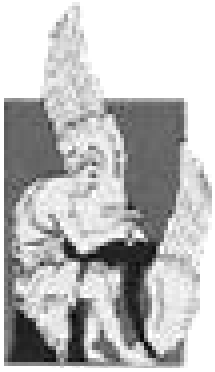


Technical University of Crete

Electronic and Computer Engineering

Department

Thesis

Design and Implementation of a TCP/IP core for reconfigurable logic

By Christophoros Kachris

Supervisor Professor: Professor Apostolos Dollas

Committee: Professor Apostolos Dollas
Professor Michalis Paterakis
Ass. Professor Dionisios Pneumatikatos

July 2001

…to my parents.
Christophoros Kachris

Acknowledgments

For this thesis I would like to thank professors Apostolos Dollas and Dionisios Pneumatikatos for their useful recommendations.

I would also like to thank Mr. Markos Kimionis and Ioannis Hatzakis for their help in technical matters.

Moreover I thank the under-graduate and post-graduate students for their help and at the end I would like to thank my parents and my friends for their support.
I

Contents

Chapter 1
Introduction

Enough research will tend
to support your theory.
Murphy's Law

In our days, with the ongoing extension of technology, we are witnessing the emergence of new computing machines, which will bring us far beyond the desktop PC. Devices featuring advanced connectivity and Internet functionality will soon become the standard in computing. In fact, we're on the verge of a revolution that will bring us a wave of smart, electronic devices that can be controlled, gather information, and distribute data via the web.

Virtually every embedded designer is looking to use Internet, to enhance or expand the reach of embedded systems. This need for connecting devices directly into Internet has leads many great manufacturers to implement ASICs (Application Specific Integrated Circuits) or reusable libraries for microcontrollers, specially designed for this purpose.

The main scope of this thesis is to implement an IP Core (Intellectual Property Core) that will mainly support the TCP/IP protocol stack and will be able to be incorporated in any embedded system based on FPGA. In particular the IP Core will support the following protocols:

IP (Internet Protocol),
ICMP (Internet Control Message Protocol),
UDP (User Datagram Protocol) and
TCP (Transfer Control Protocol).

In addition the IP core has to support the Ethernet local area network protocol. The special feature of this project is its flexibility, since it consists of independent blocks that can easily be changed in order to support future changes in protocols. The choice of the FPGA family that the IP Core will be synthesized can change significantly the speed of design in order to support 10,100 or even 1Gbits Ethernet networks in the future, given the rapid improvement of FPGAs performance.

The applications of this TCP/IP protocol stack vary in a wide range of devices. A lot of appliances will be able not only to gather and distribute information via the Internet but also to be re-configured through the web. The potential applications of this protocol stack can be extended to a wide range of devices in different fields, such as Ethernet network analyzers, smart Ethernet hubs or even hardware-based packet sniffers. Another potential application of TCP/IP core is devices that will support VoIP (Voice over Internet Protocol).

The second chapter of this document provides a general description of different types of related ASICs and FPGAs, the protocols that support each of them and the applications that they are destined. The third chapter covers the supported protocols, examines the headers of these protocols and set the specifications of our design. The fourth chapter provides a detailed description of the architecture and analyzes each module of this architecture while the fifth chapter describes the implementation board and examines several issues of Ethernet transceivers. The sixth chapter covers the several steps of simulation and verification of this design and the last chapter examines the potential applications of this design and the future work that can be done in order to upgrade this IP core.

Chapter 2

Relevant research

Never set a problem if you
are not aware of its solution.
Murphy's Law

The last months many companies have understand the need for Internet connectivity of embedded systems and have present either ASICs or reusable libraries that support whole or a part of the TCP/IP protocol stack.

The first quarter of 2001 many manufactures such as Seiko Inc, Triscent Inc., Huyndai Inc and others have implemented ASICs that support the IP v4.0 protocol stack. Unfortunately the continuous changes in protocols, and generally in technology, are so rapid that some times the time to make an ASIC exceeds the time that the protocol is used. For example, the current ASICs don't support IP v6.0 protocol stack and new ASICs have to be made to comply with the new protocols. Furthermore the most of these ASICs support only the TCP/IP protocols, which mean that other circuits must be used for the physical layer and for the specific application that will be used.

Many embedded systems designers have implemented protocols stacks in micro controllers, in order to reduce the number of circuits that will be used in the application. For example, a web server has been implemented in only one micro controller [35]. But this solution has many disadvantages. First of all, the code for the protocol stacks cannot be reused in micro controllers even of the same company, but of different family because of their discrepancies. Moreover, the system clock of the micro controllers in not enough to respond to the new fast local networks, but only to the older 10Mbits/sec Ethernet networks.

On the other hand, the use of an IP Core (Intellectual Property Core) is much more convenient for a designer, since it is technology independent and can be reused in any FPGA (Field Programmable Gate Array), of any manufacturer, of any technology. An IP Core is a set of code written in hardware description languages such as VHDL (Very high speed integrated circuit Hardware Description Language) or Verilog that can be used as part of a project. In the case of an embedded system implemented in FPGA that have to support Internet functionality, the choice of a TCP/IP Core is ideal, since it can be easily incorporated in the whole project. Especially with the new SoCs (Systems On a Chip) that contain programmable gate arrays, static RAMs and some type of microcontroller core, the Internet connectivity can be implemented in only one chip.

2.1 ASIC with TCP/IP functions

The ASIC that have been created are not uniform, meaning that there in not any standard TCP/IP protocol stack especially for embedded systems that is supported. Each ASIC has each own features and some times it is targeting a special application. Some of them include some type of memory (SRAM or Flash) in order to reduce the total amount of chip for an application. Below we describe the ASICs that have been presented by some companies and we describe the main features of each ASIC.

2.1.1 Hyundai Electronics

Hyundai Electronics Industries Co. has developed a CMOS IC with a complete TCP/IP protocol suite to facilitate internet connection for embedded application [40]. The "HMS91C7432" chip implements complete TCP/IP protocol suite that includes PPP; IP; ICMP;TCP;UDP; DNS; SMTP; POP3 protocol and additionally a general MODEM driver. The "HMS91C7432" is built-in with 96Kb SRAM (12K x 8) for communication and buffering and a full duplex UART as DTE for ease of serial modem connection. The ASIC has also an 8 bits Data/Command port and 4 control pins to facilitate control and communication between the Host MCU and the modem. There are only 20 simple commands, each of which is a single byte long order to establish and to complete the

whole internet communication and 45 respond codes for the Host MCU to monitor the communication status.

A typical application of this chip, according to the company, is a data bank email composer as shown below.

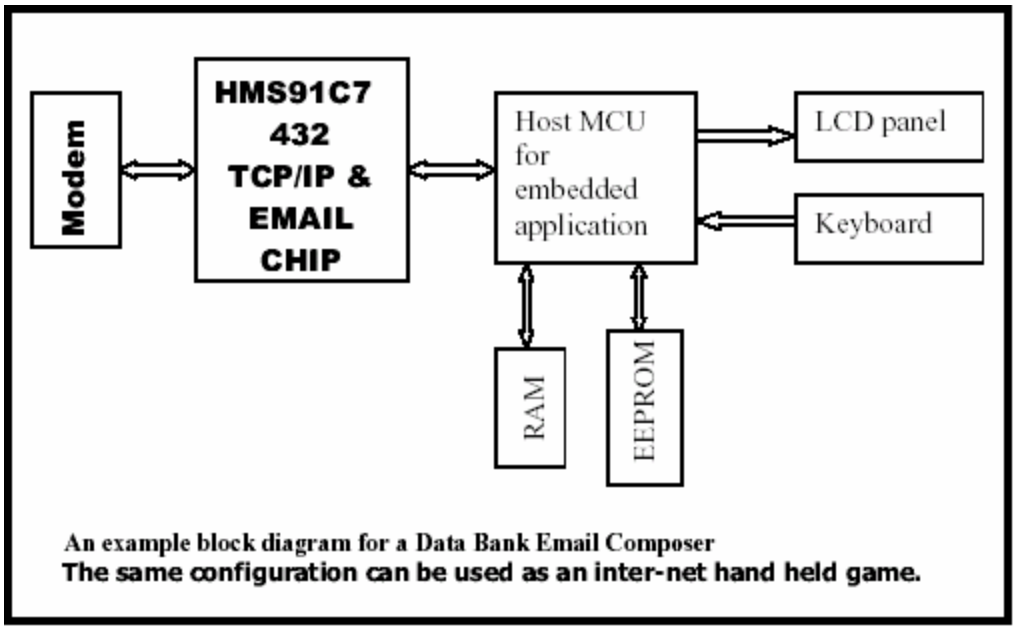


Figure 2.1.1 A typical application of HMS91C7432 chip

The features of this chip concisely are:

- Implementation of the complete TCP/IP protocol suite
- Built-in Email sending and reception function.
- Standard SMTP protocol stack.
- Standard POP3 protocol stack.
- Standard PPP protocol stack to facilitate dial-up network log on.
- Standard DNS protocol stack, resolve URL with dynamic DNS server.
- Serial modem driver built-in.
- Support V.90 56Kflex modem or lower.

- 8 bit parallel interface to the user application.
- Serial DTE port for ease of modem interface.

2.1.2 Seiko Instruments Inc.

Seiko has designed the IC “iChip S-7600A”, by using low power CMOS design, which contains TCP/IP Protocol Stacks that act as an accelerator between MPU and Internet or network which uses TCP/IP protocol [39]. “iChip S7600A” is designed to provide Internet connectivity to devices using popular microcontrollers and provides the functionality necessary for remote management and monitoring applications, portable email, Internet downloads, network access, and much more. This chip is a completely self-contained, drop-in solution for any device requiring networking connectivity and provides a high connect speed with low power consumption, integrating full TCP/IP, PPP, and UDP protocols, and 10K of on-chip SRAM for those protocol supports. The transport and network layers contain:

- Two general sockets that provide connectivity between the application layer and the transport layer.
- The TCP/UDP module that allows for reliable (retransmission) and unreliable (no retransmission) datagram deliveries.
- An IP module that provides connectionless packet delivery.
- The PPP module that provides point-to-point connection link between two peers.

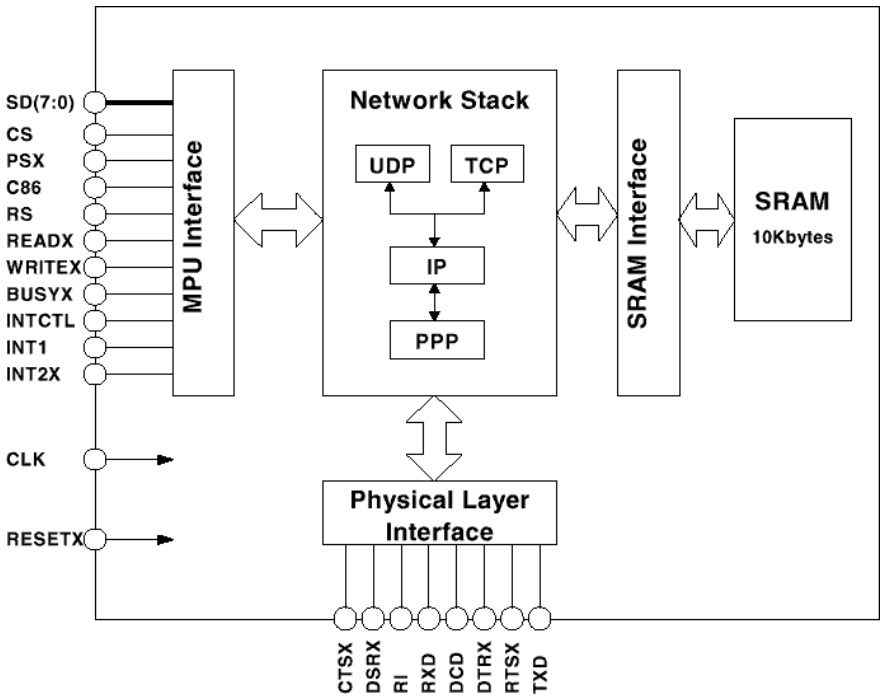


Figure 2.1.2 iChip S-7600A block diagram

The block diagram of this chip is shown above.

