the confidence and patience in the completion of part 1 and part 2 of the thesis. I wish to record my indebtedness and appreciation to everyone who has been so helpful and supportive in this project work and brought it to success.

I would like to express my deep gratitude to my supervisor Mr Noorzaily Mohd. Noor for the tremendous help he has given me during this project, technical advice and thoughtful comment. And also to my examiner Mr. Yamani Mohd. Idna Idris for his guidance and sharing his experience and knowledge. The valuable advice and motivation will be cherished thus to develop a personal values of mine in the future.

Also taking this opportunity expressing my thanks to all fellow members and especially the family of Computer Science and Networking for their constructive criticism and support to face the difficulties and challenging time.

Finally, last but not least, I am much obliged to my dear parents who have been given invaluable support and inspiration to me throughout my university life. My gratefulness also goes to all the unnamed others who directly or indirectly helped me to complete this interesting and challenging project. With this sheet of paper, I can only say thank you with all my heart.

1.1 Introduction

Most systems today, which require embedded Internet connectivity, make use of a 32-bit processor core and implement the TCP/IP protocol stack in software.

This architecture however often results in strong processor performance requirements and keeps system costs for Internet applications high.

Towards improving performance, we undertake a hardware implementation of a signaling protocol to eliminate the burden on the host CPU, drastically reduce latency in the server and help to fully accelerate data transmission in a memory-less fashion.



Chapter 1

Introduction

The system will support:

- throughput for all packet sizes
- up to 64K connections
- stateless capability
- complete TCP/IP solution

1.1 Introduction

Most systems today, which require embedded Internet connectivity, make use of a 32-bit processor core and implement the TCP/IP protocol stack in software.

This realization however often results in strong processor performance requirements and keeps system costs for Internet application high.

Towards improving performance, we undertook a hardware implementation of a signaling protocol to eliminate the burden on the host CPU, dramatically reduce bottlenecks in the server and help in faster and accurate data transmission in a tremendous network environment.

The system is designed to enhance performance and power consumptions of embedded systems. It performs all protocol functions of TCP/IP and UDP/IP connections for sustained bit rates of up to 100 Mbps independent of packet payload sizes and other connection parameters. It provides IP connectivity even without any external processor interaction, which makes it also an ideal internet access solution for existing applications.

The system will support:

- 100 mb/s throughput for all packet sizes
- support of up to 64K connections
- stand-alone capability
- complete TCP/IP solution

1.2 Problem Definition

Through the analysis process of the project I discovered few problems that should be overcome which are:

- Limited references
- Limited given time to finish the first part of the project, which is to analysis and design the system.
- The difficulties of implementing theoretical definitions and approaches into real system.
- The execution of the overall system is hard to implement because of other different protocols, which is included.

1.3 Scope

The project scope determines part of the project process, which will overcome the burden of the overall system development.

- By implementing the Internet Protocol (IP) engine in hardware which off-loads performance intensive Internet protocols from processor sub-systems and allows for separate system optimization.

- Several options to overcome the large cost of the hardware implementation:

Simulation: Simulation is one of the important steps of complex hardware design. Open Hardware designers may simulate their designs only without implementing them. In this way they did the design using free simulators without the cost of implementation.

The use of Programmable Logic: These days the programmable logic devices become very popular and have lot of hardware resources that can compete old ASICs. These devices showed some good examples of real complex designs built using them. They can be programmed in field using a PC or small programmer. This approach becomes too close to the software designs, since any one can design his/her own hardware and program it on one of these devices.

The entire IP Engine can be constructed out of interrelated sub modules, but this is very complicated when it is implemented directly. It's difficult to follow which input lines correspond to certain variables and what their values would be. It's more efficient to use some other method to construct the IP Engine that models it in a method that is easier to understand. It is for this reason that programmable logic language like VHDL were created. VHDL is a very popular language for describing modeling and synthesis of digital circuits and systems. Its powerful but narrow field of usage makes it difficult to find software packages that easily implement testing of the VHDL code.

1.4 Objectives

While software implementations require very fast processors to follow Ethernet transmission speeds, it provides a sustained bit rate of 100 Mbps up to the TCP/UDP layer of the Internet protocol stack. Instead of assigning considerable resources to interrupt driven processor context switches and memory access operations, the protocol processor implements a hardware architecture, which directly operates on the communication data stream. With a total of 100k logic gates and an operating frequency of only 25MHz, this hardware engine consumes less power and provides a competitive, small footprint solution.

The main objectives of implementing the Internet Protocol Engine in hardware are:

- Improving price, performance and power consumption of embedded systems.
- Reducing network latency
- Reducing system overhead
- Accelerates network performance to full wire speed
- 100 Mb/s throughput for all packet sizes
- Support of up to 64k connections
- Complete TCP/IP solution

1.5 Constraints

The expected constraints will be faced during system development are:

- Take a lot of time (including VHDL or Verilog design and simulation).
- There are several key limitations to the design of the IP stack, most of which are due to the limited amount of hardware, RAM and buffer space available on an FPGA (assuming the IP stack shouldn't take up over half the FPGA in size).
- The lack of buffer space also creates problems if multiple datagrams are being received and reassembled at once, or if the transport layer protocols are busy then datagrams will have to be dropped as no IP buffers will be free unless more IP buffers or transport layer buffers are allocated. Increasing the number buffers results in a large increase of memory usage and logic needed for controlling them.
- Require a robust FPGA.

1.6 Scheduling

The bar chart below shows the activities of each process phase that will be carried out through the development of the system. It will take an approximate time of 9 months to finish the whole thesis project. Starting on the first phase, which is system analysis from June until July. At this phase, information is collected on systems available and study is made on the methodology that will be used in this project.

The second phase starts from August until September, which is working on the system design. At the beginning of October the second part of the thesis will be started by the implementation of the system, which is the system coding. System testing will be carried out at the middle of December until the end of January. The system will be tested to check if it's free from errors.

The last phase of system development is the system evaluation. It starts at the end of January until the end of February. The required system output will be checked in this phase.

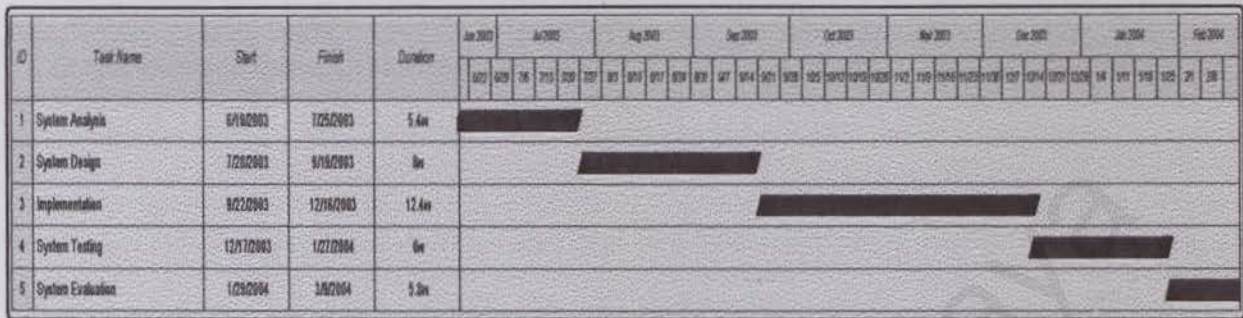


Figure 1.1 Activities Schedule Bar Chart

2.1 Introduction

A **protocol** is set of rules and conventions used to impose a standardized, structured language for the communication between multiple parties. For example, a protocol might define the order in which information is exchanged between two parties. In fact, a data exchange can *only* take place between two computers using the same protocol.

Transmitting data across computer networks is an arduous task. Network functionality has been decomposed into modules called **layers** to simplify and separate the tasks associated with data transmission. Each layer is a unit of code that performs a small, well-defined set of tasks. A **protocol suite** (or **protocol stack**) is a set of many such layers, and is usually a part of the operating system kernel on machines connected to the Internet.

A protocol stack is organized such that the highest level of abstraction resides at the top layer. For example, the highest layer may deal with streaming audio or video frames, whereas the lowest layer deals with raw voltages or radio signals. Every layer in a stack builds upon the services provided by the layer immediately below it.

2.2 Network Protocol Layers

Computers on a network communicate in agreed upon ways called protocols. The complexity of networking protocol software calls for the problem to be divided into smaller pieces. A layering model aids this division and provides the conceptual basis for

understanding how software protocols together with hardware devices provide a powerful communication system.

2.3 Layering Models

In the early days of networking, before the rise of the ubiquitous Internet, the International Organization for Standardization (ISO) developed a layering model whose terminology persists today.

	Name of Layer	Purpose of Layer
Layer 7	Application	Specifies how a particular application uses a network.
Layer 6	Presentation	Specifies how to represent data.
Layer 5	Session	Specifies how to establish communication with a remote system.
Layer 4	Transport	Specifies how to reliably handle data transfer.
Layer 3	Network	Specifies addressing assignments and how packets are forwarded.
Layer 2	Data Link	Specifies the organization of data into frames and how to send frames over a network.
Layer 1	Physical	Specifies the basic network hardware.

Table 2.1. ISO 7-Layer Reference Model

The 7-layer model has been revised to the 5-layer TCP/IP reference model to meet the current needs of protocol designers.

2.4 The TCP/IP Stack

The picture below is an example of a simple data transfer between 2 computers and shows how the data is sent and received through the TCP/IP stack.

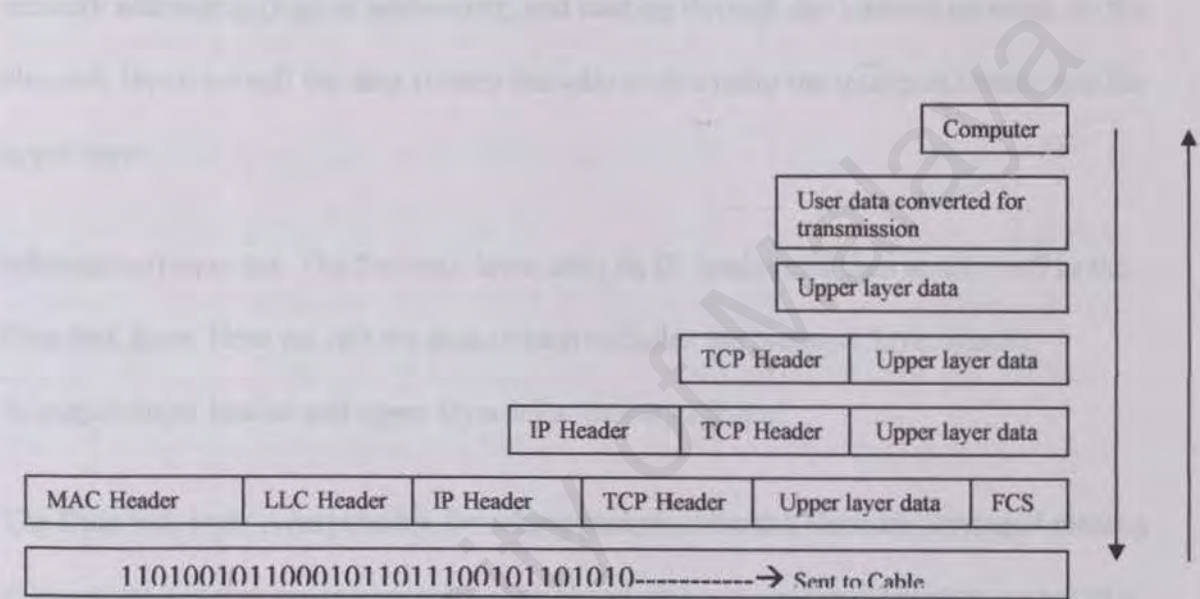


Figure 2.1. Shows Data Sent and Received through the Stack

The computer in the above diagram needs to send some data to another computer. The Application layer is where the user interface exists, here the user interacts with the application he or she is using, and then this data is passed to the Presentation layer and then to the Session layer. These three layers add some extra information to the original data that came from the user and then passes it to the Transport layer. Here the data is

broken into smaller pieces (one piece at a time transmitted) and the TCP header is added. At this point, the data at the Transport layer is called a *segment*.

Each segment is sequenced so the data stream can be put back together on the receiving side exactly as transmitted. Each segment is then handed to the Network layer for network addressing (logical addressing) and routing through the Internet network. At the Network layer, we call the data (which includes at this point the transport header and the upper layer

information) a *packet*. The Network layer adds its IP header and then sends it off to the Data link layer. Here we call the data (which includes the Network layer header, Transport layer header and upper layer information) a *frame*.

The Data link layer is responsible for taking packets from the Network layer and placing them on the network medium (cable). The Data link layer encapsulates each packet in a frame that contains the hardware address (MAC) of the source and destination computer (host) and the LLC information, which identifies to which protocol in the previous layer (Network layer) the packet should be passed when it arrives to its destination. Also, at the end, we will notice the FCS field that is the Frame Check Sequence. This is used for error checking and is also added at the end by the Data link layer.

If the destination computer is on a remote network, then the frame is sent to the router or gateway to be routed to the destination. To put this frame on the network, it must be put into a digital signal. Since a frame is really a logical group of 1's and 0's, the Physical

layer is responsible for encapsulating these digits into digital signals, which is read by devices on the same local network.

At the receiving process, computer will synchronize the digital signal by reading the few extra 1's and 0's as mentioned above. Once the synchronization is complete and it receives the whole frame it will pass it to the layer above it, which is the Data link layer.

The Data link layer will do a Cyclic Redundancy Check (CRC) on the frame. This is a computation, which the computer does, and if the result it gets matches the value in the FCS field, then it assumes that the frame has been received without any errors. Once that's out of the way, the Data link layer will strip off any information or header, which was put on by, the remote system's Data link layer and pass the rest (now we are moving from the Data link layer to the Network layer, so we call the data a *packet*) to the above layer which is the Network layer.

At the Network layer the IP address is checked and if it matches (with the machine's own IP address) then the Network layer header is stripped off from the packet and the rest is passed to the above layer, which is the Transport layer. Here the rest of the data is now called a *segment*.

The segment is processed at the Transport layer, which rebuilds the data stream (at this level on the sender's computer it was actually split into pieces so they can be transferred) and acknowledges to the transmitting computer that it received each piece. It is obvious

that since we are sending an ACK back to the sender from this layer that we are using TCP and not UDP.

We will find that when analyzing the way data travels from one computer to another most people never analyze in detail any layers above the Transport layer. This is because the whole process of getting data from one computer to another involves usually layers 1 to 4 (Physical to Transport) or layer 6 (Session) at the most, depending on the type of data.

2.5 TCP/IP Protocols

This chapter discusses the protocols available in the TCP/IP protocol suite. The following figure shows how they correspond to the 5-layer TCP/IP Reference Model. This is not a perfect one-to-one correspondence for instance, Internet Protocol (IP) uses the Address Resolution Protocol (ARP), but is shown here at the same layer in the stack.

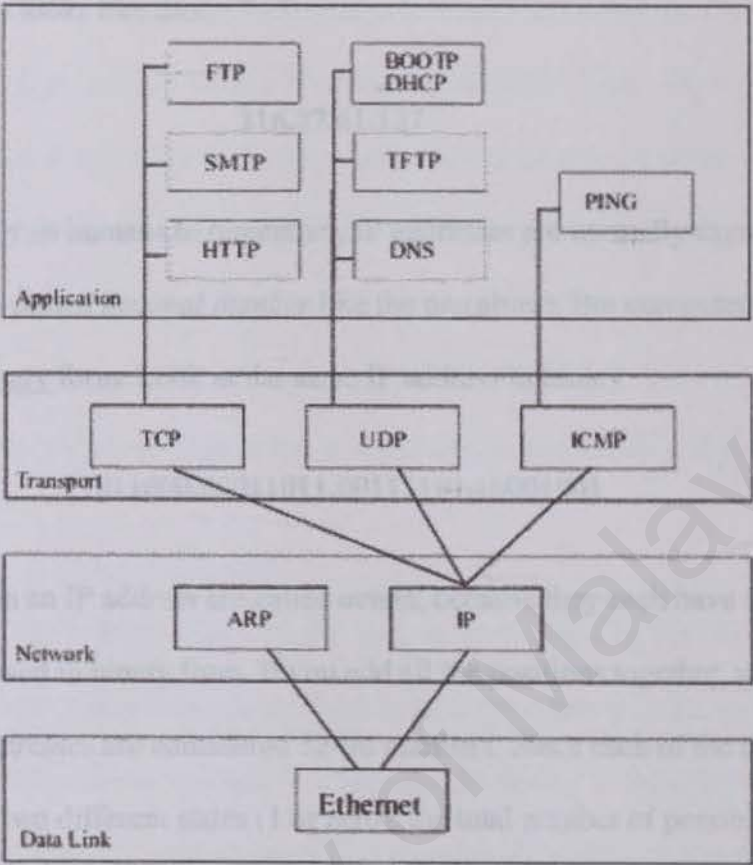


Figure 2.2. TCP/IP Protocol Flow

2.6 Internet Protocol (IP)

Every machine on the Internet has a unique identifying number, called an **IP Address**.

The IP stands for **Internet Protocol**, which is the language that computers use to communicate over the Internet. A protocol is the pre-defined way that someone who wants to use a service talks with that service. The "someone" could be a person, but more often it is a computer program like a Web browser.

A typical IP address looks like this:

216.27.61.137

To make it easier for us humans to remember, IP addresses are normally expressed in decimal format as a *dotted decimal number* like the one above. But computers communicate in binary form. Look at the same IP address in binary:

11011000.00011011.00111101.10001001

The four numbers in an IP address are called **octets**, because they each have eight positions when viewed in binary form. If you add all the positions together, you get 32, which is why IP addresses are considered 32-bit numbers. Since each of the eight positions can have two different states (1 or zero), the total number of possible combinations per octet is 2^8 or 256. So each octet can contain any value between zero and 255. Combine the four octets and you get 2^{32} or a possible 4,294,967,296 unique values!

Out of the almost 4.3 billion possible combinations, certain values are restricted from use as typical IP addresses. For example, the IP address 0.0.0.0 is reserved for the default network and the address 255.255.255.255 is used for broadcasts.

The octets serve a purpose other than simply separating the numbers. They are used to create **classes** of IP addresses that can be assigned to a particular business, government or other entity based on size and need. The octets are split into two sections: **Net** and **Host**. The Net section always contains the first octet. It is used to identify the network that a

computer belongs to. Host (sometimes referred to as **Node**) identifies the actual computer on the network. The Host section always contains the last octet. There are five IP classes plus certain special addresses. When the Internet was in its infancy, it consisted of a small number of computers hooked together with modems and telephone lines. You could only make connections by providing the IP address of the computer you wanted to establish a link with. For example, a typical IP address might be 216.27.22.162. This was fine when there were only a few hosts out there, but it became unwieldy as more and more systems came online.

The first solution to the problem was a simple text file maintained by the Network Information Center that mapped names to IP addresses. Soon this text file became so large it was too cumbersome to manage. In 1983, the University of Wisconsin created the **Domain Name System** (DNS), which maps text names to IP addresses automatically. This way you only need to remember `www.um.edu.my`, for example, instead of UM's IP address.

2.7 IP Address

IP defines an addressing scheme that is independent of the underlying physical address (e.g. 48-bit MAC address). IP specifies a unique 32-bit number for each host on a network. This number is known as the Internet Protocol Address, the IP Address or the Internet Address. These terms are interchangeable. Each packet sent across the Internet contains the IP address of the source of the packet and the IP address of its destination.

For routing efficiency, the IP address is considered in two parts: the prefix, which identifies the physical network, and the suffix, which identifies a computer on the network. A unique prefix is needed for each network in an Internet. For the global Internet, network numbers are obtained from Internet Service Providers (ISPs). ISPs coordinate with a central organization called the Internet Assigned Number Authority (IANA).

2.8 IP Address Classes

The first four bits of an IP address determine the class of the network. The class specifies how many of the remaining bits belong to the prefix (aka Network ID) and to the suffix (aka Host ID). The first three classes, A, B and C, are the primary network classes.

Class	First 4 Bits	Number Of Prefix Bits	Max Of Networks	Number Of Suffix Bits	Max Of Hosts Per Network
A	0xxx	7	128	24	16,777,216
B	10xx	14	16,384	16	65,536
C	110x	21	2,097,152	8	256
D	1110	Multicast			
E	1111	Reserved for future use.			

Table 2.2. IP addressing classes

When interacting with mere humans, software uses dotted decimal notation; each 8 bits is treated as an unsigned binary integer separated by periods. IP reserves host address 0 to denote a network. 140.211.0.0 denotes the network that was assigned the class B prefix 140.211.

2.9 Netmasks

Netmasks are used to identify which part of the address is the Network ID and which part is the Host ID. This is done by a logical bitwise-AND of the IP address and the netmask. For class A networks the netmask is always 255.0.0.0; for class B networks it is 255.255.0.0 and for class C networks the netmask is 255.255.255.0.

2.10 Subnet Address

All hosts are required to support subnet addressing. While the IP address classes are the convention, IP addresses are typically subnetted to smaller address sets that do not match the class system. The suffix bits are divided into a subnet ID and a host ID. This makes sense for class A and B networks, since no one attaches as many hosts to these networks as is allowed. Whether to subnet and how many bits to use for the subnet ID is determined by the local network administrator of each network.

If subnetting is used, then the netmask will have to reflect this fact. On a class B network with subnetting, the netmask would not be 255.255.0.0. The bits of the Host ID that were used for the subnet would need to be set in the netmask.

2.11 Directed Broadcast Address

IP defines a directed broadcast address for each physical network as all ones in the host ID part of the address. The network ID and the subnet ID must be valid network and subnet values. When a packet is sent to a network's broadcast address, a single copy travels to the network, and then the packet is sent to every host on that network or subnetwork.

2.12 Limited Broadcast Address

If the IP address is all ones (255.255.255.255), this is a limited broadcast address; the packet is addressed to all hosts on the current (sub)network. A router will not forward this type of broadcast to other (sub)networks.

2.13 IP Routing

Each IP datagram travels from its source to its destination by means of routers. All hosts and routers on an Internet contain IP protocol software and use a routing table to determine where to send a packet next. The destination IP address in the IP header contains the ultimate destination of the IP datagram, but it might go through several other IP addresses (routers) before reaching that destination.

Routing table entries are created when TCP/IP initializes. The entries can be updated manually by a network administrator or automatically by employing a routing protocol such as Routing Information Protocol (RIP). Routing table entries provide needed

information to each local host regarding how to communicate with remote networks and hosts. When IP receives a packet from a higher-level protocol, like TCP or UDP, the routing table is searched for the route that is the closest match to the destination IP address. The most specific to the least specific route is in the following order:

- A route that matches the destination IP address (host route).
- A route that matches the network ID of the destination IP address (network route).
- The default route. If a matching route is not found, IP discards the datagram.

2.14 ARP

The Address Resolution Protocol is used to translate virtual addresses to physical ones. The network hardware does not understand the software-maintained IP addresses. IP uses ARP to translate the 32-bit IP address to a physical address that matches the addressing scheme of the underlying hardware (for Ethernet, the 48-bit MAC address).

TCP/IP can use any of the three. ARP employs the third strategy, message exchange. ARP defines a request and a response. A request message is placed in a hardware frame (e.g., an Ethernet frame), and broadcast to all computers on the network. Only the computer whose IP address matches the request sends a response.

2.15 IP Packet Structure

Before introducing the system proposed and compare it with the existing system it is more helpful to study the IP packet structure and learn how different fields effect the processing of datagrams.

All IP packets or datagrams consist of a header part and a text part. The IP Header consists of a 20-byte fixed part plus a variable part. Its size is optimized to maximize the packet-processing rate without utilizing excessive resources. The header begins with a 4-bit version field that keeps track of the version of the IP protocol to which the datagram belongs. This field helps smooth the transition from one version of IP to another, which can take months or even years. All IP packets are structured the same way - an IP header followed by a variable-length data field.

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Version				IHL			TOS									Total length															
Identification																Flags		Fragment offset													
TTL							Protocol									Header checksum															
Source IP address																															
Destination IP address																															
Options and padding																															

Figure 2.3 Packet Structure

Version: The Version field indicates the format of the Internet header.

IHL: Internet Header Length is the length of the Internet header in 32 bit words, and thus points to the beginning of the data. Note that the minimum value for a correct header is 5.

Type of Service: The Type of Service provides an indication of the abstract parameters of the quality of service desired. The type of service is used to specify the treatment of the datagram during its transmission through the internet system.

Total Length: Total Length is the length of the datagram, measured in octets, including Internet header and data. This field allows the length of a datagram to be up to 65,535 octets. Such long datagrams are impractical for most hosts and networks.

Identification: An identifying value assigned by the sender to aid in assembling the fragments of a datagram.

Flags: Various Control Flags.

Fragment Offset: This field indicates where in the datagram this fragment belongs. The fragment offset is measured in units of 8 octets (64 bits). The first fragment has offset zero.

Time to Live: This field indicates the maximum time the datagram is allowed to remain in the Internet system. If this field contains the value zero, then the datagram must be destroyed.

Protocol: This field indicates the next level protocol used in the data portion of the Internet datagram.

Header Checksum: A checksum on the header only. Since some header fields change (e.g., time to live), this is recomputed and verified at each point that the Internet header is processed. The checksum field is the 16 bit one's complement of the one's complement sum of all 16-bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero.

Options: The options may appear or not in datagram's. In some environments the security option may be required in all datagram's. The option field is variable in length. There may be zero or more options.

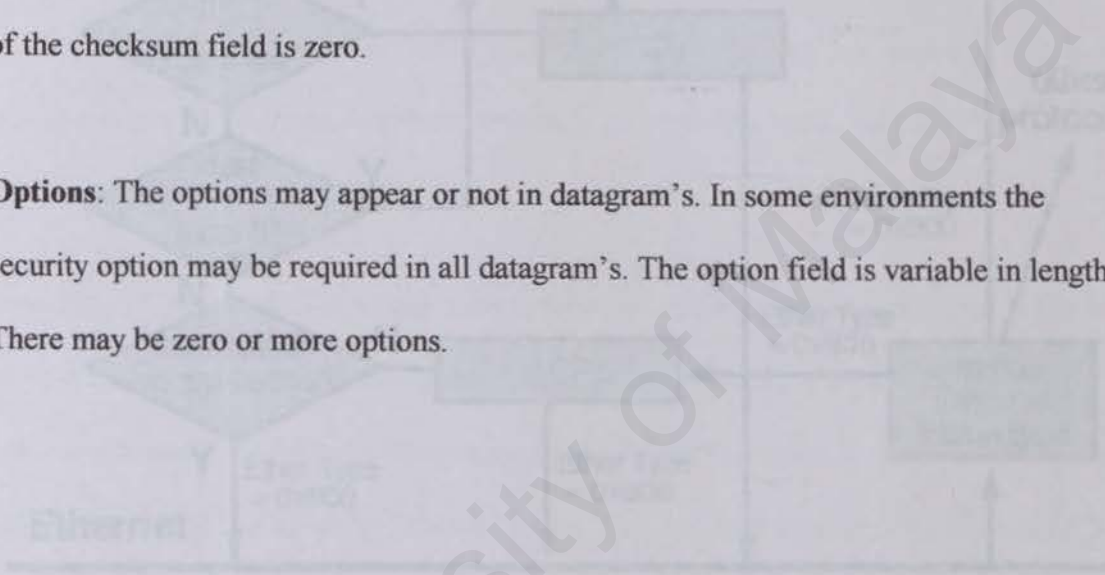


Figure 2-1 The structure of a generic Internet datagram

The IP packet is placed in an Ethernet frame as follows:

IP Header: Multi-part Address. The IP destination address is checked to see if the system should also receive a copy of the packet. This happens if this is an IP-mapped broadcast address (or a multicast address) or a local address that originates one of the registered IP addresses. If a copy is required, it is sent to the loop back

2.16 IP Packet Processing

Transmission of a frame over Ethernet

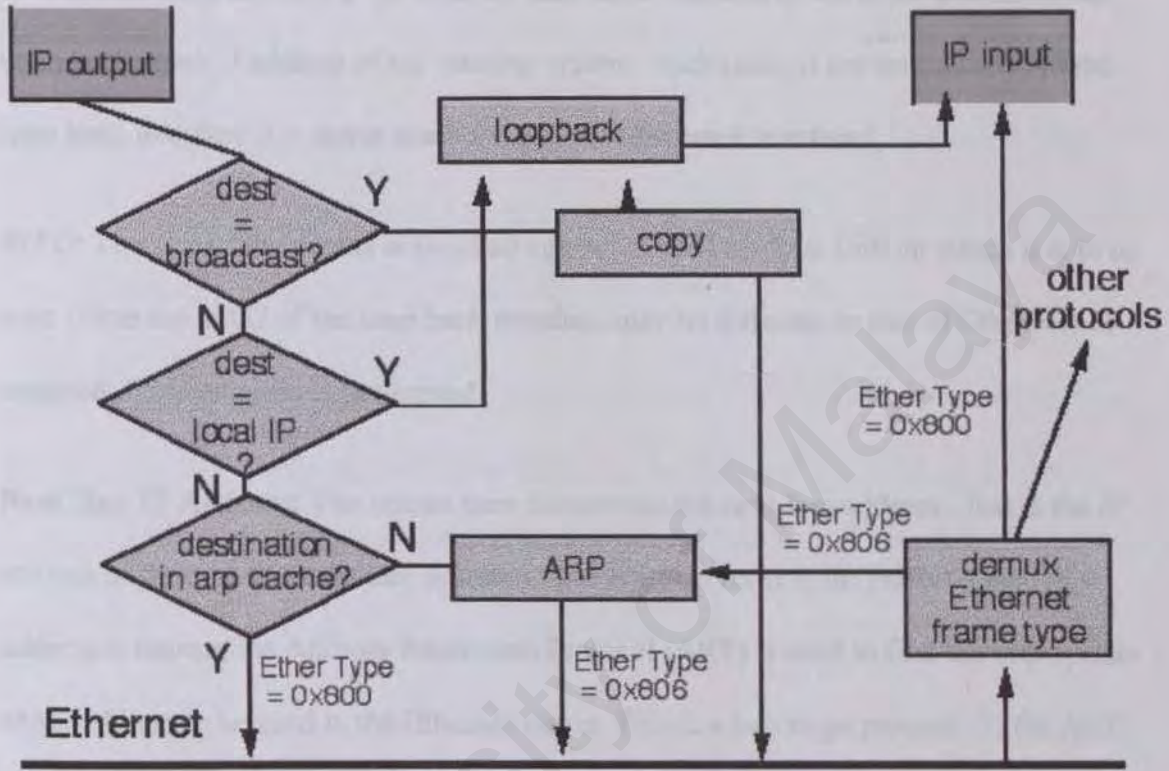


Figure 2.4 Transmission of a frame over Ethernet

The IP packet is placed in an Ethernet frames as follows:

IP Broadcast/Multicast Address: The IP destination address is checked to see if the system should also receive a copy of the packet. This happens if this is an IP network broadcast address (or a multicast address is used that matches one of the registered IP multicast filters set by the IP receiver). If a copy is required, it is sent to the loop back

interface. This directly delivers the packet to the IP input routine. The original packet continues to be processed.

IP Unicast Address: The IP destination address is checked to see if the address is the unicast (source) IP address of the sending system. Such packets are sent directly to the loop back interface (i.e. never reach the physical Ethernet interface).

MTU: The size of the packet is checked against the MTU of the link on which it is to be sent. (Note the MTU of the loop back interface may be different to that of Ethernet). If required, fragmentation is performed.

Next Hop IP Address: The sender then determines the next hop address - that is the IP address of the next Intermediate System/End System to receive the packet. Once this address is known, the Address Resolution Protocol (ARP) is used to find the appropriate MAC address to be used in the Ethernet frame. This is a two stage process: (i) the ARP cache is consulted, to see if the MAC address is already known, in which case the correct address is added and the packet queued for transmission. (ii) If the MAC address is not in the ARP cache, the ARP protocol is used to request the address, and the packet is queued until an appropriate response (or timeout) occurs.

Encapsulation: The Ethernet frame is completed, by inserting the Destination, Source and Ethernet Type fields.

Transmit: The frame is transmitted using the MAC procedure for Ethernet.

2.17 IP Fragmentation processing at a Router

To fragment/segment a long internet packet, an Intermediate System using the Internet Protocol (for example, a router), creates two new IP packets and copies the contents of the IP header fields from the long packet into BOTH new IP headers.

The data of the long packet is divided into two portions on a 8 byte (64 bit) boundary. All packets which have a more fragments (MF) flag set, must have an integral multiple of 8 bytes, but those that do not have this flag set need not do.

If we call the number of 8 byte blocks in the first portion NFB (for Number of Fragment Blocks). The first portion of the data is placed in the first new IP packet, and the total length field is set to the length of the FIRST IP packet. The more-fragments flag (MF) is set to one.

The second portion of the data is placed in the second new IP packet, and the total length field is set to the length of the SECOND packet. The more-fragments flag (MF) carries the same value as the long packet. The fragment offset field of the second new IP is set to the value of that field in the long IP packet plus the NFB.

2.18 IP Fragmentation processing at the Receiving End System

An end system that accepts an IP packet (with a destination IP address that matches its own IP source address) will also reassemble any fragmented IP packets before these are passed to the next higher protocol layer.

The system stores all received fragments (i.e., IP packets with a more-fragments flag (MF) set to one, OR where the fragment offset is non-zero), in one of a number of buffers (memory space). Packets with the same 16-bit Identification value are stored in the same buffer, at the offset specified by the fragment offset field specified in the packet header.

Packets which are incomplete remain stored in the buffer until either all fragments are received, OR a timer expires, indicating that the receiver does not expect to receive any more fragments. Completed packets are forwarded to the next higher protocol layer.

2.19 Reception of a frame from Ethernet

The following summary shows the processing performed by an end system in an IP network. It is assumed that the system is connected to an Ethernet network.

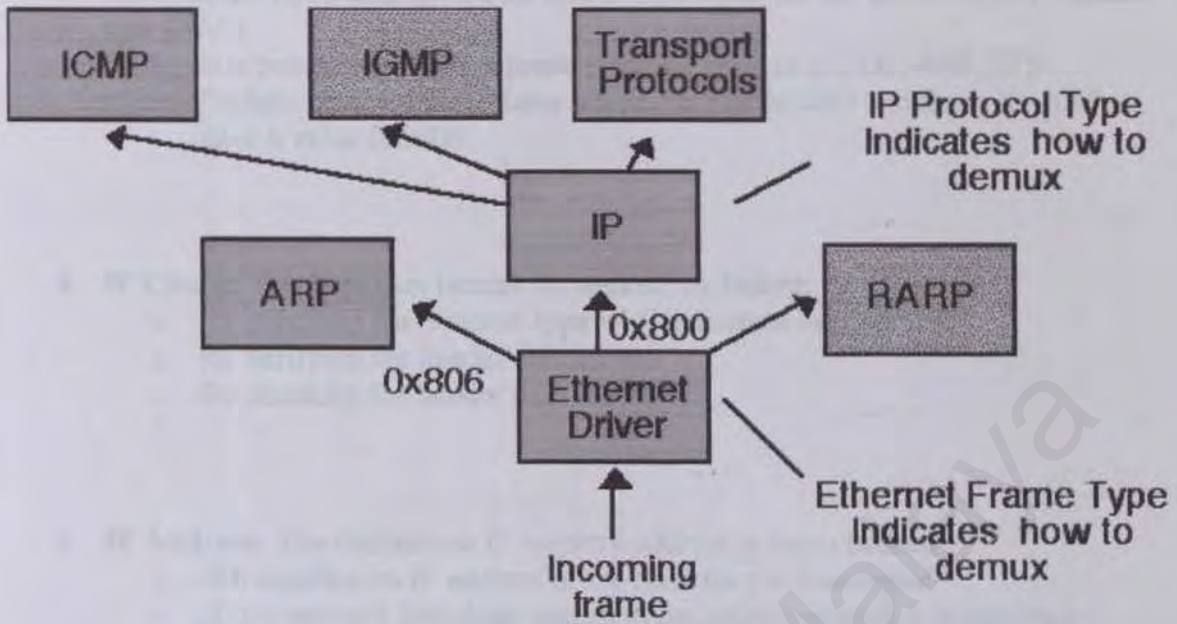


Figure 2.5 Reception of a frame from Ethernet

The received frames are processed as follows:

1. **MAC Protocol:** The Ethernet controller in the network interface card verifies that the frame is:
 - Not less than the minimum frame length not greater than the maximum length (1500 B)
 - Contains a valid CRC at the end
 - Does not contain a residue (i.e. extra bits which do not form a byte)
2. **MAC Address:** The frame is then filtered based on the MAC destination address and accepted only if:
 - It is a broadcast frame (i.e. all bits of the destination address field are set to 1)
 - It is a multicast frame to a registered MAC group address
 - It is a unicast frame to the node's own MAC address
 - Or the interface is acting in promiscuous mode (i.e. as a bridge)

3. **MAC SAP:** The frame is then demultiplexed based on the specified MAC packet type (SAP)
 - It is passed to the appropriate protocol layer (e.g. LLC, ARP, IP)
 - Packets destined for IP have a type field of 0x0800 and those for ARP have a value 0x0806
4. **IP Check:** The IP packet header is checked, including:
 - By checking the protocol type =4 (i.e. current version of IP)
 - By verifying the header checksum
 - By checking the header packet length
5. **IP Address:** The destination IP network address is then checked:
 - If it matches an IP address of the node then it is accepted
 - If it is network broadcast packet to the node's network it is accepted
 - If it is a multicast packet to an IP multicast address which is in use then it is accepted
 - If it is none of these, it is forwarded using the routing table (if possible) or discarded
6. **IP Fragmentation:** Packets for the node are then checked concerning whether reassembly is required:
 - The fragmentation offset value and more flags are inspected
 - Fragments are placed in a buffer until other fragments are received to complete the packet.
7. **IP SAP:** The IP protocol field (SAP) is checked:
 - The SAP field identifies the transport protocol (1 = ICMP; 6 = TCP; 17= UDP)
 - The complete packet is passed to the appropriate transport layer protocol.

2.20 System Comparison Architecture

2.20.1 Existing System

The diagram below shows the architecture of the existing system, which is implemented in software.

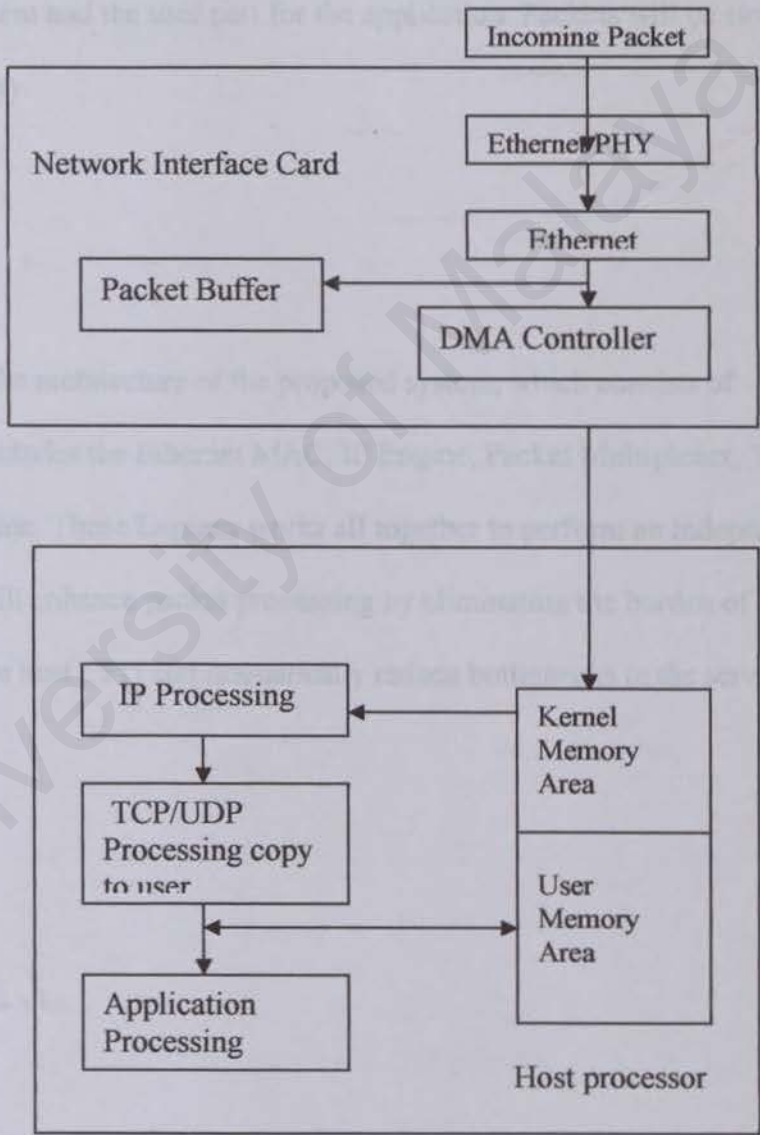


Figure 2.6 Existing System Architecture

In general, this system receives packets through the network Interface Card, process them and then store the packets in buffer before sending it to the main memory. Later on, the IP header and the TCP/UDP header will be processed by the operating system. The checksum operation will be made on the packet to make sure it's free from errors. The operating system has the memory where it is divided into two parts, which are the kernel part for the operating system and the user part for the application. Packets will be stored and retrieved from memory.