2.1.3 Connect One

The company Connect One has developed the "iChip", an ASIC that can be used as mediator device between a host processor and an Internet communication platform [41]. "iChip" constitutes a complete Internet messaging solution for non-PC embedded devices. It acts as a mediator device to completely offload the host processor of Internet-related software and activities. An industry-standard asynchronous serial link connects iChip to the host processor. Programming, monitoring and control are fully supported using Connect One's AT+i extension to the standard AT command set.

The main features of this chip are:

- Microprocessor-controllable through a standard serial connection or optional parallel bus.
- Supports remote firmware update by host, email, or direct modem-to modem communications.
- Includes onboard 128KB SRAM and 256KB or 512KB flash memory.
- Driven by Connect One's "AT+i" extension to the AT command set.
- Stand-alone Internet communication capabilities.
- Opens up to 5 TCP or UDP sockets.
- Onboard non-volatile memory stores all functional and Internet related parameters.
- Supports several layers of status reports.
- Supports the following Internet Protocols and formats:
  PPP, LCP, IPCP, IP, TCP, UDP, DNS, SMTP, POP3, HTTP
- Supports data modems up to 56 Kbps throughput or provides 10BaseT Ethernet LAN connectivity.

The block diagram of this chip is depicted below.

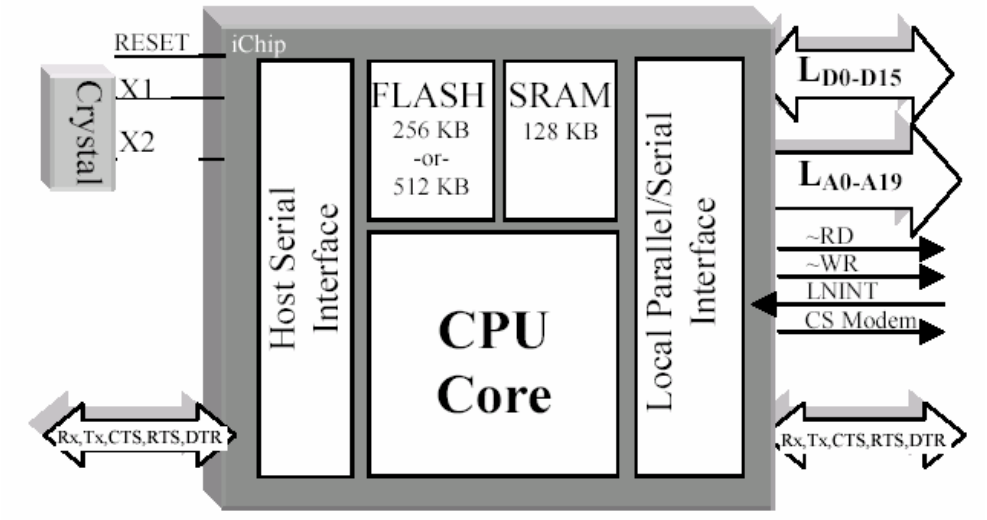


Figure 2.1.3 iChip architecture

2.1.4 eDevice Inc.

The company eDevice has designed the SmartStack, a unique embedded software solution that allows Original Equipment Manufacturers (OEMs) to add Internet connectivity to their equipment easily and at low cost. SmartStack uses a single Digital Signal Processor (DSP) to execute the Media Access Control (MAC) layer – the standard functions needed to connect to a Local Area Network (LAN - and the Internet protocols at the same time. Implementation of the Address Resolution Protocol (ARP) with the MAC layer allows SmartStack to associate the Ethernet physical address with the IP address assigned to the device. Integration of SmartStack for Ethernet into any piece of equipment converts it to a stand-alone client that can be connected to any Ethernet LAN with access to the Internet. A simple software interface allows for communication with the attached equipment. This attached device uses an AT-like command set to configure parameters or launch actions like sending/receiving emails or downloading files from an FTP server. Using these AT-like commands, network parameters can be easily configured allowing quick integration into a LAN environment thanks to SmartStack.

The supported protocols of this DSP are:

- ARP
- IP
- TCP
- UDP
- SMTP
- POP3
- HTTP
- FTP
- Telnet

The block diagram of this ASIC is shown below.

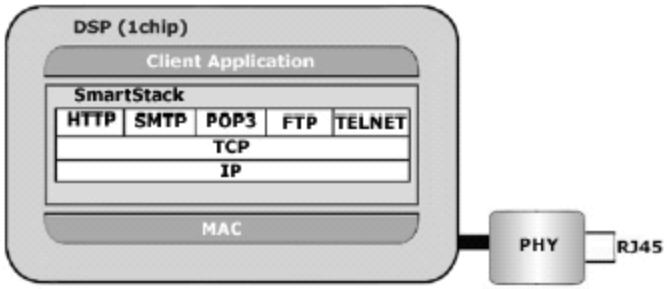


Figure 2.1.4 SmartStack's block diagram

2.2 Reusable libraries for embedded applications

A lot of companies have developed protocols stacks not in ASICs but in form of microprocessors libraries. These libraries are technology-dependent which means that they are designed for a particular processor manufacturer such as Intel, Atmel or MicroChip.

2.2.1 Embedded Power Co.

Embedded Power is one of the companies that have developed library solutions for Internet functionality of embedded systems [43]. They have designed "RTXCnet", a special Network Communication Suite that is an array of fully portable software components for networking communications in embedded applications. Each of these components is fully integrated with the RTXC real-time OS kernel. In a fraction of the development time required with other products, the RTXCnet network components allow the embedded designer to build applications that transfer files, commands, and data all over the world. The library "RTXCnet" includes a comprehensive suite of high performance components to support the most popular protocols in use today, such as TCP, IP, ARP, UDP, ICMP, SNMP, PPP, SLIP, HTTP, FTP and DHCP. The main advantage of this solution is that the embedded system designer can select exactly the protocols that the systems must support (such as RTXip, RTXtcp, RTXftp), thus maximize the available space for his own application.

2.2.2 LiveDevices Co.

The LiveDevices Standard TCP/IP stack provides the core protocols required for full Internet connectivity, together with a rich set of basic application services [44]. Resource requirements are minimal, and multiple concurrent connections are supported. This stack supports IP, TCP, UDP and ICMP protocols but at this time the only current target is the Microchip PIC18CXXX. The main advantage of this solution is the capability of the designer to control the properties of the TCP/IP, such as the port, through a windows-based program. Additionally the code is dependent of each additional port that is supported. The typical RAM sizes, according to the company, are shown below.

Component	**RAM (bytes)**	**ROM (bytes)**
PPP	132	4147
SLIP	2	269
Modem control	11	528
IP	126	3152
UDP	0	712
TCP	3	3872
Incremental costs		
Per UDP connection	15	0
Per TCP connection	28	0

2.2.3 Rabbit Semiconductors Co.

Rabbit Semiconductors has introduced the TCP/IP core, specially designed for their Rabbit 2000 microprocessor [37]. The TCP/IP source code is provided additionally with ICMP, HTTP (includes facilities for SSI, CGI routines, cookies, and basic authentication), SMTP, FTP and TFTP (client and server) capabilities. The main advantage of this solution is that the users can directly write to TCP or UDP sockets to develop custom applications.

The supported protocols of this stack are:

- Socket Level TCP.
- Socket Level UDP.
- ICMP
- HTTP
- SMTP
- FTP
- TFTP

2.2.4 CMX Systems Co.

A company that also provides Internet ready solutions for a wide range of processors is CMX Systems [46]. They have developed a TCP/IP stack that support the most of the known protocols such as, IP, UDP, TCP,PPP, SLIP, ARP, HTTP, Web Server, FTP Server. According to the company in the future TFTP, POP3 and SMTP protocols will also be supported. This TCP/IP stack can be incorporated into a wide range of known microprocessors such as 8051, Atmel AVR, MicroChip PIC and STMicroElectronics ST10. Many companies have adopted this protocol stack such as Triscend Co. and Texas Instruments DSP.

2.2.5 XCoNet Project

Chips on the Net is a project developed by Xilinx and University of Hawaii and its purpose is to connect integrated circuits directly to the Internet. This project has been implemented into a Xilinx demo board and it is connected with an Ethernet transceiver board, an LCD panel and four 512Kbytes SRAMs. The design supports TCP/IP protocol and has been synthesized into a Xilinx Virtex FPGA (XSV800). It is the only project until now that support TCP/IP stack for FPGAs, but we are not aware if it can be considered as IP core, thus it can be incorporated into other FPGAs as part of a larger design.

Chapter 3

Specifications

Measure with micrometer,
mark with chalk,
cut with axe.
Murphy's Law

This chapter presents the required specifications that the design must support. In order to be more accurate there is a brief presentation of each supported protocol, an analysis of the protocol's header and a short description of header's field of each protocol is supported by our design. Before analyzing the protocols it is necessary to present the OSI architecture.

3.1 OSI Layers

The International Standards Organization (ISO) established a framework for standardizing communications systems called the Open Systems Interconnection (OSI) Reference Model. The OSI architecture defines the communication process as a set of seven layers, with specific functions isolated and associated with each layer. Each layer covers lower layer processes, effectively isolating them from higher layer functions. This way, each layer performs a set of functions necessary to provide a set of services to the layer above it.

7	**Application Layer** *User Applications*	
6	**Presentation Layer** *Upper layer protocols*	
5	**Session Layer** *Sockets*	
4	**Transport Layer** *UDP, TCP*	**IP Core**
3	**Network Layer** *ARP, IP, ICMP*	
2	**Data Link Layer** *LLC, MAC*	
1	**Physical Layer** *PMI, MII, PMD, MMD*	**Transceiver**

Figure 3.1 OSI layers

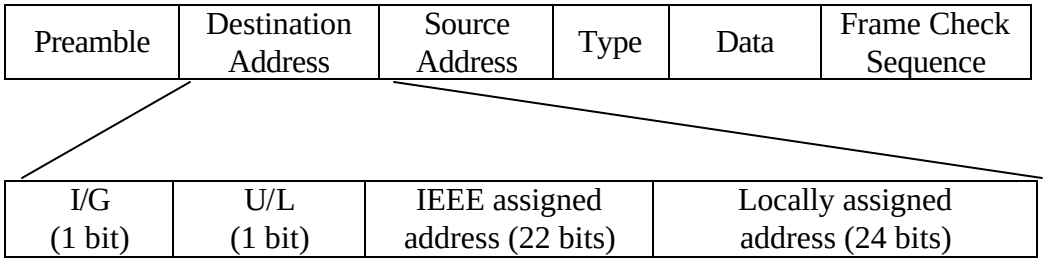
In a typical PC the lower two layers are supported by the Network Interface Card (NIC), while the upper six are supported by the operating system and the applications. Our implementation supports the Data link (both Logical Link Control and Media Access Control), the Network and the Transport layer. The physical layer is supported by the Ethernet transceiver. In addition the design partially supports the Session layer, since it provides a socket for the embedded applications. The FPGA communicates with the transceiver through the Medium Independent Interface (MII) which is described thoroughly in chapter 5.

The supported Transport layer protocols are UDP and TCP, while the supported Network layer protocols are ARP, IP and ICMP. The supported Data link layer network is Ethernet. Each of these supported protocols has its own header. When a protocol header is passed to the layer beneath, the datagram including the layer's header is treated as the entire datagram for that receiving layer, which adds its own protocol header to the front.