2.20.2 System Proposed

The figure below shows the architecture of the proposed system, which consists of protocol processor that includes the Ethernet MAC, IP Engine, Packet Multiplexer, TCP Engine and the UDP Engine. These Engines works all together to perform an independent protocol processor that will enhance packet processing by eliminating the burden of protocol processing on the host CPU and dramatically reduce bottlenecks in the server.

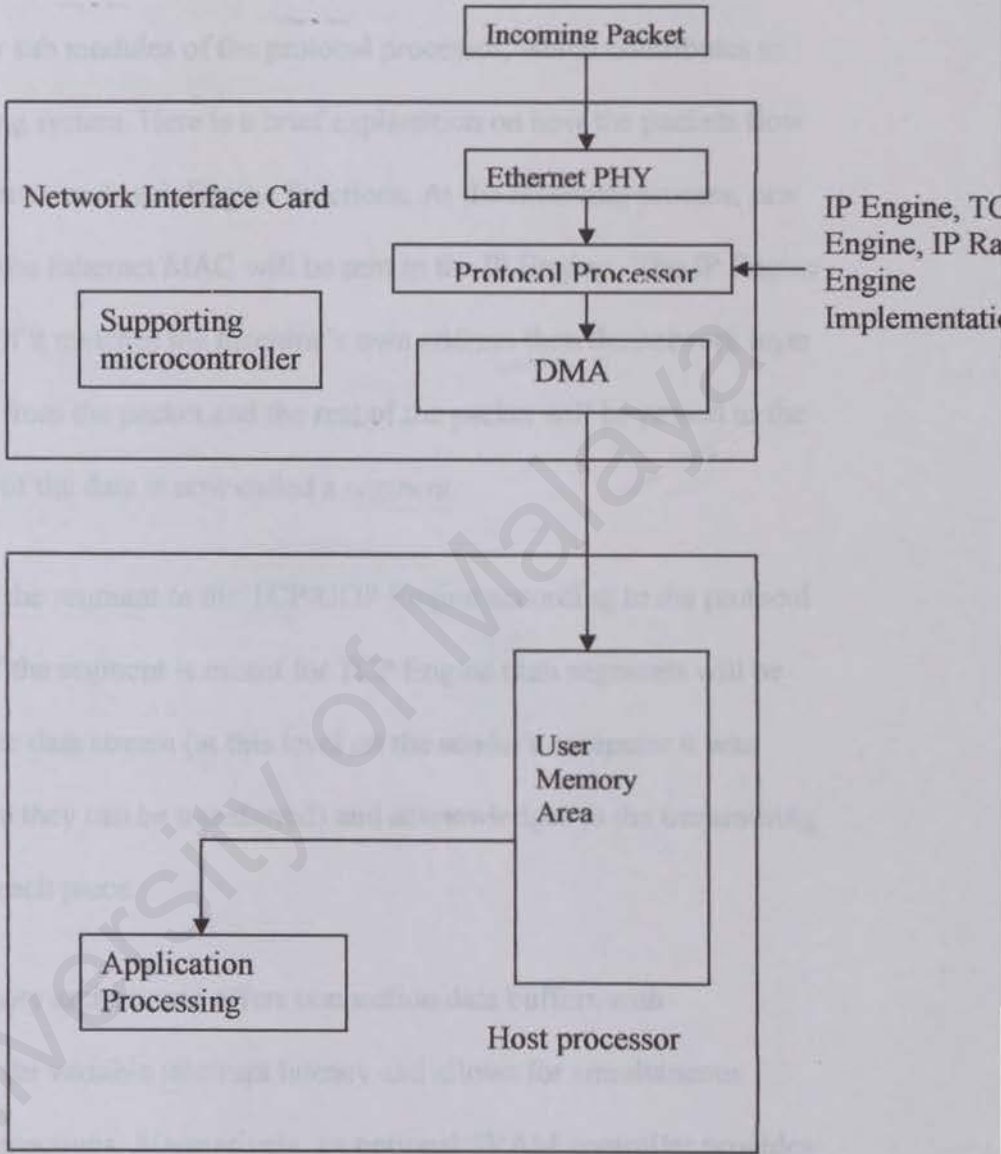


Figure 2.7 Proposed System Architecture

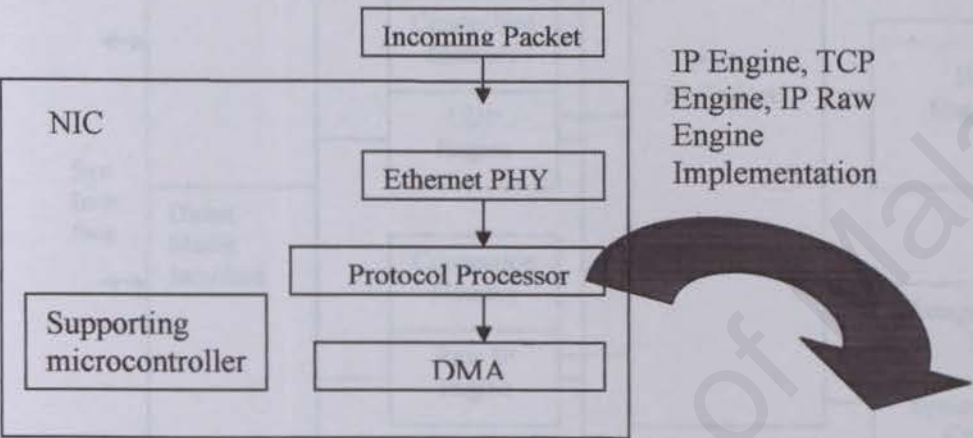
2.20.3 Protocol Processor (proposed system)

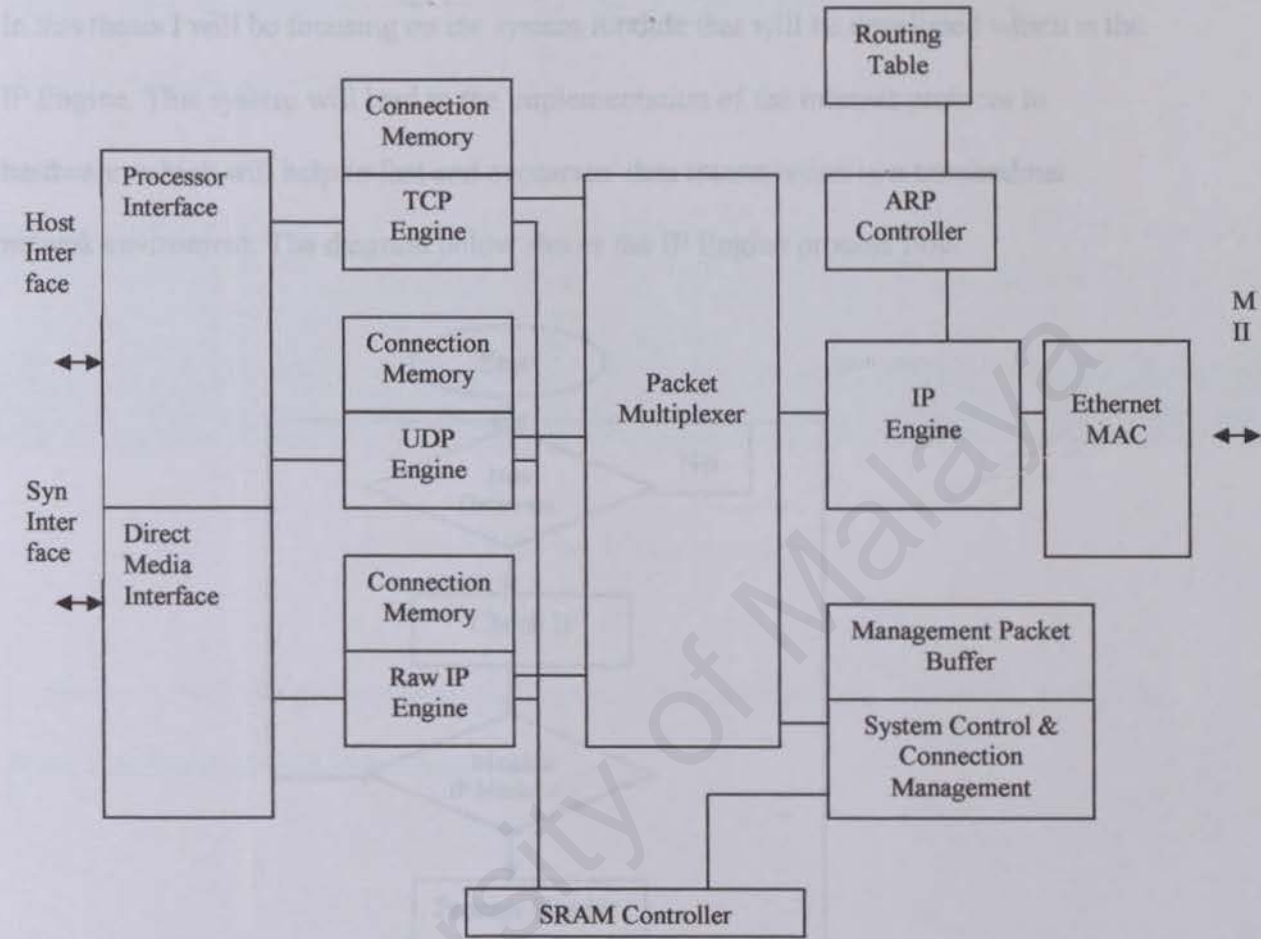
Figure 2.7 shows the inner sub modules of the protocol processor, which contributes to perform a packet processing system. Here is a brief explanation on how the packets flow through the protocol processor and each Engine functions. At the receiving process, raw data that is received from the Ethernet MAC will be sent to the IP Engine. The IP Engine will check the IP address, if it matches the machine's own address then the network layer header will be stripped off from the packet and the rest of the packet will be passed to the multiplexer. Here the rest of the data is now called a *segment*.

The multiplexer will send the segment to the TCP/UDP Engine according to the protocol addressed in the header. If the segment is meant for TCP Engine then segments will be processed by rebuilding the data stream (at this level on the sender's computer it was actually split into pieces so they can be transferred) and acknowledges to the transmitting computer that it received each piece.

Scalable, distributed memory architecture offers connection data buffers with programmable thresholds for variable interrupt latency and allows for simultaneous operation of up to 64k connections. Alternatively, an optional SRAM controller provides a high-speed memory interface for single memory architectures. IP connection set up and management is made easy by a message-based interface of the comprehensive control unit, which fully supports IP management protocols and allows for system configuration.

Its auto-configuration and remote management capability enable it to provide IP connectivity even without any external processor interaction, which makes it an ideal communication extension for existing applications.





2.20.4 IP Engine

In this thesis I will be focusing on the system module that will be developed which is the IP Engine. This system will lead to the implementation of the internet protocol in hardware, which will help in fast and accurate data transmission in a tremendous network environment. The diagram below shows the IP Engine process flow.

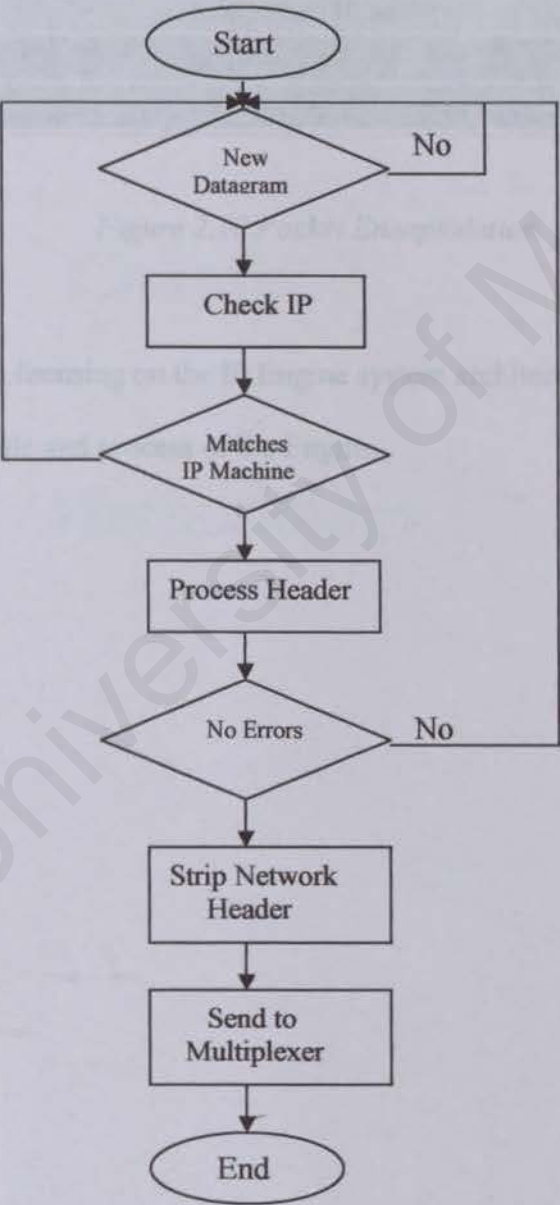


Figure 2.9 IP Engine Process Flow

Figure 2.10 shows the minimal header encapsulation where the network header is stripped out of the packet and the rest of the header is passed to the multiplexer. This process made at the IP Engine.

01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
Protocol							S	Reserved							Header checksum																
Destination IP address																															
Original Source IP address																															
Data																															

Figure 2.10 Packet Encapsulation

In chapter 5 i will be focusing on the IP Engine system architecture, which will explain in detail each sub module and process of the Engine.

2. Methodology

This chapter illustrates the methodology that has been used in this project and the advantages of the implemented architecture in developing the IP design. The methodology used in the hardware description language is VHDL.

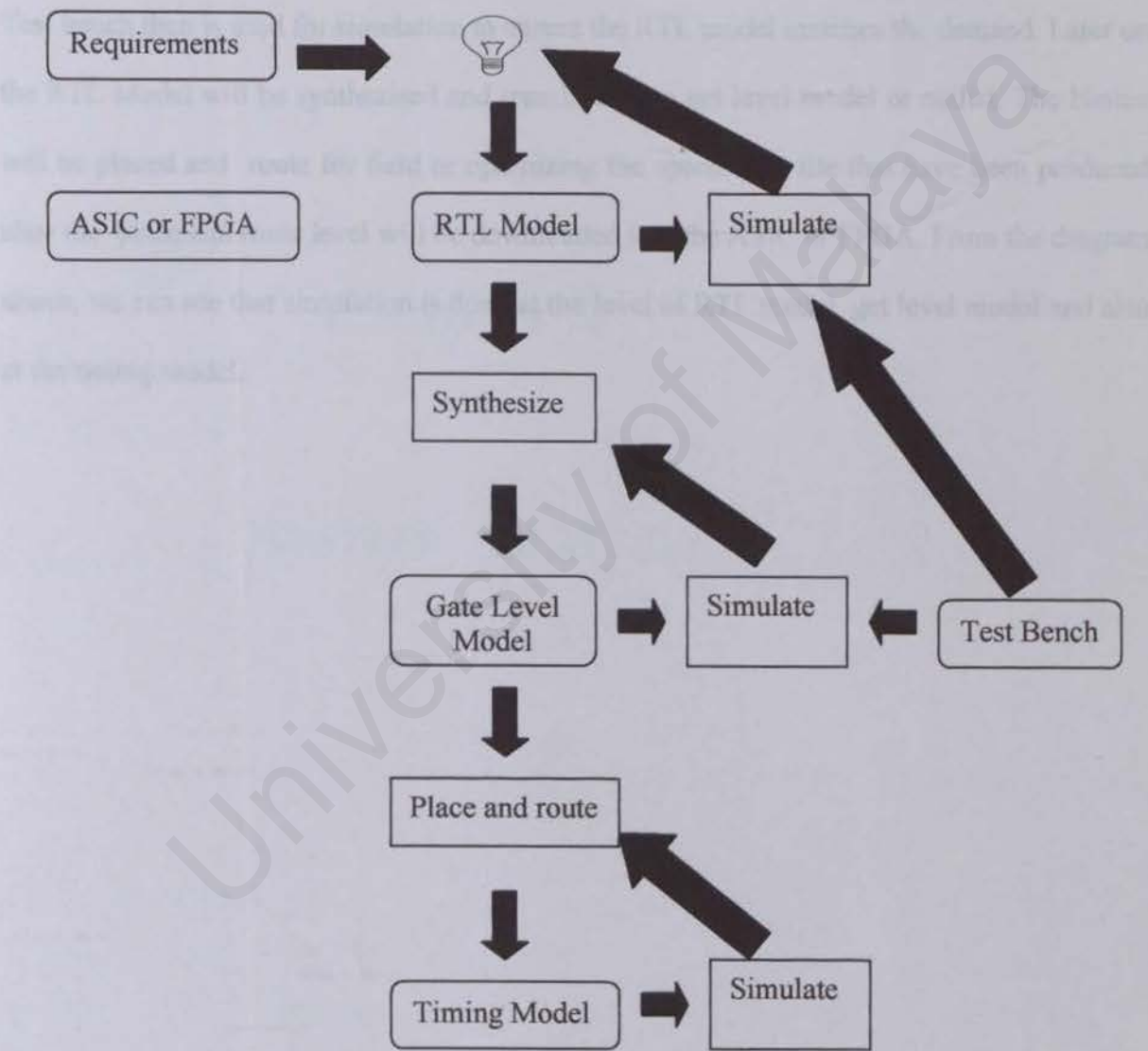


Chapter 3

Methodology

3.1 Methodology

This chapter illustrate the methodology that have been used in this project and the advantages of the implemented architecture in developing the IP Engine. The methodology used is the hardware description language is VHDL.



The diagram above shows the basic VHDL methodology process architecture beginning by inserting the architecture into the FPGA or ASIC. Firstly, we have to think of the demand of each architecture. According to the demand we can develop the RTL model and the test bench using VHDL.

Test bench then is used for simulation to ensure the RTL model matches the demand. Later on the RTL Model will be synthesized and translated into gate level model or netlist. The Netlist will be placed and routed for field or optimizing the speed. The file that have been produced after the place and route level will be downloaded into the ASIC or FPGA. From the diagram above, we can see that simulation is done at the level of RTL model, gate level model and also at the timing model.

4.1 VHDL

The hardware description language will be used in this project is VHDL, hardware description language. Firstly, it is increasingly being used for description and modeling of digital system, leading to use of this language for system design field especially robotic and microprocessors development. It is useful in describing hardware for the purpose of simulation, modeling, verifying design and documentation. It provides a convenient and compact format for the hierarchical representation of functional and wiring detail of digital systems.

4.2 What is VHDL



Chapter 4

System Analysis

The VHDL language provides an integrated and generic of the following languages:

- Sequential language
- Concurrent language
- Text-based language
- Wave generation language

Therefore, the language has convenient that enable the user to express the concept of sequential behavior of a digital system with or without timing. It also allows user to model

4.1 VHDL

The hardware description language will be used in this project is VHDL hardware description language. Recently, it is increasingly being used for description and modeling of digital system, leading to use of this language for system design field especially robotics and microprocessors development. It is useful in describing hardware for the purpose of simulation, modeling, testing design and documentation. It provides a convenient and compact format for the hierarchical representation of functional and writing detail of digital systems.

4.2 What is VHDL

VHDL is an acronym for VHSIC Hardware Description Language. VHSIC is an acronym for very high-speed integrated circuit. It is hardware description language that can be used to model a digital system at many levels of abstraction, ranging from the algorithmic level to the logic level. The complexity of the digital system being modeled could vary from that of a simple gate to a complete digital electronic system, or anything in between. The digital system can also be explicitly modeled in the same description.

The VHDL language can be regarded as an integrated amalgamation of the following languages:

- Sequential language
- Concurrent language
- Next-list language
- Wave generation language

Therefore, the language has construct that enable the user to express the concurrent or sequential behavior of a digital system with or without timing. It also allows users to model

the system as an interconnection of components. Test waveforms can also be generated using the same constructs. All the above constructs may be combined to provide a comprehensive description of the system in a single model.

The language not only defines the syntax but also vary clear simulation semantics for each language construct. Therefore, models written in this language can be verified using a VHDL simulator. It is a strongly typed language and is often verbose to write. It inherits many its features, rop-daily the sequential language part from the ADA programming language. Because VHDL provides an extensive range of modeling capabilities, it is often difficult to understand. Fortunately, it is possible to quickly assimilate a core subset of the language that is both easy and simple to understand without learning the more complex features. This subset is usually sufficient to model most applications. However the complete language has sufficient power to capture the description of the most complex chips to a complete electronic system.

4.3 VHDL Advantages

VHDL offers the following advantages for digital design:

- **Standard:** VHDL is an IEEE standard. Just like any standards (such as graphic x-window standard, bus communication interface standard, high level programming languages and so on), it reduces confusion and makes interfaces between tools, companies and products easier. Any development to the standard would have better chances of lasting longer and have less chance of becoming obsolete due to incompatibility with others.
- **Industry support:** With the advent of more powerful and efficient VHDL tools has come the growing support of the electronic industry. Companies use VHDL tools not only with regard to defense contracts, but also for their commercial designs.
- **Portability:** The same VHDL code can be simulated and used in many design tools whose limited capability may not be competitive in later markets. The VHDL standard also transforms design data much easier than a design database of a proprietary design tools.
- **Modeling Capability:** VHDL was developed to model all levels of designs, from electronic boxes to transistors. VHDL can accommodate behavioral constructs and mathematical routines that describe complex model, such as queuing networks and analog circuits. It allows the use of multiple architectures and associated with the same design during various stages of the design process.

- **Reusability:** Certain common designs can be described, verified and modified slightly in VHDL for future use. This eliminates reading and marking changes to schematic pages, which is time consuming and subject to error. For example, a parameterized multiplier VHDL code can be reused easily by changing the width parameter so that the same code can do either 16 by 16 or 12 by 8 multiplication.
- **Technology and Foundry Independence:** The functionality and behavior of the design can be described with VHDL and verified, making it foundry and technology independent. This frees the designer to proceed without having to wait for the foundry and technology to be selected.
- **Documentation:** VHDL is a design description language, which allows documentation to be located in a single place by embedding it in the code. The combining of comments and the code that actually dictates what the design should do to reduce the ambiguity between specification and implementation.
- **New Design Methodology:** Using VHDL and synthesis creates a new methodology that increases the design productivity, shortens the design cycle and lower costs. It amounts to a revolution comparable to that introduced by the automatic semi-custom layout synthesis tools of the last few years.

4.4 VHDL and Verilog comparison

This section compares and contrasts the individual aspects of both languages the VHDL and the Verilog.

i. Capability

Hardware structure can be modeled equally effectively in both VHDL and Verilog. When modeling abstract hardware, the capability of VHDL can sometimes only be achieved in Verilog when using the PLI. The choice of which to use is not therefore based solely on technical capability but on:

- Personal preferences
- EDA tool availability
- Commercial, business and marketing issues

The modeling constructs of VHDL and Verilog cover a slightly different spectrum across the levels of behavioral abstraction; see Figure 1.

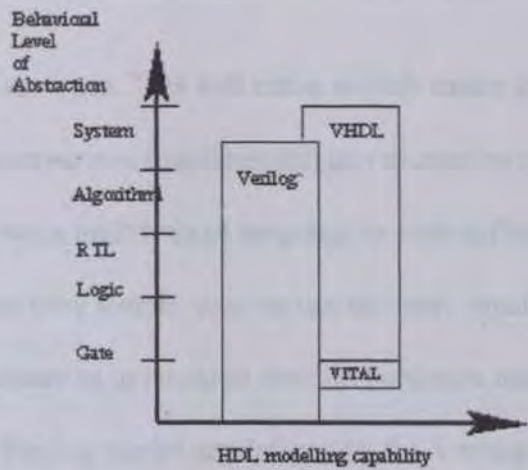


Figure 1. HDL modeling capability

ii. **Compilation**

VHDL multiple design-units (entity/architecture pairs), that reside in the same system file, may be separately compiled if so desired. However, it is good design practice to keep each design unit in it's own system file in which case separate compilation should not be an issue. The Verilog language is still rooted in its native interpretative mode. Compilation is a means of speeding up simulation, but has not changed the original nature of the language. As a result care must be taken with both the compilation order of code written in a single file and the compilation order of multiple files. Simulation results can change by simply changing the order of compilation.

iii. **Data types**

In VHDL a multitude of language or user defined data types can be used. This means dedicated conversion functions are needed to convert objects from one type to another. The choice of which data types to use should be considered wisely, especially

enumerated (abstract) data types. This will make models easier to write, clearer to read and avoid unnecessary conversion functions that can clutter the code. VHDL may be preferred because it allows a multitude of language or user defined data types to be used. In Verilog data types are very simple, easy to use and very much geared towards modeling hardware structure as opposed to abstract hardware modeling. Unlike VHDL, all data types used in a Verilog model are defined by the Verilog language and not by the user.

iv. Design reusability

VHDL procedures and functions may be placed in a package so that they are available to any design-unit that wishes to use them.

There is no concept of packages in Verilog. Functions and procedures used within a model must be defined in the module. To make functions and procedures generally accessible from different module statements the functions and procedures must be placed in a separate system file and included using the include compiler directive.

v. High level constructs

There are more constructs and features for high-level modeling in VHDL than there is in Verilog. Abstract data types can be used along with the following statements:

- Package statements for model reuse
- Configuration statements for configuring design structure
- Generate statements for replicating structure
- Generic statements for generic models that can be individually characterized, for example, bit width.

All these languages statements are useful in synthesizable models.

Except verilog for being able to parameterize models by overloading parameter constants, there is no equivalent to the high-level VHDL modeling statements in Verilog.

5.1 System Design

The system module which will be developed is the IP Engine. This system will lead to the implementation of the system proposed in hardware, which will help in fast and accurate design implementation in a streamlined manner. This chapter will explain in detail each process of the IP Engine, by illustrating the system design, flow charts and block diagrams that will ease the understanding of the system and its functions.



Chapter 5

System Design

5.1 System Design

The system module which will be developed is the IP Engine. This system will lead to the implementation of the internet protocol in hardware, which will help in fast and accurate data transmission in a tremendous network environment. This chapter will explain in detail each process of the IP Engine by illustrating the system design, flow charts and block diagrams that will ease the understanding of the system and module functions.

The IP Engine internal block diagram is designed at the part one of this thesis and will be developed at the second part of the thesis using the peakFPGA software. Each sub process module will be developed and then integrated to perform the whole Engine as an Internet Protocol Engine which will process packets independently. The diagram below shows the Engine Black Box by explaining each in and out pin, followed by the internal Black Box of the IP Engine describing the sub process modules and the signals used to activate each process.

5.2 IP Engine Block Diagram

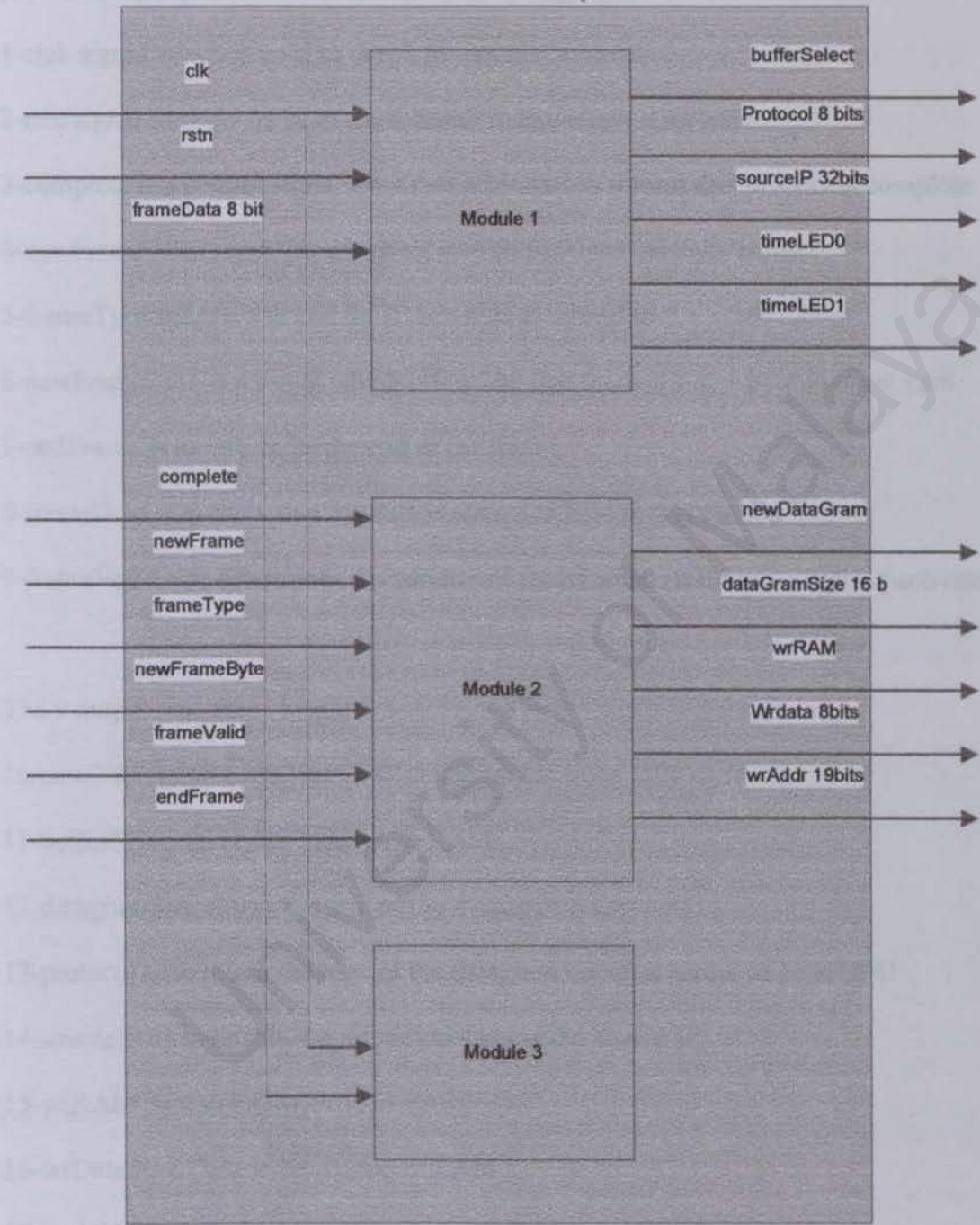


Figure 5.1 IP Engine Block Diagram

Figure 5.1 shows the IP Engine Block Diagram that consist of 9 input pins and 9 output pins. The input pins are :

- 1-clck signal which is used to check for positive transition.
- 2-rstn signal to show its in an asynchronous active low reset mode.
- 3-complete is a control signal from ram arbitrator to inform that process is complete.
- 4-newFrame is an input pin which recieve frame from the layer below
- 5-frameType tells its a frame for IP when its activated to 1.
- 6-newFrameByte is a signal which informing that there is a new byte in the stream.
- 7-endFrame is to inform its the end of the frame.
- 8-frameData is to show that the data is streamed here in this frame.
- 9-frameValid will determines the validity of frame when endframe is high or activated.

The 9 output pins are:

- 10-newDatagram shows that an IP datagram has been fully recieved.
- 11-bufferSelect indicates location in RAM.
- 12-datagramSize shows the size of the datagram is recieved.
- 13-protocol determines the type of the datagram which is meant to TCP/UDP
- 14-sourceIP its to lets the upper protocol know the source IP.
- 15-wrRAM its a signal to write to RAM.
- 16-wrData its a Data to be written in RAM.
- 17-wrAddr to send th address lines to be written in RAM.
- 18-timeLED0/LED1 indicates if buffer 0/1 is busy.

5.3 Internal Block diagram

The diagram below shows the internal block diagram of the IP Engine sub modules which illustrate the sub process of the Engine. This design is for the process of recieving a packet from the Ethernet and processing it, then passing it to the upper layer of the TCP/IP stack.

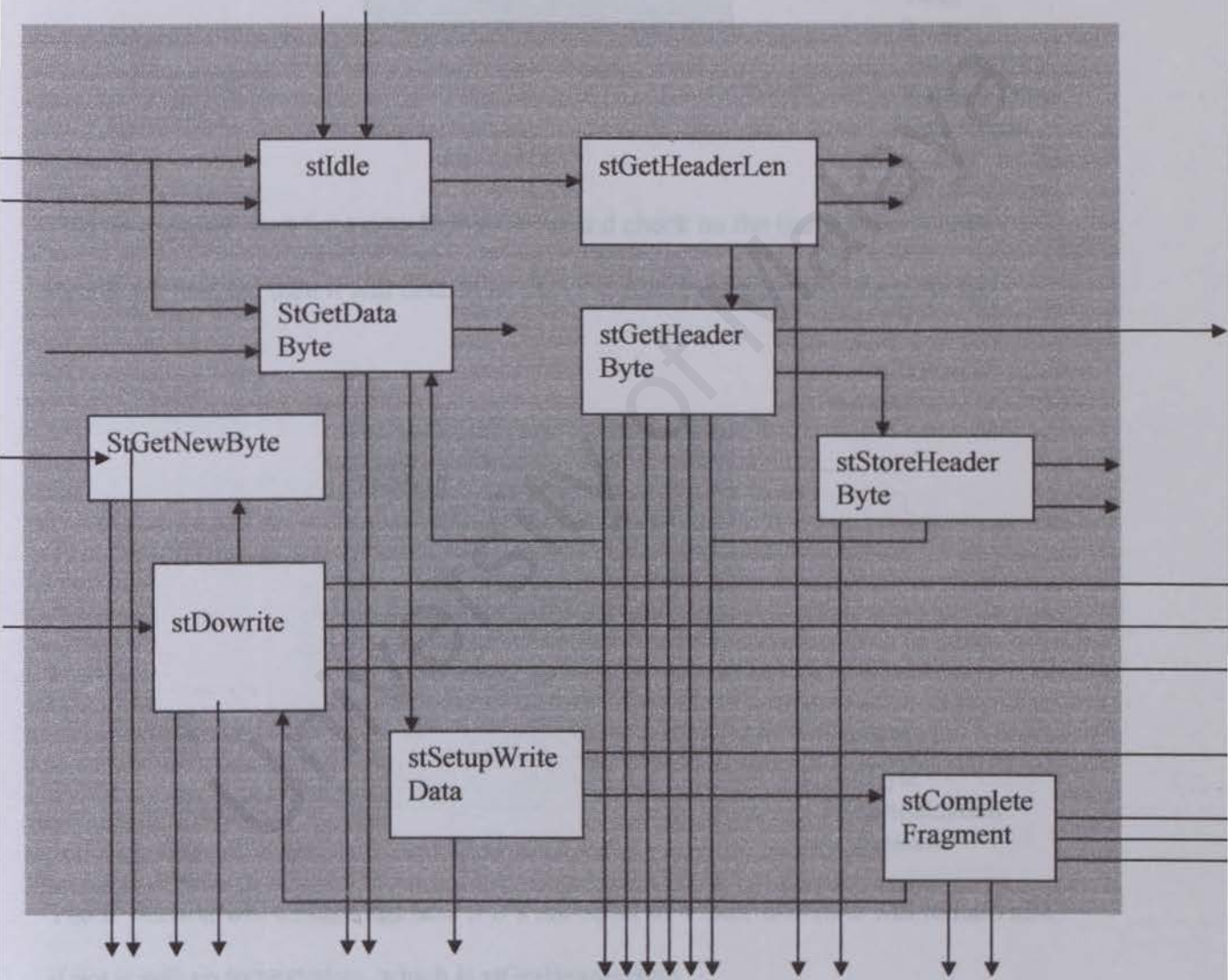
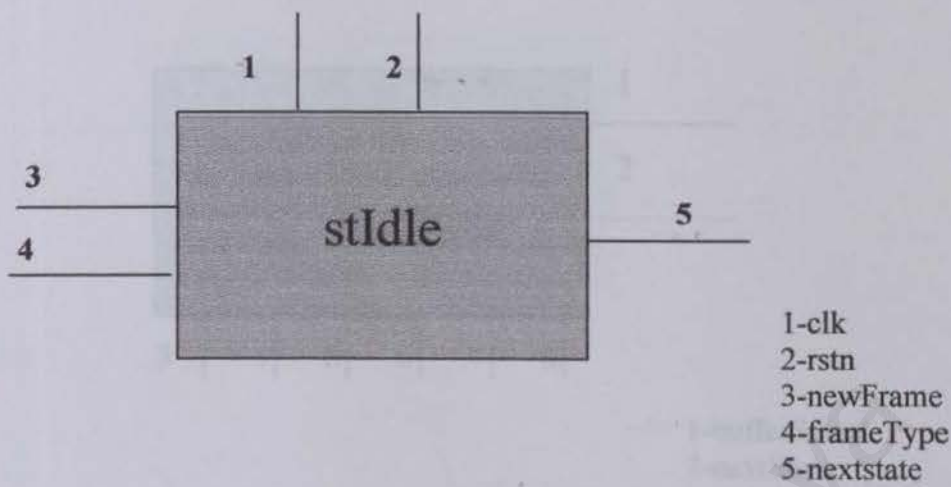
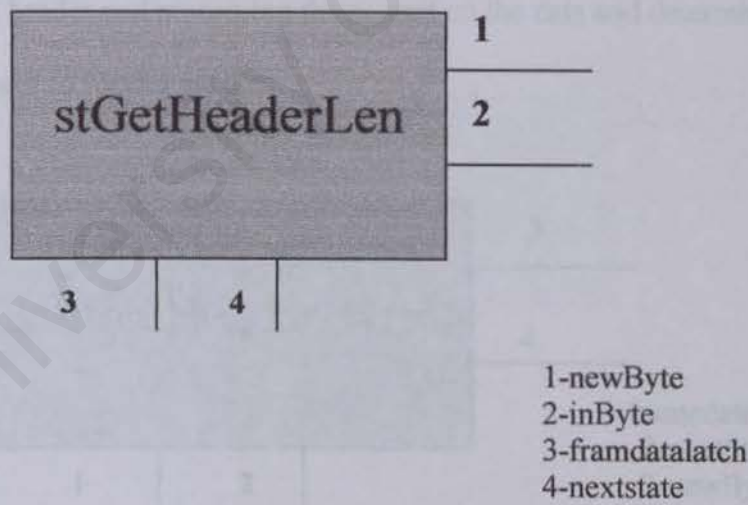


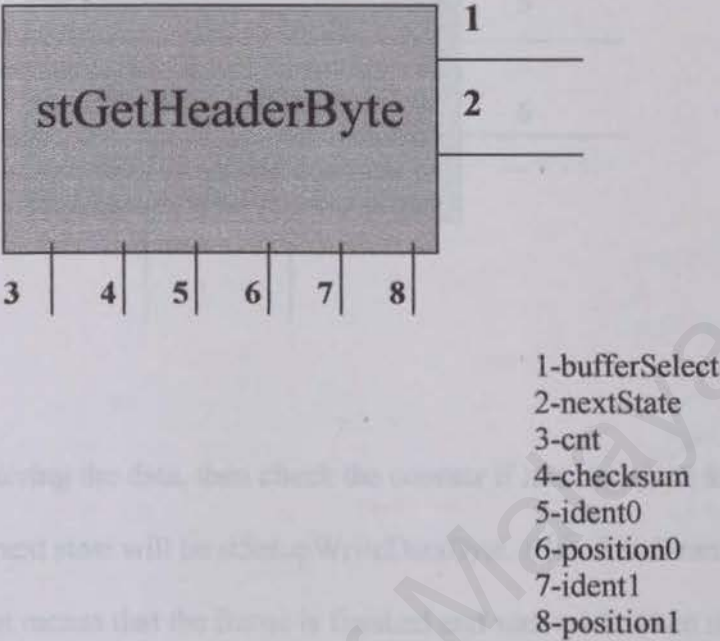
Figure 5.2 IP Engine internal Block Diagram



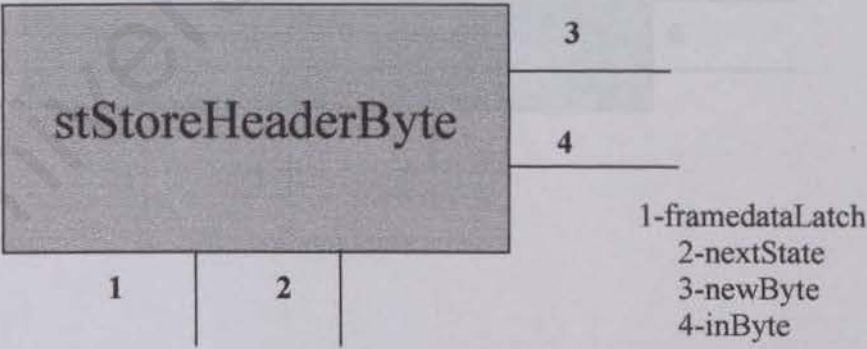
This process will wait for a new frame arrival and check on the frame type. If both signals are enabled then it will determine the next state by getting the header length.