3.2 Ethernet networks

Ethernet is one of the most commonly used Local Area Networks (LANs).It was originally developed at Xerox's Palo Alto Research Center as a step towards an electronic office communication system, and it has since grown in capability and popularity [3]. Ethernet is a hardware system that provides the data link and physical layers of the OSI model. There are several different versions of Ethernet, each one with a different data transfer rate, such as Thin Ethernet (10Base2), Thick Ethernet and Twisted-Pair Ethernet (10BaseT). The most common are Ethernet 10BaseT and 100BaseTX with 10Mbits and 100Mbits data transfer rate respectively.

Our implementation is designed to support both 10BaseT and 100BaseTX Ethernet networks. The header of a typical Ethernet frame is shown below.



Preamble: 8 bytes

The preamble field consists of eight bytes of alternating 1 and 0 bits. The purpose of this field is to announce the frame and to enable all receivers on the network to synchronize themselves to the incoming frame. In addition, this field ensures that there is a minimum spacing period of 9.6 ms, in 10Mbits networks, between frames for error detection and recovery operations.

Destination and Source Address: 6 bytes

The source and destination address identifies the transmitter and the recipient of the frame respectively. This field consists of three subfields: I/G bit, U/L bit and 46-bit address. The one bit I/G subfield is set to a 0 to indicate that the frame is destined to an individual station, or 1 to indicate that the frame is address to more than one station (group address). The U/L subfield indicates whether the destination address is an address assigned by the IEEE (universally administrated) or by the organization via software (locally administrated). The first three bytes of the address represent the address assigned by the IEEE to the manufacturer of the NIC. The vendor then assigns the last three bytes for each one of its NICs.

Type: 2 bytes

This field identifies the higher-level protocol contained in the data field, thus telling the receiving device how to interpret the data field. This implementation supports the ARP and IP type which means that the type value must be 0x0806 or 0x0800 respectively. In any other case the packet is discarded.

Frame Check Sequence Field: 4 bytes

The frame check sequence field provides a mechanism for error detection. Each transmitter computes a cyclic redundancy check (CRC) that covers the address fields, the type and the data field. The transmitter then places the computed CRC in the four bytes-wide CRC field. The CRC algorithm treats these fields as one long binary number [53]. In summary, the data D is multiplied by X^n and divided by the generator polynomial G(X), where G(X) is:

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

The quotient Q is then discarded, and the remainder R is added to the divided $X^n * D$, according to the following equation:

$(X^n * D) + R = (Q* G) + 0$

At the receiving end, the first part of the transmitted information is the original data D and the second part is the remainder R. This entire quantity is divided by the same generator polynomial G(X), and the quotient Q is discarded. If the transmitted data has no errors the remainder of this division is 0.

In our design the division is implemented with linear-feedback shift registers, which is a method that yields the same results as subtract and division process. This method is explained extensively in the next chapter.

3.3 Address Resolution Protocol

The ARP protocol is responsible for converting IP addresses to physical addresses in order to eliminate the need for applications to know about the physical addresses [24]. Essentially, ARP is a table with a list of the IP addresses and their corresponding physical addresses. The implementation is capable of responding to any ARP request that addresses its IP address by sending a packet with its physical address, but it does not implement the ARP table.

The header of a typical ARP packet is shown below.

<table>
<tr><td colspan="2">Hardware Type (16 bits)</td></tr>
<tr><td colspan="2">Protocol Type (16 bits)</td></tr>
<tr><td>Hardware Address Length</td><td>Protocol Address Length</td></tr>
<tr><td colspan="2">Operation Code</td></tr>
<tr><td colspan="2">Sender Hardware Address</td></tr>
<tr><td colspan="2">Sender IP Address</td></tr>
<tr><td colspan="2">Recipient Hardware Address</td></tr>
<tr><td colspan="2">Recipient IP Address</td></tr>
</table>

Hardware type: 2 bytes

The type of hardware interface.

Protocol type: 2 bytes

The type of protocol used by the sending device.

Hardware Address Length: 1 byte

The length of each hardware address in the datagram, given in byte.

Protocol Address Length: 1 byte

The length of the protocol address in the datagram, given in byte.

Operation Code: 2 bytes

The operation code indicates whether the datagram is an ARP request or an ARP reply. If the datagram is a request, the value is set to 1 and if it is a reply the value is set to 2.

Sender Hardware Address: 6 bytes

The hardware address of the sending device.

Sender IP Address: 4 bytes

The IP address of the sending device.

Recipient Hardware Address: 6 bytes

The hardware address of the recipient device.

Recipient IP Address: 4 bytes

The IP address of the recipient device.

3.4 Internet protocol

The Internet Protocol (IP) is a primary protocol of the OSI model responsible for addressing the datagrams between computers and managing the fragmentation process of these datagrams [21].

The header of an IP datagram when encapsulates a packet is shown below.

Version	Length	Type of Service	Total Length	
Identification			Flags	Fragmentation Offset
Time to Live		Protocol	Header Checksum	
Source IP Address				
Destination IP Address				
Options (variable length)				

Version: 4 bits

This is a 4-bit field that contains the IP version number of the protocol. The current design supports only version 4, but it can be easily adjusted to support version 6.

Internet Header Length: 4 bits

This 4-bit number indicates the total length of the IP header, including the Options field in number of 32 bit words (4 bytes), and thus points to the beginning of the data. Since this is limited to a 4-bit number, the maximum length of an IP header is fixed at 15 * 4 bytes, or 60 bytes. If no options are present, the value of the header length is always 0x5 (20 bytes) which is the minimum value for a correct header. This will be the value in all IP datagrams sent by this implementation since these packets don't include options.

Type of Service: 8 bits

The Type of Service provides an indication of the abstract parameters of the quality of service desired. These parameters are to be used to guide the selection of the actual service parameters when transmitting a datagram through a particular network. Several networks offer service precedence, which somehow treats high precedence traffic as more important than other traffic (generally by accepting only traffic above certain precedence at time of high load). The major choice is a three way tradeoff between low-delay, high-reliability, and high-throughput. In the current design this byte is ignored and is set to 0 when transmitting a datagram. The bits of Type of Service are shown below.

TOS Bit Definition

Bit	7	6	5	4	3	2	1	0
Definition	0	C	R	T	D	Precedence[2:0]		

Bits 0-2: Precedence.

Bit 3: 0 = Normal Delay, 1 = Low Delay.

Bit 4: 0 = Normal Throughput, 1 = High Throughput.

Bit 5: 0 = Normal Reliability, 1 = High Reliability.

Bit 6, 7: Reserved for Future Use.

Total Length: 16 bits

This field represents the total number of bytes in the IP datagram. Using this field in conjunction with the header length, the start of the data field of the IP datagram can be calculated. Since this field is limited to 16 bits, the largest IP datagram that can be sent is 64K.

Identification: 16 bits

An identifying value assigned by the sender to aid in assembling the fragments of a datagram. In this implementation the identification field is ignored when receiving datagrams since this design does not support fragmented IP packets. This field is initialized to 0 and is incremented by 1 when transmitting.

Flags: 3 bits

The Flags field is a 3-bit field, the first bit of which is left unused. The remaining two bits are dedicated to flags called DF (Don't fragment) and MF (More fragments), which control the handling of the datagrams when fragmentation is desirable. Because the order of the fragments' arrival might not correspond to the order in which they were sent, the

MF flag is used in conjunction with the Fragment Offset field to indicate to the receiving machine the full extent of the message.

Fragment Offset: 13 bits

If the MF flag is set to 1, the fragment offset contains the position in the complete message of the sub message contained within the current datagram. This enables IP to reassemble fragmented packets in the proper order. Offsets are always given relative to the beginning of the message. This is a 13-bit field, so offsets are calculated in units of 8 bytes, corresponding to the maximum length of 65,536 bytes. Given the fact that the core is aimed at embedded systems manipulating usually small packets, our design does not support IP fragmentation, so these fields are ignored. Moreover the current web explorers use the TCP fragmentation to send large packets over networks that support small packets like Ethernet (1500 bytes maximum).

Time to Live: 8 bits

This field sets an upper limit on the number of routers through which a datagram can pass, and therefore limits the lifetime of the datagram. It is set by the sender and decrements once each time it passes through a router. When this field reaches 0x00, the datagram is discarded, and the sender notified by an ICMP message.

.

Protocol: 8 bits

This field indicates the type of data encapsulated in the current IP datagram. The table below displays the protocols supported by our implementation.

Supported Protocols

Code	Protocol
0x01	ICMP
0x06	TCP
0x11	UDP

Header Checksum: 16 bits

This checksum is calculated over the IP header and includes only the options that are sent. Since some header fields change (e.g., time to live), this is recomputed and verified at each point that the internet header is processed. In this design, this field is ignored when receiving packets, since the whole packet is checked through CRC, but is computed when transmitting packets.

The checksum algorithm is:

> *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.*

Source Address: 32 bits

This 32-bit address represents the source of the IP datagram when one is received, and our IP address when a datagram is sent.

Destination Address: 32 bits

This 32-bit address represents the IP datagram's destination and our IP address when a datagram is received. Upon receiving a datagram, if the destination IP address is not equals to our design's address then the datagram is discarded.

Options: variable

This field is used to transmit various parameters such as security and handling restrictions, record routing, and timestamp information. The options field must end on a 32-bit boundary and pad bytes are added in as necessary (pad bytes are always 0x00). These options are not supported in the current design since most of them are reserved for future use or they are supported only for special use such as military purposes.

3.5 Internet Control Message Protocol

ICMP messages are used to report error conditions when processing datagrams [22]. In addition to reporting error conditions, ICMP is also used to query other machines. This is the protocol used to implement the PING application. ICMP messages are embedded within the data field portion of an IP. The structure of the header in the case of echo request or echo reply is shown in the table below.

Message Type	Sub Code	Checksum
Identifier		Sequence Number
Optional Data (Variable length)		

Message Type: 8 bits

This field is used to indicate the general class of the ICMP message. The implementation supports only Echo Requests (type 0x08) and Echo Replies (type 0x00). These classes are used when a host sends an echo request and waits an echo reply message in order to determine if another machine is reachable. The design sends an echo reply automatically in response to any echo request, but it does not support echo requests.

Sub Codes: 8 bits

This field is used to indicate sub codes within each message type. Our implementation supports only sub codes “0x00” with the Echo Request (0x08) and Echo Reply (0x00) message types.

Checksum: 16bits

The checksum is the 16-bit one’s complement of the one's complement sum of the ICMP message starting with the ICMP Type. For computing the checksum, the checksum field should be zero. The ICMP fields are the only ones computed for this checksum.

Identifier: 16 bits

This is normally the process ID of the sending process. When an Echo Request is received, the Identifier is copied into the Identifier field of the Echo Reply. When sending an Echo Reply the Identifier will be fixed at 0x01.

Sequence Number: 16 bits

This field is usually used to keep track of Echo Requests sent. It is incremented with each Echo Request sent. When an echo Request message is received, the Sequence number is copied into the Sequence Number field of the Reply. Our implementation will always start the sequence number at 0x00 and increment it each time an ECHO Request is sent.

3.6 User Datagram Protocol

The User Datagram Protocol is a connectionless protocol which does not provide reliability, meaning there is no indication to the sending device that a message has been received correctly [20]. This protocol also does not offer error-recovery capabilities. UDP is much simpler than TCP and is usually used with the Trivial File Transfer Protocol (TFTP).

The header of a UDP datagram is shown below.

Source Port Number	Destination Port Number
Length	Checksum
Data (if any)	

Source Port Number: 16 bits

The source port is a 16-bit field that identifies the local UDP user, which is usually an upper-layer application program.

Destination Port Number: 16 bits

The destination port number is a 16-bit field that identifies the remote machine's UDP user.

Length: 16 bits

This 16-bit field indicates to the UDP receiver the size of the UDP data. The UDP module extracts the UDP data length based on this field and only receives the correct amount of data.

Checksum: 16 bits

The checksum is calculated by taking the 16-bit one's complement of the one's complement sum of the 16-bit words in the header and data together. Unlike the ICMP and IP header in the UDP checksum some fields of the IP header (called pseudo-header) are also used for the UDP checksum calculation. These fields are the source IP address, the destination IP address, the protocol type and the UDP total length.

3.7 Transmission Control Protocol

The Transmission Control Protocol provides a considerable number of services to the IP layer and the upper layers. Most important, it provides a connection-oriented protocol to the upper layers that allows an application to make sure that a datagram sent out over the network was received in its entirety. If a datagram is corrupted or lost, TCP handles the retransmission, thus providing reliable communications in the higher layer applications.

TCP provides two methods to establish a connection: active and passive. An active connection establishment occurs when TCP issues a request for the connection, based on an instruction from an upper-level protocol that provides the socket number. A passive open approach takes place when the upper-level protocol instructs TCP to wait for the arrival of connection requests from a remote system. When TCP receives the request, it assigns a port number. The design provides only passive open connections. This means that a host must initiate not only the establishment of the connection but also the closing of this connection.

The header of a TCP datagram is shown below.

<table>
<tr><td colspan="8">Source Port</td><td>Destination Port</td></tr>
<tr><td colspan="9">Sequence Number</td></tr>
<tr><td colspan="9">Acknowledgment Number</td></tr>
<tr><td>Data offset</td><td>Reserved</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td>Window</td></tr>
<tr><td colspan="8">Checksum</td><td>Urgent Pointer</td></tr>
<tr><td colspan="9">Options and Padding</td></tr>
</table>

Source and Destination Port: 16 bits

This is a 16-bit field that identifies the local and the remote TCP application. In the design only one port is supported and has a fixed number.

Sequence Number: 32 bits

This 32-bit field indicates to the TCP receiver whether the incoming segment is to be stored in SRAM. If the sequence number matches the receive buffer's next expected sequence number, it stores the data to the buffer. Otherwise, the TCP receiver ignores the incoming segment. This scheme works even if the received sequence number arrives out of order because the remote TCP module guarantees to resend unacknowledged segments When the TCP module makes a connection, the initial sequence number is derived from a random number generator in order to avoid the confusion with earlier connections. After a connection is established, the sender module fills this field with the last acknowledge number received from the TCP peer.

Acknowledge Number: 32 bits

The TCP receiver checks this number to disable re-sending data. If the received acknowledge number is larger than the last received number and same or smaller than what was expected, the internal acknowledge number is updated. The TCP sender fills this field with a valid acknowledgement number except for when it first requests a TCP connection as a client. The acknowledge number that will be sent is incremented by the number of data received. It is also incremented in response to SYN, FIN without data because they have virtually one byte worth of data in the TCP sequence / acknowledge scheme. The TCP receiver checks both SEQ and ACK numbers to validate the incoming TCP segments.

Data Offset: 4 bits

This 4-bit header length field contains an integer that specifies the length of the segment header measured in 32-bit multiples. The TCP receiver state machine uses this field to determine if TCP options exist. The TCP receiver checks this field when it receives SYN flag. When this field is 0x60, then the TCP module knows that the current TCP segment contains options.

Reserved: 6 bits

This 6-bit field is reserved. The TCP receiver module does not check this field. The TCP sender module always fills this field with zeros.

Flags: 6 bits

The next 6-bits field specifies codes. The codes are summarized in the Table below.

Code Specification

Bit (left to right)	Meaning (if bit set to 1)
URG	Urgent pointer field is valid
ACK	Acknowledgment field is valid
PSH	This segment requests a push
RST	Reset the connection
SYN	Synchronize sequence number
FIN	Sender has reached end of its byte stream

The URG bit, if set, indicates that the incoming TCP segment's urgent pointer is valid. The TCP module does not support urgent pointer scheme. The module also uses ACK flag when receiving new packets. The PSH flag indicates that the data will be stored in the SRAM. When the module receives a packet with RST flag set then the TCP connection is reset. The SYN flag is used for the TCP module to initiate a TCP connection. If this flag is received after the connection is established the segment is ignored. The FIN flag is used to terminate connections. The TCP module supports only one case of closing which is server initiated.

Windows Size: 16 bits

This 16-bit field is used to advertise how many bytes are reserved for receiving incoming TCP data. After the TCP receiver reads this value, the TCP sender limits the number of bytes to be sent if this field indicates smaller number of data than what is needed to be

sent. The TCP sender also inserts the number of acceptable bytes to this field on every TCP segment it sends out.

Checksum: 16 bits

This 16-bit field is ignored for incoming TCP segments since the reliability of the packet is checked by CRC but the TCP sender always calculates this field before passing the segment on to the IP module. Just like the UDP header, the TCP checksum is calculated taking account some fields of the IP header (called pseudo-header). These fields are the source IP address, the destination IP address, the protocol type and the TCP total length.

Urgent Pointer: 16 bits

This 16-bit field, upon the reception of URG flag, is used to determine where the urgent pointer resides in the TCP data. This field is ignored by the TCP module since all data are treated the same and they are stored in the SRAM in order to be available to any application.

TCP Options: 16 bits

The only option that this design accepts is Maximum Segment Size option and that only when it receives SYN flag, which initiates the connection. This is detected by the TCP header field when this field is 0x60 instead of 0x50. The MSS option is listed in the table below. Any other option received is ignored.

MSS Option

Byte 0	Byte 1	Byte 2	Byte 3
Option Code	Length in bytes	MSB	LSB
0 x 02	0 x 04	MSB	LSB

TCP Data: Variable Length

The TCP data are stored in the SRAM in conjunction with the TCP sequence number. Data storing begins at the address 0x01. When a new TCP packet arrives, targeting the same port, then the new data are stored in the next available address. When we close the TCP port the SRAM address register initiates to 0x01.

Chapter 4
Architecture

Any system must be designed to withstand
the worst possible set of circumstances.
Murphy's Law

This chapter presents the architecture of our design. The first section describes the design's dataflow through the modules. The following sections are an in-depth examination of each module of IP core. Each protocol module is actually a finite state machine (fsm) that decodes each field of the protocol's header when receiving a packet and creates each field of the protocol's header before transmitting a packet. In the last two sections we describe the checksum and the CRC module.

4.1 Overall Architecture

When a new packet arrives to the system, an Ethernet transceiver chip converts the analog signals of Ethernet to 8-bit width digital signals (the function of an Ethernet transceiver is described in the next chapter). The design reads the data and at the same time decodes the protocol's header. The main control unit is actually the RxEther module since this unit activates all the other modules. The incoming data can be read by every protocol module (RxEther, ARP, ICMP, IP, UDP and TCP), except TxEther. The data are also read by the receiving CRC-32 module in order to check the packet for errors. The TCP module can route the packet to the SRAM in order to be used by a potential application. When the system has received the whole packet it decides whether it has to reply to this packet. In this case the packet that is going to be transmitted is created in an internal RAM module (TxRAM). Each protocol header is created by the corresponding module. All modules can write to TxRAM data bus and to TxRAM address bus. This bus is initially set to high impedance, thus every module can write to this bus without any

conflict. All the protocol modules control the TxRAM *write enable* signal while TxEther controls the *read enable* signal. Every module writes its data to the corresponding address of TxRAM. The first byte is used to indicate the total size of the packet that is going to be transmitted. The TxRAM data bus can be read by the checksum modules, thus they can produce the corresponding checksum field. This bus can also be read by the transmitting CRC-32 module which provides the transmitted packet's CRC field. The overall data flow is depicted below (see figure 4.1).

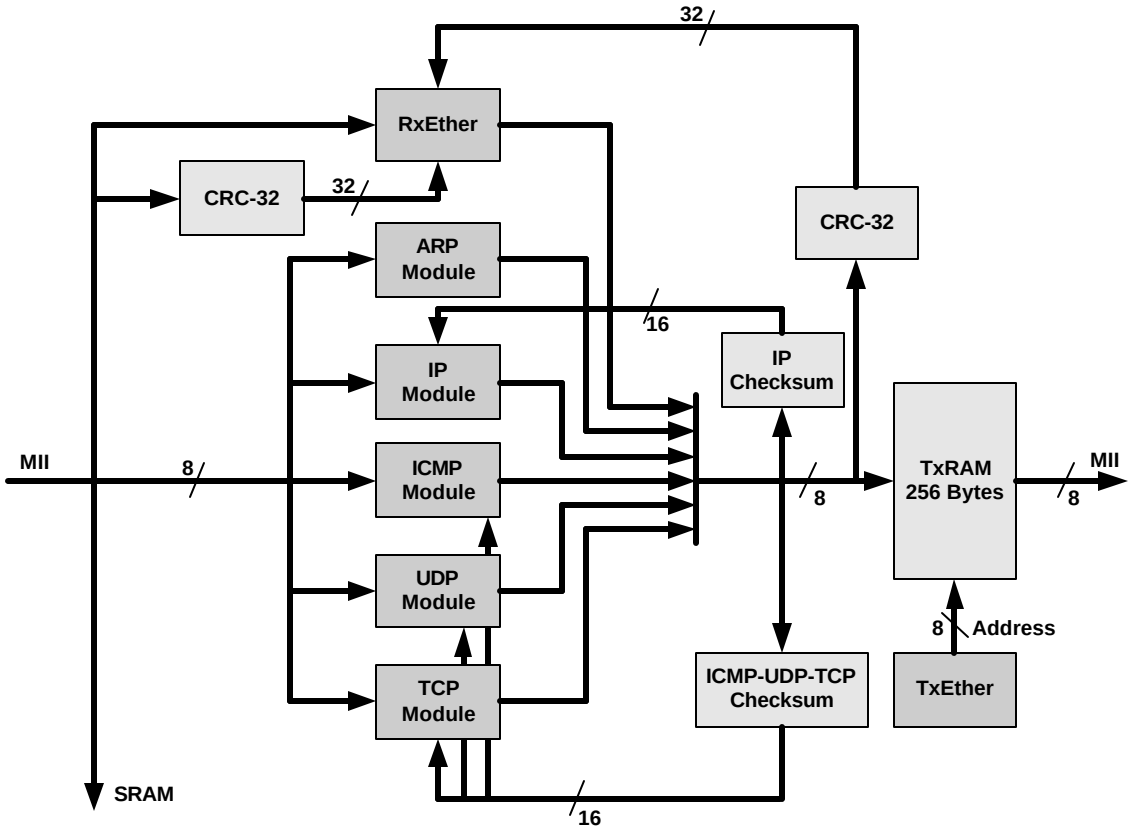


Figure 4.1 Dataflow block diagram

The basic features of the supported protocols that are described below in each module are

- ARP (without look up table)
- ICMP (echo request-reply)
- UDP (one connection)
- TCP (one connection, passive open port)

In case that an application module must be added, it can be simply added like the other protocol modules. It must be able to write to TxRAM data bus and must be triggered by the RxEther module.

4.2 RxEther module

The RxEther module is the basic one that controls all the other modules. When the signal RX_DV (data valid) goes high then the fsm goes to *read preamble* state and waits until the end of the preamble ("AB"). Next, fsm examines the local MAC address. If the address's first byte is FF, which means that it is a group address, the ARP module is activated.