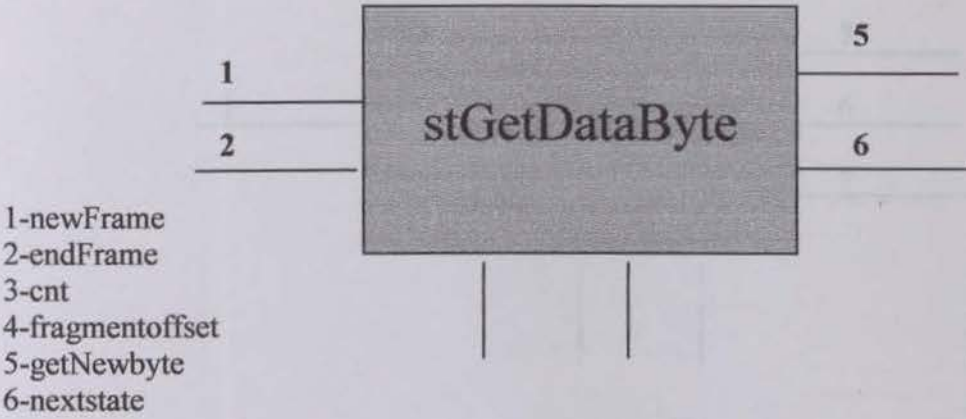
The IP version will be checked here if it's not equal to 4 then next state will remain idle, if not it will go to next state, which is stGetHeaderByte.



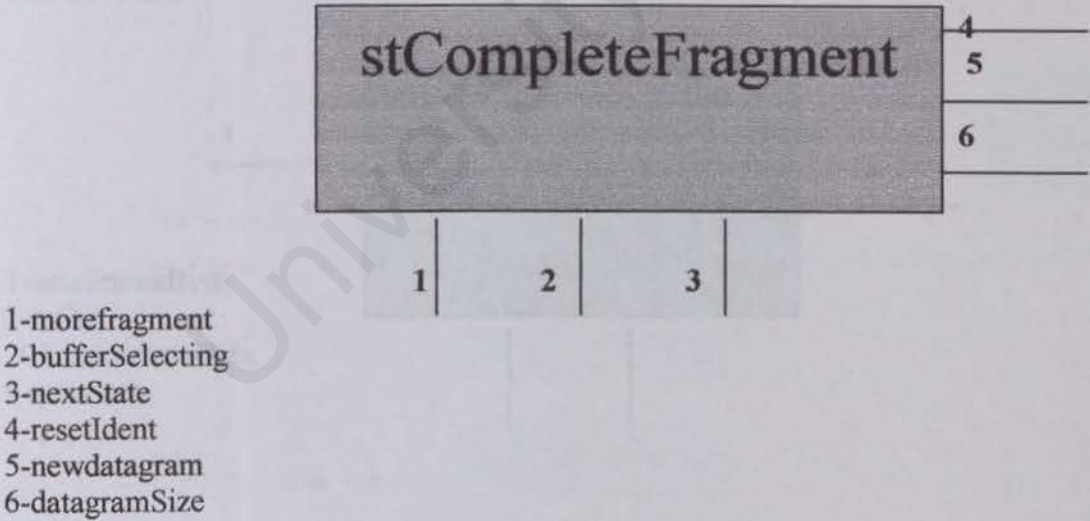
If we finished getting the header and processing them, start on the data and determine which buffer should be used to handle the data.



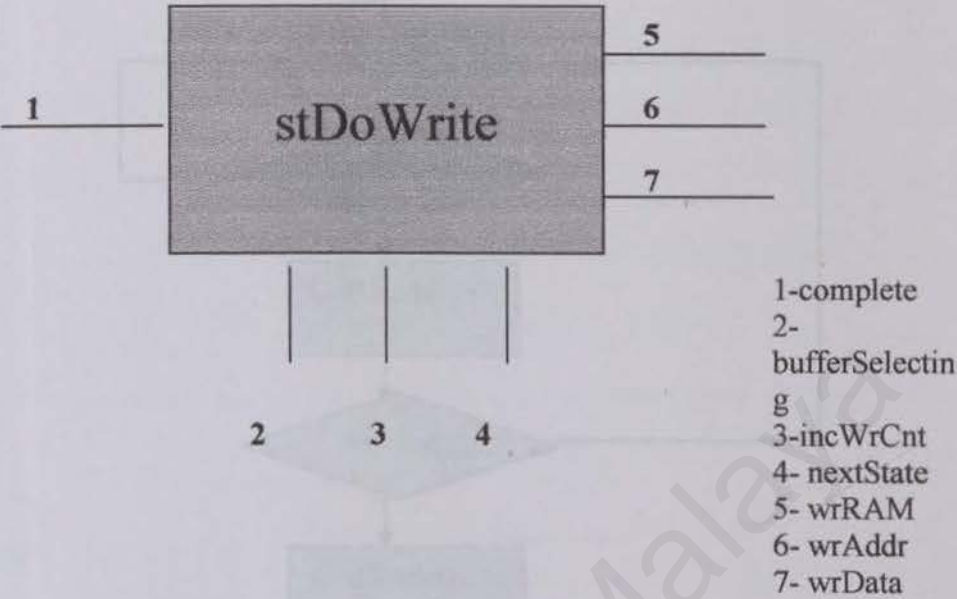
Operate on each value of the header received according to count and store the header in RAM.



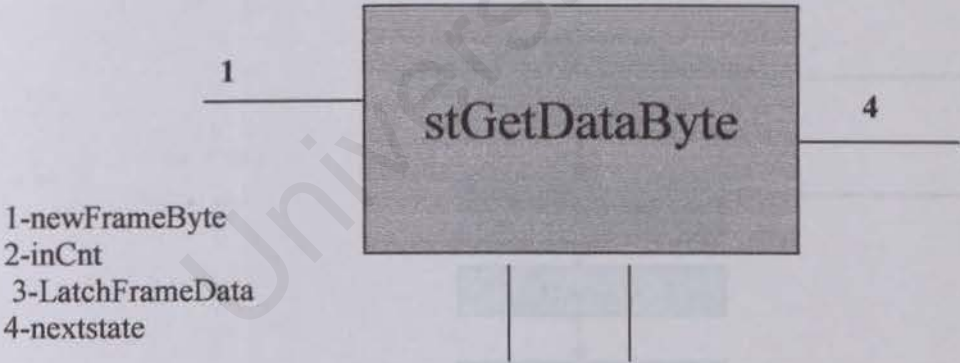
If we haven't finished receiving the data, then check the counter if it is not equal to datagram length then the next state will be stSetupWriteDataByte. Else if endFrame and frameValid is enabled, that means that the frame is finished and was valid. Then it will go to the next state which is stSetupwriteData.



At this state a signal will be sent to the transport layer informing that the datagram is finished or await of a next frame.



At this state we wait for RAM write request to be serviced if complete, then we can write data to RAM.



The last process is to check if there is newFameByte. If yes then wait for it to arrive.

5.4 Process Flow

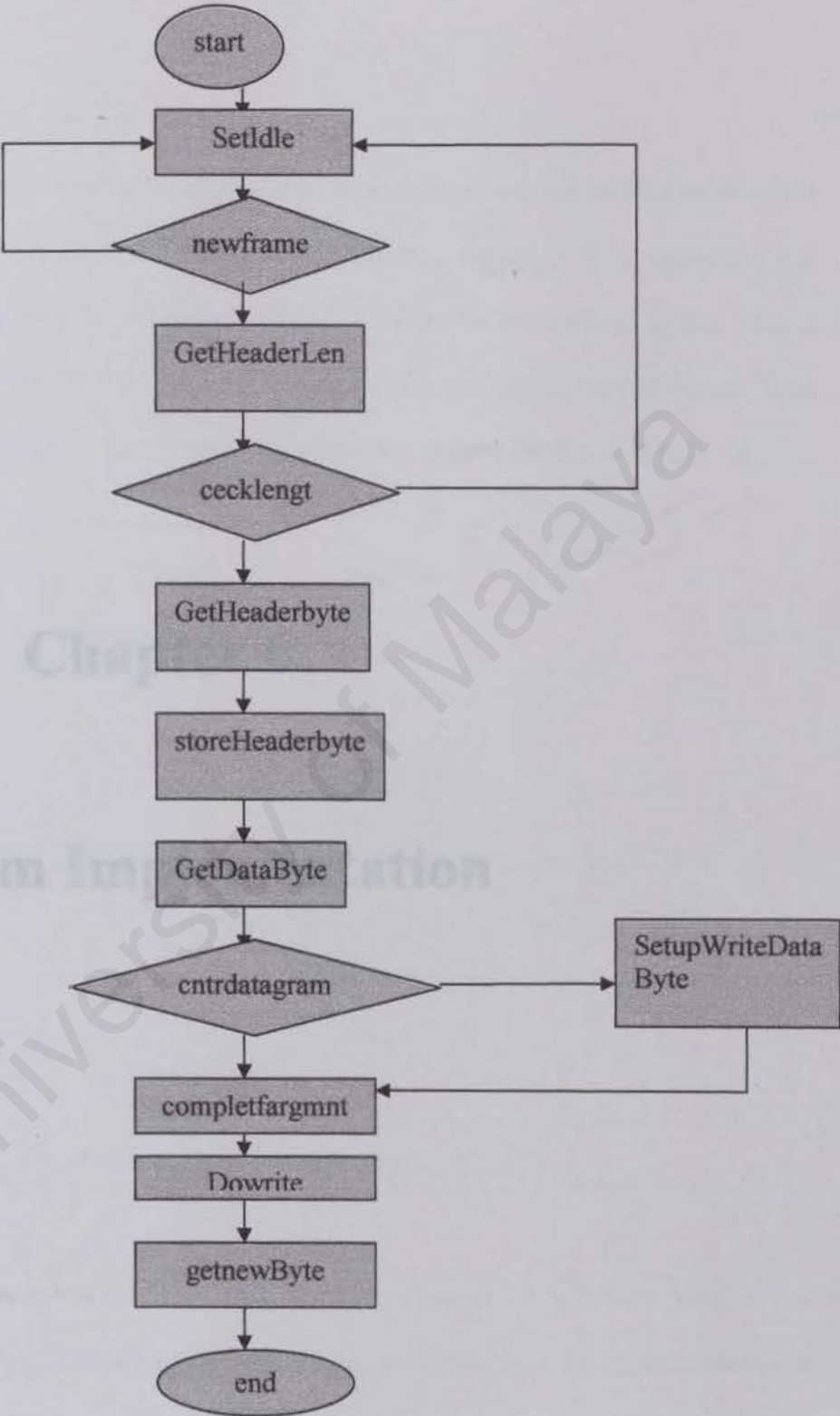


Figure 5.3 Process Flow

6.1 Introduction

After the design of the architecture has been fixed, the implementation of the IP Engine was then being started. Figure 6.1 shows a simplified design process including both synthesis and simulation for one or more programmable logic devices. The key for understanding the process and its fundamental laws is to use VHDL, to know the importance of test development. Test development should begin as soon as the general requirements of the system are known.



Chapter 6

System Implementation

VHDL is used for design entry. After being captured into a design entry system using a text editor, the VHDL source code must be passed directly to synthesis tools for implementation in a specified type of device. The design is simulated, allowing it to be functionally verified.

6.1 Introduction

After the design of the architecture has been fixed, the implementation of the IP Engine was then bieng started. Figure 6.1 shows a simplified design process including both synthesis and simulation for one or more programmable logic design. The key for understanding this process and to understand how to use VHDL, is to know the importance of test development. Test development should begin as soon as the general requirements of the system are known.

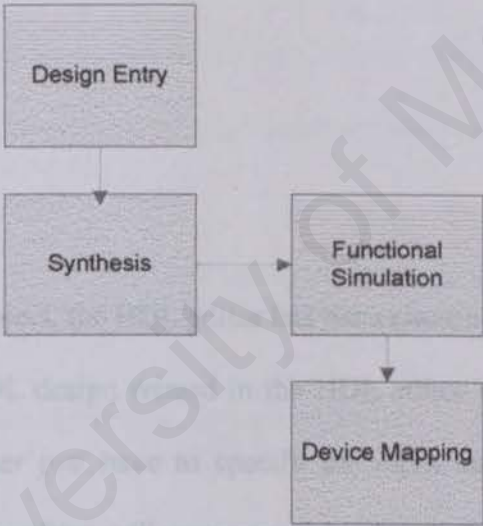


Figure 6.1 Steps In Implementing A VHDL Module

VHDL is used for design entry. After bieng captured into a design entry system using a text editor, the VHDL source code must be passed diresctly to synthesis tools for implementation in a specified type of device. The its input to simulation, allowing it to be functionally verified.

On the test development side, script files or VHDL test benches can be created to exercise the circuit to verify that it meets the functional and timing constraints of the specification. These script files may be entered using a text editor, or may be generated from other forms of test stimulus information such as graphical waveforms.

6.2 Design Entry

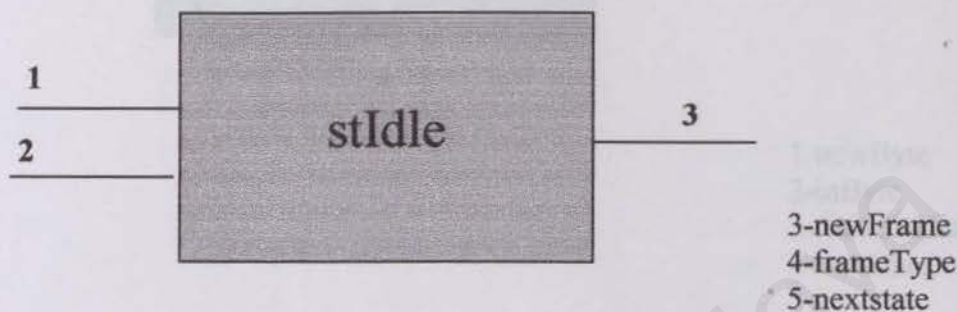
An external editor has been used for the design entry. Xilinx Foundation offers three types of design entry editor for the preferences of the user. Three of these editors are listed below:

- HDL editor
- FSM editor
- Schematic Editor

In this IP Engine modeling project, the HDL editor has been chosen as the design entry editor for all the entities. There is a HDL design wizard in the HDL editor to ease the users to code the ports of every entity. The user just has to specify the input and output ports of the entity following the wizard, then the editor will generate the entity declaration codes. After that, the users have to code the architecture of the entity that has been created using VHDL whether in behavioral or structural form. Besides, the VHDL editor also has a synthesis tool for syntax checking of the written codes.

6.3 Modeling Entity

Here is the behavior description of each entity is listed below:

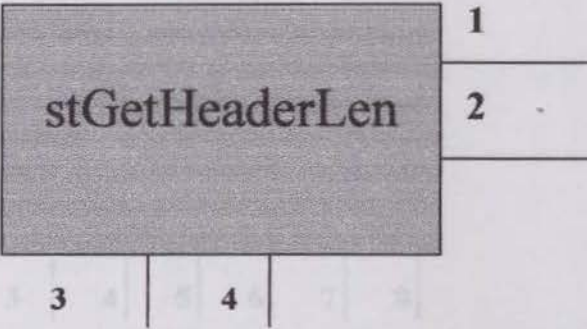


This process will wait for a new frame arrival and check on the frame type. If both signals are enabled then it will determine the next state by getting the header length.

when stIdle =>

```

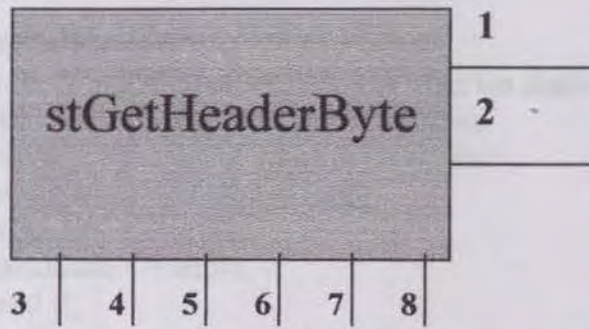
-- wait for the arrival of a new frame that has a frameType of 1
if newFrame = '0' or frameType = '0' then
    nextState <= stIdle;
else
    -- reset the counters for the next datagram
    rstCnt <= '1';
    rstWrCnt <= '1';
    newHeader <= '1';
    nextState <= stGetHeaderLen;
    -- get header length and version information
    getNewByte <= '1';
end if;
  
```



- 1-newByte
- 2-inByte
- 3-framdatalatch
- 4-nextstate

The IP version will be checked here if it's not equal to 4 then next state will remain idle, if not it will go to next state, which is stGetHeaderByte.

```
when stGetHeaderLen =>
    -- check ip version
    if frameDataLatch (7 downto 4) /= 4 then
        nextState <= stIdle;
    else
        nextState <= stGetHeaderByte;
        -- send data to checksum machine
        inByte <= frameDataLatch;
        newByte <= '1';
        -- get the header length in bytes, rather than 32-bit words
        nextHeaderLen <= frameDataLatch (3 downto 0) & "00";
    end if;
```

- 1-bufferSelect
- 2-nextState
- 3-cnt
- 4-checksum
- 5-ident0
- 6-position0
- 7-ident1
- 8-position1

If we finished getting the header and processing them, start on the data and determine which buffer should be used to handle the data.

when stGetHeaderByte =>

```
-- if we've finished getting the headers and processing them, start on the data
-- once finished, refragmenting will come next
if cnt = headerLen then
    -- only operate on data meant for us, or broadcast data
    if checksum = 0 then
        -- determine which buffer should be used to handle the data
        if ident0 = targetIdent and timeout0 /= FULLTIME then
            -- the ident matches and the timeout counter has not expired
            nextBufferSelect <= '0';
            -- accept the frame if its offset matches what we think it should be
            -- this drops out of order and duplicate frames.
            if position0 = fragmentOffset & "000" then
                nextState <= stGetDataByte;
            else
                nextState <= stIdle;
            end if;
        end if;
    end if;
end if;
```

```

elseif ident1 = targetIdent and timeout1 /= FULLTIME then
    -- the ident matches and the timeout counter has not expired

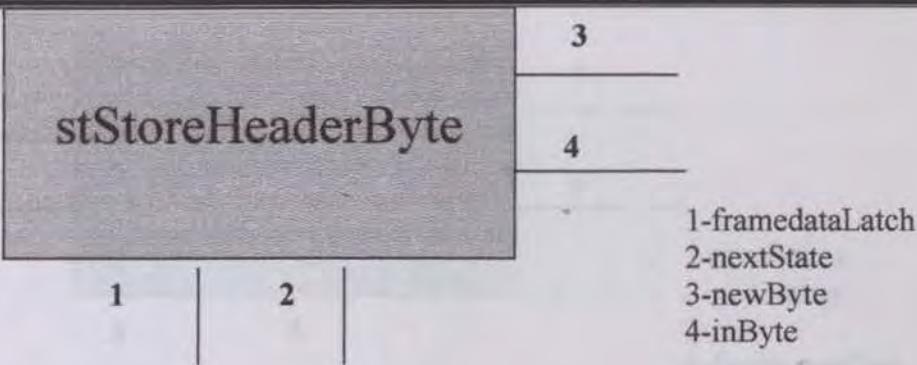
    nextBufferSelect <= '1';
    -- accept the frame if its offset matches what we think it should be
    -- this drops out of order and duplicate frames.
    if position1 = fragmentOffset & "000" then
        nextState <= stGetDataByte;
    else
        nextState <= stIdle;
    end if;
elseif (ident0 = 0 or timeout0 = FULLTIME) and fragmentOffset = 0 then

    -- The ident doesn't match either of the buffers so check if buffer 0
    -- is free. If ident = 0 or the timeout has expired then the buffer is free
    -- This must be the first fragment if it is to go here so also check the offset
    nextState <= stGetDataByte;
    nextBufferSelect <= '0';

elseif (ident1 = 0 or timeout1 = FULLTIME) and fragmentOffset = 0 then
    -- The ident doesn't match either of the buffers so check if buffer 1
    -- is free. If ident = 0 or the timeout has expired then the buffer is free
    -- This must be the first fragment if it is to go here so also check the offset
    nextState <= stGetDataByte;
    nextBufferSelect <= '1';
    else
        nextState <= stIdle;
    end if;
else
    -- ignore frame as it wasn't for us
    nextState <= stIdle;
end if;

-- otherwise get the next header byte from RAM
else
    nextState <= stStoreHeaderByte;
    getNewByte <= '1';
end if;

```

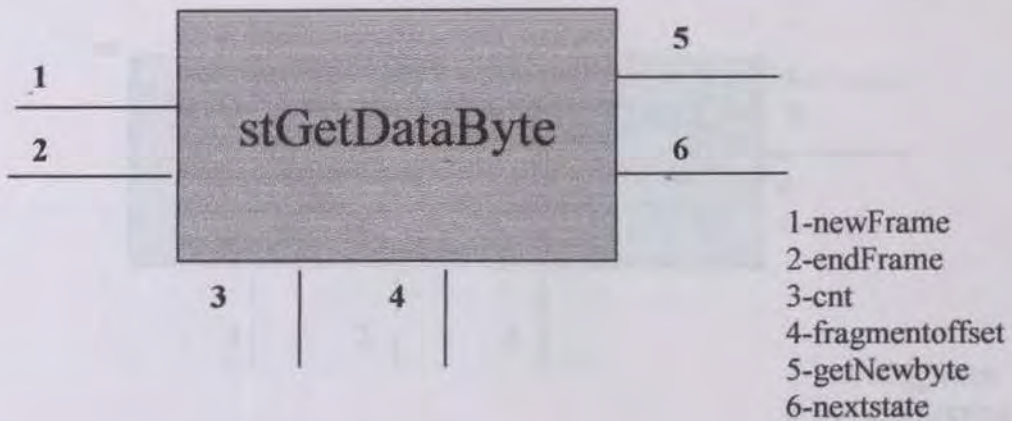



Operate on each value of the header received according to count and store the header in RAM.

```
when stStoreHeaderByte =>
  nextState <= stGetHeaderByte;
  -- operate on each value of the header received according to count
  -- count will be one higher than the last byte received, as it is incremented
  -- at the same time as the data is streamed in, so
  -- when the data is seen to be available, count should also be one higher

  -- Send data to checksum process
  newByte <= '1';
  inByte <= frameDataLatch;

  -- Operate on data in the header
  case cnt(4 downto 0) is
    when "00011" =>
      nextDatagramLen (10 downto 8) <= frameDataLatch (2 downto 0);
    when "00100" =>
      nextDatagramLen (7 downto 0) <= frameDataLatch;
    when "00101" | "00110" =>
      shiftInIdentification <= '1';
    when "00111" =>
      shiftInFragmentOffset <= '1';
      latchMoreFragments <= '1';
    when "01000" =>
      shiftInFragmentOffset <= '1';
    when "01010" =>
      latchProtocol <= '1';
    when "01101" | "01110" | "01111" | "10000" =>
      shiftInSourceIP <= '1';
    when "10001" | "10010" | "10011" | "10100" =>
      shiftInTargetIP <= '1';
    when others =>
  end case;
```

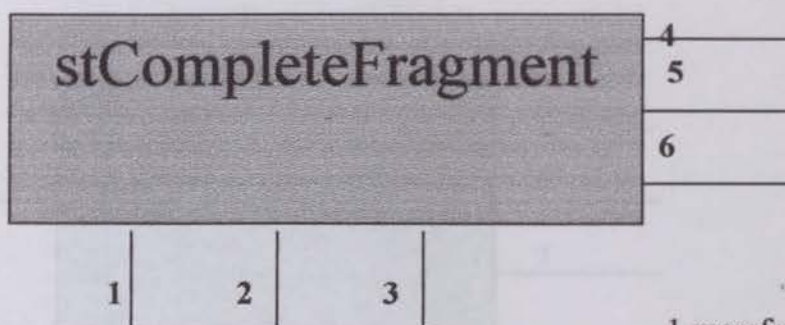



If we haven't finished receiving the data, then check the counter if it is not equal to datagram length then the next state will be stSetupWriteDataByte. Else if endFrame and frameValid is enabled, that means that the frame is finished and was valid. Then it will go to the next state which is stSetupwriteData.

```

when stGetDataByte =>
    -- if we haven't finished receiving the data, then
    if cnt /= datagramLen then
        nextState <= stSetupWriteDataByte;
        -- read an IP data byte from the data stream...
        getNewByte <= '1';
    elsif endFrame = '1' and frameValid = '1' then
        -- this means that the frame is finished and was valid
        -- so update the buffer data and go to final state
        nextState <= stCompleteFragment;
        resetTimeout <= '1'; -- start/restart timer
        latchIdent <= '1';    -- allocate buffer to data
        if fragmentOffset = 0 then -- check if this is the first fragment
            resetPosition <= '1'; -- give position initial value
        else
            updatePosition <= '1'; -- or add to the amount of data stored
        end if;
    elsif endFrame = '1' then
        -- the frame is complete but not valid so ignore it
        nextState <= stIdle;
    else
        -- the frame is not complete so keep looping until it is
        nextState <= stGetDataByte;
    end if;

```



- 1-morefragment
- 2-bufferSelecting
- 3-nextState
- 4-resetIdent
- 5-newdatagram
- 6-datagramSize

At this state a signal will be sent to the transport layer informing that the datagram is finished or await of a next frame.

when stCompleteFragment =>

-- Signal the transport protocols if the datagram is finished

-- or await next frame.

nextState <= stIdle;

if moreFragments = '0' then

-- Last frame so :

newDatagram <= '1'; -- notify higher protocols it's ready

resetIdent <= '1'; -- free buffer for next time

if bufferSelectSig = '0' then -- output datagram size from correct buffer

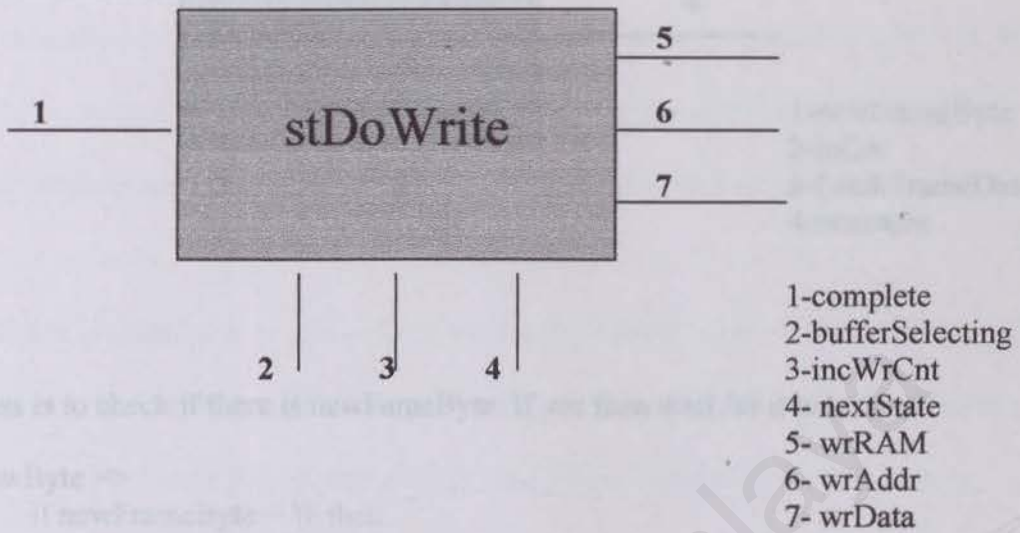
datagramSize <= position0;

else

datagramSize <= position1;

end if;

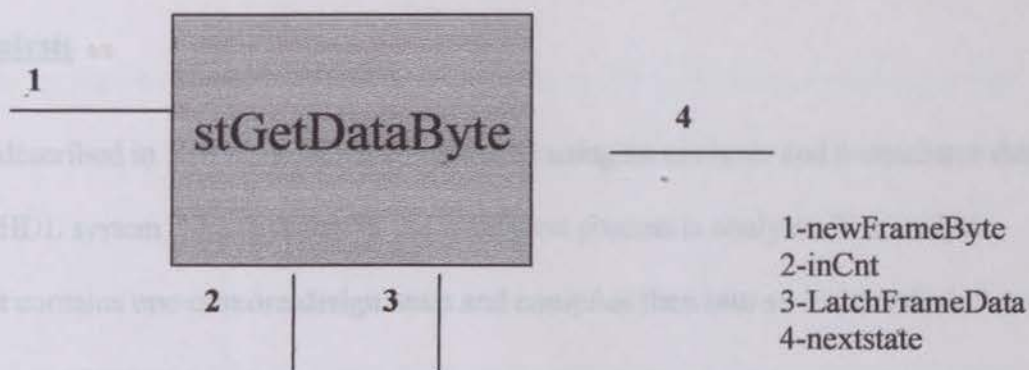
end if;



At this state we wait for RAM write request to be serviced if complete, then we can write data to RAM.

when stDoWrite =>

```
-- Wait for RAM write request to be serviced
if complete = '0' then
  -- keep signals asserted until complete is high
  nextState <= stDoWrite;
  wrRAM <= '1';
  -- The address is based on the fragment offset and buffer
  if bufferSelectSig = '0' then
    wrAddr <= "001" & (wrCnt + (fragmentOffset & "000"));
  else
    wrAddr <= "010" & (wrCnt + (fragmentOffset & "000"));
  end if;
  wrData <= frameDataLatch;
else
  -- when write is finished, go to returnState
  nextState <= returnState;
  incWrCnt <= '1';
end if;
```

The last process is to check if there is newFameByte. If yes then wait for it to arrive.

```

when stGetNewByte =>
  if newFrameByte = '0' then
    -- wait for new byte to arrive
    nextState <= stGetNewByte;
  else
    -- latch new byte and go to returnState
    nextState <= returnState;
    incCnt <= '1';
    latchFrameData <= '1';
  end if;

```

6.4 Model Analysis

Once entity is described in VHDL, it was then validated using an analyzer and a simulator that are part of a VHDL system. The first step in the validation process is analysis. The analyzer takes a file that contains one or more design units and compiles then into an intermediate form.

During compilation, the analyzer validates the syntax and performs static semantic checks. The generated intermediate form is stored in a specific design library that has been designated as the working library. The language analyzer always compiles descriptions into this library, therefore at the given time, only one library is updated.

A design library is a location in the host environment where compiled descriptions are stored. Each design library has a logical name that is used when referring to a library name to a physical storage location which are provided externally by the host environment and is not defined by the language. One possible way of providing the mapping of physical names to logical names is by specifying the mapping in a special file that the VHDL system could interpret.

6.5 Synthesis

After the VHDL entities are described, the next step is to synthesis those entities. Synthesis in the domain of digital design is a process of translation and optimization. For example, layout synthesis is a process of taking design netlist and translating it into a form of data that facilitates placement and routing, resulting in optimizing timing and chip design.

Logic synthesis, on the other hand, is the process of taking a form of input, translating it into form, and then optimizing it in terms of propagation delay and/or area.

After the VHDL code is translated into the internal form, the optimization process can be performed based on constraints such as speed, area, power and so on. After the synthesis process being completed, then the whole module will be simulated to testify that the behavior description is correct. The simulation will then be discussed in the next chapter.

Chapter 7

System Testing & Evaluation

7.1 Simulations and Testing

A simulator is nothing more than a program written in some high-level language. The program is designed to model the features of a computer system that the performance analysis is interested in studying. Since the simulator is just a program, it can be easily modified (at least in principle) to change the behavior of the system being studied. For instance, you may want to change a processor simulator to increase the size of pipeline stages, or to modify some characteristics of the cache.



Chapter 7

System Testing & Evaluation

While simulators are very flexible, it is impossible to model a system with complete details and complex interactions. Therefore, simulators usually make assumptions about the system to make the simulation tractable. These assumptions limit the accuracy of the results of a simulation study compared to the results obtained when measuring real systems. However, simulations are useful for predicting the performance of a system before it is built.

Simulators used in computer system performance evaluation come in two main varieties. Execution-driven simulators actually execute an application program, such as a standard benchmark program, as they perform the simulation of the system being evaluated. That is, the output of the application program being simulated will be the same as if the program were executed on actual system. At the same time, the simulator will be modeling the behavior of the system being evaluated. The output of the simulator then is the information about the system.

7.1 Simulations and Testing

A simulator is nothing more than a program written in some high-level language, the program is designed to model the features of a computer system that the performance analyst is interested in studying. Since the simulator is just a program, it can be easily modified (at least in principle) to change the behavior of the system being studied. For instance, you may want to change a processor simulator to increase the number of pipeline stages, or to modify some characteristics of the cache.

While simulators are very flexible, it is impossible to model all of the numerous details and complex interactions that occur in real systems. Simplifying assumptions must be made to make the simulation tractable. These simplifications limit the accuracy of the results of a simulation study compared to the results obtained when measuring real systems. Even with their limitations, simulators are very powerful tools in the computer systems performance evaluation tool chest. They are particularly useful when trying to predict the performance of a system that has not yet been built.

Simulators used in computer systems performance evaluation come in two main varieties. *Execution-driven* simulators actually execute an application program, such as a standard benchmark program, as they perform the simulation of the system being evaluated. That is, the output of the application program being simulated will be the same as if the program were executed on some real system. At the same time, the simulator will be modeling the behavior of the system being evaluated. The output of the simulator then, is the information about the system.

In contrast to execution-driven simulators, *trace-driven* simulators read and process a list of events. The simulation steps performed then depend on the type and sequence of events read. A cache simulator, for instance, could be driven with the sequence of addresses that were referenced by a processor when executing some benchmark program. A list of input events, such as the list of addresses for the cache simulator, can be recorded from the execution of an actual system, or from an execution-driven simulator. Another alternative is to generate a sequence of random numbers that follow a desired probability distribution. The assumption in this latter case is that the distribution of random numbers will be similar to the sequence of events that would be observed in a real system.

At this point in design cycle, the block diagram has been generated and the process begins to test and debug the design using the VHDL simulator. The simulator used is a PeakFPGA Simulator. Note that in contrast to the conventional approach there is no need for gate level yet, and the design will be tested at the higher level of behavior called RTL. The code for the VHDL design and test patterns will type in binary and hexadecimal base, which are the input source for the VHDL simulation tools. Besides, the simulator has a tool called script editor that can assist in creating portions of the test code and are useful in reducing the amount of effort in entering I/O signals and the structure of the major design blocks. In either case, the simulation process similar to traditional logic simulation. The source files are compiled and checked for error, then are linked together

with any other blocks that are part of the design. The simulation flow is the same in that desired input and output signals could be monitored either as graphical waveforms or as tabular listing in order to verify the proper behavior of the circuit. This simulator also provides the basic capabilities to allow the simulation to be executed for a short and long time interval to be a step through operation one at a time, and to run some desired breakpoint condition.

7.2 Cycle Simulation

Cycle simulation is a technique for simulating digital circuits that do not take into account the detailed circuit timing. Rather cycle simulation computes the steady state response of the circuit at each clock cycle boundary. The main benefit of cycle simulation over event driven simulation is faster simulation speed, provided the circuit being simulated has event activity over 15%. Although a design can never be exhaustively tested by functional verification alone, by using cycle simulation, on larger circuits in the same amount of time. Since only steady state responses need to be computed for each clock cycle, it is possible to perform circuit levelization operations at compile time as opposed to simulation time which reduces the number of circuit evaluations.

7.3 System Testing

The system is tested using the simulator which will validate the description of each module by showing the result corresponding to each given value. The table below shows the process of state machine process according to a given input value.

Input Conditions	Output/States
newByte & frameType ='1'	It will receive a packet from the layer below and process it by going to the next state which is stGetHeaderLen.
frameDataLatch = "0100"	This will check the IP version, if its four then accept packet and go to next state which is stGetHeaderByte
cnt = headerLen and checksum = 0	If the counter number of byte received = header length and header is free from error then continue the process of receiving the rest of packet.
endFrame = '1' and frameValid = '1'	This means that the frame is finished and was valid, so update the buffer data and go to next state stSetupWriteDataByte
if complete = '0'	Wait for RAM write request to be serviced

Table 7.1 State Machine process According to inputs

The snapshot below shows the output simulation of the receiving Packet module.

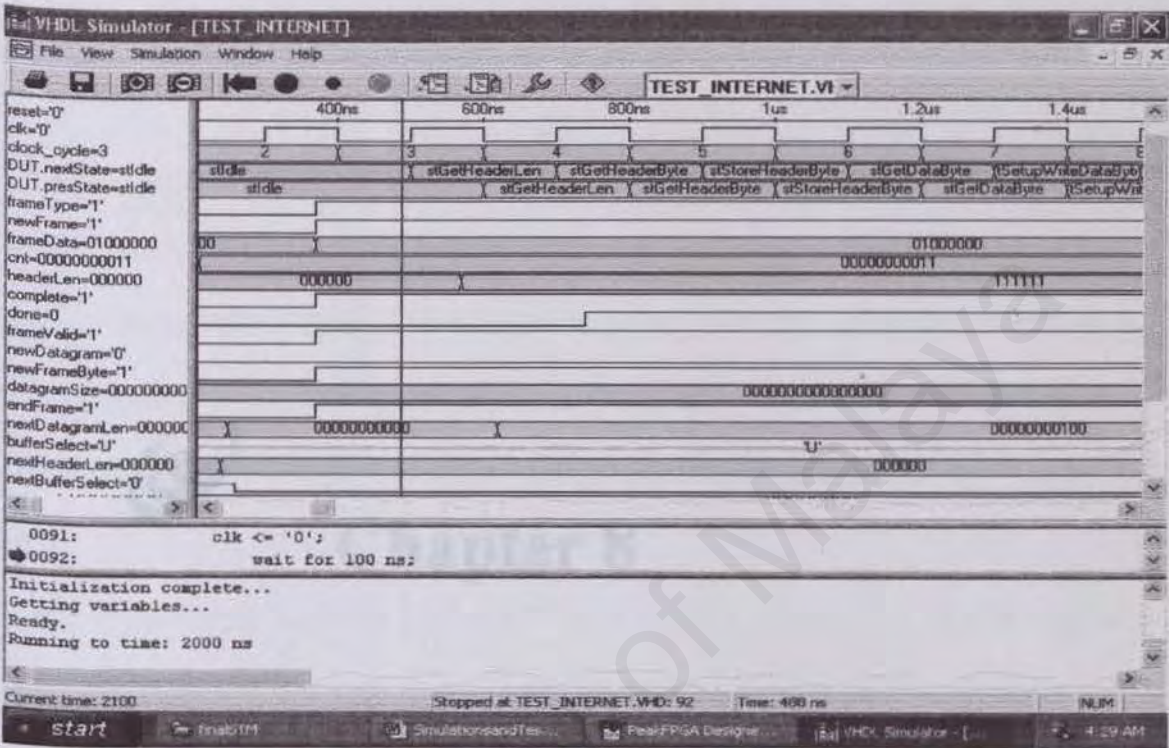


Figure 7.1 A Snapshot of the System Simulation output

Presented the stacks of hardware implementations are inflexibility and inability to handle complex tasks. The proposed solution is to use a reconfigurable Field Programmable Gate Arrays (FPGAs) solve the inflexibility problem.

The device is a compromise between general purpose processors and is software reconfigurations at one end of the flexibility-performance spectrum, and Application Specific Integrated Circuits (ASICs) at the opposite end of this spectrum. FPGAs can be reprogrammed with updated versions as signaling protocols evolve and can handle complex handling capacities relative to software processors.



Chapter 8

As for the challenge posed by the complexity of signaling protocols, reconfiguring the logic and frequently used operations of the protocol in hardware, and migrate the

complex and infrequent operations (e.g., processing of optional parameters) to software is central to this project. The proposed work items for this project are the implementation of a typical signaling protocol in FPGA.

Discussion

Perceived drawbacks of hardware implementations are inflexibility and inability to handle complex tasks. The proposed software to be used is reconfigurable Field Programmable Gate Arrays (FPGAs) solve the inflexibility problem.

This device is a compromise between general-purpose processors used in software implementations at one end of the flexibility-performance spectrum, and Application Specific Integrated Circuits (ASICs) at the opposite end of this spectrum. FPGAs can be reprogrammed with updated versions as signaling protocols evolve while significantly improving the call handling capacities relative to software implementation.

As for the challenge posed by the complexity of signaling protocols, implementing the basic and frequently used operations of the protocol in hardware, and relegate the complex and infrequently used operations (for example, processing of optional parameters) to software is carried out in this project. The proposed work items for this project are the implementation of a typical signaling protocol in FPGAs.

This research is on developing a TCP/IP engine in hardware, which can handle all kinds of performance intensive Internet protocols from processor systems. When compared to software implementations, which often consumes 50-90% of available processor cycles, the new offload architecture offers cost-effective system advantages.

Implementation of signaling protocols in hardware poses a tremendously different set of problems than implementing core-plane protocols such as IP. This research will demonstrate hardware handling of processing IP load. This prototype implementation of a signaling protocol has demonstrated the potential for 100x-1000x speedup vis-a-vis software implementations on state-of-the-art processors.



Chapter 9


Conclusion

This research is on developing a TCP/IP engine in hardware, which completely off-loads performance intensive Internet protocols from processor systems. When compared to software implementations, which often consumes 60-90% of available processor cycles, the new offload architecture offers cost-effective system advantages.

Implementation of signaling protocols in hardware poses a considerably larger number of problems than implementing user-plane protocols such as IP. The implementation will demonstrate the hardware handling of processing IP headers. Overall, this prototype implementation of a signaling protocol in FPGA hardware has demonstrated the potential for 100x-1000x speedup vis-à-vis software implementations on state-of-the-art processors.

Two type of references which I have used for writing this report which are

Books:

- TCP/IP Illustrated, Volume 1: The Protocols by W. Richard Stevens, 1994.
- TCP/IP Illustrated, Volume 1: The Implementation by Gary K. Wright and W. Richard Stevens, 1991.
- TCP/IP Network Administration by Craig Hunt, O'Reilly and others, 1998.
-  Interacting With TCP/IP Volume 1: Protocols, Tools And Architecture by Douglas Comer, Fourth edition, 2004.
- Data Telecommunication by Charles N. Kuhlman, Jr.
- TCP/IP LEARN Web Screen For Embedded Systems by Jeremy Bodman.
- Data Communications by Martin Comer, 1974.
- VHDL And Verilog Computer Aided Codeded by Douglas J. Smith.
- VHDL guide by Digital Party.

References

Journals And Articles From The Internet:

- Braden R., ed. "Requirements for Internet Hosts - Communication Layers," RFC 1122, October 1989.
- Carpenter B., ed. "Architectural Principles of the Internet," RFC 1938, June 1996.
- Domain Name FAQ <http://www.internic.net/faq/100/>

Two type of references which I have used for writing this report which are:

Books:

- TCP/IP Illustrated, Volume 1: The Protocols by W. Richard Stevens. 1994.
- TCP/IP Illustrated, Volume 1: The Implementation by Gary R. Wright and W. Richard Stevens. 1995.
- TCP/IP Network Administration by Craig Hunt, from O'Reilly. Second edition, 1998.
- Internetworking With TCP/IP Volume 1: Principles, Protocols And Architecture by Douglas Comer. Fourth edition, 2000.
- Data Telecommunication by Carle's N. Thurwachter, Jr.
- TCP/IP LEAN Web Servers For Embedded Systems by Jeremy Bentham.
- Data Communications by Myron E. SVEUM.
- VHDL And Verilog Compared And Contrasted by Douglas J. Smith.
- VHDL guide by Douglas Perry.

Journals And Articles From The Internet:

- Braden R., ed. "Requirements for Internet Hosts -- Communication Layers," RFC 1122. October 1989.
- Carpenter B., ed. "Architectural Principles of the Internet," RFC 1958. June 1996.
- Domain Name FAQ <http://www.internic.net/faq.html>.

-
- Jacobsen, Braden, Borman, ed. "TCP Extensions for High Performance," RFC 1323. May 1992.
 - Postel, Jon, ed. "Transmission Control Protocol," RFC 793. September 1981.
 - "User Datagram Protocol," RFC 768. August 1980.
 - "Internet Control Message Protocol," RFC 792. September 1981.
 - "Internet Protocol," RFC 791. September 1981.
 - Reynolds, Braden, ed. "Internet Official Protocol Standards," RFC 2600. March 2000.
 - Tanenbaum A.S. *Computer Networks, 3/e*. Prentice Hall. 1996.
 - Understanding IP addressing <http://www.3com.com/nsc/501302.html>.
 - www.Peakfpga.com
 - www.Xilinx.com
 - www.howstauffworks.com
 - [www.vhdl-online.de~vhdl-](http://www.vhdl-online.de/~vhdl-)
 - <http://www.gmvhdl.com/VHDL.html>



Appendices A: VHDL Source Code

-- IP layer for network stack project. This accepts byte-streams of data from
 -- the ethernet layer and decodes the IP information to send data to the upper
 -- protocols. Reassembly is implemented and two incoming packets can be
 -- reassembled at once. Reassembly only works if incoming packets come in
 -- order.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.global_constants.all;
```

entity internet is

```
port (
    clk: in std_logic;           -- clock
    rstn: in std_logic;          -- asynchroneuse active low reset
    complete: in std_logic;       -- control signal from ram arbitrator
    newFrame: in std_logic;       -- new frame received from the layer below
    frameType: in std_logic;      -- frame type = '1' for IP
    newFrameByte: in std_logic;   -- signals a new byte in the stream
    frameData: in std_logic_vector (7 downto 0); -- data is streamed in here
    endFrame: in std_logic;       -- signals the end of a frame
    frameValid: in std_logic;     - determines validity of frame when endFrame is high
    newDatagram: out std_logic;   -- an IP datagram has been fully received
    bufferSelect: out std_logic;  -- indicates location of data in RAM
    datagramSize: out std_logic_vector (15 downto 0); -- size of the datagram received
    protocol: out std_logic_vector (7 downto 0); -- protocol type of datagram
    sourceIP: out std_logic_vector (31 downto 0); -- lets upper protocol know the source IP
    wrRAM: out std_logic;         -- signal to write to the RAM
    wrData: out std_logic_vector (7 downto 0); -- data to write to the RAM
    wrAddr: out std_logic_vector (18 downto 0); -- address lines to the RAM for writing
    timeLED0: out std_logic;      -- indicates if buffer 0 is busy
    timeLED1: out std_logic;      -- indicates if buffer 1 is busy
);
```

end internet;

architecture internet_arch of internet is

-- signal declarations
 -- FSM states

```
type STATETYPE is (stIdle, stGetHeaderLen, stGetHeaderByte, stStoreHeaderByte,
    stGetDataByte, stSetupWriteDataByte, stCompleteFragment, stDoWrite,
    stGetNewByte);
signal presState: STATETYPE;
```



```

signal nextState: STATETYPE;
signal returnState: STATETYPE;    -- Used to return from RAM 'subroutines'

signal headerLen: std_logic_vector (5 downto 0);    -- IP datagram header length
signal nextHeaderLen: std_logic_vector (5 downto 0);    -- signal for the next header length
signal datagramLen: std_logic_vector (10 downto 0);    -- IP datagram total length in bytes
signal nextDatagramLen: std_logic_vector (10 downto 0);    -- signal for the next datagram
length
signal dataLen: std_logic_vector (10 downto 0);    -- IP datagram data length in bytes
signal nextDataLen: std_logic_vector (10 downto 0);    -- signal for the next data length

signal incCnt: std_logic;    -- increments byte address counter
signal rstCnt: std_logic;    -- resets byte address counter
signal cnt: std_logic_vector (10 downto 0);    -- byte address counter for the frame received

signal incWrCnt: std_logic;    -- increments the write address counter
signal rstWrCnt: std_logic;    -- resets the write address counter
signal wrCnt: std_logic_vector (15 downto 0);    -- write address counter for storing that data

signal doWrite: std_logic;    -- tell RAM controller to write data
signal getNewByte: std_logic;    -- wait for new data on the stream

signal latchFrameData: std_logic;    -- latch in the data from the stream
signal frameDataLatch: std_logic_vector (7 downto 0);    -- register to hold latched data

signal targetIP: std_logic_vector (31 downto 0);    -- stores target IP (destination)
signal shiftInTargetIP: std_logic;    -- signal to shift in target IP

signal shiftInSourceIP: std_logic;    -- stores source IP
signal latchProtocol: std_logic;    -- signal to shift in source IP

-- checksum signals
signal checkState: std_logic;
CONSTANT stMSB: std_logic := '0';
CONSTANT stLSB: std_logic := '1';

signal checksumLong: std_logic_vector (16 downto 0);    -- stores 2's complement sum
signal checksumInt: std_logic_vector (15 downto 0);    -- stores 1's complement sum

signal latchMSB: std_logic_vector (7 downto 0);    -- latch in first byte

signal newHeader: std_logic;    -- resets checksum
signal newByte: std_logic;    -- indicate new byte
signal lastNewByte: std_logic;    -- detect changes in newByte

```



```

signal inByte: std_logic_vector (7 downto 0);           -- byte to calculate
signal checksum: std_logic_vector (15 downto 0);        -- current checksum

-- bufferSelect is used both to indicate which area in RAM to write to
-- and to indicate which buffer control signals are to operate on
signal nextBufferSelect: std_logic;                     -- allows memory of bufferSelect
signal bufferSelectSig : std_logic;                     -- allows memory of bufferSelect

signal identification: std_logic_vector (15 downto 0);   -- identification field
signal shiftInIdentification: std_logic;                -- signal to shift in identification

signal fragmentOffset: std_logic_vector (12 downto 0);  -- fragment offset field
signal shiftInFragmentOffset: std_logic;                -- signal to shift in offset
signal moreFragments : std_logic;                       -- more fragments flag
signal latchMoreFragments : std_logic;                  -- signal to determine MF flag

-- The ident signals are of the form "source IP : protocol : identification" and
-- are used in reassembly.
signal targetIdent: std_logic_vector (55 downto 0);     -- incoming frame's ident
signal ident0: std_logic_vector (55 downto 0);          -- current ident for buffer 0
signal ident1: std_logic_vector (55 downto 0);          -- current ident for buffer 1
signal latchIdent: std_logic;                           -- latch targetIdent into specified buffer ident
signal resetIdent: std_logic;                            -- clear ident of specified buffer to indicate a vacant buffer

signal position0: std_logic_vector (15 downto 0);       -- stores expected offset of next fragment
signal position1: std_logic_vector (15 downto 0);       -- stores expected offset of next fragment
signal updatePosition: std_logic;                       -- add dataLen to current position
signal resetPosition: std_logic;                        -- set position to be dataLen

constant TIMERWIDTH : INTEGER := 30;                   -- can be used to vary timeout length

signal timeout0: std_logic_vector (TIMERWIDTH - 1 downto 0); -- timeout counter
signal timeout1: std_logic_vector (TIMERWIDTH - 1 downto 0); -- timeout counter
signal resetTimeout: std_logic;                         -- start timeout counter

constant FULLTIME: std_logic_vector (TIMERWIDTH - 1 downto 0) := (others => '1'); --
last value of timeout counter

signal sourceIPSig : std_logic_vector (31 downto 0);    -- internal signal for output
signal protocolSig : std_logic_vector (7 downto 0);      -- internal signal for output

begin
    -- These signals are used instead of buffer ports
    sourceIP <= sourceIPSig;
    protocol <= protocolSig;

```



```
bufferSelect <= bufferSelectSig;
```

```
-- Indicate when buffers are busy
```

```
timeLED0 <= '0' when timeout0 = FULLTIME or ident0 = 0 else '1';
```

```
timeLED1 <= '0' when timeout1 = FULLTIME or ident1 = 0 else '1';
```

```
-- Some definitions to make further code simpler
```

```
targetIdent <= sourceIPSig & protocolSig & identification;
```

```
dataLen <= datagramLen - ("000000" & headerLen);
```

```
-- main clocked process
```

```
process (rstn, clk)
```

```
begin
```

```
if rstn = '0' then -- only need to reset required signals
```

```
presState <= stIdle;
```

```
returnState <= stIdle;
```

```
ident0 <= (others => '0');
```

```
ident1 <= (others => '0');
```

```
timeout0 <= FULLTIME;
```

```
timeout1 <= FULLTIME;
```

```
elsif clk'event and clk = '1' then
```

```
-- Go to next state wither directly or via a RAM state.
```

```
* -- If a RAM write or a new byte from the data stream are requested,
```

```
-- the state machine stores nextState in returnState and goes to the
```

```
-- required state. After completion, the state machine will go to
```

```
-- returnState. This is like a 'subroutine' in the state machine.
```

```
if doWrite = '1' then
```

```
presState <= stDoWrite;
```

```
returnState <= nextState;
```

```
elsif getNewByte = '1' then
```

```
presState <= stGetNewByte;
```

```
returnState <= nextState;
```

```
else
```

```
presState <= nextState;
```

```
end if;
```

```
-- increment and reset the counter synchronously to avoid race conditions
```

```
if incCnt = '1' then
```

```
cnt <= cnt + 1;
```

```
elsif rstCnt = '1' then
```

```
cnt <= (others => '0');
```

```
end if;
```

```
-- increment and reset the write address counter synchronously
```



```

if incWrCnt = '1' then
    wrCnt <= wrCnt + 1;
elsif rstWrCnt = '1' then
    wrCnt <= (others => '0');
end if;

-- latch data read from RAM
if latchFrameData = '1' then
    frameDataLatch <= frameData;
end if;

-- these signals must remember their values once set
headerLen <= nextHeaderLen;
datagramLen <= nextDatagramLen;

-- shift registers and latches to hold important data
if shiftInSourceIP = '1' then
    sourceIPSig <= sourceIPSig(23 downto 0) & frameDataLatch;
end if;

if shiftInTargetIP = '1' then
    TargetIP <= TargetIP(23 downto 0) & frameDataLatch;
end if;

if latchProtocol = '1' then
    protocolSig <= frameDataLatch;
end if;

if shiftInFragmentOffset = '1' then
    fragmentOffset <= fragmentOffset(4 downto 0) & frameDataLatch;
end if;

if latchMoreFragments = '1' then
    moreFragments <= frameDataLatch(5);
end if;

if shiftInIdentification = '1' then
    identification <= identification(7 downto 0) & frameDataLatch;
end if;

-- bufferSelect will remember its previous value
bufferSelectSig <= nextBufferSelect;

-- handle timeout counters, resetTimeout will only reset the current buffer
if resetTimeout = '1' then
    if bufferSelectSig = '0' then

```

```

        timeout0 <= (others => '0');
    else
        timeout1 <= (others => '0');
    end if;
else
    -- increment timeout counters but don't let them overflow
    if timeout0 /= FULLTIME then
        timeout0 <= timeout0 + 1;
    else
        timeout0 <= FULLTIME;
    end if;
    if timeout1 /= FULLTIME then
        timeout1 <= timeout1 + 1;
    else
        timeout1 <= FULLTIME;
    end if;
end if;

-- the following signals will operate only on the current buffer which
-- is chosen with bufferSelect.
if bufferSelectSig = '0' then
    -- manage the ident register of the buffer
    if latchIdent = '1' then
        ident0 <= targetIdent;
    elsif resetIdent = '1' then
        ident0 <= (others => '0');
    end if;

    -- manage the position register of the buffer
    if resetPosition = '1' then
        position0 <= "00000" & dataLen;
    elsif updatePosition = '1' then
        position0 <= position0 + dataLen;
    end if;
else
    -- manage the ident register of the buffer
    if latchIdent = '1' then
        ident1 <= targetIdent;
    elsif resetIdent = '1' then
        ident1 <= (others => '0');
    end if;

    -- manage the position register of the buffer
    if resetPosition = '1' then
        position1 <= "00000" & dataLen;
    end if;
end if;

```



```
resetTimeout <= '0';
```

```
case presState is
```

```
when stIdle =>
```

```
-- wait for the arrival of a new frame that has a frameType of 1
```

```
if newFrame = '0' or frameType = '0' then
```

```
    nextState <= stIdle;
```

```
else
```

```
-- reset the counters for the next datagram
```

```
rstCnt <= '1';
```

```
rstWrCnt <= '1';
```

```
newHeader <= '1';
```

```
nextState <= stGetHeaderLen;
```

```
-- get header length and version information
```

```
getNewByte <= '1';
```

```
end if;
```

```
when stGetHeaderLen =>
```

```
-- check ip version
```

```
if frameDataLatch(7 downto 4) /= 4 then
```

```
    nextState <= stIdle;
```

```
else
```

```
    nextState <= stGetHeaderByte;
```

```
-- send data to checksum machine
```

```
inByte <= frameDataLatch;
```

```
newByte <= '1';
```

```
-- get the header length in bytes, rather than 32-bit words
```

```
nextHeaderLen <= frameDataLatch(3 downto 0) & "00";
```

```
end if;
```

```
when stGetHeaderByte =>
```

```
-- if we've finished getting the headers and processing them, start on the data
```

```
-- once finished, refragmenting will come next
```

```
if cnt = headerLen then
```

```
-- only operate on data meant for us, or broadcast data
```

```
if checksum = 0 then
```

```
-- determine which buffer should be used to handle the data
```

```
if ident0 = targetIdent and timeout0 /= FULLTIME then
```

```
-- the ident matches and the timeout counter has not expired
```

```
    nextBufferSelect <= '0';
```

```
-- accept the frame if its offset matches what we think it should be
```

```
-- this drops out of order and duplicate frames.
```

```
if position0 = fragmentOffset & "000" then
```

```
    nextState <= stGetDataByte;
```

```
else
```



```

        nextState <= stIdle;
    end if;
    elsif ident1 = targetIdent and timeout1 /= FULLTIME then
        -- the ident matches and the timeout counter has not expired

        nextBufferSelect <= '1';
        -- accept the frame if its offset matches what we think it should be
        -- this drops out of order and duplicate frames.
        if position1 = fragmentOffset & "000" then
            nextState <= stGetDataByte;
        else
            nextState <= stIdle;
        end if;
    elsif (ident0 = 0 or timeout0 = FULLTIME) and fragmentOffset = 0 then
        -- The ident doesn't match either of the buffers so check if buffer 0
        -- is free. If ident = 0 or the timeout has expired then the buffer is free
        -- This must be the first fragment if it is to go here so also check the offset
        nextState <= stGetDataByte;
        nextBufferSelect <= '0';
    elsif (ident1 = 0 or timeout1 = FULLTIME) and fragmentOffset = 0 then
        -- The ident doesn't match either of the buffers so check if buffer 1
        -- is free. If ident = 0 or the timeout has expired then the buffer is free
        -- This must be the first fragment if it is to go here so also check the offset
        nextState <= stGetDataByte;
        nextBufferSelect <= '1';
    else
        nextState <= stIdle;
    end if;
else
    -- ignore frame as it wasn't for us
    nextState <= stIdle;
end if;

-- otherwise get the next header byte from RAM
else
    nextState <= stStoreHeaderByte;
    getNewByte <= '1';
end if;

when stStoreHeaderByte =>
    nextState <= stGetHeaderByte;
    -- operate on each value of the header received according to count
    -- count will be one higher than the last byte received, as it is incremented
    -- at the same time as the data is streamed in, so
    -- when the data is seen to be available, count should also be one higher

```

-- Send data to checksum process

newByte <= '1';

inByte <= frameDataLatch;

-- Operate on data in the header

case cnt(4 downto 0) is

when "00011" =>

nextDatagramLen (10 downto 8) <= frameDataLatch (2 downto 0);

when "00100" =>

nextDatagramLen (7 downto 0) <= frameDataLatch;

when "00101" | "00110" =>

shiftInIdentification <= '1';

when "00111" =>

shiftInFragmentOffset <= '1';

latchMoreFragments <= '1';

when "01000" =>

shiftInFragmentOffset <= '1';

when "01010" =>

latchProtocol <= '1';

when "01101" | "01110" | "01111" | "10000" =>

shiftInSourceIP <= '1';

when "10001" | "10010" | "10011" | "10100" =>

shiftInTargetIP <= '1';

when others =>

end case;

when stGetDataByte =>

-- if we haven't finished receiving the data, then

if cnt /= datagramLen then

nextState <= stSetupWriteDataByte;

-- read an IP data byte from the data stream...

getNewByte <= '1';

elsif endFrame = '1' and frameValid = '1' then

-- this means that the frame is finished and was valid

-- so update the buffer data and go to final state

nextState <= stCompleteFragment;

resetTimeout <= '1'; -- start/restart timer

latchIdent <= '1'; -- allocate buffer to data

if fragmentOffset = 0 then -- check if this is the first fragment

resetPosition <= '1'; -- give position initial value

else

updatePosition <= '1'; -- or add to the amount of data stored

end if;

elsif endFrame = '1' then

-- the frame is complete but not valid so ignore it

nextState <= stIdle;


```

else
    -- the frame is not complete so keep looping until it is
    nextState <= stGetDataByte;
end if;

```

```

when stSetupWriteDataByte =>
    nextState <= stGetDataByte;
    --Set up to write the byte that was read in stGetDataByte to RAM
    doWrite <= '1';
    wrData <= frameDataLatch;

```

```

when stCompleteFragment =>
    -- Signal the transport protocols if the datagram is finished
    -- or await next frame.
    nextState <= stIdle;
    if moreFragments = '0' then
        -- Last frame so :
        newDatagram <= '1';      -- notify higher protocols it's ready
        resetIdent <= '1';      -- free buffer for next time
        if bufferSelectSig = '0' then -- output datagram size from correct buffer
            datagramSize <= position0;
        else
            datagramSize <= position1;
        end if;
    end if;
end if;

```

```

when stDoWrite =>
    -- Wait for RAM write request to be serviced
    if complete = '0' then
        -- keep signals asserted until complete is high
        nextState <= stDoWrite;
        wrRAM <= '1';
        -- The address is based on the fragment offset and buffer
        if bufferSelectSig = '0' then
            wrAddr <= "001" & (wrCnt + (fragmentOffset & "000"));
        else
            wrAddr <= "010" & (wrCnt + (fragmentOffset & "000"));
        end if;
        wrData <= frameDataLatch;
    else
        -- when write is finished, go to returnState
        nextState <= returnState;
        incWrCnt <= '1';
    end if;

```

```

when stGetNewByte =>

```



```

if newFrameByte = '0' then
    -- wait for new byte to arrive
    nextState <= stgetNewByte;
else
    -- latch new byte and go to returnState
    nextState <= returnState;
    incCnt <= '1';
    latchFrameData <= '1';
end if;

```

```

when others =>
    end case;
end process;

```

-- Perform 2's complement to one's complement conversion, and invert output

```

checksumInt <= checksumLong(15 downto 0) + checksumLong(16);
checksum <= NOT checksumInt;

```

```

process (clk,rstn)
begin

```

```

    if rstn = '0' then
        checkState <= stMSB;
        latchMSB <= (others => '0');
        checksumLong <= (others => '0');
        lastNewByte <= '0';
    elsif clk'event and clk = '1' then
        -- this is used to check only for positive transitions
        lastNewByte <= newByte;
    
```

```

    case checkState is

```

```

        when stMSB =>

```

```

            if newHeader = '1' then
                -- reset calculation
                checkState <= stMSB;
                checksumLong <= (others => '0');
            elsif newByte = '1' and lastNewByte = '0' then
                -- latch MSB of 16 bit data
                checkState <= stLSB;
                latchMSB <= inByte;
            
```

```

        else

```

```

            checkState <= stMSB;

```

```

        end if;

```

```

        when stLSB =>

```

```

            if newHeader = '1' then
                -- reset calculation
                checkState <= stMSB;
                checksumLong <= (others => '0');
            