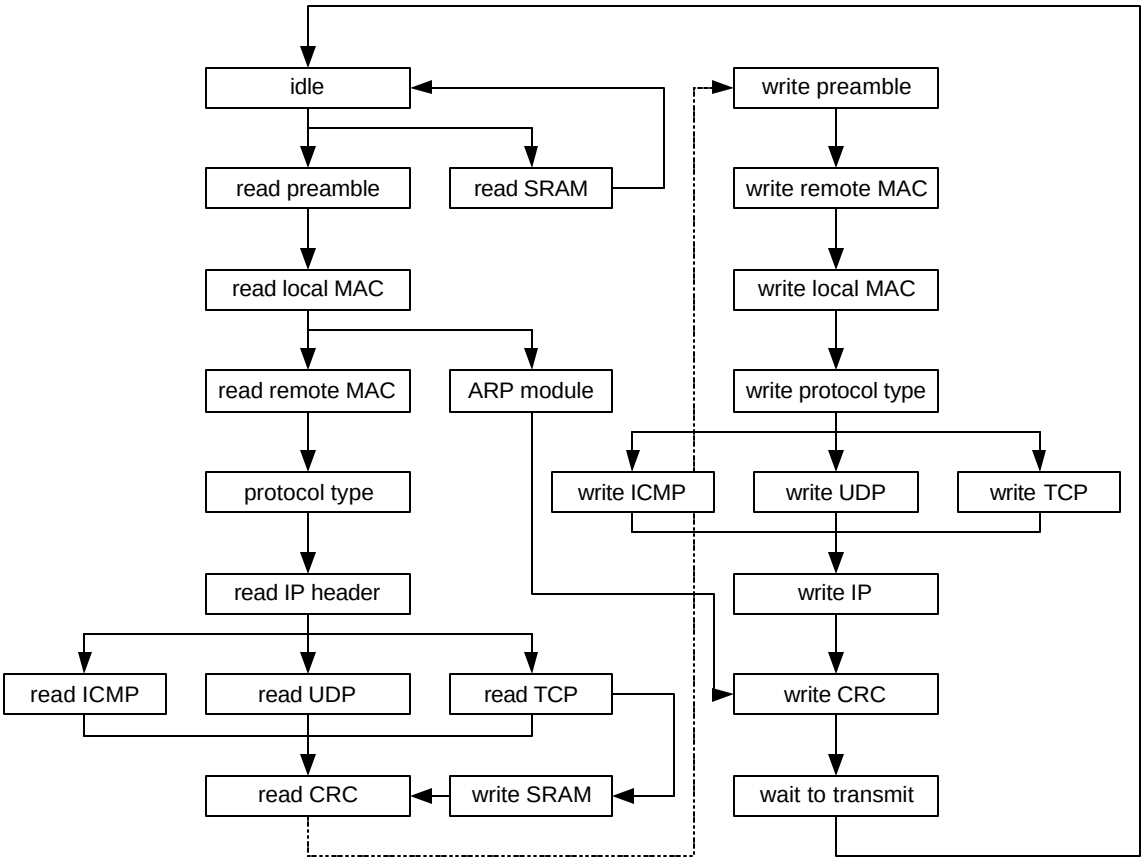


Figure 4.1 RxEther fsm

Otherwise the module keeps examining the local MAC address and if it matches with its own address it goes to the next state. In this state, *read remote MAC*, the remote physical address is stored in the proper register and then fsm goes to *protocol type* state. This fsm state examines the type and if it is "0x0800" (the Internet protocol type), it accepts the packet.

The next state is *read IP header* state which activates the IP module. This module decodes the IP header and returns a signal that indicates the Transport's protocol type. Depending on this signal fsm goes to the proportional state. In the case of *read TCP* state, if the TCP module activates a special signal then the fsm goes to *write SRAM* state, which is responsible for writing the incoming data to the SRAM. Otherwise the fsm goes to *read CRC* state which examines the CRC register. If the register does not have the default value "38FB2284", the packet is discarded. This value is the only valid CRC value and includes the CRC field of the Ethernet header.

The transport module (ICMP, UDP or TCP) decides whether the RxEther module must reply to the incoming packet. If so, then the fsm transits to *write preamble* state which writes the appropriate sequence of preamble (7 "AA", following one "AB") to the internal RAM. The next states are *write remote MAC* and *write local MAC* states that write the remote and the local MAC addresses respectively. The following state is *write protocol type* state which writes the data 0x0800 to the internal RAM. Depending on the transport protocol of the incoming packet, fsm goes to the proportional state (*write ICMP, write UDP* or *write TCP).* The next state is *write IP* state that activates the IP module, in order to write the IP header to the RAM. This state is following the transport's protocol states because the IP header must know the length of the packet that is going to be transmitted. The last state is *write CRC* state which writes the CRC register to the corresponding field.

When fsm is at the *idle* state we can set a signal to 1 (SRAM_RD='1') in order to activate the *read SRAM* function which sends the contents of the SRAM to the corresponding pins of the FPGA. This way we can move the contents of the SRAM to the ports of the FPGA.

This function has been implemented for testing purposes and is described in detail in the next chapter.

4.3 ARP module

The Address Resolution Protocol (ARP), as we presented it in section 3.1, is responsible for converting IP addresses to physical addresses in order to eliminate the need for applications to know about the physical addresses. The ARP module is actually an fsm that decodes and synthesizes each field of the received and transmitted packet respectively. The first state, *read local MAC,* examines if the MAC address is set to high ("FF FF FF FF FF FF") which would mean that is a group address. The second state, *read remote MAC,* stores the remote MAC address in order to know the sender of the packet. The next two states, *read protocol type* and *read ARP general,* read the incoming data, which are useless in the current design. The *read destination MAC* and *read destination IP* include the MAC and IP address of the sender. The *read source MAC* state simply reads the data without storing it. That happens because the sender does not know the recipient MAC address, only the recipient IP address. The next state is *read CRC* state which in case the field is correct the fsm goes to the transmission states.

In these states, the module writes to the internal RAM the corresponding registers that have been read. The only difference is that the source MAC address (all high) is replaced by the actual MAC address. Finally the fsm transits to *write length* state that writes to the first byte of TxRAM the total size of the packet that is going to be transmitted.

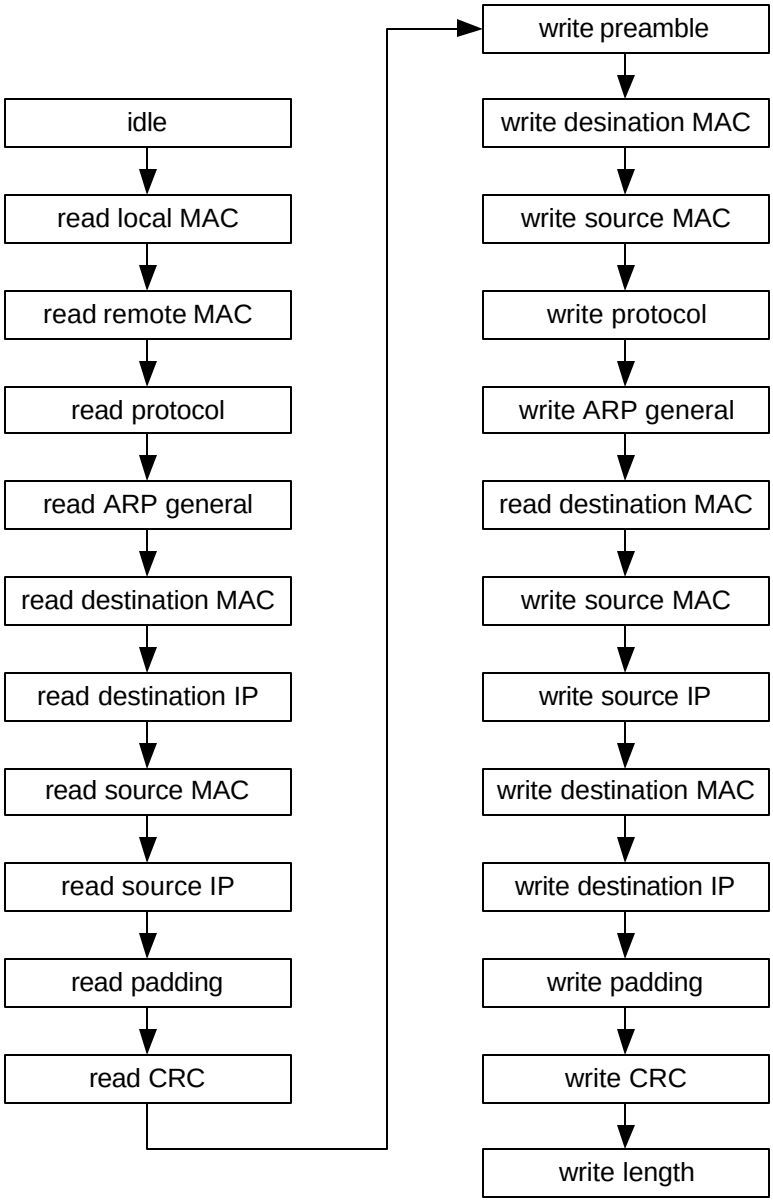


Figure 4.3 ARP fsm

4.4 IP module

The Internet Protocol (IP) module is divided into 2 fsms; the first one is activated when the system is receiving packets and the second one when it is transmitting packets. When receiving a packet the fsm goes to *read version* state, which reads the IP version (0x4), the header length (in 32-bit words, 0x5) and the service type (usually 0x00). The next

state, *read size*, reads the packet length in bytes in order to know the size of the packet. Then the fsm goes to *read offset* and *read time to live* states that simply ignore these fields. The next state, *read protocol type*, is very important as it determines the next module that will be activated after the IP header. The next state reads the IP header checksum, but does not verify it, since we use the Ethernet CRC field in order to know if the packet has any errors. The following states, *read remote address* and *read local IP*, store the IP address of the sender and the IP address of the recipient respectively in order to use them when creating the IP header of the transmitted packet.

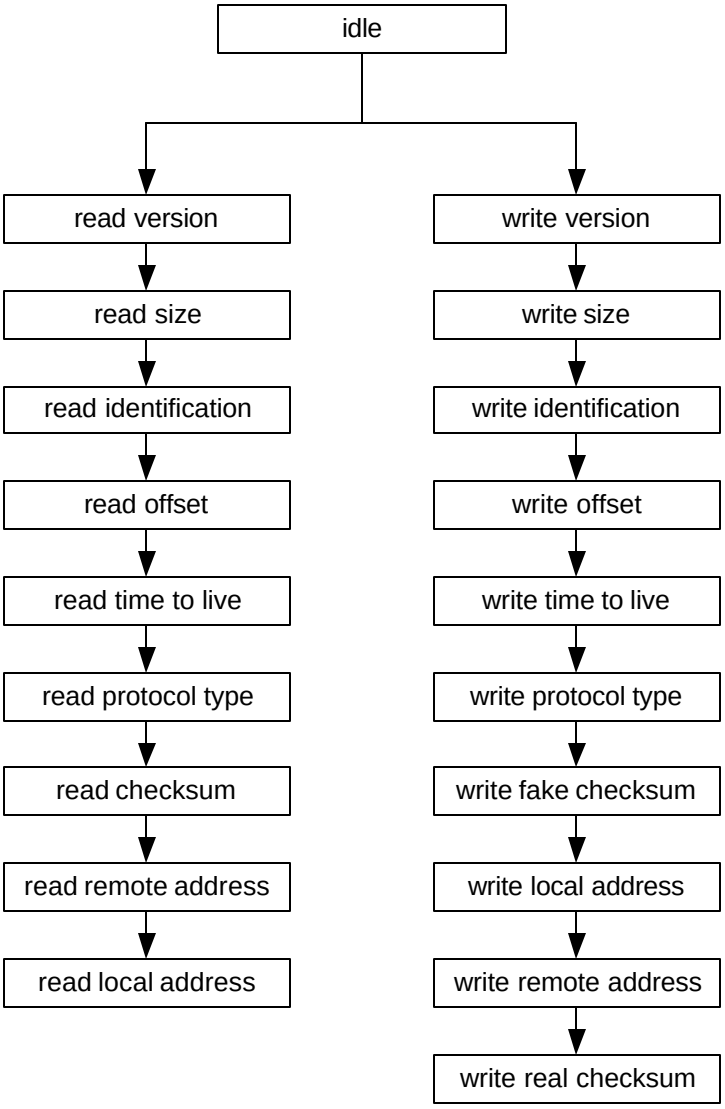


Figure 4.4 IP fsm

When it is time for the IP module to create the IP header of the packet, it uses the necessary registers that were loaded when the module was receiving a packet. Otherwise the module fills theses field according to other functions. For example, in the *write size* state, the IP module reads from RxEther the current size of the packet and adds the IP header size (20 bytes) and the CRC field size (4 bytes) to obtain the right size of the packet. The transmitted identification number of the packet is obtained by inverting the receiving identification number in order to create a random number. The transmitted time to live is set to 0x80=128 which is the default value for most network programs, such as ping and trivial ftp. When the fsm goes to *write checksum* state then the module writes a dummy checksum since the checksum module has not received all the fields yet. When the IP module has written all the fields of the IP header it goes to *write real checksum* state which reads the right checksum from the checksum module and writes it to the internal RAM.