```

```

elsif newByte = '1' and lastnewByte = '0' then
-- add with 2's complement arithmetic (convert to 1's above)
checkState <= stMSB;
checkSumLong <= ('0' & checkSumInt) + ('0' & latchMSB & inByte);
else
    checkState <= stLSB;
end if;
when others =>
    checkState <= stMSB;
end case;
end if;
end process;
end internet_arch;

```

Appendices B: Peak FPGAs User Manual

PeakFPGA is now a part of oVivado DXP.

The powerful and versatile VHDL, Verilog, and SystemVerilog synthesis and logic simulation, PeakFPGA, has now been integrated into the oVivado DXP design tool. With oVivado DXP, you can now design and simulate your digital logic designs and electronic design tool, oVivado DXP.

oVivado DXP delivers all of the features and capabilities of PeakFPGA, plus a whole lot more! With oVivado DXP you can now design and simulate your digital logic designs using methods oVivado offers you to help you design. oVivado-based circuit design, schematic-based FPGA design, Verilog-based VHDL and Verilog-based VHDL, and VHDL-driven FPGA design.



Running a Sample Project

This tutorial will introduce you to the features of PeakFPGA, including how to design through the simulation and synthesis steps, and how to use the project tool.

Appendices B: PeakFPGA User Manual

The steps within this tutorial are as follows:

Step 1: Open the sample project

Step 2: Prepare the project

Step 3: Simulate the project

Step 4: Synthesize the project

For the Quick Tour, click on the link and first of each page to skip through the topics.

Step 1: Open the sample project

Begin by launching the PeakFPGA application and clicking on the Open Existing Project button as shown below:

PeakFPGA is now a part of nVisage DXP

The powerful and versatile VHDL-based FPGA design entry, simulation and synthesis solution, PeakFPGA, has now been incorporated into Altium's new multi-dimensional front-end electronic design tool, nVisage DXP.

nVisage DXP delivers all of the features and functions of PeakFPGA plus a whole lot more! With nVisage DXP you can capture your design using multiple, integrated design entry methods. nVisage allows you to freely mix: Schematic-based circuit design, Schematic-based FPGA design, Text-based VHDL and CUPL code or mixed schematic and VHDL-driven FPGA design.

Running a Sample Project

This tutorial will introduce you to the features of PeakFPGA by taking you step-by-step through the simulation and synthesis of an sample project included in the product. The sample project we've used is a controller for a video frame capture unit that has been implemented in VHDL as a state machine.

The steps within this tutorial are as follows:

- Step 1: Open the sample project
- Step 2: Prepare the project for simulation
- Step 3: Simulate the project
- Step 4: synthesize to an FPGA

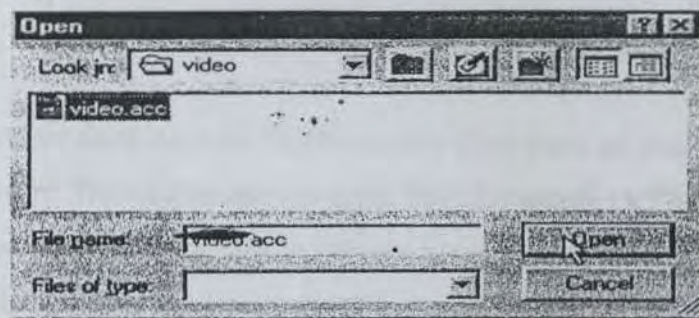
Use the Quick Jump menus at the head and foot of each page to skip through the topics.

Step 1: Open the sample project

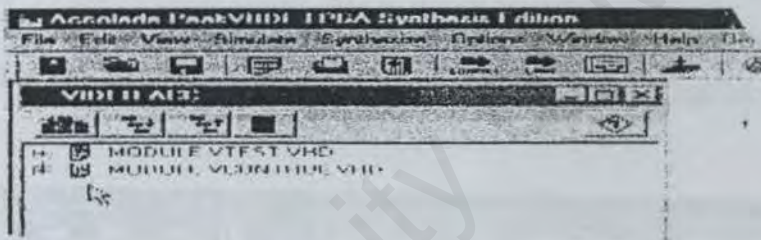
Begin by launching the PeakFPGA application and clicking on the **Open Existing Project** toolbar button as shown below:



Navigate to the **examples** directory and choose the **VIDEO.ACC** file from the **Video** subdirectory as shown below:

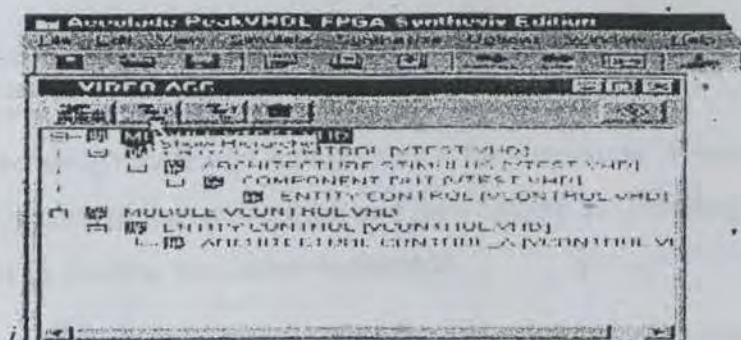


When the project is open, you will see the Hierarchy Browser child window, listing the two VHDL source files associated with this project:



The Hierarchy Browser is the place where you will select files for editing, invoke processes for simulation and synthesis, and otherwise manage your design files. The Hierarchy Browser also gives you valuable information about the structure of your VHDL design, such as the relationship of lower-level VHDL design units (entities, architectures, components, etc.).

To view the complete hierarchy for your design, for example, you can select the **Show Hierarchy** button in the Hierarchy Browser toolbar (or click on the small + icons next to each file name) to expand the view and see what each design module (VHDL source file) is composed of:



In this simple project there are two VHDL source files, each of which includes one entity and one architecture. Notice that the top-most VHDL module (**VTEST.VHD**) also includes a component entry that makes reference to an entity in the second module, **VCONTROL**. This is how the Hierarchy Browser displays VHDL hierarchy information, and how it determines file dependencies (and order of compilation) when processing your designs.

To edit a VHDL source file, simply double-click on the desired module name, or on any lower-level design unit name (such as an entity or architecture name) in the Hierarchy Browser. Double-clicking on a name in the Browser invokes the built-in text editor as shown below:



Note that in this example we have double-clicked on the entry for **ARCHITECTURE STIMULUS**, and the editor has jumped to the corresponding section of VHDL code.

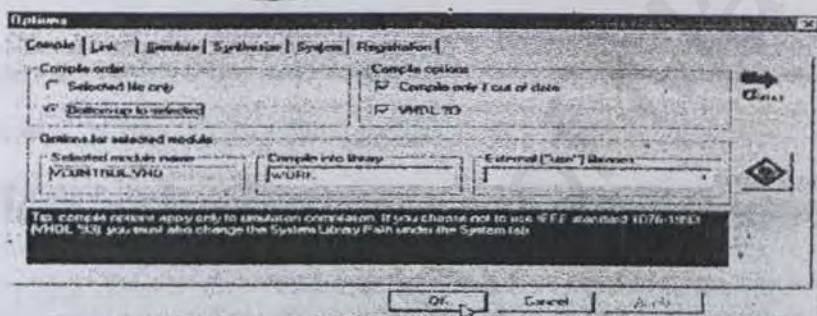
Note: PeakFPGA allows you to specify an external text editor to be invoked when project files are edited. Refer to the PeakFPGA help files for more information.

Step 2: Prepare the project for simulation

The first step in processing this design is to prepare it for simulation. This involves two steps: first compiling each of the source files, and then linking the resulting compiled output files together to create a simulation executable.

PeakFPGA can, if desired, perform these steps automatically (using the dependency checking features of the Hierarchy Browser) each time you invoke the simulator. For illustrative purposes, however, we will compile each file in this sample project individually.

Before we compile this design for simulation, let's first look at the compiler options available. To see the compile options, select the **Options** menu or click on the **Options** icon (the one that looks like a wrench) to open the Options dialog:



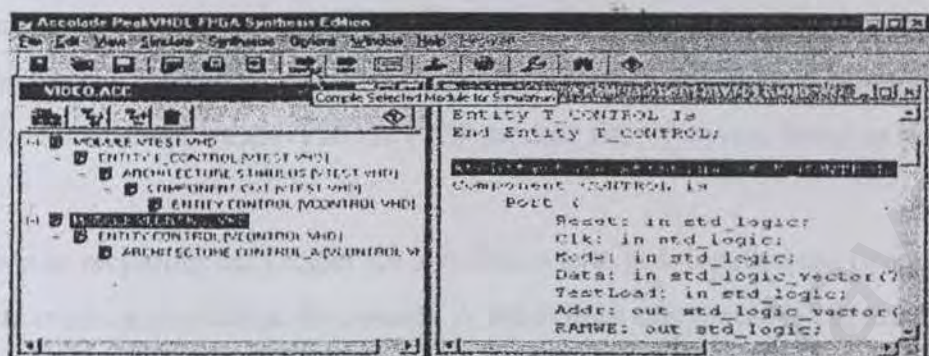
The **Compile** tab (which is the default tab) of the Options dialog shows the options available during compilation. These options are documented in the Help system (directly accessible via that question mark icon to the right). The options we have selected for this project are:

Bottom-up to select. This option instructs PeakFPGA to look for any lower-level VHDL files (those that the current file depends on) and compile them before compiling the selected file.

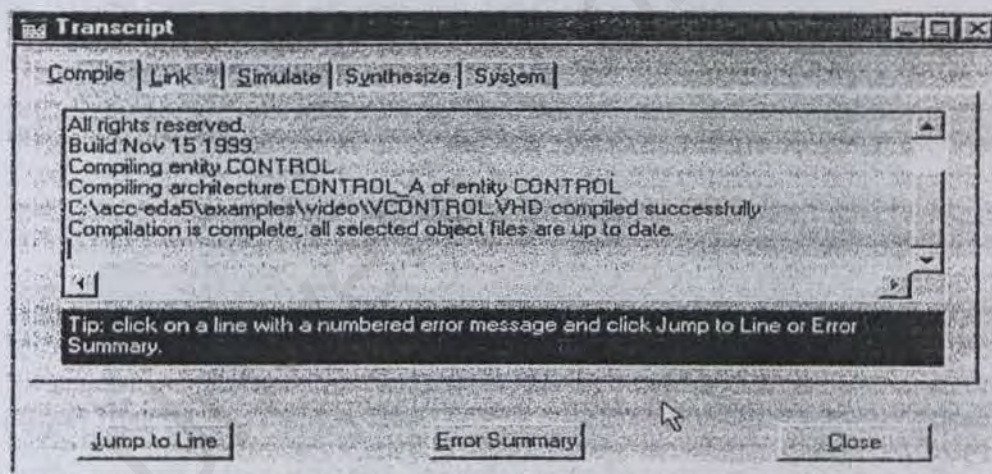
Compile only if out of date. This option prevents files from being re-compiled if they are already compiled and up-to-date. This can be a big time saver for larger designs consisting of many source files.

Compile into library. This option specifies that the currently highlighted module (the one being compiled) is to be compiled into a library called **WORK**. (As specified by the VHDL language standard, this is the default compile library.)

To compile a VHDL module (in this case the **VTEST.VHD** module), you first highlight the module in the Hierarchy Browser (as shown below) and select the **compile** button (or select **Compile** from the **Simulate** menu):



When you start the compile process, a transcript window appears and displays status messages, as well as reporting syntax or other errors found in your source file:



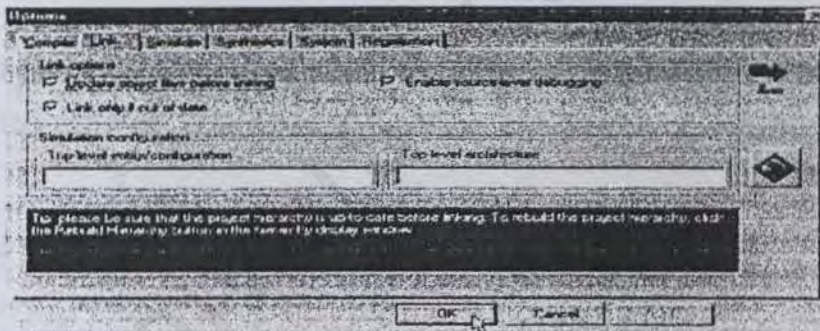
There were no errors in this sample source file, so we repeat the process by selecting and compiling the **VTEST.VHD** module.



Note: it is not actually necessary to compile every source file individually as we are doing in this sample. Instead, we could have simply compiled the top-level module, which in this case is **VTEST.VHD**. The Hierarchy Browser would have automatically compiled the lower-level **VCONTROL.VHD** module first if it were found to be out of date.

The next step in preparing the project for simulation is to link together the two compiled modules and create a *simulation executable*. A simulation executable is a special kind of executable file that can be executed in PeakFPGA's VHDL simulation and debugging environment.

As with compiling, there are options available (in the **Link** tab of the Options dialog) that control the linking process. For this sample, the options set are:



Update object files before linking. This option specifies that the Hierarchy Browser will check to make sure all relevant VHDL modules have been successfully compiled before starting the link process. When this option is selected, it is not necessary to manually compile the VHDL source files (as we did in the previous step) before invoking the Link process.

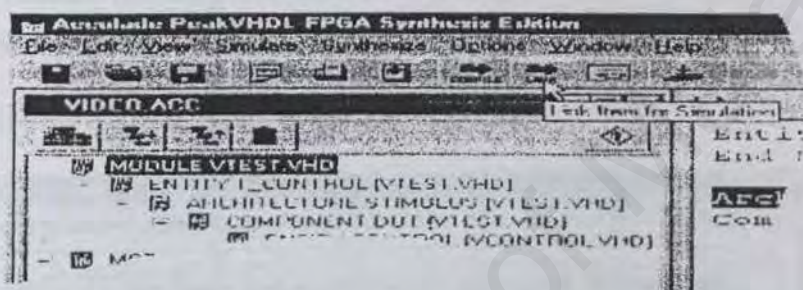
Link only if out of date. This option (which is similar to the **Compile only if out of date** option discussed previously) prevents the link process from being run if the simulation

executable is already up to date as indicated by its time stamp. (If the simulation executable is new than all VHDL files and object files that it depends upon, it will not be re-linked.)

Enable source-level debugging. This option instructs the code generation software (the portion of the linker that actually generates Windows executable code) to insert additional information to allow debugging of the design at the source code level.

Simulation configuration. These two text entry fields (which are left blank in this example) allow you to re-specify the default top-level entity and architecture used for simulation. This can be a convenience for certain kinds of test benches.

To start the Link process, select the top-level design unit (or the top-level module) and select the **Link** button (or select **Link** from the **Simulate** menu) as shown below:



The transcript will again appear and the Link process will execute. Errors (if any) will be reported to the transcript. The project is now compiled, linked and ready for simulation.

Step 3: Add functionality to the new module

The VHDL source file created by the New Module Wizard is not a complete VHDL file. (PeakFPGA cannot read your mind and know what you want this new module to do, no matter how descriptive a name you give it.) The next step, then, is to complete the source file by adding the needed functionality.

At this point it is a good idea to scan through the generated VHDL source file and get a feel for what has been created. If you examine the file, you will find that it has created a comment header followed by a few standard library references, which are commented to

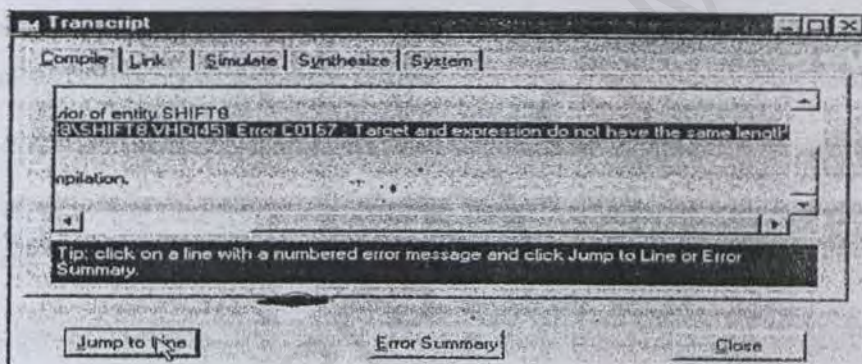
help you understand their purpose. After this header you'll see the entity declaration with the ports that you described using the Wizard. (The entity declaration for this example was shown in the previous screen image.)

After the entity declaration you will find a template architecture declaration containing some sample code that you can modify for your particular needs.

Step 4: Compile the VHDL module

Once we have entered the VHDL code and modified the template to our liking, we can check our work by invoking the simulation compiler. To invoke the compiler, make sure the appropriate module (at this point there is only one) is selected in the Hierarchy Browser and select the **Compile** button from the toolbar.

A Transcript window appears as shown below:



Notice that the compiler has reported an error (an incorrect expression width specified at line 45 in the file). When errors such as this appear in the transcript, it generally means that we have made some mistake in entering the VHDL code. Fortunately it is easy to find such errors.

Notice that the transcript window includes two buttons in addition to the **Close** button in the lower part of the dialog. The **Jump to Line** and **Error Summary** buttons will (respectively) open the text editor and take you to the appropriate line in the source file,

and provide you with more detailed error message information and (in some cases) suggestions on how to resolve the problem.

Step 5: Create a test bench

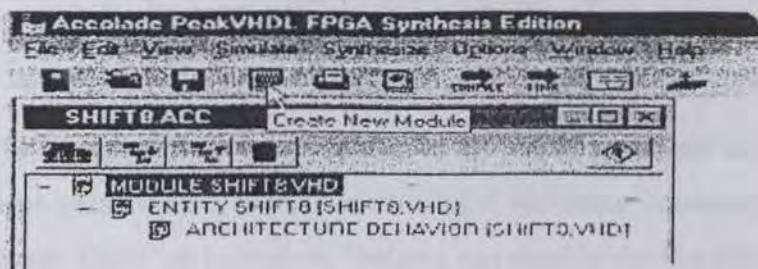
We now have a completed VHDL module, ready for synthesis into an FPGA (or for use in a larger hierarchy of VHDL modules). But how do we know that the VHDL we wrote is functionally correct? The answer, of course, is to simulate it.

Simulation in VHDL, as we saw in the first tutorial, requires that you not only describe the design (or component of a design) itself, but that you also provide a test bench. A test bench is a VHDL source file that describes stimulus to be applied to the design, which for this purpose is often referred to as the *unit under test* or *device under test*. Test benches can be quite simple, applying a sequence of inputs to the unit under test, or can be much more complex, perhaps reading stimulus information from external files and automatically comparing simulation results.

Regardless of their complexity, all test benches share some common traits: they all reference the lower-level design module (the unit under test) as a component, and they all include some means of providing stimulus to the unit under test.

In general, you can expect your test benches to be similar in size and complexity to the actual design being verified. In fact, for many designs (including our simple shifter) the test bench can significantly greater in size due to the overhead of declaring signals, creating the component declaration and component instance, etc. PeakFPGA's test bench Wizard saves you time by automatically generating much of this overhead code.

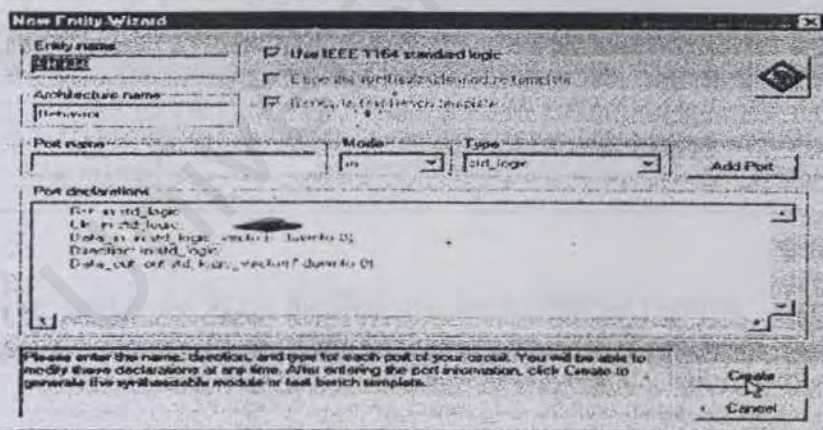
To start the Test Bench Wizard, first highlight the module that will be tested (in our case **SHIFT8.VHD**), then click the **Create New Module** toolbar button as shown:



As in the earlier step, the New Module dialog appears with the three Wizard buttons. This time, select the **Test Bench Wizard** button.

You'll see right away that this Wizard looks very much like the previous Wizard. The only difference this time is that the fields for the **Entity name**, **Architecture name**, and the **Port declarations** are already filled out for you, based on the port declarations found in the already-generated VHDL module. (If the port declarations list is empty, you may have neglected to select an existing VHDL module first, or the selected module does not have port declarations in a form recognizable to the Wizard. In the latter situation you may need to enter the port list yourself, or paste it in from the clipboard after copying it from the original lower-level file.)

Here is the Test Bench Wizard dialog for our design:



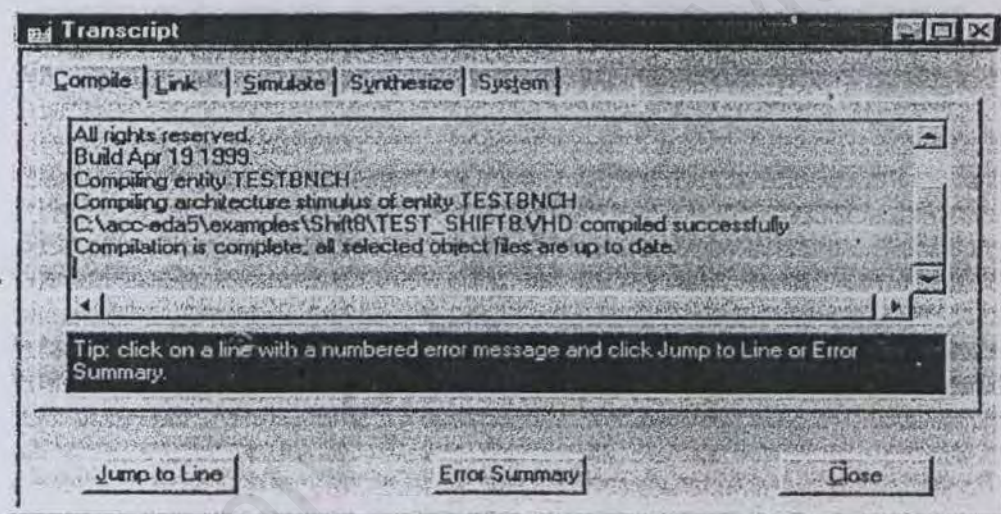
At this point, all we need to do is quickly verify that the port list is intact (has the ports that we defined in the original module) and click the **Create** button.

The Wizard prompts for a VHDL file name, and supplies a default name. And as before, we can simply accept this name and allow the file to be added to the project.

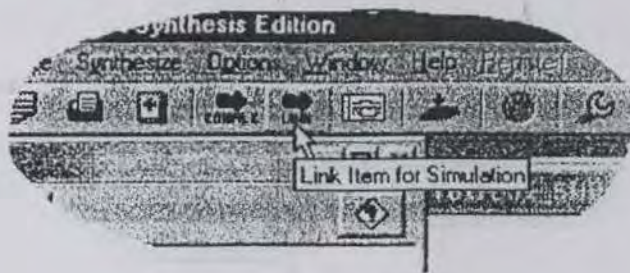
When the test bench template has been added to the project, we need to click the **Rebuild Hierarchy** button (described earlier) to establish the hierarchy information in the Hierarchy Browser. Once we have done that, we can modify the test bench template to complete the stimulus and add any other verification-related VHDL source file statements.

Step 6: Simulate the design

Now that we have the test bench we are ready to simulate the design. First we compile the test bench by highlighting the **TEST_SHIFT8.VHD** module and clicking the **Compile** button. If we are lucky enough to have no VHDL coding errors, our transcript looks like this:

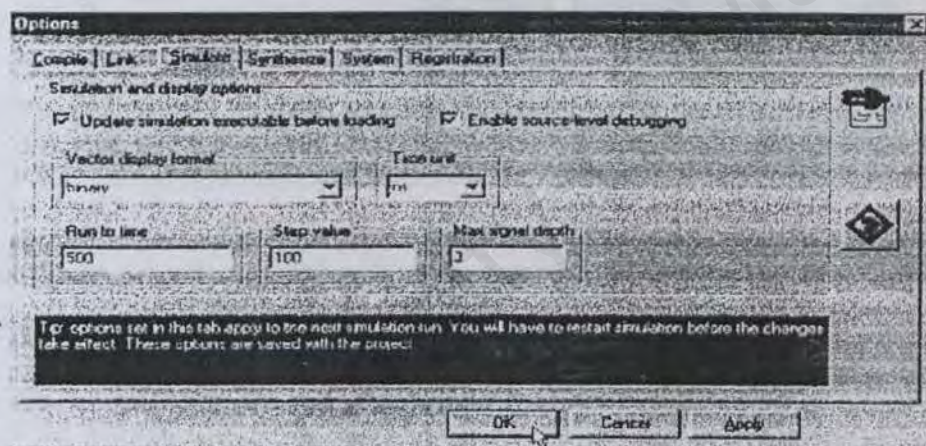


Next we link the project by again highlighting the test bench module (**TEST_SHIFT8.VHD**) and clicking the **Link** button:

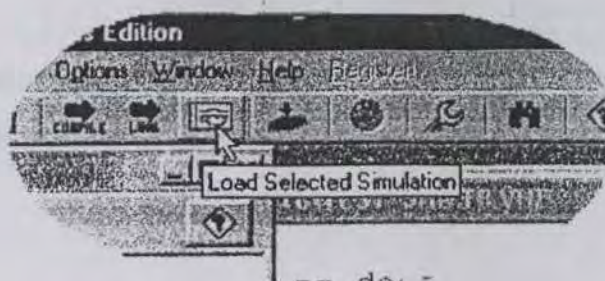


Linking the two compiled modules together provides us (behind the scenes) with a file named **TEST_SHIFT8.VX**. This is the *simulation executable*, and it is this file that the VHDL simulator will load and execute when we select the **Load** button.

Before selecting **Load**, however, let's take a moment to examine and modify the simulation options. We open the Options dialog and click on the **Simulate** tab. In the Simulate Options dialog, we'll set the **Vector display format** to binary (to more clearly see the shifter behavior) and set the **Run to time** to **500 ns** as shown below:



Now, after clicking the **Close** button to dismiss the Options dialog, we can select the **Load selected simulation** button (first making sure that the test bench is the highlighted module) to start the simulator:

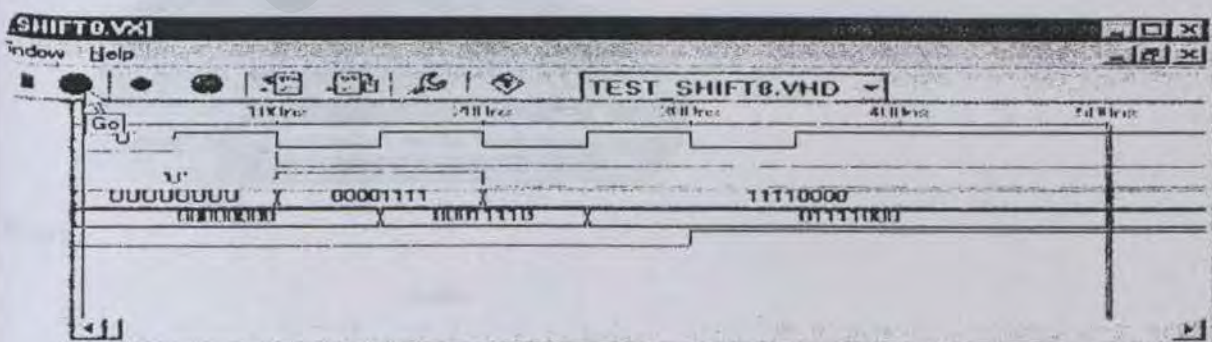


As in the first tutorial, the Select Display Objects dialog appears. We will simply use the **Add Primaries** button to make all top-level signals in the design available for display. We'll also spend a moment arranging the objects to place the signals of interest in a more useful vertical display order:



When the signals are selected and ordered to our liking, we can click the **Close** button to dismiss the selection dialog.

Click the **Go** button to run the simulation. Notice that the simulation results show us that the shifter appears to be working properly, wrapping the left-most or right-most bit (depending on direction) to the opposite side of the collection of bits while shifting them accordingly:



So far so good! We've created a new module, and we've verified that it works as expected. Now we can synthesize the module to create an FPGA compatible netlist.

Step 7: Synthesize the design

Synthesis is a straightforward process, particularly when dealing with a single VHDL source file. For this example we'll select the Altera family of devices. Notice when we select **Altera** there are fewer synthesis options:



The specific set of options available for synthesis is determined by the device family that has been selected. For your reference, all of the synthesis options are documented in the on-line help. Simply click the **Help** button in the dialog.

After choosing a target FPGA family, setting relevant options and closing the Options dialog, we start the synthesis process by highlighting the **SHIFT8.VHD** module and selecting the **Synthesize** toolbar button as shown



below:

When synthesis is complete, we can review the generated transcript:

```
C:\VCC-EDA5\EXAMPLES\SHIFT8\PEAKFPGA...
METAMOR VHD C:\VCC-EDA5\EXAMPLES\SHIFT8\PEAKFPGA.LOG
Copyright (c) 1992-2000 Protel Inte
Compiling for: Altera , Macrocell i
analyze....
elaborate design "SHIFT8"
process : P1
Inferred structure :
flip flop: ar Data_out
flip flop: ar Data_out
flip flop: ar Data_out
flip flop: ar Data_out
flip flop: ar Data_out
flip flop: ar Data_out
```

When synthesis is complete and you have an FPGA netlist, you can then move on to place-and-route using the FPGA place-and-route tools provided by your FPGA vendor.

Appendices C: Main Reference



IPAC - The Internet Protocol in Hardware for System-on-Chip Applications

Most systems today, which require embedded Internet connectivity, make use of a 32 bit processor core and implement the TCP/IP protocol stack in software. This realization, however, often results in strong processor performance requirements and keeps system costs for Internet applications high.

Committed to continuously improve price, performance, power consumption of embedded systems, ADESCOM has developed a TCP/IP engine in hardware, code-named IPAC, which off-loads performance intensive Internet protocols from processor sub-systems and allows for separate system optimization of both parts. Instead of having the need for expensive, high-performance cores, SoC designers can now tailor processor sub-systems to fit the application. This leads to more application performance, reduced clock frequencies and power consumption, or allows for use of 8/16 bit processor architectures and even the complete elimination of cores.

IPAC-E100

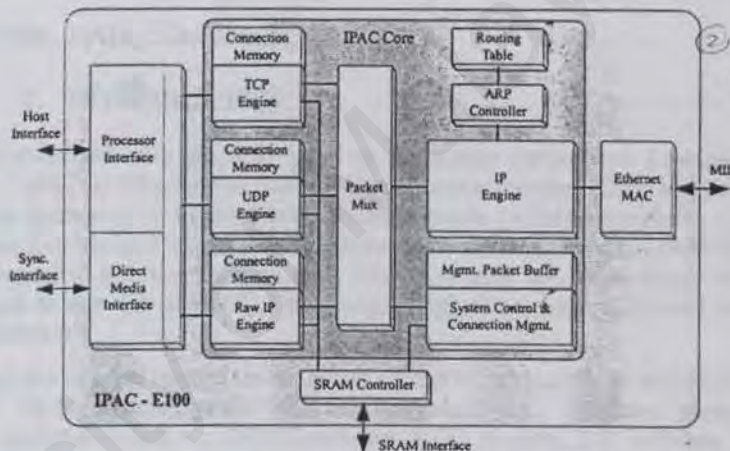
IPAC-E100 is the first member of ADESCOM's family of embedded Internet soft-macros around the IPAC core. Its integrated Ethernet MAC provides an MII interface to a 10/100 Ethernet transceiver. To the system side, IPAC-E100 offers an Intel/Motorola compliant processor bus with DMA support and an optional, programmable synchronous interface for direct connectivity to audio/video and data streams.

IPAC-E100 performs all protocol functions of TCP/IP and UDP/IP connections for sustained bit rates of up to 100 Mbps independent of packet payload sizes and other connection

parameters. A scalable memory architecture offers connection data buffers with programmable thresholds for variable interrupt latency and allows for simultaneous operation of up to 64k connections. Alternatively, an optional SRAM controller provides a high speed interface to external memory.

IP connection set up and management is made easy by a set of user registers of

IPAC's comprehensive control unit, which fully supports IP management protocols and allows for system configuration. Its auto-configuration and remote management capability enable IPAC-E100 to provide IP connectivity even without any external processor interaction, which makes it also an ideal Internet access solution for existing applications.



- ▶ 100 Mb/s throughput for all packet sizes
- ▶ Support of up to 64k connections
- ▶ Complete TCP/IP solution
 - IPv4
 - TCP, UDP, raw IP
 - ICMP
 - client DHCP and DNS
 - ARP
- ▶ IP multicast and IGMP
- ▶ Integrated 10/100 Ethernet MAC with MII interface
- ▶ Connection data buffers with programmable thresholds

- ▶ Scalable buffer memory
- ▶ SRAM controller for external memory
- ▶ 8/16/32 bit processor interface with DMA support
- ▶ Stand-alone capability
 - direct media interface
 - auto-configuration
 - remote management
- ▶ Fully synthesizable soft-macro
 - VHDL RTL source code
 - synchronous design
- ▶ Also available as pre-configured FPGA macro for time-to-market solutions



ADESCOM

256 Calvin Place • Santa Cruz • CA 95060 • U.S.A.

www.adesc.com

A hardware implementation of a signaling protocol

Haobo Wang, Malathi Veeraraghavan and Ramesh Karri
Polytechnic University, New York

ABSTRACT

Signaling protocols in switches are primarily implemented in software for two important reasons. First, signaling protocols are quite complex with many messages, parameters and procedures. Second, signaling protocols are updated often requiring a certain amount of flexibility for upgrading field implementations. While these are two good reasons for implementing signaling protocols in software, there is an associated performance penalty. Even with state-of-the-art processors, software implementations of signaling protocol are rarely capable of handling over 1000 calls/sec. Correspondingly, call setup delays per switch are in the order of milliseconds. Towards improving performance we implemented a signaling protocol in reconfigurable FPGA hardware. Our implementation demonstrates the feasibility of 100x-1000x speedup vis-à-vis software implementations on state-of-the-art processors. The impact of this work can be quite far-reaching by allowing connection-oriented networks to support a variety of new applications, even those with short call holding times.

Keywords: Hardware, Signaling protocols, VHDL, FPGA, SONET/SDH, GMPLS

1. INTRODUCTION

Signaling protocols are used in connection-oriented networks primarily to set up and release connections. Examples of signaling protocols include Signaling System 7 (SS7) in telephony networks¹, User Network Interface (UNI) and Private Network Network Interface (PNNI) signaling protocols in Asynchronous Transfer Mode (ATM) networks^{2,3}, Label Distribution Protocol (LDP)⁴, Constraint-based Routing LDP (CR-LDP)⁵ and Resource reReservation Protocol (RSVP)⁶ in Multi-Protocol Label Switched (MPLS) networks, and the extension of these protocols for Generalized MPLS (GMPLS)⁷⁻¹¹, which supports Synchronous Optical Network (SONET), Synchronous Digital Hierarchy (SDH) and Dense Wavelength Division Multiplexed (DWDM) networks.

Signaling protocols are implemented in the end devices that request the setup and release of connections as well as in the switches of connection-oriented networks. These switches could be circuit-switched, e.g., telephony switches, SONET/SDH switches, DWDM switches, or packet-switched, e.g., MPLS switches, ATM switches, X.25 switches. The end devices requesting the setup/release of connections could be end hosts, e.g., PCs, workstations, or other network switches with interfaces into a connection-oriented network, e.g., Ethernet switches or IP routers with an ATM interface.

Signaling protocol implementations in switches are primarily done in software. There are two important reasons for this choice. First, signaling protocols are quite complex with many messages, parameters and procedures. Second, signaling protocols are updated often requiring a certain amount of flexibility for upgrading field implementations. While these are two good reasons for implementing signaling protocols in software, the price paid is performance. Even with the latest processors, signaling protocol implementations are rarely capable of handling over 1000 calls/sec. Correspondingly, call setup delays per switch are in the order of milliseconds¹².

Towards improving performance, we undertook a hardware implementation of a signaling protocol. We used reconfigurable hardware, i.e., Field Programmable Gate Arrays (FPGAs)^{13,14} to solve the inflexibility problem. These devices are a compromise between general-purpose processors used in software implementations at one end of the flexibility-performance spectrum, and Application Specific Integrated Circuits (ASICs) at the opposite end of this spectrum. FPGAs can be reprogrammed with updated versions as signaling protocols evolve while significantly improving the call handling capacities relative to software implementation. As for the challenge posed by the complexity of signaling protocols, our approach is to only implement the basic and frequently used operations of the signaling protocol in hardware, and relegate the complex and infrequently used operations (for example, processing of optional parameters, error handling, etc.) to software.

* haobo_w@photon.poly.edu; phone (718) 260-3384; mv@poly.edu; phone (718) 260-3493; http://kunene.poly.edu/~mv; ramesh@india.poly.edu; phone (718) 260-3596; fax (718) 260-3740; Polytechnic University, 6 Metrotech Center, NY, USA 11201

We modeled the signaling protocol in VHDL* and then mapped onto two FPGAs on the WILDFORCE™ reconfigurable board – a Xilinx® XC4036XLA FPGA with 62% resource utilization and a XC4013XLA with 8% resource utilization. From the timing simulations, we determined that a call can be processed in 6.6μs assuming a 25MHz clock (this includes the processing time for four signaling messages, *Setup***, *Setup-Success*, *Release*, and *Release-Confirm*) yielding a call handling capacity of 150,000 calls/sec. Optimizing this implementation will reduce the protocol processing time even further.

The impact of this work is quite far-reaching. By decreasing call processing delays, it becomes conceivable to set up and tear down calls more often leading to a finer granularity of resource sharing and hence better utilization. For example, if a SONET circuit is set up and held for a long duration, given that data traffic using the SONET circuit is bursty, the circuit utilization can be low. However, if fast call setup/teardown is possible, circuits can be dynamically allocated and held for short durations, leading to improved utilization.

Section 2 presents background material on connection setup and teardown procedures and surveys prior work on this topic. Section 3 describes the signaling protocol we implemented in hardware. Section 4 describes our FPGA implementation while Section 5 summarizes our conclusions.

2. BACKGROUND AND PRIOR WORK

In this section, as background material, we provide a brief review of connection setup and release. We also describe prior work on this topic.

2.1 Background

An end device that needs to communicate with another end device initiates connection setup. When the ingress switch (e.g., switch SW1 in Figure 1) receives such a request, it uses the destination address carried in the *Setup* message to determine the next-hop switch toward which it should route the connection. This task can be accomplished in different ways. It could be a simple routing table lookup if the routing table is pre-computed. Routing table pre-computation could be done either by a centralized network management station and downloaded to all switches or by a routing process implemented within each switch that processes distributed routing protocol messages and then executes a shortest-path algorithm, such as Bellman-Ford or Dijkstra's¹⁵. Alternately, the signaling protocol processor could perform an on-the-fly route computation upon receipt of a *Setup* message. Typically switches use a combination of pre-computed route lookups and on-the-fly computation if no pre-computed route exists to meet the requirements of the connection.

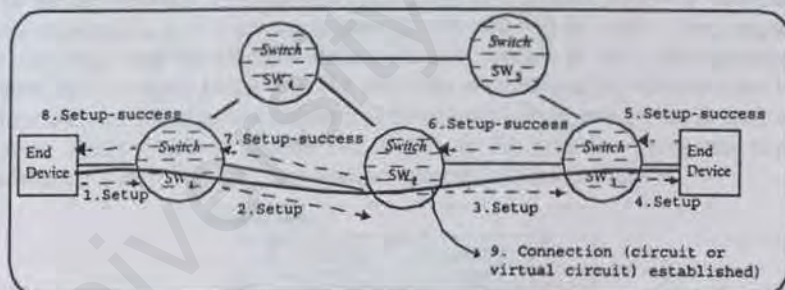


Figure 1: Illustration of connection setup

After determining the next-hop switch toward which the connection should be routed, each switch performs the following four steps:

1. Check for availability of required resources (link capacity and optionally buffer space) and reserve them.
2. Assign "labels" for the connection. The exact form of the "label" is dependent on the type of connection-oriented network in question. For example, in SONET/SDH switches, the label identifies a time slot, while in ATM networks, it is a Virtual Path Identifier/Virtual Channel Identifier (VPI/VCI) pair.

* VHDL stands for VHSIC Hardware Description Language, where VHSIC stands for Very High Speed Integrated Circuits

** Here we use a generic name for the message, i.e., *Setup*. Different signaling protocols call this message by different names, e.g., Label request message in LDP.

3. Program the switch fabric to map incoming labels to outgoing labels. This will allow user data bits flowing on the connection after it is set up to be forwarded through the switch fabric based on these configurations. We refer to this configuration information as a *Switch-Mapping* table.
4. Set control parameters for scheduling and other run-time algorithms. For example, in packet switched networks, if weighted fair queueing is used in the switch fabric to schedule packets, the computed equivalent capacity and buffer space allocated for this connection are used to program the scheduler. Even in circuit-switched networks, such as a SONET network, there could be certain parameters. An example is the transparency requirement for how the SONET switch handles bytes in the overhead portions of the incoming and outgoing signals⁸.

In a classical connection setup procedure as illustrated in Figure 1, the setup progresses from the calling end device toward the called end device, and the success indication messages travel in the reverse direction. In this scenario, the first step should be performed in the forward direction so that resources are reserved as the setup proceeds, but the last three steps could be performed as signaling proceeds in the forward direction or in the reverse direction. Other variants of this procedure are possible such as reverse direction resource reservation⁶.

After connection setup, user-plane data arriving at a switch is forwarded by the switch hardware according to the *Switch-Mapping* table. Upon completion of data exchange, the connection is released with a similar end-to-end release procedure. Typically release messages are also confirmed. Switches processing the release messages free up bandwidth, optionally buffer, and label resources for usage by the next connection.

To support the above-described connection setup and release procedures, signaling messages with parameters in each message, some mandatory and some optional, are defined in a typical signaling protocol. In addition, other messages to support notifications, keep-alive exchanges, etc. are also present in signaling protocols.

With regards to implementation, we illustrate the internal architecture of a switch (unfolded view) in a connection-oriented network in Figure 2. The user-plane hardware consists of a switch fabric and line cards that terminate interfaces carrying user data. In packet switches, the line cards perform network-layer protocol processing to determine how to forward packets. In circuit switches, the line cards are typically multiplexers/demultiplexers. The control-plane unit consists of a signaling protocol engine, which could have a hardware accelerator as we are proposing, or be completely implemented in the software resident on the microprocessor. The routing process handles routing protocol messages and manages routing tables. Network Interface Cards (NICs) are shown in the control-plane unit. These cards are used to process the lower layers of the signaling protocols on which the signaling messages are carried. For example, in SS7 networks, the NICs process the Message Transfer Part (MTP) layers, which are the lower layers of the SS7 protocol stack. In optical networks, the expectation is that an out-of-band IP network will be used to carry signaling messages between switches. In this case, the NICs may be Ethernet cards. It is also possible to carry the signaling messages on the same interface as the user data. An example occurs in ATM networks where signaling messages are carried on VCI 5 within interfaces that carry user data on other virtual channels. Management-plane processing is omitted from this figure, e.g., Management Information Bases (MIBs), agents, etc. Also, all the software processes required for initialization, maintenance of the switch, error handling, etc., and various other details are not shown.

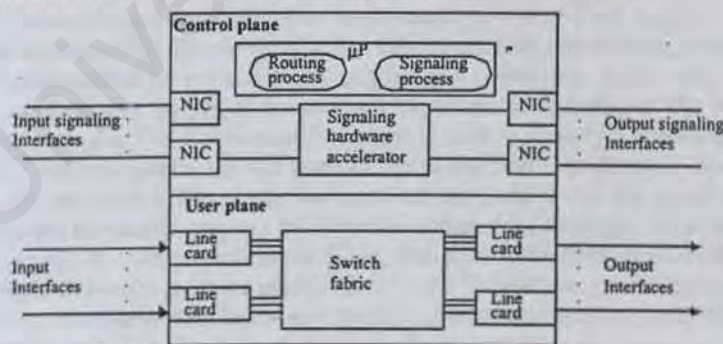


Figure 2: Unfolded view of a switch

We note that the signaling hardware accelerator unit shown in Figure 2 is part of our proposal and not typical in current-day switches. The illustration in Figure 2 shows that the processing of signaling messages is comparable to packet processing in a packet router, where a *Setup* message comes in on one interface and is "forwarded" on another interface;

in reality, many actions are performed on the *Setup* message, which makes the signaling protocol engine more complex than a simple router.

2.2 Prior work

There are many signaling protocols as listed in Section 1. In addition, many other signaling protocols have also been proposed in the literature¹⁶⁻²⁵. Some of these protocols such as Fast Reservation Protocol (FRP)²⁵, fast reservation schemes^{18, 19}, YESSIR¹⁶, UNITE²⁰ and PCC²¹ have been designed to achieve low call setup delays by improving the signaling protocols themselves. FRP is the only signaling protocol that has been implemented in ASIC hardware. Such an ASIC implementation is inflexible because upgrading the signaling protocol implementation entails a complete redesign of the ASIC. More recently, Molinero-Fernandez and McKeown²⁶ are implementing a technique called TCP Switching in which the TCP SYNchronize segment is used to trigger connection setup and TCP FINish segment is used to trigger release. By processing these inside switches, it becomes comparable to a signaling protocol for connection setup/release. They are implementing this technique in FPGAs.

3. SIGNALING PROTOCOL

In this section, we describe the signaling protocol that we implemented in hardware. It is not a complete signaling protocol specification because our assumption is that all aspects of the signaling protocol other than those described below will be implemented in the software signaling process shown in Figure 2. Therefore, often in this description, we will leave out details that are handled by the software.

3.1 Signaling messages

We defined a set of four signaling messages, *Setup*, *Setup-Success*, *Release*, and *Release-Confirm*. Figure 3 illustrates the detailed fields of these four messages.

	Bit 31	1615	Bit 0
Setup Message	Message Length	TTL	Msg.Type (0001) Connection Reference (prev.)
	Destination IP Address		
	Source IP Address		
	Previous Node's IP Address		
	Bandwidth	Reserved	Interface Number Timeslot Number
	Pad Bits		Checksum
Setup-Success Message	Message Length	Bandwidth	Msg.Type (0010) Connection Reference (prev.)
	Connection Reference(own)	Reserved	Checksum
Release/Release-Confirm Message	Message Length	Cause	Msg.Type (0011/0100) Connection Reference (prev.)
	Connection Reference(own)	Reserved	Checksum

Figure 3: Signaling messages

The *Setup* message is of variable length while the other three messages are of fixed length. The *Message Length* field specifies the length of the message. The *Time-to-Live (TTL)* field is used to avoid routing loops. It is initialized by the sender to some value and decremented by every switch along the end-to-end path. If the value reaches 0, a *TTL expired* error is recognized, error handling is in the part of the protocol implemented in software. The *Message Type* field is used to distinguish the different messages. The *Connection Reference* is used to identify a connection locally. The *Source IP Address* and *Destination IP Address* specify the end hosts of the connection. The *Previous Node's IP Address* specifies the previous node along the connection. The reason we included this field is that the lower layers of the protocol on which these signaling messages are carried may not indicate the sender of the message, but a switch would need to know the downstream switch's identity in order to process the *Setup*. The *Bandwidth* field specifies the bandwidth requirement of the connection. The *Interface/timeslot* pairs are used to identify the "labels" assigned to the connection, which are used to program the switch fabric. Since there may be an odd number of interface/timeslot pairs, 16-bit *Pad Bits* field is used to make all messages 32-bit aligned. The *Checksum* field covers the whole message.

In *Setup-Success* message, the *Bandwidth* field records the allocated bandwidth. In *Release* and *Release-Confirm* messages, the *Cause* field explains the reason of release. Some fields are common to all messages, such as *Message Length*, *Message Type* and *Connection Reference*. These fields are in the same relative position for all messages. Such an arrangement simplifies hardware design.

3.2 State transition diagram for a connection at a switch

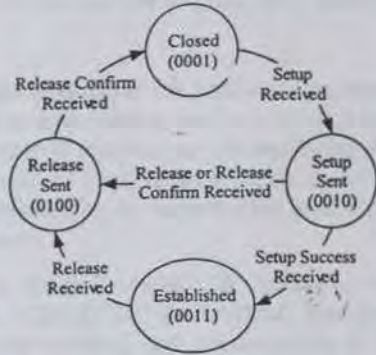


Figure 4: State transition diagram

In connection-oriented networks, each connection goes through a certain sequence of states at each switch. The state of each connection must be maintained at each switch. In our protocol, we define four states, *Setup-Sent*, *Established*, *Release-Sent* and *Closed*. Figure 4 shows the state transition diagram of a connection at a switch. Initially, the connection is in the *Closed* state. When a switch accepts a connection request, it allocates a connection reference to identify the connection, reserves the necessary resources including the labels, programs the switch fabric, marks the state associated with the connection as *Setup-Sent* after sending the *Setup* message to the next switch on the path. When the switch receives a *Setup-Success* message for a particular connection, which means all switches along the path have successfully established the connection, then the state of the connection is changed to *Established*. *Release-Sent* means the switch has received the *Release* message, freed the allocated resources, and sent the outgoing *Release* message to the next node. When the switch receives the *Release-Confirm* message, the connection is successfully terminated, and the state of the connection returns to *Closed*.

3.3 Data tables

Routing table	Index		Return value				
	Destination address		Next node address		Next node interface#		
CAC table	Index		Return/Written value				
	Next node address		Total bandwidth		Available bandwidth		
Conn. table	Index			Return value			
	Neighbor address		Neighbor interface#		Own interface#		
State table	Index		Return/Written value				
	Own connection reference	Connection reference		State	Bandwidth	Node address	
		Previous	Next			Previous	Next
Switch mapping table	Index		Return/Written value				
	Own connection reference	Sequential offset(0 to BW-1)	Incoming Ch. ID		Outgoing Ch. ID		
			Interface#	Timeslot#	Interface#	Timeslot#	

Figure 5: Data tables used by the signaling protocol

There are five tables associated with the signaling protocol, namely, *Routing* table, *Connection Admission Control (CAC)* table, *Connectivity* table, *State* table, and *Switch-Mapping* table, shown in Figure 5. The *Routing* table is used to determine the next-hop switch. The index is the destination address; the fields include the address of the next switch and the corresponding output interface. The *CAC* table maintains the available bandwidth on the interfaces leading to neighboring switches. The *Connectivity* table is used to map the interface numbers used at neighboring switches to local interface numbers. This information will be used to program the switch fabric.

The *State* table maintains the state information associated with each connection. The connection reference is the index into the table. The fields include the connection references and addresses of the previous and next switches, the bandwidth allocated for the connection, and most importantly, the state information as defined in Figure 4.

Switch fabrics, such as PMC-Sierra's PM5372, Agere's TDCS6440G and Vitesse's VSC9182, have similar programming interfaces. For example, VSC9182 has an 11-bit address bus A[10:0] and a 10-bit data bus D[9:0]. The switch is programmed by presenting the output interface/timeslot number on A[10:0] and the input interface/timeslot number on

D[9:0]. We define a generic *Switch-Mapping* table to emulate this programming interface, with the connection reference as the index, the incoming interface/timeslot pair and the outgoing interface/timeslot pair as the fields.

3.4 Discussion

Aspects of signaling protocols that make it difficult for hardware implementation include the maintaining of state information, the usage of timers, the need to initiate messages from a switch instead of simply forwarding messages (e.g., a release message aborting a connection setup if resources are not available), the Tag-Length-Value (TLV) structure used to carry parameters within messages instead of fixed location fields, choices specific to parameters (e.g., values made global or local significance), and most importantly, the current drive toward generalizing protocols with goal of making them applicable to a large variety of networks.

Starting with the last reason first, consider the evolution of LDP. It has evolved from LDP to CR-LDP to CR-LDP with extensions for GMPLS networks, such as SONET/SDH and DWDM. This complex protocol is now targeting almost all connection-oriented networks both packet-switched and circuit-switched. This drive impacts almost all fields in parameters within messages. For example, the address field identifying the destination address of the connection allows for different address families, IP, telephony E.164, ATM End System Addresses, etc. Next, with regards to choices made for specific parameters, consider a simple parameter such as a connection identifier or connection reference. Most signaling protocols have this parameter. If this is chosen to be globally unique, then connection related data tables need to be searched with a much larger key than if this is chosen to be locally significant. Next, the TLV structure was designed for flexibility, allowing protocol designers to add parameters in arbitrary order. But this construct makes parameter extraction in hardware a complex task. Finally, with regards to state information, signaling protocol engines have to maintain the states of a connection as shown in Figure 4. While the type of state information is quite different, the notion of maintaining some state information is already in practice in IP packet and ATM cell forwarding engines for policing purposes. Other aspects that complicate signaling protocols are the support for a variety of procedures, such as third-party connection control and multiparty connection control.

The signaling protocol described in this section is limited to the part implemented in hardware. Thus, the specification of error handling, aborting setups for lack of resources, checking timers, handling connections more complex than simple two-party connections, etc. have been delegated to the remaining part of the protocol implemented in software. Our approach is to define a large enough subset of the protocol that a significant percentage of users' requirements can be handled with this subset. Infrequent operations are delegated to the slower software path. Nevertheless, there are many aspects of the complex CR-LDP-like protocols that we have omitted here. Examples include TLV processing, handling larger parameters (such as global connection references, called "label-switched path identifier" in CR-LDP), handling many choices such as the different types of addresses, etc. We are currently implementing CR-LDP for SONET networks in VHDL for an FPGA implementation. This is an NSF-sponsored project²⁷. At the end of that experiment, we hope to answer the question of whether a complex signaling protocol such as CR-LDP can be implemented in this mode of handling frequent operations in hardware and infrequent operations in software, or whether simpler lightweight signaling protocols targeted for specific networks need to be defined, as we have done here.

4. FPGA-BASED IMPLEMENTATION OF SIGNALING PROTOCOL

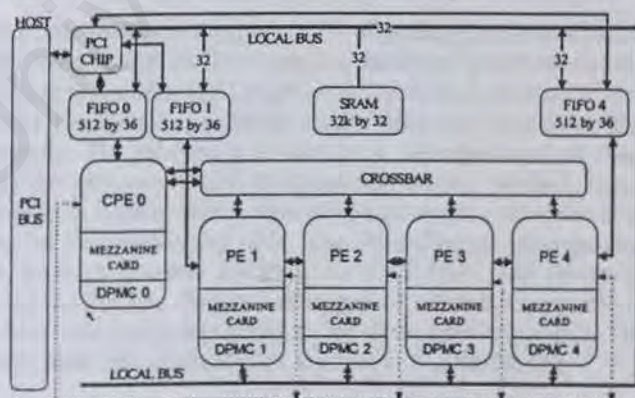


Figure 6: Architecture of WILDFORCE™ board

To demonstrate the feasibility and advantage of hardware signaling, we implemented a signaling hardware accelerator in FPGA. We used the WILDFORCE™ multi-FPGA reconfigurable computing board shown in Figure 6, which consists of five XC4000XLA series Xilinx® FPGAs, one XC4036XLA (CPE0) and four XC4013XLA (PE1-PE4). These five FPGAs can be used to implement user logic while the crossbar provides programmable interconnections between the FPGAs. In addition, there are three FIFOs on the board, and one Dual Port RAM (DPRAM) attached to CPE0. The board is hooked to the host system through PCI bus. The board supports a C language based API through which the host system can dynamically configure the FPGAs and access the on-board FIFOs and RAMs.

Figure 7 illustrates our prototype implementation. We use CPE0, PE1, FIFO0, FIFO1 and DPRAM. The CPE0 implements the signaling hardware accelerator state machine, the *State* and *Switch Mapping* tables, FIFO0 controller, and DPRAM controller. The DPRAM implements the *Routing*, *CAC* and *Connectivity* tables. FIFO0 and FIFO1 work as receive and transmit buffers for signaling messages. PE1 implements the FIFO1 controller and provides the data path between CPE0 and FIFO1.

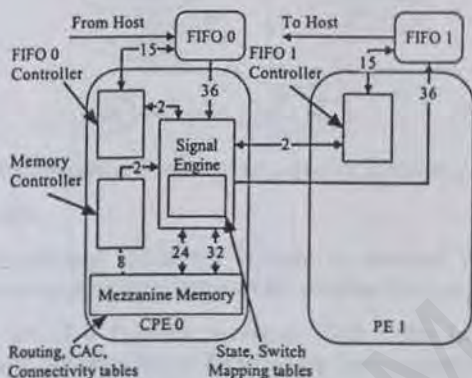


Figure 7: Implementation of signaling protocol on WILDFORCE™ board

In the following subsections, we describe the design consideration about routing table, and the state transition diagram of the hardware accelerator. We also present two novel approaches for managing timeslots and connection references.

4.1 Routing table look up

In recent years, there has been significant progress in fast table lookup in both research literature and commercial products²⁸⁻³⁰. Lookup co-processors are widely available, such as Silicon Access Networks' iAP, SiberCore Technologies' Ultra-9M, Netlogic Microsystems' NSE4256, MOSAID Semiconductor's DC9288, etc. These chips can easily process up to 100 million lookups/sec³⁰. In our prototype implementation, we assumed that routing table lookups can be offloaded to an external co-processor, and used equivalent three memory accesses to emulate a routing table lookup.

4.2 State transition diagram of the signaling hardware accelerator

Figure 8 shows the detailed state transition diagram of the signaling hardware accelerator. When a signaling message arrives, it is temporarily buffered in FIFO0. The signaling hardware accelerator reads the messages from FIFO0 and delimits the messages according to the *Message Length* field. The *Checksum* field is verified. The *State* table is consulted to check the current state of the connection. Based on the *Message Type* field, the signaling hardware accelerator processes messages accordingly. The processing of the *Setup* message involves checking the *TTL* field, reading the *Routing* table to determine the next switch and corresponding output interface, updating the *CAC* table, reading the *Connectivity* table to determine the input interface, allocating a connection reference to identify the connection, allocating timeslots and programming the *Switch-mapping* table. The *Setup-Success* message requires no special processing. The processing of the *Release* message involves updating the *CAC* table, and releasing the timeslots reserved for the connection. When processing the *Release-Confirm* message, the allocated connection reference is freed and thus, the connection is terminated. After processing any message, the *State* table is updated. The new message is generated and buffered in FIFO1 temporarily, and then transmitted to the next switch on the path.

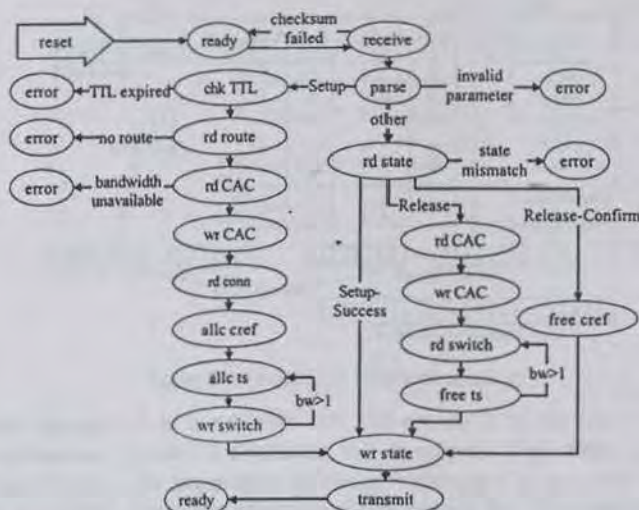


Figure 8: State transition diagram of the signaling hardware accelerator

4.3 Managing the available timeslots

The management of timeslots and connection references is easy in software through simple array manipulations. However, this poses a challenge in hardware implementations. Our solution is to use a priority decoder.

Figure 9 illustrates our implementation of a timeslot manager. Each entry in the timeslot table is a bit-vector, corresponding to an output interface with the bit-position determining the timeslot number and the bit-value determining availability of the timeslot ('0' available, '1' used). The priority decoder is used to select the first available timeslot. When an interface number is provided by the signaling state machine to the timeslot manager, the bit-vector corresponding to the interface is sent into the priority decoder and the first available timeslot is returned. Then the bit corresponding to the timeslot is marked as used (from 0 to 1) and the updated bit-vector is written back to the table. In the example shown in Figure 9, the timeslot manager was asked to find a free timeslot on interface 3. It returns timeslot 14 and marks it as 'used.' De-allocating a timeslot follows a similar pattern but the timeslot number is needed as an input in addition to the interface number in order for the timeslot manager to free the timeslot.

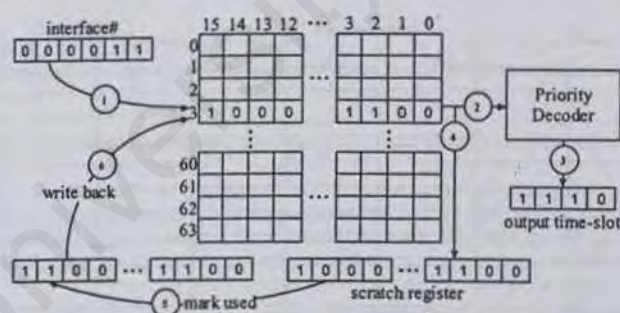


Figure 9: Timeslot manager

4.4 Managing the connection references

A connection reference is used to identify a connection locally. It is allocated when establishing a connection and de-allocated when terminating it. A straightforward implementation of a connection reference manager is a bit-vector combined with a priority decoder. The priority decoder finds the first available bit-position (a bit marked as '0'), sends its index as the connection reference and updates the bit as used (a bit marked as '1'). However, this approach is impractical when there are a large number of connections. While our actual implementation only used 32 connections per switch, we designed the connection reference manager to handle 2^{12} simultaneous connections, which requires a bit-vector with 4096 entries. This is too large for the simple priority decoder implementation as used for timeslots.

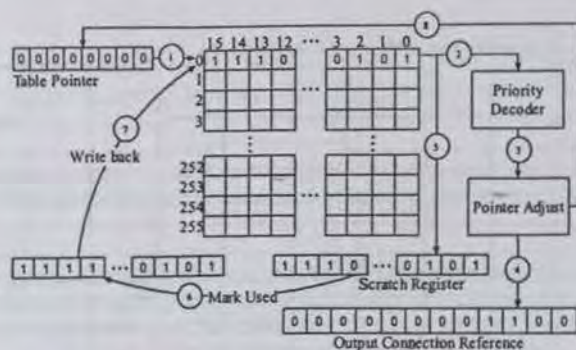


Figure 10: Connection reference manager

Our improvement to this basic approach is to use a table with 256 entries of 16-bit vectors to record the availability of a total of 4096 connection references. Figure 10 illustrates this approach. With 4096 connections, we need a 12-bit connection reference. The first 8 bits of the connection reference correspond to the table pointer, while the remaining 4 bits correspond to the first available connection reference from among the 16 pointed to by the table pointer. The connection reference manager starts with the table pointer set to 0. If any of the 16 connection references corresponding to this row of the table are available (i.e., a bit position is 0), the priority decoder will identify this index and write the output connection reference as a concatenation of the 8-bit table pointer and the 4-bit index extracted. In the example shown in Figure 10, the 12th bit in the first row is a 0. Therefore it outputs the connection reference number 12. The bit-position is marked as used as illustrated with steps 5 – 7 of Figure 10.

De-allocating follows a similar approach; the bit corresponding to the connection reference is reset to 0 and the updated bit-vector is written back to the table. We can parallelize this approach by partitioning the table into several smaller tables, each with a pointer and a priority decoder, forming several smaller managers. All these managers work concurrently. A round-robin style counter can be used to choose a connection reference among the managers. Thus, this approach can be generalized if more than 4096 connections are to be handled.

4.5 Simulation

We developed a prototype VHDL model for the signaling hardware accelerator, used Synplify[®] for synthesizing the design and Xilinx[®] Alliance for the placement and routing of the design. CPE0 (Xilinx[®] XC4036XLA FPGA) uses 62% of its resources while PE1 (XC4013XLA) uses 8% of its resources.

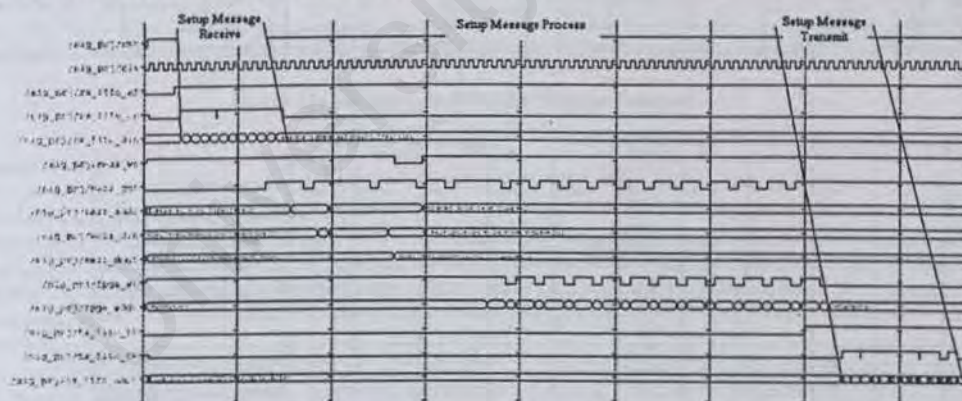


Figure 11: Timing simulation for Setup message

We performed timing simulations of the signaling hardware accelerator using ModelSim[®] simulator. The simulation results are shown in Figure 11-Figure 14. From the timing simulation of the Setup message (Figure 11), it can be seen that while receiving and transmitting a Setup message (requesting a bandwidth of OC-12 at a cross connect rate of OC-1) consumes 12 clock cycles each, processing of the Setup message consumes 53 clock cycles. Overall, this translates into 77 clock cycles to receive, process and transmit a Setup message.

Processing *Setup-Success* (Figure 12), *Release* (Figure 13) and *Release-Confirm* (Figure 14) messages consumes about 70 clock cycles total since these messages are much shorter (2 32-bit words versus 11 32-bit words for *Setup*) and require simpler processing. A detailed breakdown of the clock cycles consumed to process each of these signaling messages is shown in Table 1.

	Setup	Setup Success	Release	Release Confirm
Clock cycles	77-101*	9	51	10

Table 1: Clock cycles consumed by the various messages.

Assuming a 25 MHz clock, this translates into 3.1 to 4.0 microseconds for *Setup* message processing and about 2.8 microseconds for the combined processing of *Setup-Success*, *Release* and *Release-Confirm* message. Thus, a complete setup and teardown of a connection consumes about 6.6 microseconds. Compare this with the millisecond-based software implementations of signaling protocols. We are currently optimizing the design to operate at 100 MHz thereby reducing the processing time even further. We are also exploring pipelined processing of signaling messages by selectively duplicating the data path to further improve the throughput.

5. CONCLUSIONS

Implementation of signaling protocols in hardware poses a considerably larger number of problems than implementing user-plane protocols such as IP, ATM, etc. Our implementation has demonstrated the hardware handling of functions such as parsing out various fields of messages, maintaining state information, writing resource availability tables and switch mapping tables, etc., all of which are operations not encountered when processing IP headers or ATM headers. We also demonstrated the significant performance gains of hardware implementation of signaling protocols, i.e., call handling within a few μ s. Overall, this prototype implementation of a signaling protocol in FPGA hardware has demonstrated the potential for 100x-1000x speedup vis-à-vis software implementations on state-of-the-art processors. Our current work is implementing CR-LDP for SONET networks in hardware.

ACKNOWLEDGMENTS

This work is sponsored by a NSF grant, 0087487, and by NYSTAR (The New York Agency of Science, Technology and Academic Research) through the Center for Advanced Technology in Telecommunications (CATT) at Polytechnic University.

We thank Reinette Grobler for helping specify the signaling protocol, and Brian Douglas and Shao Hui for the initial prototype VHDL model of the protocol.

REFERENCES

1. Travis Russell, *Signaling System #7, 2nd edition*, McGraw-Hill, New York, 1998.
2. The ATM Forum Technical Committee, "User Network Interface Specification v3.1," af-uni-0010.002, Sept. 1994.
3. The ATM Forum Technical Committee, "Private Network-Network Specification Interface v1.0F (PNNI 1.0)," af-pnni-0055.000, March 1996.
4. L. Andersson, P. Doolan, N. Feldman, A. Fredette, B. Thomas, "LDP Specification," IETF RFC 3036, Jan. 2001.
5. B. Jamoussi (editor), et al., "Constraint-Based LSP Setup using LDP," IETF RFC 3212, Jan. 2002.
6. R. Braden, L. Zhang, S. Berson, S. Herzog, S. Jamin, "Resource ReSerVation Protocol (RSVP) Version 1 Functional Specification," IETF RFC 2205, Sept. 1997.
7. E. Mannie (editor), "GMPLS Architecture," IETF Internet Draft, draft-many-gmpls-architecture-00.txt, March 2001.
8. E. Mannie (editor) et al., "GMPLS Extensions for SONET and SDH Control, IETF Internet Draft, draft-ietf-ccamp-gmpls-sonet-sdh-01.txt, June 2001.

*Based on a worst-case search through a four option routing table

Hardware/Software-Architecture and High Level Design Approach for Protocol Processing Acceleration

Mirko Benz, Georg H. Overbeck
Department of Computer Science
Dresden University of Technology
D-01062 Dresden, Germany
{benz, overbeck}@ibdr.inf.tu-dresden.de

Klaus Feske, Jens Grusa
FhG IIS Erlangen
Department EAS Dresden, Zeunerstr. 38
D-01069 Dresden, Germany
{feske, grusa}@eas.iis.fhg.de

Abstract

Developing hardware support for transport layer protocol processing is a very complex and demanding task. However, for optimal performance hardware acceleration can be required. To cope with this situation we present a high level design approach which targets the development of configurable and reusable components. Therefore we outline the integration of advanced tools for the development of controller systems into our design environment. This process is illustrated based on a TCP/IP header analysis and validation component for which initial performance results are presented. The development of these specialised components is embedded in an approach to develop flexible and configurable protocol engines that can be optimised for specific applications.

1 Introduction and Related Work

Today's communication environments are mainly influenced by the tremendous success of the Internet. As a result the Internet Protocol (IP) and standard layers above - especially TCP [18, 19] - are now the common denominator. This means that although these protocols have a number of limitations concerning functionality, flexibility and performance other protocol approaches like XTP [17] have failed to gain broad acceptance. This is also partly true for other superior technologies like ATM which compete with IP. Hence it is important to transfer the alternatives and ideas developed in various research projects to improve implementations of these standard protocols.

On the other hand, the Internet has encouraged huge investments in fibre optical networks and technologies to exploit them more efficiently like Wave Division Multiplex (WDM). Furthermore, new technologies like xDSL and cable modems will also provide high speed communication in the access networks. Altogether this will contribute to an emerging global high speed networking infrastructure based on IP.

In contrast to using the same base protocol everywhere, communication devices are extremely diversified. This includes standard workstation and server class computers as well as laptops up to Wireless Application Protocol (WAP) mobile phones. Therefore, architectures for protocol processing acceleration have to be adaptive to various network interfaces, their properties as well as processor architectures or optimisation goals concerning performance as well as memory, CPU and power limitations. Furthermore, the ongoing development within and the commercialisation of the Internet have produced quite a number of applications, protocols and service proposals and standards for higher layers or extensions to IP. Examples are IP security for virtual private networks (VPN), voice over IP, video conferencing or the WWW. Especially the real-time requirements of multimedia data transmission stimulated the

development of resource reservation protocols, priority mechanisms, accounting or the differentiated services approach.

The ongoing research and deployment of WDM technologies and the direct transmission of IP datagrams over specific wave lengths, will contribute to very high bandwidth capacities at low error rates. These optical networks will again shift the protocol processing overhead to the access routers and into the end systems. On the other hand they will probably provide no or only limited quality of service (QoS) features. Combined with data touching intensive or real time requirements of specific services this adds to processing power that is required within endsystems. Hence, flexible architectures for protocol processing acceleration are necessary to cope with these conditions.

There have been quite a number of approaches for hardware support of communication protocols in the early nineties. However they were only successful for lower layer protocols (e.g. MAC sublayer) [1, 13, 9]. The most significant aspect is probably the complexity of standard communication stacks like TCP/IP. This makes it impossible to perform all the processing in custom hardware because the design, implementation, testing, validation and maintenance effort would be too high. This would also lead to extreme costs and limit the ability to adopt standard modifications and improvements. Another obstacle is the lack of a formal specification. Hence, many research projects had concentrated on the hardware support of specialised light weight protocols [8, 15]. Although this was successfully demonstrated, hardware support for complex transport level protocols is still an open issue.

On the other hand, hardware support for communication protocols is again a very active topic. Especially so-called network processors are very popular. They include custom hardware for standard protocol specific computations as well as multiple programmable RISC cores, which makes them more flexible than custom ASIC designs that are used in routers today. This approach provides benefits concerning time to market and allows to continue development after purchase and to update the devices as required. This development is supported by improved tool support for simulation, verification and synthesis, because of the expansion of the component idea onto the hardware design in the form of intellectual property cores and subsequent reuse as well as higher abstraction levels like hardware compilation approaches. An example is the integration of high level design tools [20] in the process of developing hardware components. This is shown in [16, 7] and can improve productivity. Furthermore ASIC and especially FPGA technologies for hardware prototyping were drastically improved. Combined, these achievements facilitate the design and implementation of complex and heterogeneous hardware/software architectures for protocol processing acceleration and system-on-a-chip solutions. One such example is the hardware support for ATM, which often integrates transport layer functionality like segmentation and reassembling, congestion control (ABR) or traffic shaping in hardware.

In the following chapter we outline our protocol engine approach. First we state design goals and present our general approach. Then we outline the aspired architecture and discuss possible configurations and describe the involved components. In chapter 3 we illustrate a possible TCP/IP partitioning, explain the required protocol processing and outline a synchronisation between the software and hardware parts. Then, we describe our validation architecture for a hardware implementation of TCP/IP core functionality and present initial evaluations and performance data.

2 Protocol Engine Project

The protocol engine project is a joint effort of multiple research groups with a computer science and electrical engineering background. In this context communication protocols are analysed, evaluated and optimised. Specific protocols tailored for ATM networks and multimedia applications have been developed. For existing standard protocols hardware support is evaluated and designed. The focus of this paper is to describe the basic approach and architecture of the protocol engine with hardware support for TCP/IP in mind as well as a prototype implementation. The overall goal however, is to see networking in its entirety - ranging from applications, over protocols to the actual hardware. This way, by not looking at one specific layer alone, system performance can be improved.

2.1 Assumptions, Approach and Design Goals

Today's transport protocols like TCP are far too complex to be completely implemented in custom hardware. Furthermore this approach would limit the flexibility and maintainability of the solution. On the other hand, only a very small part of the protocol has real-time processing requirements. The rest consists of lower priority tasks like exception handling, buffer registration or connection management. This means, that only a relatively small part of the protocol has to be accelerated.

Within modern local high speed networks the error probability is very low. Hence exception handling due to corrupted data, loss or duplicates can be considered as a rare condition. As a consequence, non real-time processing like connection setup or exception handling in case of errors can still be performed by a modified software stack because relatively expensive synchronisation is tolerable. This way, investments in high performance TCP/IP implementations can be reused. Another advantage is that a step wise optimisation - based on existing implementations - and tailored to specific requirements is possible.

Advances in CPU development have greatly improved the processing power that is available in endsystems. However, due to added services which demand very high processing capacity or exhibit very tight real-time requirements hardware support could still make sense. Another issue is that the operating system becomes a bottleneck for simple protocol processing tasks because of context changes, synchronisation or the communication overhead inherent in layered architectures relative to the protocol processing itself. Hence the data path between the application and the network is very important too. From these requirements and constraints we derive the following design goals:

- Flexible, adaptive architecture to support different protocols based on IP. Allow easy extensions to support new features and services.
- Scalable performance according to specific requirements like cost, power consumption and network conditions.
- Development of components for specific protocol functions. Design of common interfaces to allow reuse, hardware emulation in early design phases and flexibility concerning the implementation architecture.
- Allow integration of specific hardware to benefit from existing solutions or for very high performance requirements. Furthermore enable easy integration of additional processing resources like DSPs or micro controllers.
- Take advantage of existing software based protocol implementations and allow stepwise hardware support.

- Consider protocol processing as a whole. This means that the network interface, the protocol processing and the communication with the application have to be regarded and optimised in its entirety.
- On wire compatibility to other (software based) implementations.
- Enable existing applications to take advantage of the protocol processing acceleration in a transparent manner.

2.2 Architecture and Configuration Opportunities

Due to the number of requirements and environmental conditions a protocol engine has to be designed specifically to address these challenges. As a consequence there will be many configurations. The general idea however is to add hardware support as required, to allow a smooth upgrade path and enable scalability. Figure 1 presents the overall architecture of the protocol engine with all possible components.

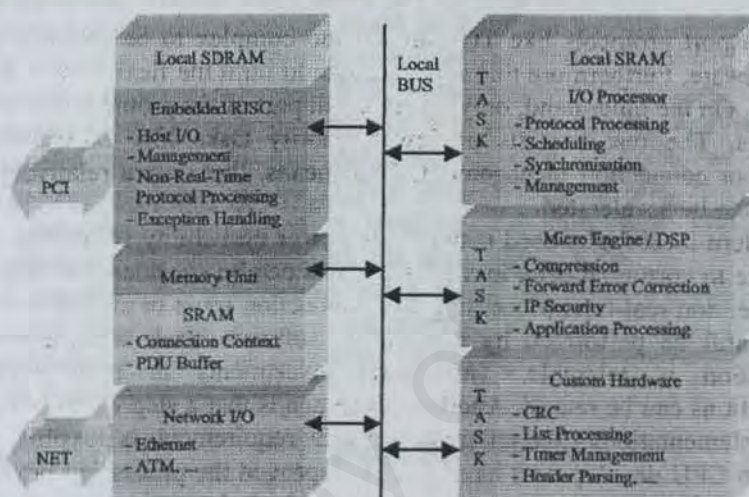


Fig. 1: General Protocol Engine Architecture

Depending on the specific application requirements there can be various configurations of this protocol engine. One opportunity consists in using currently emerging network processor designs which contain a standard embedded RISC processor combined with multiple programmable micro engines and very few network specific hardware like the LEVEL ONE IXP1200 architecture [11]. These architectures are especially suited for technology demonstrations of emerging standards like the integrated services approach. Because they are fully programmable, specification modifications can be quickly adopted. Furthermore they offer relatively high performance.

Another direction would be to leave the majority of the protocol processing tasks on the host and support only the standard data path with specific hardware. The high bandwidth of today's networks combined with extremely low error rates makes this approach feasible. As a consequence only in the rare case of exceptions due to error or connection management synchronisation handling would be required. Supported by a high level design tool a scheme for the development of such protocol accelerators was outlined in [4] for TCP/IP packet validation within the receive path.

For greater flexibility we plan the integration of a specific I/O processor [5]. This processor will possess a customisable instruction set. Hence it can be optimised for stream parsing and bit operations for example. This can lead to a reduction of code size and required processing cycles which is a general goal for power consumption sensitive applications.

On the other hand we plan to use the I/O processor to control the operation of the protocol engine. This means that it can be programmed to control the processing flow. As such it must communicate with the other entities of the protocol engine. Hence it must have knowledge of the number and capabilities of the integrated components and optimise their utilisation. We plan to develop a message abstraction layer for the exchange of such processing requests (TASK). This enables the I/O processor to communicate with other processing elements without having to know their implementation architecture in an asynchronous fashion. Therefore, this adaptation contains a common message buffer and a bus access part as well as a specific interface to the component. This way, specific hardware could be emulated during early stages of the development process. This could be a very high performance DSP or protocol specific hardware. On the other hand this abstraction opens the way to scalability since multiple components of the same type could be integrated as well. In this configuration the I/O processor would be responsible to synchronise the processing results of the entities and communicate with a standard embedded RISC processor.

Due to the complexity of transport layer protocols it is usually not beneficial to perform the entire protocol processing in specialised components. Hence we intend to use a general processor which runs a modified software TCP/IP stack. Here non real-time processing tasks like connection management or exception handling are performed. Furthermore, it is responsible for efficient communication with the application on the host system. To avoid operating system overhead we consider to use a modified implementation of the virtual interface architecture (VIA) for efficient communication [21]. Further information, for example how applications could transparently – without modifications – benefit from this acceleration can be found in [2, 3].

3 TCP/IP Partitioning and Fast Path Processing

Judged by the lines of code TCP is a relatively complex protocol [12]. However, assuming bulk data transfer within local area networks only a fraction of the code is actually required to process most packets. This is called the fast path. This state is reached after the connection is established and holds on as long as normal data packets with no control flags in the header are transmitted. A further requirement is the absence of error conditions due to loss, congestion or data corruption. Most of these conditions are relatively rare within today's local high speed networks. To achieve good performance it is therefore necessary to optimise the fast path. As a consequence it is not necessary or beneficial to implement complex protocols completely in hardware but to only support the common path with specific accelerators. Hence the majority of the protocol will still reside in software.

Within the fast path the sending instance checks whether available data can be transmitted. This is for example the case if the transfer unit of the network can be fully utilised. Large messages can be fragmented or small ones are aggregated. Thus, message boundaries are not preserved. The header fields are set and a checksum is computed then IP is invoked for the actual transmission. At the receiver, fast path processing consists of header analysis, context lookup, checksum calculation, packet validation, packet reassembling and hand over to the higher layers. Important optimisations include header prediction, context caching and integration of checksum computation in buffer copy operations [10, 14]. As a consequence of

these optimisations it normally takes very few instructions to process a packet. Assuming faster and faster processors while the protocol processing remains essentially the same the ratio between actually required processing and overheads is getting worse. Since protocol processing is not the bottleneck (if not additional services like IP security are used) user mode implementations present an alternative. However, without intelligent hardware support they can not fully exploit today's networks.

3.1 Fast Path Protocol Processing

Figure 2 illustrates the tasks that are involved in the fast path processing for bulk data transmission. The sender accepts the application's data and transmits them if certain criteria are met. The receiving protocol instance takes and validates the data and eventually hands them over to the application. Both protocol instances are coupled by a window based flow control that ensures that enough buffer space is available at the receiver. Usually for every other received protocol data unit the receiver generates an acknowledgement. Based on this information the sender can release transmitted data that was saved for eventual retransmission. Furthermore, the transmission window is enlarged enabling the transmission of new data.

The fast path consists of three major components: `TcpSend`, `TcpRecv` and `SendAck`. These tasks may run concurrently but since they access shared context data, synchronisation has to be applied. Furthermore, they signal each other required processing like the receiver indicating necessity of sending an acknowledgement. Depending on the communication behaviour only some tasks are active on each instance. The context data include information describing the connection and variables for flow and congestion control as well as for error detection. These data sets are kept separate for every connection. Hence connections can be processed concurrently. Statistics data however, are gathered for every connection and thus have to be periodically synchronised with the software stack.

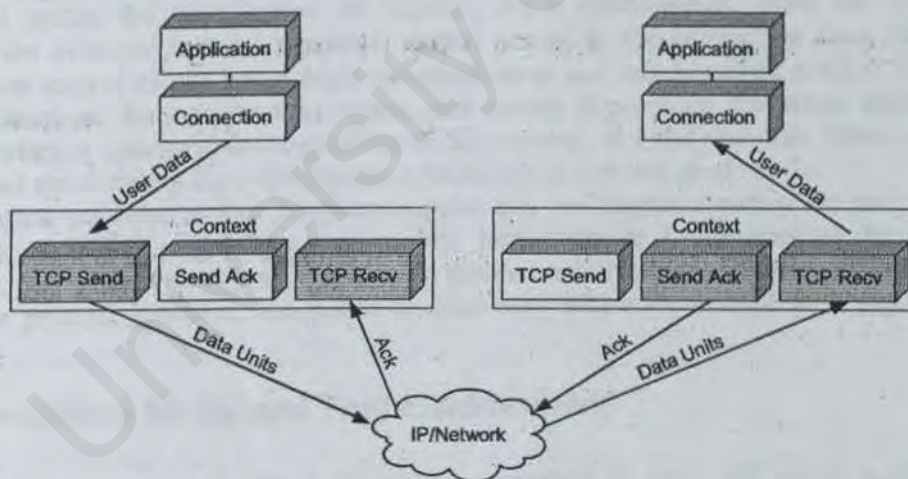


Fig. 2: TCP Fast Path Processing Flow for Bulk Data Transfer

3.2 Software Stack Synchronisation

According to the proposed partitioning the complex tasks of connection management and error handling are still performed in software. Thus, we can benefit from existing well performing and stable implementations. Therefore, when an application initiates a connection establishment the software stack is invoked. On the receiving side the protocol data unit can not be mapped onto a known fast path entry, therefore it is passed to the software stack. If the user decides to accept the communication the context information for this connection is transferred to the fast path processing unit and marked active. The same happens on the client side. The number of connections that should be accelerated may be limited to participants of the same or corresponding networks because normally only here very high performance is required and the above mentioned conditions are met. The data units that are exchanged afterwards are entirely processed by this unit and transferred directly to the application. This happens without invoking the operating system or the TCP/IP software stack. This means that no interrupts have to be dealt with, no operating system contexts or processor modes have to be changed and caches could remain intact.

When a connection is idle for some time, errors occur or the user terminates it, the context information and remaining user data are transferred to the software stack. Then the communication is treated as it would have been without an accelerator. In case an error condition was successfully managed, the fast path unit could be reinitialised for this connection.

4 A TCP/IP Receive Side Processing Component

Hardware support and parallelism to improve protocol processing performance were not considered within the specification of TCP/IP. As a consequence, there are a lot of dependencies between protocol functions, shared access to the connection state data (the transmission control block) and a high communication and synchronisation effort between protocol functions. To make matters worse, this mainly depends on the actual transported data. Therefore, a functional decomposition of the protocol is a difficult task. Hence, a prior analysis and simulation of the entire system is necessary to achieve good results.

In this section we describe how hardware support can be integrated within the receive path processing of TCP/IP. Again, we assume low error rates and high network bandwidth. Therefore, the required processing is mainly to assure error free data reception and hand over to the user process. However, this has to be done very efficiently to cope with gigabit data rates.

4.1 Simulation Model and Test Environment

Figure 3 represents the simulation test bench currently in use. The major goals were prototyping of the fast path unit, exploration of different configurations, bottleneck investigation and derivation of initial performance data. For this to be accomplished, the interface to the network (e.g. MAC in the case of Ethernet) is currently emulated. Furthermore, only one connection is actively processed.

All models except the interfaces to the application and to the remote TCP/IP instance are written in VHDL. The SRam model allows to dynamically update its contents via a file

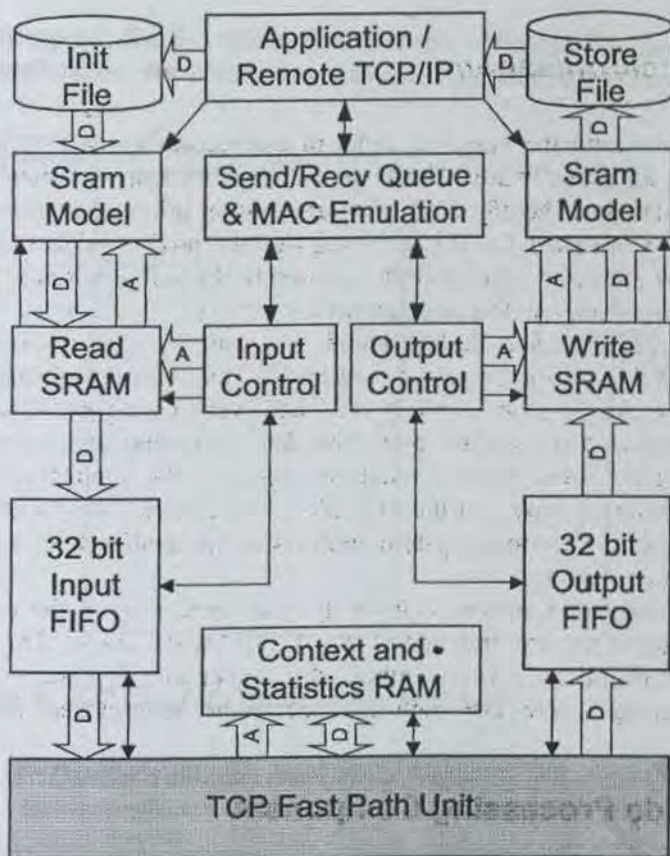


Fig. 3: Simulation Model

received the packet. After the packet is transferred the queue entry is cleared and can be reused by the application. The fast path unit is invoked if a configurable threshold for the Input FIFO filling is reached. The processing of this unit is described in the next section. Here a local SRam can be utilised to store connection and statistic information. The data extraction from the Output FIFO and the transfer to the application or network emulation works in a similar manner. The models for FIFO elements and the local Ram target Block Select Ram of Xilinx Virtex FPGA devices.

4.2 TCP Fast Path Unit Receive Processing Flow

Figure 4 represents the general processing flow within the TCP fast path unit. After leaving the reset state the unit communicates with the input FIFO and waits for available data. It then checks whether the received packet contains synchronisation information. If this is the case, the connection context data is initialised. Since currently only the receive path is implemented a filter for those packets is inserted within the processing flow. Other packets are directly transferred to the output FIFO. The next step consists of extracting the connection description (IP addresses and TCP ports) and a comparison with the currently loaded context information. If the packet does not belong to the accelerated connection it is simply forwarded. Otherwise it is analysed as explained in section 3.1. Within the data validation step the TCP checksum is

interface. This is for example used by the application interface when a packet should be transmitted. First the user data is transferred to the init file. Then the contents of the SRam is updated. The position and length of this data is stored in a transmission queue that is available for each connection. Via this queue a synchronisation between the software application and the hardware unit is performed. The input control generates a header and transfers the described buffer to the Input FIFO. The header contains a packet type field. This could be application (send/rcv), network (send/rcv) as well as synchronisation and statistics. Furthermore, it contains the length and packet specific data - for example the connection to which this packet belongs or the network interface that

computed for the entire packet. If this succeeds the data is transferred to the corresponding application. Furthermore an acknowledgement packet for the received data is generated.

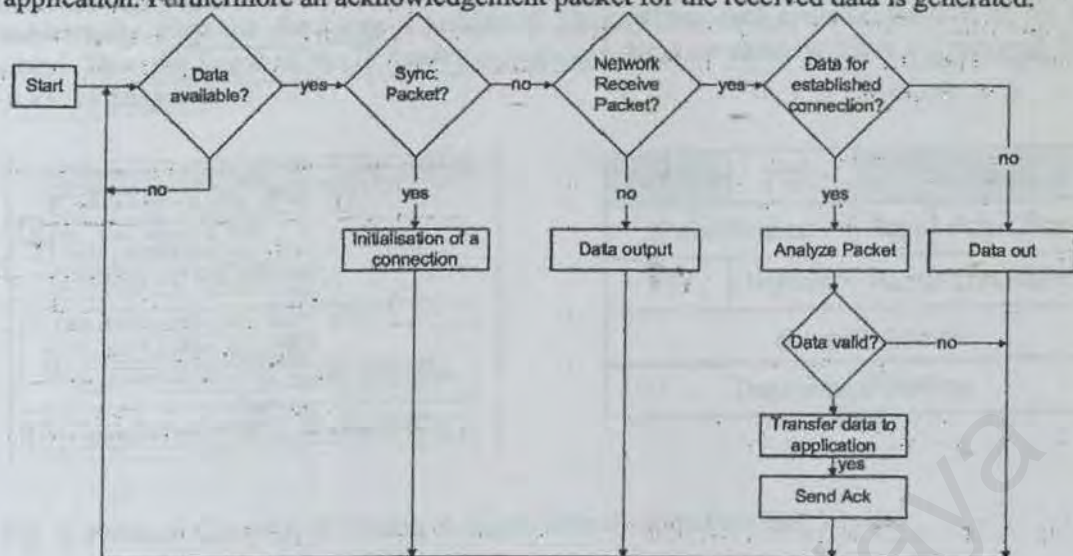


Fig. 4: TCP Fast Path Unit Processing Flow

4.3 Design Flow, Implementation and First Results

Conventionally, the controller design of structured data stream processing is not well supported by EDA tools. So we are facing a bottleneck in the design process especially for protocol processing hardware components. To fill the gap, we incorporated modelling and synthesis facilities of the Protocol Compiler from Synopsys [20] into a proved FPGA based rapid prototyping (RPT) design flow [6] and utilised it for designing fast path protocol processing components.

Our RPT design flow starts at RT-Level with a VHDL description of the design specification. As a new element in the sequence of this flow, the Protocol Compiler is set on top of the whole process, by means of which the high level specification is graphically composed. Furthermore the Protocol Compiler provides the following features: formal protocol analysis, back annotation simulation, controller logic partitioning and synthesis, and VHDL code generation. Being similar to the Backus-Naur-notation the Frame Modelling Language FML [20] and the graphical symbolic format (Figure 5) closely match requirements of high-level protocol specifications like

- Recognition of header patterns and synchronisation,
- Parsing and reassembling of structured data streams,
- Interface issues between data stream processing modules (such as synchronisation or stall).

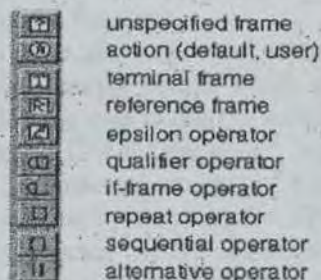


Fig. 5: Types of frames and frame operators

As an example Figure 6 shows the Protocol Compiler description of a simplified 32 bit IP header analysis implementation. A 32 bit wide register p_data_in is used to process the data sequentially. First the checksum is computed. Here, within each cycle portions of 16 bit are added. Then the length of the IP header is extracted. Next we check whether the received data is a IPv4 fragment.

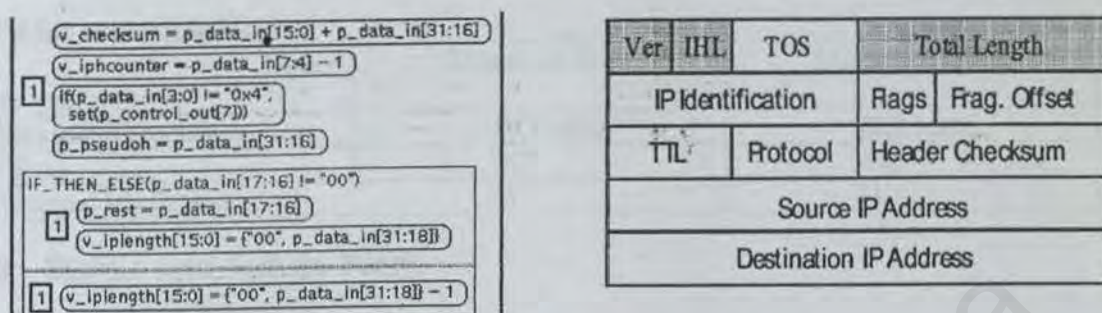


Fig. 6: Protocol Compiler IP Header Analysis Specification (excerpt, 32 bit)

Within the next step the pseudo header is initialised which is required for the checksum computations of higher layers. The next step calculates the number of 32 bit words (v_iplength) that have to be processed and determines eventually remaining bytes (p_rest). All these actions are performed within one clock cycle. Then the rest is read in units of 32 bytes. Within these steps header fields like the source IP address are extracted. Furthermore option fields are taken into account. After the header fields are extracted we perform validation operations as outlined in the previous section to verify the received fragment.

One part of the top-level is shown in figure 7. It illustrates three alternatives of processing after receiving valid data. The first four bits of the incoming packet decide which of the alternatives will run. If DIN is equal to „5“ (decimal) the following data is used to initialise a new connection. The packet is transferred directly to the output-FIFO if DIN is „3“, because only the receive path is implemented at this time (see part 4.2). A normal analysis of a packet begins only if DIN is equal to „1“. After starting the packet analysis by checking the Ethernet MAC address referenced to „ethernet_input“ the fast path unit parses the IP-Header and the TCP-Header. The last step is the generation of an acknowledgment if the encapsulated data in the packet was valid.

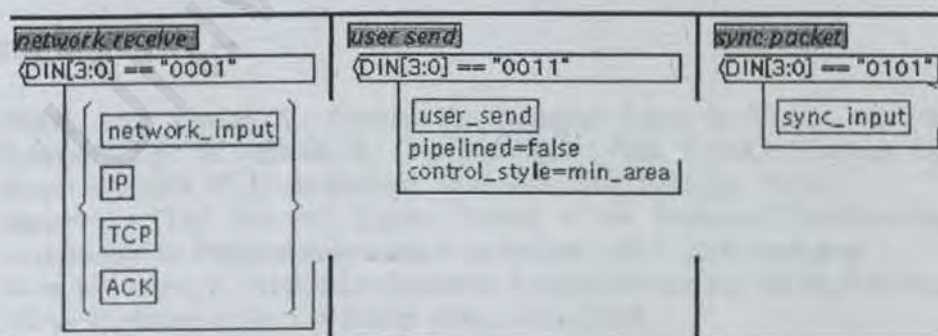


Fig. 7: Fast Path Unit Top Level Protocol Compiler Description

The Protocol Compiler offers the possibility to synthesize the design applying certain optimisation criteria. For example in figure 5 the attribute „Control_Style“ is set to „Min Area“. That means the partition „user_send“ will be optimised for minimum area during high level synthesis. Table 1 shows results of the synthesis for a Xilinx Virtex FPGA XV300 device and the differences between the Control_Style „Min Area“ and „Min Delay“.

Table 1: Synthesis Results

Property	Option	Minimum Area	Minimum Delay
Maximum Frequency		19,1 MHz	22,3 MHz
Slices		967	982
External I/O (IOBs)		71	71

5 Status and Future Work

We have presented a concept for the design and implementation of protocol processing accelerators. This was demonstrated with a packet classification and validation unit for hardware supported TCP/IP receive path processing. First steps in modelling and synthesis of the hardware partition were made utilising a usual FPGA rapid prototyping design flow, which we extended by a graphical high level design entry and by reusable protocol components. This approach supports an application-oriented modelling style in order to enhance design efficiency and quality for structured data stream processing controllers. Additionally, this leads to a quick design exploration, easy changeability, and design cycle reduction. A further controller design improvement can be expected by utilising reuse methodologies. Consequently, our future work aims at extending our rapid prototyping design flow by inserting a library of reusable protocol templates and components.

The design of accelerators based on these components is relatively difficult because a number of conditions have an influence on the achievable performance. Due to the inherent complexity of transport layer protocols it is furthermore advantageous to perform the non real-time processing on standard programmable architectures. Furthermore we will evaluate the integration of signal processors as well. As a consequence there are a lot of design alternatives combined with protocol engine configurations and requirements to take into account. Hence an automated and integrated design approach to determine which architecture is best suited for a specific protocol processing task would be beneficial. Further issues would be performance prediction as well as simulation and validation of the entire communication system.

6 References

- [1] Balraj, T.S.; Yemini, Y.: "Putting the Transport Layer on VLSI - the PROMPT Protocol Chip", in: Pehrson, B.; Gunningberg, P.; Pink, S. (ed.): *Protocols for High-Speed Networks, III*, North Holland, Stockholm, May 1992, pp. 19-34
- [2] Benz, M.: "The Protocol Engine Project - An Integrated Hardware/Software Architecture for Protocol Processing Acceleration", SDA 2000 workshop
- [3] Benz, M.; Engel, F.: "Hardware Supported Protocol Processing for Gigabit Networks", SDA - Workshop on System Design Automation, 1998
- [4] Benz, M.; Feske, K.: "A Packet Classification and Validation Unit for Hardware Supported TCP/IP Receive Path Processing", SDA 2000 workshop

INTERNET PROTOCOL (IP)

(see RFC 791)

The Internet Protocol (IP) provides a frame for encapsulating other protocols like TCP and UDP. The IP header informs the recipient among other things of: the destination and source addresses of the packet, number of octets in the packet, whether the packet can be fragmented or not, how many hops can the packet traverse, the protocol that the packet carries, etc. The IP version currently utilized is 4.

IP HEADER FORMAT

OCTET 1	Version (4 bit)+IHL (4 bit)	(VER, IHL)
OCTET 2	Type of service	(TOS)
OCTET 3,4	Total Length	(TOL)
OCTET 5,6	Identification	(ID)
OCTET 7,8	Flags (3 bit)+Fragment Offset (13 bit)	(FLG, FRO)
OCTET 9	Time to Live	(TTL)
OCTET 10	Protocol	(PRO)
OCTET 11,12	Header Checksum	(IP_SUM)
OCTET 13,14,15,16	Source Address	(SRC)
OCTET 17,18,19,20	Destination Address	(DEST)
OCTET 21,22,23	Options	(OPT)
OCTET 24	Padding	
OCTET 25, 26 ...	Data	

In the example shown the IP packet is encapsulated in a PPP frame including the flag sequence (7E), address (FF 03), protocol (00 21) and FCS (0B 81). In turn, the IP packet encapsulates a UDP message.

Example:

```
7E FF 03 00 21 45 00 00 40 00 01 00 00 3C 11 E0 31 CE D9 8F 1F C7 B6 78
CB 04 63 00
35 00 2C AB DA 00 01 01 00 00 01 00 00 00 00 00 04 70 6F 70 64 02 69
78 06 6E 65
74 63 6F 6D 03 63 6F 6D 00 00 01 00 01 0B 81 7E
```

```
Start      7E
Address    FF 03
SEP        00 21
IP Header  45 00 00 40 00 01 00 00 3C 11 E0 31 CE D9 8F 1F C7 B6 78
CB
Data       04 89 00 35 00 2C AB B4 00 01 01 00 00 01 00 00 00 00 00
00 04 70 6F
           70 64 02 69 78 06 6E 65 74 63 6F 6D 03 63 6F 6D 00 00 01
00 01
FCS        0B 81
Stop       7E
```

IP Header

VER=4 IHL=5 TOS=0 TOL=64 ID=1 FLG=00 FRO=00 TTL=60 PRO=17 IP_SUM=E031
SRC=206.217.143.31. DEST=199.182.120.203. OPT=00000000

Click Next for IP Checksum calculation code.

IP CHECKSUM CALCULATION

Click [here](#) for a short Description of the Internet checksum.

The IP Header Checksum is computed on the header fields only.
Before starting the calculation, the checksum fields (octets 11 and 12) are made equal to zero.

In the example code,
u16 buff[] is an array containing all octets in the header with octets 11 and 12 equal to zero.
u16 len_ip_header is the length (number of octets) of the header.

```
/*
*****
***
Function: ip_sum_calc
Description: Calculate the 16 bit IP sum.
*****
****
*/
typedef unsigned short u16;
typedef unsigned long u32;

u16 ip_sum_calc(u16 len_ip_header, u16 buff[])
{
    u16 word16;
    u32 sum=0;
    u16 i;

    // make 16 bit words out of every two adjacent 8 bit words in
the packet
    // and add them up
    for (i=0; i<len_ip_header; i=i+2) {
        word16 = ((buff[i]<<8) & 0xFF00) + (buff[i+1] & 0xFF);
        sum = sum + (u32) word16;
    }

    // take only 16 bits out of the 32 bit sum and add up the
carries
    while (sum >> 16)
        sum = (sum & 0xFFFF) + (sum >> 16);

    // one's complement the result
    sum = ~sum;

    return ((u16) sum);
}
```

TRANSPORT CONTROL PROTOCOL (TCP)

(RFC 793)

TCP is a very clever design! One can quickly notice the resemblance between TCP and a coherent telephone conversation between two or more people.

The sender starts with "Hello. May I speak with Alex?". The recipient replies

"Sure. May I ask who's calling?". And the sender answers "This is Peter". This 3 step process is how TCP initiates a transmission. A SYN packet including the sending address is sent, the recipient answers with an acknowledge-syn packet (ACK SYN) including his address and the sender acknowledges with an ACK packet. Note that the recipient answer "Yes. This is Alex speaking" is not allowed since the sender has not identified himself at this point.

From here, the conversation can follow in both directions, provided that either one of the parties has clearly understood what the other one has said. TCP does this through a Sequence (SEQ) and an Acknowledge (ACK) numbers. For simple explanatory purpose, every sent packet has a SEQ number which is equal with the number of octets sent (and acknowledged to be received) and an ACK number equal to the number of octets received up to the current packet. For the receiver, these numbers reverse. If these numbers don't match the packet is re-transmitted or the transmission stops in case the error can't be corrected. Just as for a real conversation one part can send a bulk of packets before receiving an acknowledge. And even more, "What did you just say?", "Can you say it again?" or "Slow down a little, I can't follow" are usual situations implemented by TCP.

The conversation can finish with "Bye" followed by "OK, I'll talk to you later" from the receiver which in TCP terms it's called graceful. TCP implements a graceful end by sending a FIN packet followed by an ACK FIN packet from the receiver. Or, the conversation may have a not-so-graceful graceful end when one part just hangs-up. In this case TCP sends a RST (reset) packet closing the connection. TCP is the protocol mostly utilized on the Internet. It's usually called TCP/IP because in all cases TCP packets are encapsulated in IP packets.

TCP HEADER FORMAT

OCTET 1,2 (SRC_PORT)	Source Port
OCTET 3,4 (DEST_PORT)	Destination Port
OCTET 5,6,7,8 (SEQ)	Sequence Number
OCTET 9,10,11,12 (ACK)	Acknowledgement Number
OCTET 13,14 (DFO, FLG)	Data Offset (4 bit)+Reserved (6 bit)+ Control Flags (6 bit)
OCTET 15,16 (WIN)	Window
OCTET 17,18 (TCP_SUM)	Checksum
OCTET 19,20 (URF)	Urgent Pointer
OCTET 21,22,23 (OPT)	Options
OCTET 24	Padding
OCTET 25,26...	Data

TCP/IP Packet Example:

```

7E 21 45 00 00 4B 57 49 40 00 FA 06 85 77 C7 B6 78 0E CE D6 95 50 00 6E
04 9F 74 5B EE A2
59 9A 00 0E 50 18 24 00 E3 2A 00 00 2B 4F 4B 20 50 61 73 73 77 6F 72 64
20 72 65 71 75 69
72 65 64 20 66 6F 72 20 61 6C 65 78 75 72 2E 0D 0A 67 B2 7E

```

```

Start      7E
SEP        21
IP Header  45 00 00 4B 57 49 40 00 FA 06 85 77 C7 B6 78 0E CE D6 95
50
TCP Header 00 6E 04 9F 74 5B EE A2 59 9A 00 0E 50 18 24 00 E3 2A 00
00
Data       2B 4F 4B 20 50 61 73 73 77 6F 72 64 20 72 65 71 75 69 72
65 64 20 66 6F 72
           20 61 6C 65 78 75 72 2E 0D 0A
FCS        67 B2
Stop       7E

```

TCP Header:

```

SRC_PORT=110 DEST_PORT=1108 SEQ=745BEEA2 ACK=599A000E DFO=5
FLG=16 WIND=9216
TCP_SUM=E32A URF=0000 No Options

```

Data:

-OK Password required for alexis r n

Control Flags (FLG=18):

```

FLG=00110000
Urgent Pointer      URG=0
Acknowledgment      ACK=1
Push Function       PSN=1
Reset connection    RST=0
Synchronization    SYN=0
Finished data       FIN=0

```

State = ACK-PSH

Click Next for TCP Checksum calculation code.

TCP CHECKSUM CALCULATION

(See also short Description of Internet Checksum).

To calculate TCP checksum a "pseudo header" is added to the TCP header. This includes:

IP Source Address	4 bytes
IP Destination Address	4 bytes
TCP Protocol	2 bytes
TCP Length	2 bytes

The checksum is calculated over all the octets of the pseudo header, TCP header and data.

If the data contains an odd number of octets a pad, zero octet is added to the end of data. The pseudo header and the pad are not transmitted with the packet.

In the example code,

u16 buff[] is an array containing all the octets in the TCP header and data.

u16 len_tcp is the length (number of octets) of the TCP header and data.

BOOL padding is 1 if data has an even number of octets and 0 for an odd number.

u16 src_addr[4] and u16 dest_addr[4] are the IP source and destination address octets.

```
/*
*****
***
Function: tcp_sum_calc()
*****
***
Description:
    Calculate TCP checksum
*****
****
*/

typedef usingend short u16;
typedef unsigned long u32;

u16 tcp_sum_calc(u16 len_tcp, u16 src_addr[], u16 dest_addr[], BOOL
padding, u16 buff[])
{
    u16 prot_tcp=6;
    u16 padd=0;
    u16 word16;
    u32 sum;

    // Find out if the length of data is even or odd number. If odd,
    // add a padding byte = 0 at the end of packet
    if (padding&1==1){
        padd=1;
        buff[len_tcp]=0;
    }
}
```


<http://www.netfor2.com/udp.htm>

USER DATAGRAM PROTOCOL (UDP)

(see RFC 768)

User Datagram Protocol is utilized to send data that doesn't necessarily need to be very reliable. The UDP packet is encapsulated in an IP packet which in turn is encapsulated in a PPP packet. Both UDP and IP have checksums octets and the PPP packet has its FCS octets however this can only guarantee that the data and the destination are correct. However, there is a possibility that this data does not belong to an expected message sequence but is rather part of another message that just happened to have the same destination. This issue is addressed by the TCP protocol.

UDP is a simple to implement protocol because it does not require to keep track of every packet sent or received and it does not need to initiate or end a transmission. Because of this it is mainly designed for communications where you either don't care what the response will be or you pretty much know it. UDP messages are generally faster than TCP provided that the communication link functions properly. UDP is widely utilized to send DNS (Domain Name Search) requests, to exchange chat messages, or to access telephone numbers via Internet.

UDP HEADER FORMAT

OCTET 1,2	Source Port
OCTET 3,4	Destination Port
OCTET 5,6	Length
OCTET 7,8	Checksum
OCTET 9,10....	Data

UDP PACKET (Example)

```
04 89 00 35 00 2C AB B4 00 01 01 00 00 01 00 00 00 00 00 00 04 70 6F 70
64 02 69 73 06 6E
65 74 63 6F 6D 02 62 6F 6D 00 00 01 00 01
```

UDP Header	04 89 00 35 00 2C AB B4
Data	00 01 01 00 00 01 00 00 00 00 00 00 04 70 6F 70 64 02 69 73 06 6E 65 74 63 6F 6D 02 62 6F 6D 00 00 01 00 01

Source Port	04 89
Destination Port	00 35
Length	00 2C
Checksum	AB B4
Data	DNS Message

Click Next for UDP Checksum calculation code.

UDP CHECKSUM CALCULATION

(see also short Description of Internet Checksum)

To calculate UDP checksum a "pseudo header" is added to the UDP header. This includes:

IP Source Address	4 bytes
IP Destination Address	4 bytes
Protocol	2 bytes
UDP Length	2 bytes

The checksum is calculated over all the octets of the pseudo header, UDP header and data.

If the data contains an odd number of octets a pad, zero octet is added to the end of data.

The pseudo header and the pad are not transmitted with the packet.

In the example code,

u16 buff[] is an array containing all the octets in the UDP header and data.

u16 len_udp is the length (number of octets) of the UDP header and data.

BOOL padding is 1 if data has an even number of octets and 0 for an odd number.

u16 src_addr[4] and u16 dest_addr[4] are the IP source and destination address octets

```
/*
*****
***
Function: udp_sum_calc()
Description: Calculate UDP checksum
*****
***
*/
typedef usingend short u16;
typedef unsigned long u32;

u16 short udp_sum_calc(u16 len_udp, u16 src_addr[], u16 dest_addr[],
BOOL padding, u16 buff[])
{
    u16 prot_udp=17;
    u16 padd=0;
    u16 word16;
    u32 sum;

    // Find out if the length of data is even or odd number. If odd,
    // add a padding byte = 0 at the end of packet
    if (padding!=1){
        padd=1;
        buff[len_udp]=0;
    }