4.5 ICMP module

The Internet Control Message Protocol module is responsible for replying to information requests that usually come from the ping program. This module supports only this function, which means that the type field of the ICMP header when receiving a packet must be 0x08 and the code field must be 0x00. The ICMP module is divided into two fsms, like the IP fsm; the first one is activated when receiving a packet and the second one when creating the transmitted packet. When the fsm is in *read identifier* and *read sequence* state the two fields are stored into the proportional registers, so they can be retrieved when the module writes the identifier and sequence number of the transmitted packet, respectively.

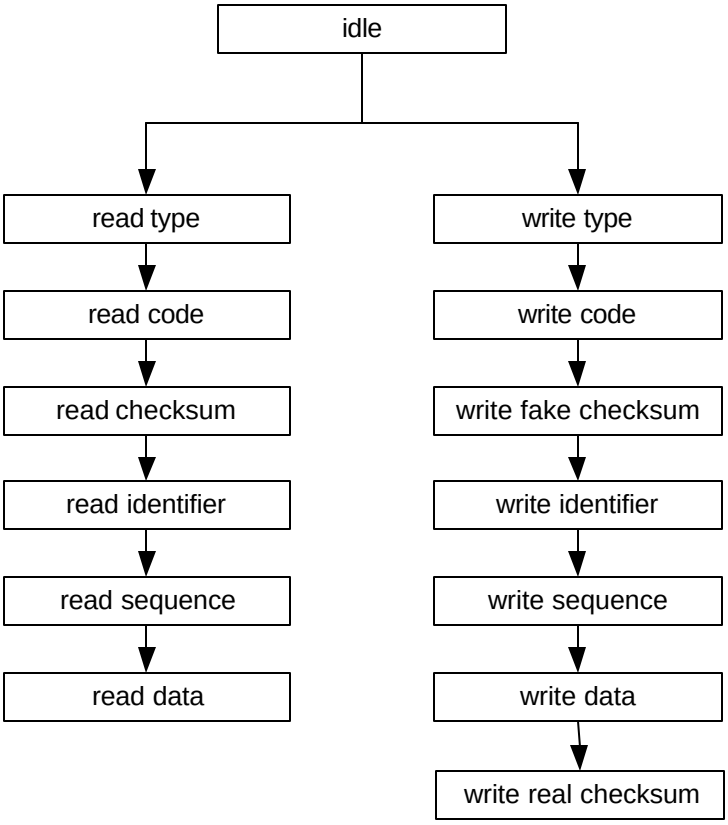


Figure 4.5 ICMP fsm

When the fsm goes to *write data* state the module writes 32 dummy data in order to pad the ICMP packet according to the ICMP protocol. In this module we use one more state, *write real checksum* state, in order to write the correct checksum which includes not only the ICMP header but the data, too.

4.6 UDP module

The Uses Datagram Protocol (UDP) is widely used as a connectionless protocol by many programs such as the Trivial File Transfer Protocol (TFTP) and the Remote Call Procedure (RCP). The UDP fsm also consists of two distinct fsms; the first one is activated when receiving a UDP packet and the second one when creating the UDP header of the transmitted packet. Since UDP is connectionless its header is rather simple. When receiving a packet the UDP module reads some fields of the IP header (for pseudo-

header purpose) and then stores the remote (source) port and the local (destination) port into two distinct registers. The next state, *read length*, reads the packet size including header and data. The checksum field is ignored when receiving a packet as we use again the Ethernet CRC field to check if there are any errors in the datagram. When we produce the UDP header we read the local and remote port registers and write them to the internal RAM. We also output the pseudo-header fields without writing these fields to the TxRAM. The UDP checksum is written at the last state of the fsm since it must include both the header and the data. We must note that even though the UDP module does not have to transmit any acknowledgment, when receiving a datagram, in this design the UDP module send a fake reply by sending back the data to the sender in order to achieve a form of communication. This way we can be aware of the proper or not reception of UDP packets by the implementation.

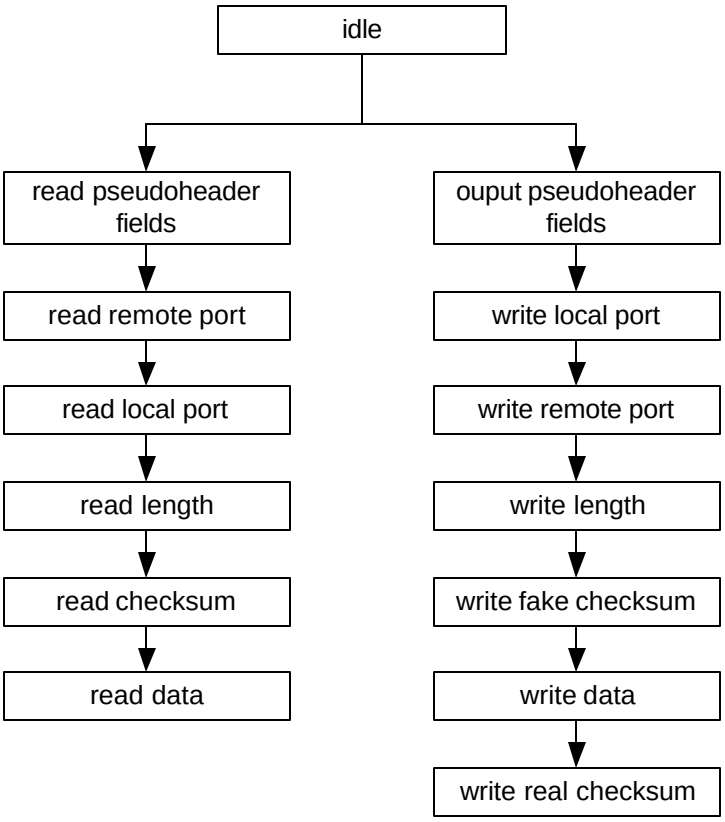


Figure 4.6 UDP fsm

4.7 TCP module

The Transmission Control Protocol module is the most basic and the most complex module of this design. It consists of three independent fsms; the first one is activated when the module receives a TCP packet (see figure 4.7a), the second one when creating the TCP header of the transmitted packet (see figure 4.7a) and the third one is implemented according to the official TCP fsm document (RFC793). This fsm describes the different states of TCP depending on the receiving packets (see figure 4.6b).

The receiving fsm is activated when the design is still reading the IP header. The reason is that the TCP checksum includes some fields of the IP header (pseudo header). The module reads and stores the IP length, the IP destination address (remote) and the IP source address (local), while discarding all the other unnecessary IP fields (*read IP dummy* state). This way, when the module creates the TCP checksum in the header of the new packet, it includes the TCP header, the above IP fields and the data. After *read local IP* state, the fsm goes to *read remote port* (source port) and *read local port* (destination port) state, where it stores these fields so that it can be used when creating the transmitted packet. The local port has been set to 1001, which means that only packets with this value at the destination port are valid and all others are discarded. The next states, *read sequence number* and *read acknowledgment number*, store these values in the appropriate registers, so they can be used by the TCP fsm. The next state, *read data offset*, determines if the TCP header has any options. If so, the fsm goes to *read options* state after *read urgent* state, otherwise the fsm goes to *read data* state. After reading the data offset, the fsm goes to *read flags* state which determines the next TCP fsm state. The following states *read window* and *read checksum*, simply read these fields without any action. The urgent pointer that is read in the next state is not supported since any action taken, using this field, depends on the upper application. If the fsm goes to *read options* state then it reads the maximum segment size type (0x02), the options length and the maximum segment size in bytes. If the fsm goes to *read data* state then a signal is activated in order for RxEther to start saving these data to the SRAM.

The transmitting fsm is responsible for writing the header of the transmitted packet. The first four states create the right IP fields in order to be included into the TCP checksum field (pseudo header), although they are neither written to the RAM, nor are included in the CRC field. The following states write all the other TCP fields by reading the appropriate registers. Some of these registers are written by the receiving fsm while all other registers are written by the TCP fsm. If the fsm is in *write urgent* state and options have to be written then the fsm goes to *write options* state, otherwise it writes the TCP checksum and then goes to *idle* state.

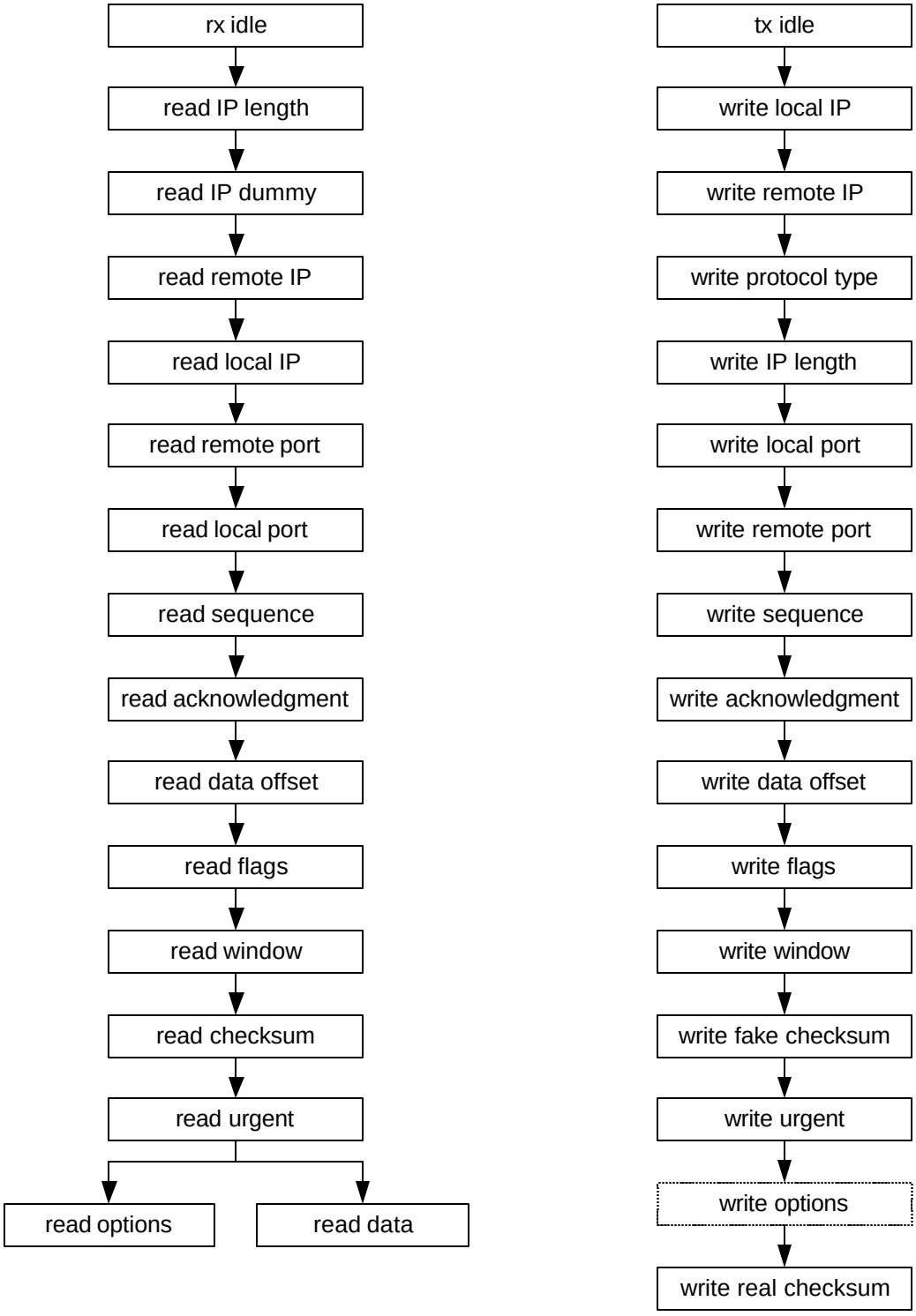


Figure 4.7a TCP read-write fsm

The TCP fsm (see figure 4.6b) is implemented in order to support the main functions of TCP. It is responsible for establishing a connection, transferring data and closing a connection. It is also responsible for the values of the transmitted flags, the sequence and the acknowledgment numbers. This fsm supports only passive open ports, thus a connection establishment happens only if an active ports initiates a connection. This fsm is in the *listen* state while waiting to receive a packet with the SYN flag on, which indicates that an active port wants to establish a connection. Then the fsm sets the SYN and the ACK flags and activates the RxEther module to send the reply packet.

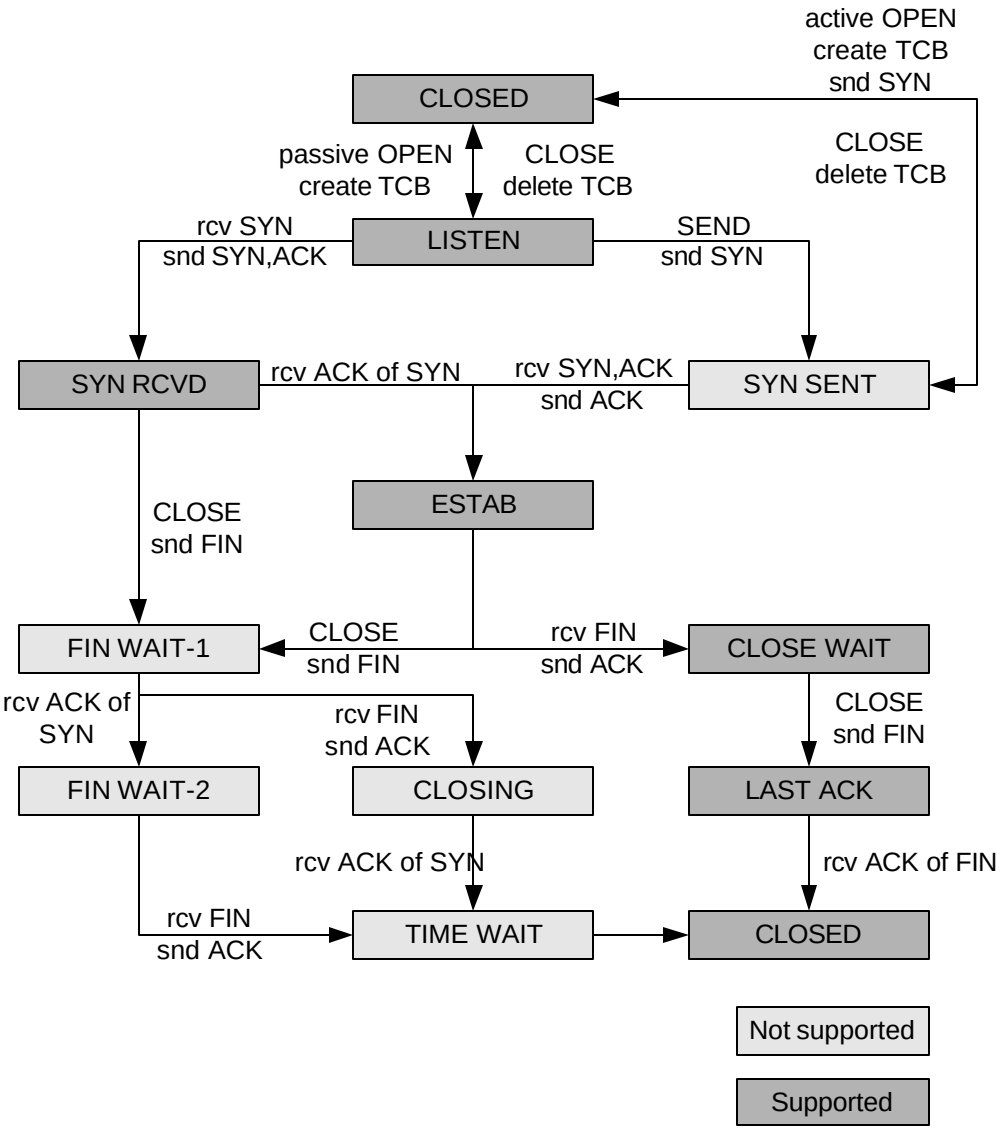


Figure 4.7b TCP fsm

With the arrival of a new packet, fsm checks the flags and if the ACK flag is on, the fsm goes from *syn_rcvd* state to *established* state. In this state, the fsm receives data packets and sends the right acknowledgment. The sequence number of the transmitted packet is once again created by inverting the receiving sequence number, thus obtaining a random number. The transmitted acknowledgment number is created by adding the receiving sequence number to the number of data bytes received.

The fsm stays in that state until it receives a packet with the FIN flag on. In this case the fsm goes to *close Wait* state and sends a packet with the ACK flag on. When the transmission of this packet has ended the fsm sends another packet with the FIN flag on and goes to *last ack* state. With the arrival of the last packet, if the ACK flag is on, the fsm goes to *closed* state and waits for a new connection to be established. We must note that the TCP fsm interacts with the RxEther fsm through the tcp_status register in order for RxEther fsm to know if it must send a new packet or not, while all the other protocols always send a new packet after receiving a datagram.

4.8 TxEther module

Under the IEEE 802 series of 10-Mbits operating standards, the data link layer of the OSI Reference Model is subdivided into two layers, the logical link control (LLC) and medium access control (MAC). The frame format that we examined in section 3.1 represents the manner in which LLC information is transported. Directly under the LLC sublayer is the MAC sublayer, which is responsible for checking the channel and transmitting data if the channel is idle or checking for the occurrence of a collision, and performing some actions if a collision is detected. Actually the MAC layer is an interface between user data and the physical placement and retrieval of data on the network. The TxEther module actually implements the transmitting MAC operations since the receiving MAC operations are rather simple and are implemented in the RxEther module.

These operations are based on the Carrier-Sense Multiple Access with Collision Detection method (CSMA/CD) that in used in Ethernet networks. According to this protocol, when a station has data to send, it first listens to determine whether any other station on the network is talking. In a CSMA/CD network, if the channel is busy, the station will wait until it becomes idle before transmitting data. Since it is possible for two stations to listen at the same time and discover an idle channel, it is also possible that the two stations could then transmit at the same time. When this situation arises, a collision will occur. Upon sensing a collision, a delay scheme is employed to prevent a repetition of the collision.

The TxEther fsm (see figure 4.7) is in *idle* state until the signal tx_start is activated by the RxEther module. When this signal arises, that means that the data to be transmitted is in TxRAM and ready to be transmitted. The fsm goes to *S1* state, which examines if the network is idle, by checking the Carrier Sense signal (CRS). If the CRS signal is on then the fsm goes to *CRS* state and stays there until CRS's falling edge. When CRS turns to 0 the fsm goes to the next state. In this state the module, using an internal counter, waits 9.6usec before trying to retransmit. This 9.6usec space between two consecutive packets ensures that all receiving electronic devices will sense the silence in the network. At the end of this delay the fsm goes again to *S1* state and if CRS is low then the fsm reads the total size of the packet, information stored in the first byte of the packet (RAM address=0x01). After reading the packet size the fsm goes to *data* state and start transmitting the packet to the network. In this state the fsm reads the data written by all the other modules from the internal RAM.

If there is no collision the fsm goes to *idle* state after the transmission of the whole packet. While transmitting data if a collision is detected the fsm goes to *jamm* state. In this state the TxEther module transmit 128 consecutive high bits which ensures that the collision lasts long enough to be detected by all stations on the network. After transmitting these jamming bits the fsm goes to *backoff* state. In this state the module waits a random number of slot times before attempting to retransmit. A slot is a frame that represents 512 bits (or 51.2usec on a 10-Mbits network). The actual number of slot times the module waits is selected by a randomization process, formerly known as a

truncated binary exponential backoff. Under this randomization process, a randomly selected integer r defines the number of slot times the module waits before trying to retransmit the packet. This integer is selected by specific range of values, which is called window. If a collision occurs on a retransmission attempt the module doubles the window, a new random number is generated from the new window and then waits for the prescribed number of slot intervals prior to attempting a retransmission. The maximum retransmission attempts before the module aborts the retransmission is 16. In the first 10 attempts the backoff window increases binary (0-1, 0-4, 0-8…) while in the last 6 attempts the window is constant (0-1023).

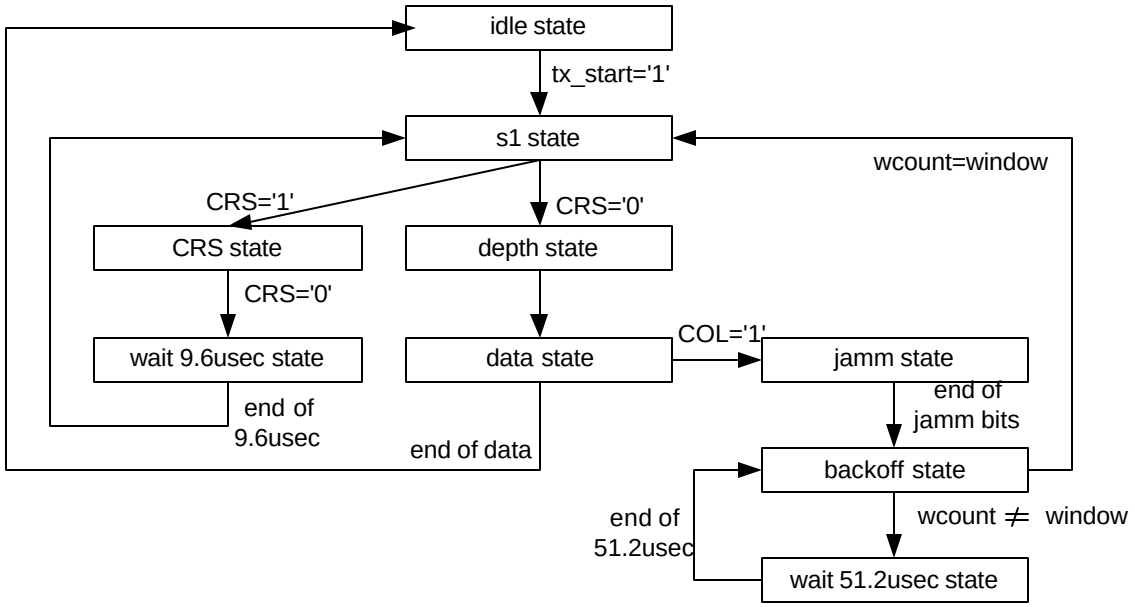


Figure 4.8 TxEther fsm (MAC fsm)

In TxEther module we use an internal integer (wcount) to count the number of slots. Each time the 51.2usec delay ends we check if this integer is equal to the random generated window. If it is equal, the fsm goes to *S1* state while doubling the possible values that the window can take the next time. The window range is produced by using an integer from 0 to 1023 and taking the remainder with 2, 4, 8, 16 … in each attempt. In this design we have not implement the random generator process and the slot number r is selected from a Look Up Table (LUT) for each attempt.