    //initialize sum to zero
    sum=0;
```


Tutorial on TCP/IP Packet Format

- Ethernet headers
- TCP headers
- IP headers
- CODE BITS (6 bits) URG ACK PSH RST SYN FIN (UAPRSF)

[You may safely ignore the "checksum incorrect" message for outgoing frames from 158.132.148.170. This is due to the so-called "checksum offloading". Refer to <http://www.ethereal.com/lists/ethereal-users/200210/threads.html#00070> for more details.]

Frame 1 (62 bytes on wire, 62 bytes captured)
Arrival Time: Dec 2, 2002 11:18:28.416594000
Time delta from previous packet: 0.000000000 seconds
Time relative to first packet: 0.000000100 seconds
Frame Number: 1
Packet Length: 62 bytes
Capture Length: 62 bytes
Ethernet II, Src: 00:01:02:83:ab:72, Dst: 00:00:0c:07:ac:01
Destination: 00:00:0c:07:ac:01 (00:00:0c:07:ac:01)
Source: 00:01:02:83:ab:72 (00:01:02:83:ab:72)
Type: IP (0x0800)
Internet Protocol, Src Addr: 158.132.148.170 (158.132.148.170), Dst Addr: 207.202.214.132 (207.202.214.132)
Version: 4
Header length: 20 bytes
Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
0000 00.. = Differentiated Services Codepoint: Default (0x00)
.... 00.. = ECN-Capable Transport (ECT): 0
.... 00.. = ECN-CE: 0
Total Length: 48
Identification: 0x4e6a
Flags: 0x04
..1.. = Don't fragment: Set
..0.. = More fragments: Not set
Fragment offset: 0
Time to live: 128
Protocol: TCP (0x06)
Header checksum: 0x0000 (incorrect, should be 0x4c36)
Source: 158.132.148.170 (158.132.148.170)
Destination: 207.202.214.132 (207.202.214.132)
Transmission Control Protocol, Src Port: 1729 (1729), Dst Port: telnet
Seq: 363961647, Ack: 0, Len: 0
Source port: 1729 (1729)
Destination port: telnet (23)
Sequence number: 363961647
Header length: 28 bytes
Flags: 0x0002 (SYN)
0... .. = Congestion Window Reduced (CWR): Not set

Destination port: 1729 (1729)
Sequence number: 3351904732
Acknowledgement number: 3639616448
Header length: 24 bytes
Flags: 0x0012 (SYN, ACK)

0... .. = Congestion Window Reduced (CWR): Not set
.0... .. = ECN-Echo: Not set
..0. = Urgent: Not set
...1 = Acknowledgment: Set
.... 0... = Push: Not set
.... .0.. = Reset: Not set
.... ..1. = Syn: Set
.... ...0 = Fin: Not set

Window size: 16384
Checksum: 0xb168 (correct)
Options: (4 bytes)
Maximum segment size: 1460 bytes

0000 00 01 02 83 ab 72 00 01 97 2d 63 48 05 00 45 00n...-
cH..E.
0010 00 2c 97 00 00 00 2c 06 1e 4e cf ca d6 84 9e 84
.....N.....
0020 94 aa 00 17 06 c1 c7 ca 01 dc d8 f0 23 c0 60 12
.....#. .
0030 40 00 b1 68 00 00 02 04 05 b4 00 00e..h.....

Frame 3 (54 bytes on wire, 54 bytes captured)

Arrival Time: Dec 2, 2002 11:18:28.678399000
Time delta from previous packet: 0.000030000 seconds
Time relative to first packet: 0.261805000 seconds
Frame Number: 3
Packet Length: 54 bytes
Capture Length: 54 bytes

Ethernet II, Src: 00:01:02:83:ab:72, Dst: 00:00:0c:07:ac:01

. Destination: 00:00:0c:07:ac:01 (00:00:0c:07:ac:01)

Source: 00:01:02:83:ab:72 (00:01:02:83:ab:72)

Type: IP (0x0800)

Internet Protocol, Src Addr: 158.132.143.170 (158.132.143.170), Dst
Addr: 207.202.214.132 (207.202.214.132)

Version: 4

Header length: 20 bytes

Differentiated Services Field: 0x00 DSCP 0x00: Default; ECN: (000)

0000 00.. = Differentiated Services Codepoint: Default (0x00)

.... ..0. = ECN-Capable Transport (ECT): 0

.... ...0 = ECN-CE: 0

Total Length: 40

Identification: 0x1e6b

Flags: 0x04

..1. = Don't fragment: Set

..0. = More fragments: Not set

Fragment offset:

Time to live: 128

Protocol: TCP (0x06)

Header checksum: 0x0000 (incorrect, should be 0x0000)

Source: 158.132.143.170 (158.132.143.170)

Destination: 207.202.214.132 (207.202.214.132)

.0... = ECN-Echo: Not set
..0. = Urgent: Not set
...0 = Acknowledgment: Not set
.... 0... = Push: Not set
... .0.. = Reset: Not set
.... ..1. = Syn: Set
.... ...0 = Fin: Not set

Window size: 16384

Checksum: 0x6619 (correct)

Options: (8 bytes)

Maximum segment size: 1460 bytes

NOP

NOP

SACK permitted

0000 00 00 0c 07 ac 01 00 01 02 83 ab 72 03 00 45 00

.....r..E.

0010 00 30 4e 6a 40 00 80 06 00 00 9e 84 94 aa cf ca

.ONj0.....

0020 d6 84 06 c1 00 17 d8 f0 23 bf 00 00 00 00 70 02

.....#.....p.

0030 40 60 66 19 00 00 02 04 05 b4 01 01 04 02

0.f.....

Frame 2 (60 bytes on wire, 60 bytes captured)

Arrival Time: Dec 2, 2002 11:18:28.678349000

Time delta from previous packet: 0.261755000 seconds

Time relative to first packet: 0.261755000 seconds

Frame Number: 2

Packet Length: 60 bytes

Capture Length: 60 bytes

Ethernet II, Src: 00:01:97:2d:63:48, Dst: 00:01:02:83:ab:72

Destination: 00:01:02:83:ab:72 (00:01:02:83:ab:72)

Source: 00:01:97:2d:63:48 (00:01:97:2d:63:48)

Type: IP (0x0800)

Trailer: 0000

Internet Protocol, Src Addr: 207.202.214.132 (207.202.214.132), Dst

Addr: 158.152.148.170 (158.152.148.170)

Version: 4

Header length: 20 bytes

Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)

0000 00.. = Differentiated Services Codepoint: Default (0x00)

.... ..0. = ECN-Capable Transport (ECT): 0

.... ..0. = ECN-CE: 0

Total Length: 44

Identification: 0x9700

Flags: 0x00

... = Don't fragment: Not set

... = More fragments: Not set

Fragment offset: 0

Time to live: 40

Protocol: TCP (0x06)

Header checksum: 0x1e4e (correct)

Source: 207.202.214.132 (207.202.214.132)

Destination: 158.152.148.170 (158.152.148.170)

Transmission Control Protocol, Src Port: telnet (23), Dst Port: 1719

1719, Seq: 3639616473, Ack: 3639616448, Len: 0

Source port: telnet (23)

Transmission Control Protocol, Src Port: 1729 (1729), Dst Port: telnet (23), Seq: 3639616448, Ack: 3351904733, Len: 0

Source port: 1729 (1729)

Destination port: telnet (23)

Sequence number: 3639616448

Acknowledgement number: 3351904733

Header length: 20 bytes

Flags: 0x0010 (ACK)

0... .. = Congestion Window Reduced (CWR): Not set

.0... .. = ECN-Echo: Not set

..0. = Urgent: Not set

...1 = Acknowledgment: Set

.... 0... = Push: Not set

.... .0.. = Reset: Not set

.... ..0. = Syn: Not set

.... ...0 = Fin: Not set

Window size: 17520

Checksum: 0xd998 (incorrect, should be 0xc4b5)

0000 00 00 0c 07 ac 01 00 01 02 83 ab 72 08 00 45 00

.....r..E.

0010 00 28 4e 6b 40 00 80 06 00 00 9e 84 94 aa cf ca

..(Nk@.....

0020 d6 84 06 c1 00 17 d8 f0 23 c0 c7 ca 01 dd 50 10

.....#.....P.

0030 44 70 d9 98 00 00

Ip.....

Frame 4 (60 bytes on wire, 60 bytes captured)

Arrival Time: Dec 2, 2002 11:18:28.937350000

Time delta from previous packet: 0.258951000 seconds

Time relative to first packet: 0.520766000 seconds

Frame Number: 4

Packet Length: 60 bytes

Capture Length: 60 bytes

Ethernet II, Src: 00:01:97:2d:63:48, Dst: 00:01:02:83:ab:72

Destination: 00:01:02:83:ab:72 (00:01:02:83:ab:72)

Source: 00:01:97:2d:63:48 (00:01:97:2d:63:48)

Type: IP (0x0800)

Trailer: 000000

Internet Protocol, Src Addr: 207.202.214.131 (207.202.214.131), Dst Addr: 158.132.148.170 (158.132.148.170)

Version: 4

Header length: 20 bytes

Differentiated Services Field: 0x00 (DSCP: 0x00: Default; ECN: 0x00)

0000 00.. = Differentiated Services Codepoint: Default (0x00)

.... ..0. = ECN-Capable Transport (ECT): 0

.... ...0 = ECN-CE: 0

Total Length: 43

Identification: 0x0000

Flags: 0x00

... = Don't Fragment: Not set

... = More fragments: Not set

Fragment offset: 0

Time to live: 44

Protocol: TCP (0x06)

Header checksum: 0x1e19 (correct)

Source: 207.202.214.131 (207.202.214.131)

Destination: 158.132.148.170 (158.132.148.170)
Transmission Control Protocol, Src Port: telnet (23), Dst Port: 1729
(1729), Seq: 3351904733, Ack: 3639616448, Len: 3

Source port: telnet (23)

Destination port: 1729 (1729)

Sequence number: 3351904733

Next sequence number: 3351904736

Acknowledgement number: 3639616448

Header length: 20 bytes

Flags: 0x0018 (PSH, ACK)

0... .. = Congestion Window Reduced (CWR): Not set

.0... .. = ECN-Echo: Not set

..0. = Urgent: Not set

...1 = Acknowledgment: Set

.... 1... = Push: Set

.... .0... = Reset: Not set

.... ..0. = Syn: Not set

.... ...0 = Fin: Not set

Window size: 17520

Checksum: 0x9fac (correct)

Telnet

Command: Do Authentication Option

0000 00 01 02 83 ab 72 00 01 97 3d 63 48 06 00 45 00-

cH..E.

0010 00 2b 97 36 00 00 2c 06 1e 19 cf ca d6 84 9e 84

.+.6.....

0020 94 aa 00 17 06 e1 c7 ca 01 dd d8 f0 28 c0 80 18

.....#.P.

0030 44 70 9f ac 00 00 ff fd 25 00 00 00

Dp.....&...

Flow control is implemented to control the flow of packets relating to a single call in order to overcome the difference between the rate at which a source system sends packets and the rate at which a destination can accept packets. If the destination can accept packets faster than the source can send them, clearly there is no problem. However, if the reverse is true a harmonization (flow control) function must be provided.

Congestion control is concerned with a similar function within the network itself. If the composite rate at which packets enter the network exceeds the rate at which packets leave, then the network becomes congested. Similarly, at a more local level, if packets arrive at a network node – for example, an IS – faster than they can be processed and forwarded, then the node becomes congested thus affecting the flow of packets relating to all calls through that node.

With a connection-oriented network such as X.25, flow control is performed on a VC basis across the local DTE–DCE and DCE–DTE interfaces. A send window is defined and when this number of packets have been sent (typically two), the sender must wait until an acknowledgment relating to either of them is received. Since this function is being performed at the periphery of the network on a per call basis, in addition to regulating the flow of packets into the network, it helps to control congestion. However, it does not prevent congestion completely.

In contrast, with a connectionless network no flow control is applied to the packets associated with a call within the network. Instead it is left to the transport protocol entity within each ES to perform flow control on an end-to-end basis. If congestion occurs within the network, flow control information is delayed and the source transport protocol entities stop sending new data into the network. Again, although sending new data helps to relieve network congestion, as with a connection-oriented network, it does not always avoid it. Therefore, with both schemes we must incorporate a congestion control algorithm within the network. Moreover, for internets comprising multiple network types, the congestion control algorithm must harmonize between the different network algorithms.

Error reporting

The way in which errors are reported varies from one network type to another. Consequently, we must establish a means of error reporting across multiple networks.

All these problems must be addressed by any internetworking solution.

9.3 Network layer structure

The role of the network layer in each ES is to provide an end-to-end, internetwork service to its local NS_user(s). This can be either a CONS or a CLNS. In both cases the NS_users should be unaware of the presence of multiple, possibly different, network types. Hence the routing and all other functions relating to the

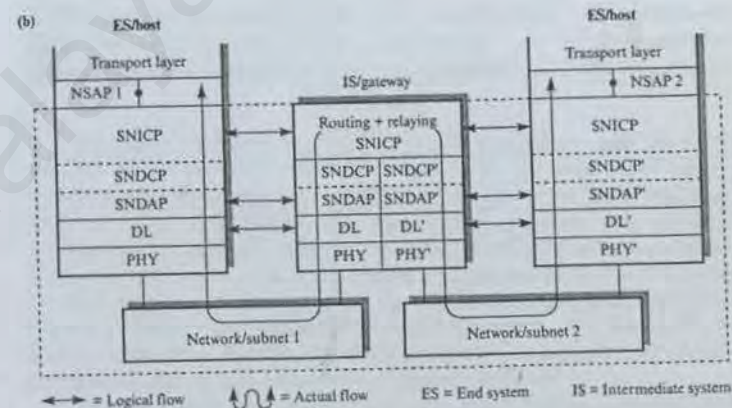
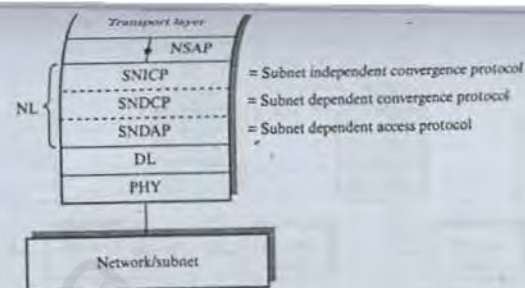


Figure 9.6
Network layer structure: (a) sublayer protocols; (b) IS structure.

relaying of NSDUs must be carried out in a transparent way by the network layer entities in each of the end and intermediate systems.

To achieve this goal, in the context of the ISO reference model the network layer in each ES and IS consists not just of a single protocol but rather of three (sublayer) protocols, each performing a complementary role in providing the network layer service. In ISO terminology, each network that makes up an internet is known as a subnet and hence the three protocols are known as follows:

- Subnetwork independent convergence protocol (SNICP)
- Subnetwork dependent convergence protocol (SNDCP)
- Subnetwork dependent access protocol (SNDAP)

The relative position of the three protocols in an ES is given in Figure 9.6(a); part (b) shows the protocols in relation to an IS.

The SNICP supports the network service provided to NS users at the interface with the internet. Its role is to carry out the various harmonizing (convergence) functions which may be necessary to route and relay user data (transport protocol data units) across the internet. Its operation is independent of the characteristics of the specific subnets (networks) used in the internet and it assumes a standard network service from them.

The SNDAP is the access protocol associated with a specific subnet (network) in the internet. Examples are the X.25 packet layer protocol for an X.25 network and the connectionless network protocol that is often used for LANs. Because the service and operational characteristics associated with the SNDAPs differ from one network type to another, an intermediate sublayer must be provided between the SNICP and the SNDAP. This is the role of the SNDCP. Clearly, the detailed mapping operation that it performs will vary for different subnet/network types.

9.4 Internet protocol standards

As we discussed in Chapter 8, multiple X.25 WANs can be interconnected by X.75-based gateways. The introduction of a standard specifying the operation of the X.25 packet layer protocol for use with LANs means that one approach to internetworking is to adopt X.25 as an internetworking protocol. The latter can be operated in either a connection-oriented mode or in a pseudoconnectionless mode by using fast select.

This solution has the appeal that the various internetworking functions are much reduced. The disadvantage is that the overheads associated with X.25 packet switching are high and hence the packet throughput of these networks is relatively low. This is also true with fast select, since the same VC/error control functions are still used. Moreover, the much improved bit error rate performance of the next generation of WANs, such as ISDN, means that frame relay and cell (fast packet) switching will be the preferred operational modes rather than conventional packet switching.

The solution adopted by ISO is based on a connectionless internet service and an associated connectionless SNICP. The SNICP is defined in ISO 8475. It is based on the internet protocol that has been developed as part of research into internetworking funded by the US Defense Advanced Research Projects Agency (DARPA). The early DARPA internet - ARPANET - was used to interconnect the computer networks associated with a small number of research and university sites with those of DARPA. When it came into being in the early 1970s, ARPANET involved just a small number of networks and associated host computers. Since that time, the internet has grown steadily. Instead of just a small number of mainframe computers at each site, there are now large numbers of workstations. Moreover, the introduction of LANs means that there are now several thousand networks/subnets. ARPANET is now linked to other internets. The combined internet, which is jointly funded by a number of agencies, is thus known simply as the Internet.

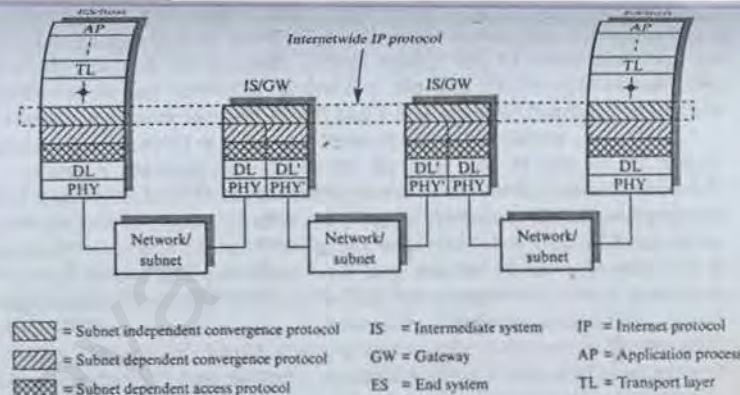


Figure 9.7
Internetwork IP
schematic.

The internet protocol is only one protocol associated with the complete protocol suite (stack) used with the Internet. The complete suite, known as TCP/IP, includes transport and application protocols which are now used as the basis of many other commercial and research networks. All the TCP/IP specifications are publicly available, as a result of which the Internet is by far the largest currently operational internet based on open standards. The two protocols we shall discuss in this chapter are the internet protocol associated with the Internet - known as the Internet IP or simply IP - and the ISO Internet Protocol known as ISO-IP or ISO CLNP, which is intended for use with OSI stacks. The general approach of both standards is illustrated in Figure 9.7.

IP is an internetworking protocol that enables two transport protocol entities resident in different ESs/hosts to exchange message units (NSDUs) in a transparent way. This means that the presence of multiple, possibly different, networks/subnets and ISs/gateways is completely transparent to both communicating transport entities. As the IP is a connectionless protocol, message units are transferred using an unacknowledged best-try approach.

Although the operational features associated with ISO CLNP are based on experience gained from the evolution and use of IP, there are differences both in terms of terminology and operational detail. Hence we shall discuss each protocol separately.

9.5 Internet IP

TCP/IP is now widely used in many commercial and research internets in addition to the Internet. Nevertheless, almost all the protocols associated with the TCP/IP have been researched and developed as part of the Internet. Indeed, new protocols are introduced relatively frequently as research associated with the combined

part of all TCP/IP implementations. Other optional protocols are intended for open systems of varying size and complexity. We shall consider only the core protocols.

9.5.1 Address structure

Recall that there are two network addresses associated with a host/ES attached to an internet. In ISO terminology, these are the network service access point (NSAP) address and the subnet point of attachment (SNPA) address. With TCP/IP, these are the IP address and the network point of attachment (NPA) address, respectively. The NPA address is different for each network/subnet type, whereas the IP address is a unique internetwide identifier. The structure of an IP address is shown in Figure 9.8. In order to give the authority establishing the internet some flexibility in assigning addresses, the address structure shown in part (a) has been adopted.

To ensure that all hosts have a unique identifier, a 32-bit integer is used for each IP address. Then three different address formats are defined to allow for the

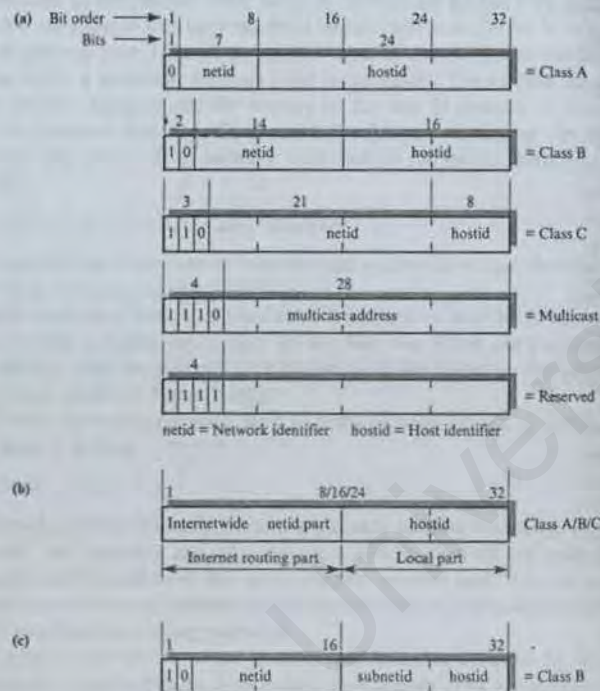


Figure 9.8
IP address formats:
(a) frame; (b) subnet
addressing;
(c) modified class B
address.

addressing scheme of the internet. The three primary classes are A, B, and C; each is intended for use with a different size of network. The class to which an address belongs can be determined from the position of the first zero bit in the first four bits. The remaining bits specify two subfields – a network identifier (netid) and a host identifier (hostid). The subfield boundaries are located on byte boundaries to simplify decoding.

Class A addresses have 7 bits for the netid and 24 bits for the hostid; class B addresses have 14 bits for the netid and 16 bits for the hostid; and class C addresses have 21 bits for the netid and 8 bits for the hostid. Class A addresses are intended for use with networks that have a large number of attached hosts (up to 2^{24}) while class C addresses allow for a large number of networks each with a small number of attached hosts (up to 256). An example of a class A network is ARPANET; an example of a class C network is a single site-wide LAN.

An address with a hostid of zero is used to refer to the network in the netid field rather than to a host. Similarly, an address with a hostid of all 1s refers to all hosts attached to the network in the netid field or, if the latter is all 1s also, then to all hosts in the internet. Such addresses are used for broadcast purposes.

To make it easier to communicate IP addresses, the 32 bits are broken into four bytes. These are converted into their equivalent decimal form with a dot (period) between each. This is known as **dotted decimal**. Example addresses are as follows:

00001010 00000000 00000000 00000000 = 10.0.0.0 = class A
= netid 10 (ARPANET)

10000000 00000011 00000010 00000011 = 128.3.2.3 = class B
= netid 128.3, hostid 2.3

11000000 00000000 00000001 11111111 = 192.0.1.255 = class C
= all hosts broadcast on netid 192.0.1

Class D addresses are reserved for **multicasting**. In a LAN, a frame may be sent to an individual, broadcast, or **group address**. The last one allows a group of hosts – for example, workstations – that are cooperating in some way, to arrange for network transmissions to be sent to all members of the group. This is often referred to as **computer-supported cooperative working (CSCW)**; class D addresses allow this mode of working to be extended across an internet.

Although this basic structure is adequate for most addressing purposes, the introduction of multiple LANs at each site can mean unacceptably high overheads in terms of routing. As Chapter 7 described, MAC bridges are normally used to interconnect LANs of the same type. This solution is attractive for routing purposes, since the combined LAN then behaves like a single network. When interconnecting dissimilar LAN types, the differences in frame format and, more importantly, frame length, mean that routers are normally used since the fragmentation and reassembly of packets/frames is a function of the network layer rather than the MAC sublayer. However, the use of routers means that each

LAN must have its own netid. In the case of large sites, there may be a significant number of such LANs.

This means that with the basic addressing scheme, all the routers relating to a site need to take part in the overall internet routing function. The efficiency of any routing scheme is strongly influenced by the number of routing nodes that make up the internet. The concept of subnets has been introduced to decouple the routers – and hence routing – associated with a single site from the overall internet routing function. Essentially, instead of each LAN associated with a site having its own netid, only the site is allocated an internet netid. The identity of each LAN then forms part of the hostid field. This refined address format is shown in Figure 9.8(b).

The same address classes and associated structure are used, but the netid now relates to a complete site rather than to a single network. Hence, since only a single gateway attached to a local site network performs internetwide routing, the netid is considered as the **internet part**. For a single netid with a number of associated subnetworks the hostid part consists of two subfields: a **subnetid part** and a **local hostid part**. Because these have only local significance, they are known collectively as the **local part**.

Because of the possibly wide range of subnets associated with different site networks, no attempt has been made to define rigid subaddress boundaries for the local address part. Instead, an **address mask** is used to define the subaddress boundaries for a particular network (and hence netid). The address mask is kept by the internet gateway and the routers at the site. It consists of binary 1s in those bit positions that contain a network address – including the netid and subnetid – and binary 0s in positions that contain the hostid. Hence an address mask of

11111111 11111111 11111111 00000000

means that the first three bytes (octets) contain a network/subnet identifier and the fourth octet contains the host identifier.

For example, if the address is a class B address – a zero bit in the second bit position – this is readily interpreted as: the first two octets are the internetwide netid, the next octet the subnetid, and the last octet the hostid on this subnet. Such an address is shown in Figure 9.8(c).

Dotted decimal is normally used to define address masks, in which case the above mask is written:

255.255.255.0

Byte boundaries are normally chosen to simplify address decoding. Hence with this mask, and assuming the netid was, say, 128.10, then all the hosts attached to this network would have this same internet routing part. The presence of a possibly large number of subnets and associated routers is thus transparent to the internet gateways for routing purposes.

To ensure IP addresses are unique, they must be assigned by the central authority that is setting up the open system environment. For a small internet, this is relatively straightforward. However, in the case of large internets, such as the

Internet, this is normally done at two levels. Firstly, a central authority is set up to allocate netids and multicast addresses. Secondly, an authority associated with each network assigns hostids on that network. The central authority for the Internet is known as the **Network Information Center (NIC)**.

9.5.2 Datagrams

Before we consider the various functions and protocols associated with the IP, let us describe the format of an IP data unit. This is known as a **datagram**. The format and contents of a datagram are shown in Figure 9.9.

The **version** field contains the version of the IP used to create the datagram and ensures that all other systems – gateways and hosts – that process the datagram during its transit across the internet interpret the various fields correctly. The current version number is 4 and is referred to as **IP version 4**, or, simply, **IPv4**.

The header can be of variable length. The **header length** specifies the actual length of the datagram in multiples of 32-bit words. The minimum length – without options – is 5. If the datagram contains options, these must be in multiples of 32 bits. Any unused bytes must be filled with **padding** bytes.

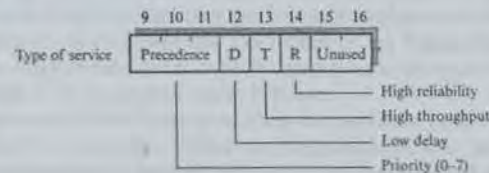
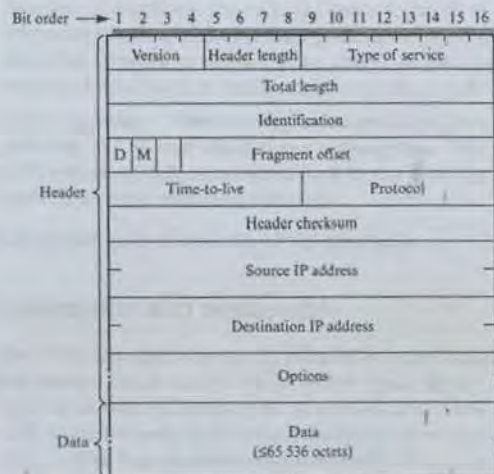


Figure 9.9
Internet datagram
format and contents.

The type of service provides the same role as the QOS parameter used in ISO networks. It allows an application process to specify the preferred attributes associated with the route and is, therefore, used by each gateway during route selection. For example, if a reliable delivery service is preferred to a best-try transfer, then given a choice the gateway should choose a connection-oriented network rather than a connectionless network. The total length defines the total length of the datagram including the header and user data parts. The maximum length is 65 536 bytes.

As we shall explain in Section 9.5.4, user messages may be transferred across the internet in multiple datagrams, with the identification field being used to allow a destination host to relate different datagrams to the same user message.

The next three bits are known as flag bits of which two are currently used. The first, known as the don't fragment or D bit, is again intended for use by intermediate gateways. A set D bit indicates that a network should be chosen that can handle the datagram as a single entity rather than as multiple smaller datagrams – known as fragments. Hence if the destination host is connected to that network (or subnet) it will receive the user data in a single datagram or not at all. The transit delay of the user data can therefore be more accurately quantified.

The second flag bit, known as the more fragments or M bit, is also used during the reassembly procedure associated with user data transfers involving multiple datagrams. The fragment offset is also used by the same procedure to indicate the position of the (data) contents of the datagram in relation to the initial user data message. We shall describe the reassembly procedure in Section 9.5.4.

The time-to-live value defines the maximum time for which a datagram can be in transit across the internet. The value, in seconds, is set by the source IP. It is then decremented by each gateway by a defined amount. Should the value become zero, the datagram is discarded. This procedure allows the destination IP to wait a known maximum time for a datagram fragment during the reassembly procedure. It also enables datagrams that are looping to be discarded.

More than one protocol is associated with the TCP/IP suite. The protocol field is used to enable the destination IP to pass the datagram to the required protocol.

The header checksum, which applies just to the header part of the datagram, is a safeguard against corrupted datagrams being routed to incorrect destinations. It is computed by treating each 16-bit field as an integer and adding them all together using 1's-complement (end-around-carry) arithmetic. The checksum is then the 1's-complement (inverse) of the sum.

The source address and destination address are the internetwork IP (NSAP) addresses of the source and destination hosts.

Finally, the options field is used in selected datagrams to carry additional information relating to the following:

- **Security** The data field may be encrypted for example, or be made accessible only to a specified user group.
- **Source routing** If known, the actual route to be followed through the internet may be specified in this field as a list of gateway addresses.

Route recording This field is used by each gateway visited, during the passage of a datagram through the internet to record its address. The resulting list can be used, for example, in the source routing field of subsequent datagrams.

- **Stream identification** This enables a source to indicate the type of data being carried in the datagram if this is not computer data, for example, samples of speech.
- **Timestamp** If present, this is used by each gateway along the path followed by the datagram to record the time it processed the datagram.

9.5.3 Protocol functions

The IP provides a number of core functions and associated procedures to carry out the various harmonizing functions that are necessary when interworking across dissimilar networks. These include the following:

- **Fragmentation and reassembly** This concerns the transfer of user messages across networks/subnets which support smaller packet sizes than the user data.
- **Routing** To perform the routing function, the IP in each source host must know the location of the internet gateway or local router that is attached to the same network or subnet. Also, the IP in each gateway must know the route to be followed to reach other networks or subnets.
- **Error reporting** When routing or reassembling datagrams within a host or gateway, the IP may discard some datagrams. This function is concerned with reporting such occurrences back to the IP in the source host and with a number of other reporting functions.

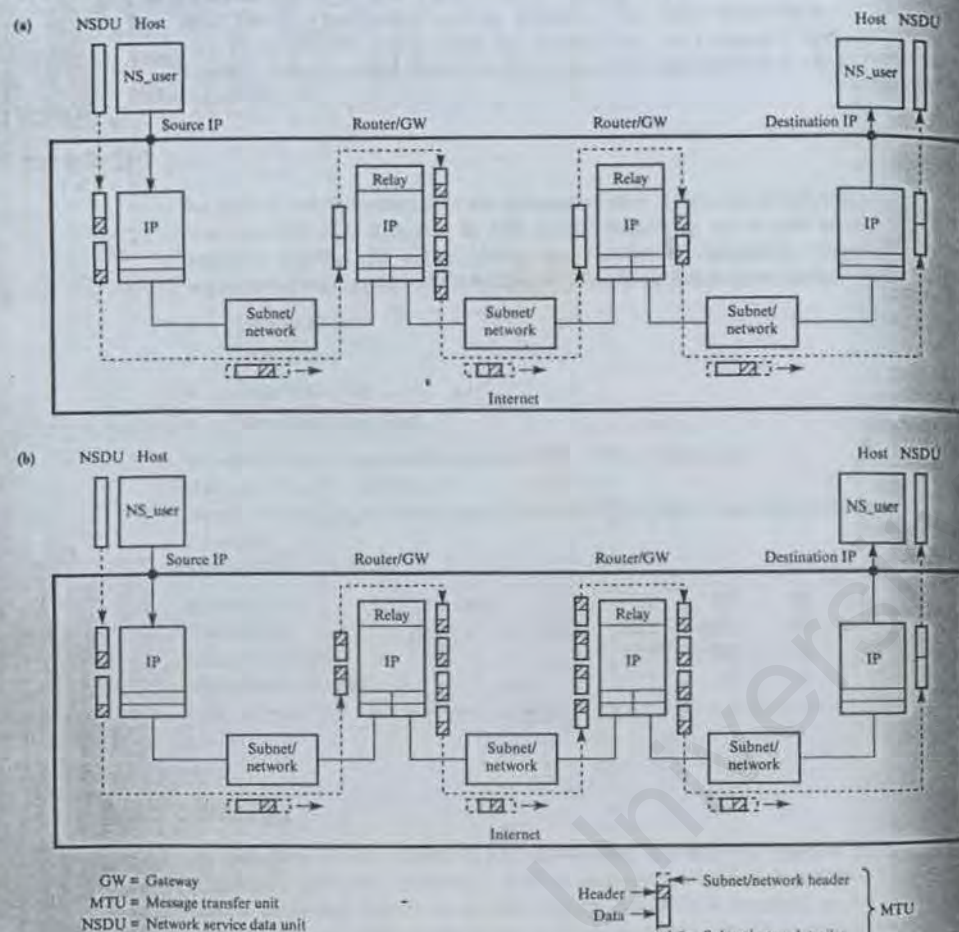
We shall discuss each of these functions separately.

9.5.4 Fragmentation and reassembly

The size of the user data – normally referred to as an NSDU – associated with an NS_user request can be up to 64K or 65 536 octets (bytes). The maximum packet sizes associated with different types of network are much less than this, ranging from 128 octets for some X.25 packet switching networks to over 8000 octets for some LANs. The fragmentation and reassembly functions associated with the IP fragment the NSDU associated with an NS_user request into smaller fragments – segments in ISO terminology – so that they can be transferred across a particular network in appropriately sized datagrams. On receipt of the fragments of data relating to the same NSDU contained in each IP datagram, the IP reassembles the NSDU before passing it on to the destination NS_user.

One of two approaches may be adopted since the maximum packet size may vary from one network to another. Either the fragmentation and reassembly functions can be performed on a per network basis – **intranet fragmentation** – or

Figure 9.10
Fragmentation
alternatives:
(a) intranet;
(b) internet.



On an end-to-end (internetwork) basis, internet fragmentation. The two approaches are shown in Figure 9.10(a) and (b), respectively.

In general, the IP in a host knows only the maximum packet size associated with its local network. Similarly, the IP in each gateway knows only the maximum packet sizes associated with the two networks to which it is connected. With intranet segmentation, the IP in the source host first fragments the NSDU data – the NSDU – into a number of individually addressed datagrams as dictated by the network to which it is attached. It initiates the sending of these either to the

We shall discuss the way in which it obtains the NPA address in Section 9.5.5. On receipt of each datagram, the IP in the host or gateway reassembles the NSDU. Next it refragments the reassembled NSDU into a possibly different set of individually addressed datagrams as dictated by the maximum packet size of the second network.

This procedure is repeated by each gateway until the datagram reaches the IP in the destination host, where the NSDU is again reassembled and passed to the destination NS_user.

With internet fragmentation, the IP in the source host carries out the same fragmentation procedure as before and sends the resulting datagrams to the IP in the first gateway. However, this time the IP does not reassemble the NSDU. Instead it either modifies the appropriate fields and sends the received datagrams directly onto the second network (if the latter can support this size of datagram), or refragments the datagram into smaller fragments (datagrams). In Figure 9.10, we assume that the maximum packet size associated with the second network/subnet is smaller than that used by the first. Consequently, the IP will segment each datagram it receives into a number of smaller datagrams, each with the same source and destination addresses.

This procedure is repeated at the next gateway. However, since in Figure 9.10 the last network/subnet can support a larger packet size than the datagrams it receives, the received datagrams are transmitted directly with only selected modifications to some header fields. As before, the IP in the destination host reassembles the user data from each datagram it receives and passes the resulting NSDU to the destination NS_user.

We can deduce, especially from the packet flows associated with the third network in Figure 9.10, that intranet fragmentation allows the maximum packet size of each network to be used, since the individual fragments are reassembled by each gateway in the route. With internet fragmentation this is not necessarily the case, but it has the advantage that the reassembly processing is not needed at each gateway.

The IP does in fact use internet fragmentation. This may at first appear surprising but it is used because of the problem of lost datagrams. Some networks operate with a best-try connectionless protocol with the possibility that one or more datagrams relating to a single NSDU may be corrupted while being transmitted. As we have seen, with intranet fragmentation the receiving IP in each gateway reassembles the complete NSDU before relaying it to the next network. If any fragments are missing (for example, a datagram is discarded because it has been corrupted), the receiving IP must decide at what point to abort the reassembly function.

To determine this the IP in the source host defines a maximum time limit that a gateway may wait for any datagrams relating to an NSDU during each reassembly operation. Known as the time-to-live, this limit is carried in the header of all datagrams relating to the NSDU. It is set by the IP in the source host and is then decremented by each IP that processes the datagram. If a datagram is

reaches zero at any point during the reassembly processing in a gateway (or host) the reassembly function is aborted and all fragments relating to that NSDU are discarded.

The time-to-live field in each datagram is in multiples of 1 second, so the amount it is decremented by each IP varies depending on the (known) mean transit delay of the associated network. In the case of internet fragmentation, the IP in each gateway still decrements the time-to-live lifetime field in each datagram it receives and discards any datagrams for which the value reaches zero. The IP in the destination host aborts its reassembly operation in the same way. In both cases, if fragments are missing and the reassembly operation is aborted, a time exceeded error message is generated and returned to the IP in the source host.

Example 9.1

An NSDU of 1000 octets is to be transmitted over a network which supports a maximum NS_user data size of 256 octets. Assuming the header in each IP datagram requires 20 octets, derive the number of datagrams (fragments) required and the contents of the following fields in each datagram header:

- Identification
- Total length
- Fragment offset
- More fragments flag

Maximum usable data per datagram = $256 - 20 = 236$ octets

Use, say, $29 \times 8 = 232$ octets

Hence five datagrams are required, four with 232 octets of user data and one with 72 octets

The fields are as follows:

Identification	20 (say)	20	20	20	20
Total length	252	252	252	252	92
Fragment offset	0	29	58	87	116
More fragments flag	1	1	1	1	0

9.5.5 Routing

As each network (or subnet) in an internet may use different types of point of attachment addresses, a system – host or gateway – attached to a network can send a datagram directly to another system only if it is attached to the same network. To route datagrams across multiple networks, the IP in each internet-network gateway must know either the point of attachment address of the destination

gateway along the route to the required destination network, if it is not. Again, the next gateway must be attached to a network to which the gateway is attached. The major problem with routing is how the hosts and gateways within the internet obtain and maintain their routing information.

Two basic approaches are used for routing within an internet: centralized and distributed. With a **centralized routing** scheme, the routing information associated with each gateway is downloaded from a central site using the network and special network management messages. The network management system endeavors to maintain their contents up to date as networks and hosts are added or removed and faults are diagnosed and repaired. In general, for all but the smallest internets, this is a viable solution only as long as each individual network has its own network management system which incorporates sophisticated configuration and fault management procedures.

With a **distributed routing** scheme, all the hosts and gateways cooperate in a distributed way to ensure that the routing information held by each system – hosts and gateways – is up to date and consistent. Routing information is retained by each system in the form of a **routing table** which contains the NPA address to be used to forward each datagram. The Internet uses such a scheme.

The routing procedure associated with the IP first reads the destination IP (NSAP) address from within a datagram and then uses this to find the corresponding point of attachment address – of a host or a gateway – from the routing table. In addition, a set of routing protocols is used to create and maintain the contents of each routing table in a distributed way. The general scheme used within a host IP is shown in Figure 9.11.

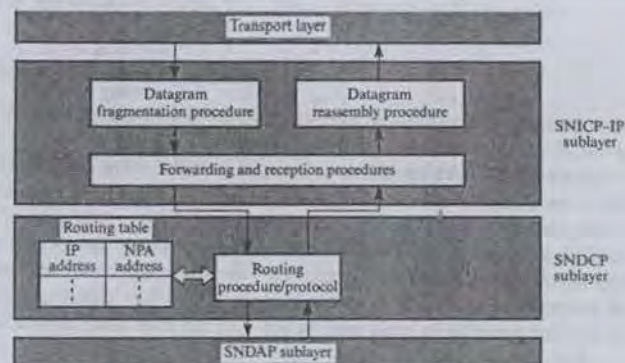


Figure 9.11
General routing
scheme within a host.

SNICP = Subnet independent convergence protocol
SNDCP = Subnet dependent convergence protocol
SNDAP = Subnet dependent access protocol
NPA = (Sub)net point of attachment (address)

Before we discuss the various routing protocols relating to the Internet, let us look at its architecture and the associated terminology. To reflect the fact that the Internet is made up of a number of separately managed and run internets, each internet is treated as an autonomous system with its own internal routing algorithms and management authority. The combined Internet is considered as a **core backbone network** to which a number of autonomous systems are attached. The general architecture is shown in Figure 9.12 together with some (very much simplified) autonomous system topologies.

To discriminate between the gateways used within an autonomous system and those used to connect an autonomous system to the core network, we use the terms **interior gateway** and **exterior gateway**, respectively. The corresponding routing protocols are the **interior gateway protocol (IGP)** and the **exterior gateway protocol (EGP)**. Since the Internet consists of an interconnected set of internets, each of which has evolved over a relatively long period of time, each autonomous system has its own IGP. However, the Internet EGP is, as indeed it must be, an internetwide standard.

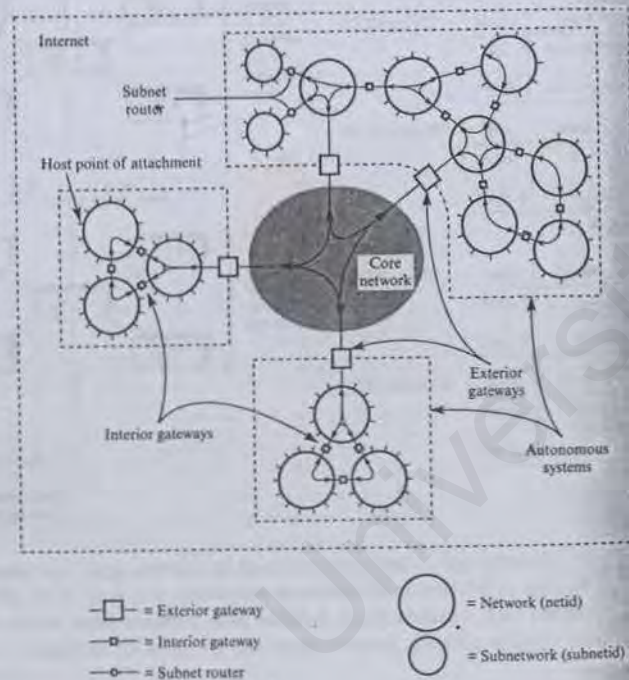


Figure 9.12
General internet
architecture and
terminology.

number of networks/subnets, in a more general application an autonomous system might consist of just one network managed and run by a single corporation. Others might consist of a set of subnets connected to a site-wide backbone network with a single exterior gateway. An example is a site with multiple LANs interconnected by routers. To simplify the discussion, we shall consider only autonomous systems that consist of multiple networks since the presence of subnets just adds another level of routing between an interior gateway and the hosts.

If every gateway and host system in an internet contained a separate entry in its routing table for all other systems, the size of the routing tables and the amount of processing and transmission capacity needed to maintain the tables would be excessive and, for the Internet, unmanageable. Instead, the total routing information is organized hierarchically as follows:

- Hosts maintain sufficient routing information to forward datagrams to other hosts or an interior gateway(s) that is (are) attached to the same network.
- Interior gateways maintain sufficient routing information to forward datagrams to hosts or other interior gateways within the same autonomous system.
- Exterior gateways maintain sufficient routing information to forward datagrams either to an interior gateway, if the datagram is for the same autonomous system, or to another exterior gateway, if it is not.

A number of routing protocols have been developed to implement this scheme. These include an intranet protocol known as the **address resolution protocol (ARP)**, a number of interior gateway protocols (IGPs), and an exterior gateway protocol (EGP). The scope of each protocol and the associated routing tables are shown in Figure 9.13. We shall discuss each separately.

Address resolution protocol

To enable an interior gateway to forward any datagrams it receives for hosts that are attached to one of its local networks, it must keep a record of the hostid and corresponding NPA address – known as an **address pair** – for all the hosts attached to each of these networks. To obtain this information, each host simply informs the local gateway of its existence by sending it its IP/NPA address pair. Typically, this is stored at the host in permanent storage (such as the hard disk) and is then broadcast. With nonbroadcast networks, the address pair of its local gateway(s) is (are) also stored and used directly. As a result, each interior gateway builds up a **local routing table** with the IP/NPA address pairs of all hosts that are attached to each of the networks to which the interior gateway is itself attached.

When a host wishes to send a datagram to another host on the same network, the IP simply sends the datagram to its local gateway for forwarding. Although this must be done for datagrams addressed to hosts on other networks, for hosts attached to the same network it can lead to excessively high overheads.

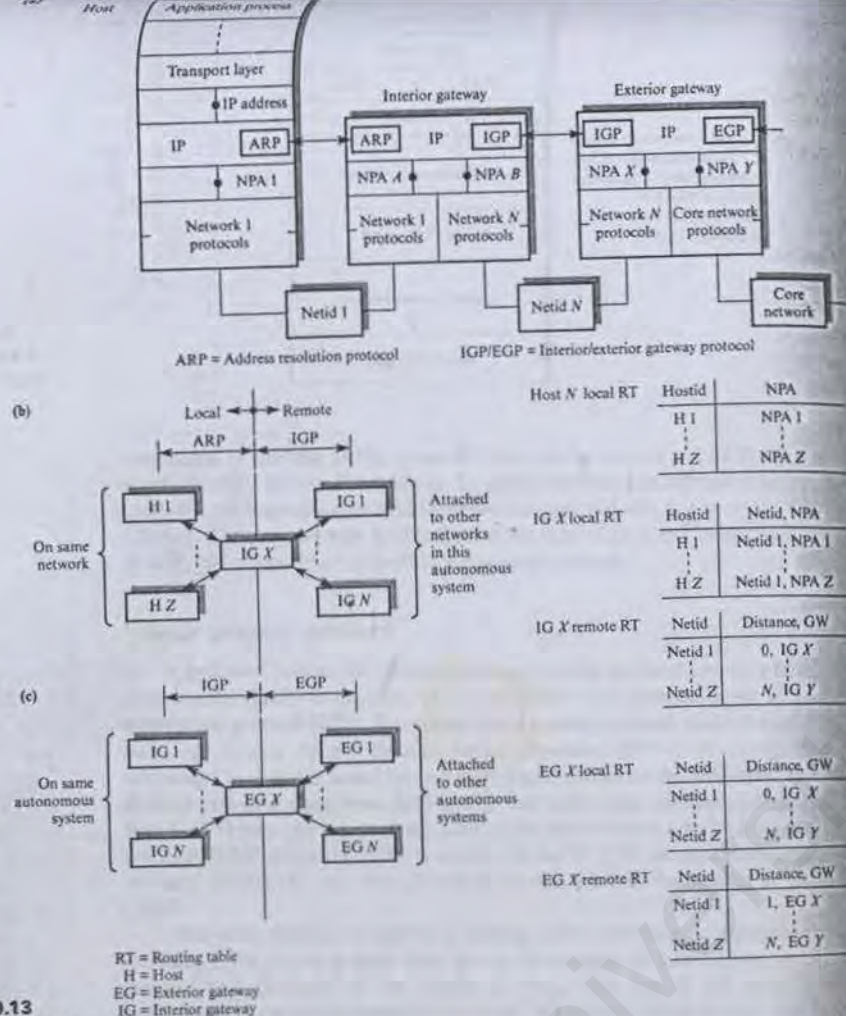


Figure 9.13

Routing protocols:
(a) general architecture; (b) ARP/IGP scope and routing tables; (c) IGP/EGP scope and routing protocols.

especially if a large number of hosts are attached to the network. To overcome this, the IP in each host endeavors to obtain the hostid/NPA address pair of all hosts on the same network with which it communicates. This enables a host to send a datagram to hosts on the same network directly without involving the gateway.

(ARP). ARP forms an integral part of the IP in each host; there is a peer ARP in each interior gateway, as shown in Figure 9.13(a).

Whenever the fragmentation procedure associated with the IP creates a datagram for forwarding, it first passes the address pointer of the memory buffer in which the datagram is stored to the ARP. The ARP maintains a local routing table which contains the hostid/NPA address pairs of all the hosts connected to this network with which the host communicates. If the destination IP address in the datagram is present in the table, then the ARP simply passes the datagram address pointer with the corresponding NPA address to the SNDAP protocol, with the netid field of the IP address set to zero to indicate this network. The SNDAP then initiates the sending of the datagram either by broadcast or directly.

If the NPA address is not present, the ARP endeavors to find it by creating and sending an ARP request message and waiting for a reply. The request message contains both its own IP/NPA address pair and the required (target) IP address. Again, this can either be broadcast – in which case it is received by the ARP in all hosts – or sent directly to the ARP in the gateway using the gateway's (known) NPA address. In the second case, the ARP in the gateway simply relays the message to the required host using its own local routing table and the required destination IP address in the request message.

The ARP in the required destination host recognizes its own IP address in the request message and proceeds to process it. It first checks to see whether the source hostid/NPA address pair is within its own routing table; if not, it enters them. It then responds by returning an ARP reply message containing its own NPA address to the ARP in the requesting host, using the latter's NPA address from the request message. On receipt of the reply message, the ARP in the source host first makes an entry of the requested hostid/NPA pair in its own routing table and then passes the waiting datagram address pointer to the SNDAP protocol together with the corresponding NPA address which indicates where it should be sent. The hostid/NPA pair is recorded by the destination since it is highly probable that the destination host will require it later when the higher-layer protocol responds to the datagram.

As we indicated earlier, the IP/NPA address pair of a host is normally held in permanent storage and read by the computer operating system at startup. With diskless hosts, this is not possible, so an associated protocol known as the reverse address resolution protocol (RARP) is used. The server associated with each set of diskless hosts has a copy of the IP/NPA address pair of all the hosts it serves. When a diskless host first comes into service, it broadcasts an RARP request message to the server containing its own physical hardware network address, that is, its NPA. On receipt of such messages, the RARP in the server responds with a reply message containing both the IP address of the requester and its own IP/NPA address pair. In practice, the format of the request and reply messages associated with ARP and RARP are the same, as shown in Figure 9.14.

The operation field specifies the particular message type: ARP request/reply, RARP request/reply. When making an ARP request, the sender writes its own hardware address (HA) and IP address in the appropriate fields together with the

Protocol type
HLEN
PLEN
Operation
Sender hardware address
Sender IP address
Target hardware address
Target IP address

HLEN = Hardware address length
 PLEN = IP address length
 Operation = 1 ARP request
 = 2 ARP response
 = 3 RARP request
 = 4 RARP response

Figure 9.14
 ARP and RARP
 message formats.

destination IP address in the **target IP** field. In the case of a RARP, the sender simply includes its own HA address. To ensure that the HA address is interpreted correctly, the **hardware type** field identifies the type of LAN, for example, CSMA/CD is 1. The **protocol type** field indicates the type of protocol being used: ARP, RARP, and others to be defined in subsequent sections.

Interior gateway protocol

As we indicated earlier, the interior gateway routing protocol can vary from one autonomous system to another. The most widely used protocol is the **IP routing information protocol (RIP)**. It is a **distributed routing protocol** which is based on a technique known as the **distance vector algorithm (DVA)**. A more recently introduced protocol is based on two algorithms known as the **link state (LS)** and **shortest-path-first algorithms (SPF)**. The **link state open shortest-path-first (link state OSPF)** protocol has been adopted as the international standard for use with the ISO CLNP. Since the DVA is specific to the TCP/IP we shall discuss it here. We shall discuss the link state OSPF in Section 9.8.3 in the context of the ISO CLNP.

The term *distance* is used as a **routing metric** between two gateways. For example, if the metric is *hops*, then this is the number of intermediate networks between two gateways. If the metric is *delay*, then this is the mean transit delay between the two gateways, and so on. Whichever metric is used, the DVA uses a distributed algorithm to enable each interior gateway in an autonomous system to build up a table containing the distance between itself and all the other networks in that system.

Initially, each gateway knows only the netid of each network to which it is attached as well as the IP/NPA address pair of each gateway attached to these networks. Typically this information is entered by management when the gateway

the adjacency table for the gateway. The format of the routing table associated with an interior gateway was shown in Figure 9.13(b). If the metric is hops, the remote routing table contains simply the netid of each of its local networks, a distance of zero, and its own IP address as the gateway from which the distance applies. Similarly, if the metric is delay, this is determined by a gateway which sends a message (datagram) to each of the gateways attached to its own networks and measures the time delay before it receives the responses. The distance is then set to, say, half of these values.

Periodically, each gateway sends the current contents of its (remote) routing table to each of its neighbors. Based on the contents of its neighbors' tables which it receives, it updates, or adds to, the contents of its own routing table. The receiving gateway simply adds the known distances to each of its immediate neighbors to the distances contained in the received tables. Since this procedure repeats, after each iteration the routing table starts to build up as new distances are reported. If a reported distance to a network is less than a current entry, the entry is updated. After a number of iterations, each gateway has an entry for each of the networks in the autonomous system. The time taken to achieve this is a function of the size of the system and the frequency with which routing information is exchanged. The time for routing information to propagate throughout the system is known as the **route propagation delay**.

For example, consider the simple network shown in Figure 9.15(a) and assume the metric is hops. The way in which the routing tables build up is shown in part (b). The initial contents of each gateway simply contain the netid of its local networks. For this network, the contents of each routing table are complete after just two exchanges of routing tables. The final routing table for each gateway contains the distances to each network in the system and the immediate neighbor gateway to be used to reach it. Thus from gateway 1, the distance to netid 6 is two hops – that is, two intermediate networks – via gateway 2. At gateway 2, netid 6 is a distance of one hop via gateway 3 to which netid 6 is attached.

We can readily deduce that a metric of hops can lead to the selection of inferior routes. For example, if the delay metric associated with each network is the same as its netid, it would be quicker to go from gateway 4 to netid 6 via gateways 1, 2, and 3 in three hops rather than 5 and 6 with two hops. The delay metric often gives a better performance. A protocol that uses delay is **HELLO**. As its name implies, the delay is determined by periodically sending **hello messages** to each of its neighbors and timing their responses.

To ensure that table entries reflect the current topology of the network when faults develop, each entry has an associated timer. If the entry is not confirmed within a defined time, then it is timed-out. This means that each gateway transmits its complete routing table at regular intervals, typically 30 seconds. For a small network this is not necessarily a problem, but for large networks the overheads associated with the distance vector algorithm can be very high. Also, gateways may have dissimilar routes to the same destination since entries are made in the order in which they are received and equal distance routes are discarded. As a result, datagrams between certain routes may loop rather than going directly to

different systems must first agree to exchange such information. This is the role of the neighbor acquisition and termination procedure. When two gateways agree to such an exchange, they are said to have become neighbors. When a gateway first wants to exchange routing information, it sends an acquisition request message to the EGP in the appropriate gateway which then returns either an acquisition confirm message or, if it does not want to accept the request, an acquisition refuse message which includes a reason code.

Once a neighbor relationship has been established between two gateways – and hence autonomous systems – they periodically confirm their relationship. This is done either by exchanging specific messages – hello and I-heard-you – or by embedding confirmation information into the header of normal routing information messages.

The actual exchange of routing information is carried out by one of the gateways, which sends a poll request message to the other gateway asking it for the list of networks (netids) that are reachable via that gateway and their distances from it. The response is a routing update message which contains the requested information. Finally, if any request message is incorrect, an error message is returned as a response with an appropriate reason code.

As with the other IP protocols, all the messages (PDUs) associated with the EGP are carried in the user data field of an IP datagram. All EGP messages have the same fixed header; the format is shown in Figure 9.16.

The version field defines the version number of the EGP. The type and code fields collectively define the type of message while the status field contains message-dependent status information. The checksum, which is used as a safeguard against the processing of erroneous messages, is the same as that used with IP. The autonomous system number is the assigned number of the autonomous system to which the sending gateway is attached; the sequence number is used to synchronize responses to their corresponding request message.

Neighbor reachability messages contain only a header with a type field of 5, a code of 0 = hello, and a 1 = I-heard-you.

Neighbor acquisition messages have a type field of 3; the code number defines the specific message type. The hello interval specifies the frequency with which hello messages should be sent; the poll interval performs the same function for poll messages.

A poll message has a type field of 2. The code field is used to piggyback the neighbor reachability information: a code of 0 = hello and a code of 1 = I-heard-you. The source network IP address in both the poll and the routing update response messages indicates the network linking the two exterior gateways. This allows the core network itself to consist of multiple networks.

The routing update message contains the list of networks (netids) that are reachable via each gateway within the autonomous system arranged in distance order from the responding exterior gateway. As indicated, this enables the requesting gateway to select the best exterior gateway through which to send a datagram for forwarding within an autonomous system. Notice that to conserve space, each netid address is sent in three bytes (24 bits) only with the most significant 8-bit hostid field missing. The latter is redundant for all class types.

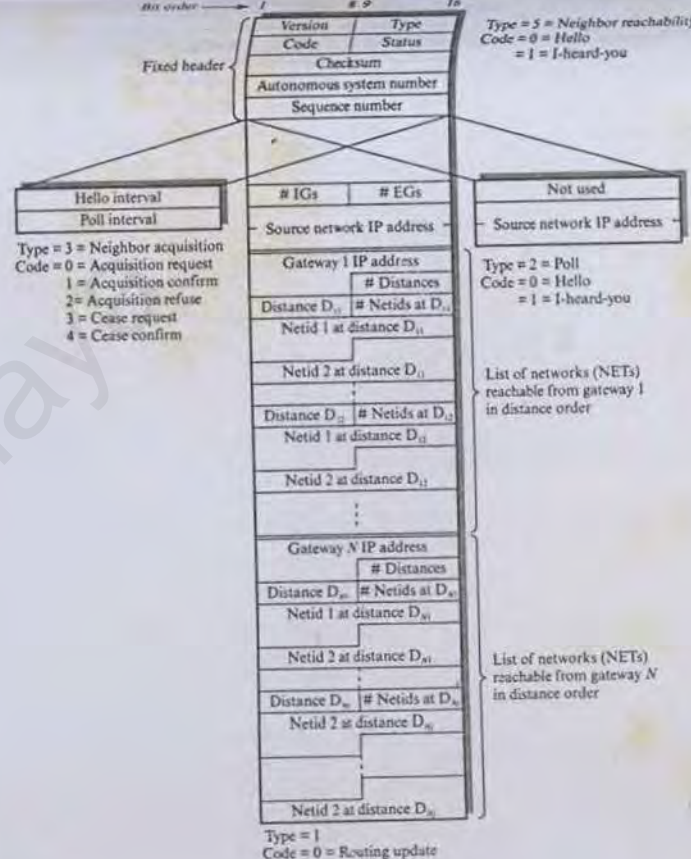


Figure 9.16
EGP message formats.

9.5.6 Internet control message protocol

The internet control message protocol (ICMP) forms an integral part of all IP implementations. It is used by both hosts and gateways for a variety of functions, and especially by network management. The main functions associated with the ICMP are as follows:

- Error reporting
- Reachability testing
- Congestion control