4.9 16bit Checksum algorithm

The 16-bit checksum algorithm is widely used in many protocols of the Transport and Network layer. The official algorithm of this checksum according to the RFC793 [23] is:

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.

Depending on the protocol, the checksum can include only the protocol's header, the header and the data or even the header of other protocols, i.e. the TCP checksum includes the TCP and part of IP header.

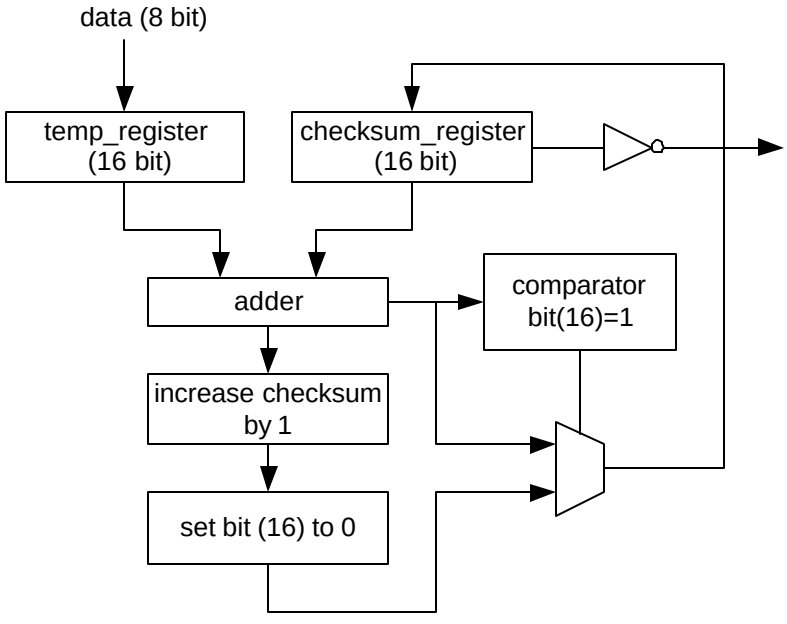


Figure 4.9 The checksum module architecture

In our implementation, on the first cycle the 8 bit data are inverted and stored in the 8 upper bits of a 16-bit wide register (see Figure 4.8). On the second cycle the data are inverted and stored in the 8 lower bits of this register. On the same cycle this register is added to the previous value of the checksum register. If there is an overflow (bit(16) =1) then the checksum register is increased by one and bit(16) is set to 0.

4.10 32bit Cyclic Redundancy Checking algorithm

The frame check sequence field provides a mechanism for error detection. Each transmitter computes a cyclic redundancy check (CRC) that covers the address fields, the type and the data field. The transmitter then places the computed CRC in the four bytes CRC field. As presented in section 3.1 the CRC field is the remainder when M(X) is divided by the following polynomial:

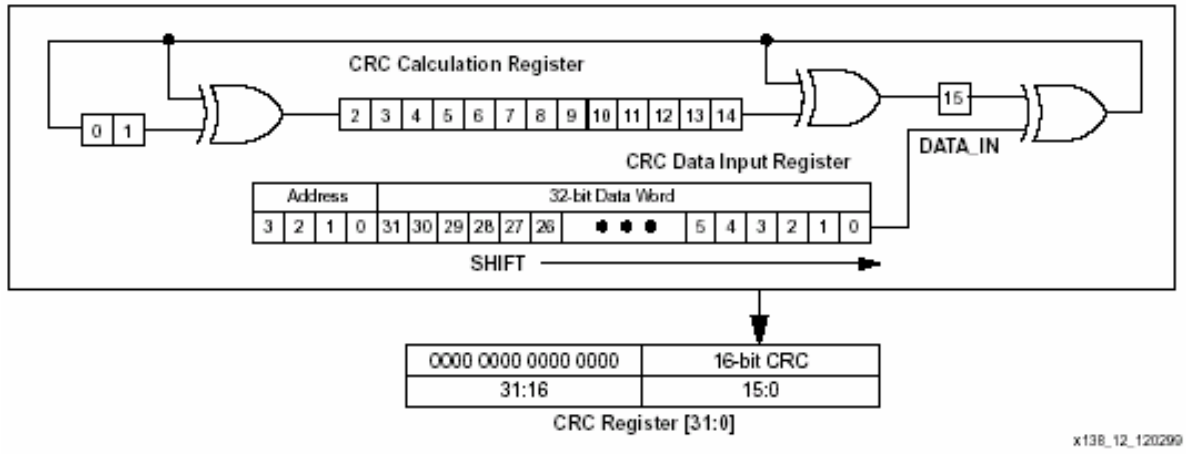
$G(X) = X^{32}+ X^{26}+ X^{23}+ X^{22}+ X^{16}+ X^{12}+ X^{11}+ X^{10}+ X^{8}+ X^{7}+ X^{5}+ X^{4}+ X^{2}+ X+1,$
where M(X) is a polynomial that covers the data bits.

In general, digital logic does not implement efficiently the division of very large number. Consequently, binary information must be converted into a more appropriate form before the CRC is used. The strings of bits to be verified is represented as the coefficients of a large polynomial, rather than as a large binary number, as shown in the following example:
$1,1000,0000,0000,0101 = X^{16}+X^{15}+X^{2}+1$

Typically, CRC calculations are implemented with linear-feedback shift registers (LFSRs). LFSRs use a method that yields the same results as subtraction and shift division process when the subtraction is performed without carry by the XOR function. To affect subtraction and shift division one bit at a time, you can shift through and examine each bit in the original frame of data. For the first bit of value 1, the divisor high-ordered bit is subtracted (XOR) from the dividend. That dividend bit, which is unnecessary and is not generated, is set to zero by the subtraction. The lower order bits of

the divisor cannot be subtracted yet, because the corresponding divisor bits have not been shifted in.

As shown in the following figure, for the simple case of the CRC-16, the algorithm is implemented by shifting the data stream into a 16-bit shift register. Register Bit(0) receives an XOR of the incoming data and the output of Bit(15). Bit(2) receives an XOR of the input to Bit(0) and the output of Bit(1). Bit(15) receives an XOR of the input to Bit(0) and the output of Bit(14).



In the case of CRC-32, which is used in Ethernet header, we use 14 XOR gates, one for each coefficient of polynomial G(X). As data is shifted into the CRC circuitry, a CRC calculation accumulates in the registers. When the CRC value is loaded into the CRC calculation register, the ending CRC checksum is loaded into the CRC Register. The value loaded into the CRC Register should be zero; otherwise, the configuration failed CRC check.

In addition, as the data is 8-bit wide we take advantage of the VHDL variables in order to process 8 bit data each clock cycle. In order to calculate the CRC field, as soon as the first bit is processed, we save this register and we immediately begin the process of the second bit. This way we achieve the serial processing of 8 bits for the CRC calculation in only one cycle.

Chapter 5

Implementation

A complex system that works is
invariably found to have evolved
from a simple system that works.
Murphy's Law

In this chapter we describe Ethernet transceivers which are essential components in order to connect our design in a 10 or 100Mbits Ethernet network. In addition we describe Pammete, a PCI generic interface board based on reconfigurable logic, on which our design is implemented.

5.1 Ethernet Transceivers

In order to connect our implementation to Ethernet networks we must use a mixed signal ASIC that will convert our digital signals into appropriate analog signals. The Ethernet network created at the Xerox Palo Alto Research Center by Dr. Robert Metcalfe is the most widely used local area network [3]. The transmission in the physical layer is baseband with Manchester coding, which means that the signal does not use a modulated carrier but it is transmitted directly on the network line. The Manchester coding is depicted below.

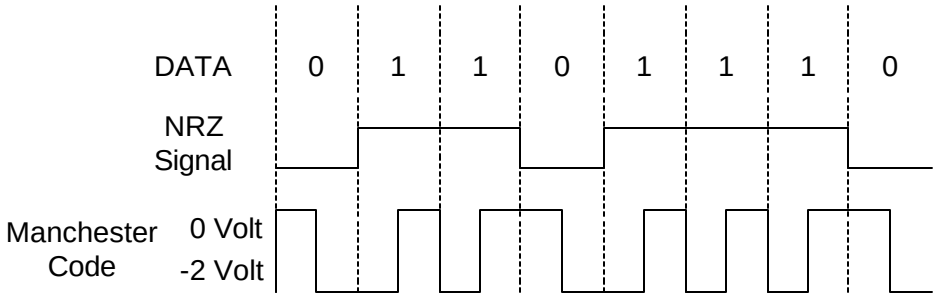


Figure 5.1a Manchester Coding

These ICs incorporate many necessary electronic parts such as parallel to serial converters, NRZ to NRZI converters and binary to ML-3 encoders and decoders. Furthermore when there is no packet to send transceivers send control pulses every 16msec in order to confirm the connection. If the receiver in the network does not sense these pulses for more than 100msec then it considers that the connection failed. In addition transceivers are responsible for carrier detection and collision detection, thus the MAC layer knows when to send a packet or send jamm bits.

Most of the Ethernet transceivers support the Media Independent Interface (MII) as specified in clause 22 of the IEEE 802.3u standard. This interface may be used to connect transceivers to a 10/100 Mb/s MAC or a 100 Mb/s repeater controller. The management interface of the MII allows the configuration and control of multiple transceivers, the gathering of status and error information, and the determination of the type and abilities of the attached devices.

The Media Independent Interface also includes a dedicated receive bus and a dedicated transmit bus. These two data buses, along with various control and indicative signals, allow the simultaneous exchange of data between the transceiver and the upper layer agent (MAC or repeater).

The receive interface consists of

- a nibble wide data bus RXD [3:0],
- a receive error signal RX_ER,
- a receive data valid flag RX_DV, and
- a receive clock RX_CLK for synchronous transfer of the data.

The receive clock can operate at either 2.5 MHz to support 10 Mb/s operation modes or at 25 MHz to support 100 Mb/s operational modes. In addition RX_CLK is the clock that drives the implementation.

The transmit interface consists of

- a nibble wide data bus TXD [3:0],
- a transmit error flag TX_ER,
- a transmit enable control signal TX_EN, and
- a transmit clock TX_CLK which runs at either 2.5 MHz or 25 MHz.

Additionally, the MII includes the carrier sense signal CRS, as well as a collision detect signal COL. The CRS signal asserts to indicate the reception of data from the network or works as a function of transmit data in Half Duplex mode. The COL signal asserts as an indication of a collision which can occur during half-duplex operation when both transmit and receive operation occur simultaneously. Although the MII interface provides a 4-bit data bus it is trivial enough to adjust this conflict by using an 8-bit register that reads the 4-bit MII data bus and provides an 8-bit data bus plus a clock with double period. This way the implementation is actually triggered by 12.5MHz in 100Mbits networks and by 1.25MHz in 10Mbits networks.

We must note at this point that according to the MII interface the data bus and the other signals (input and output) change on the falling edge of the clock. According to this specification all the receiving modules (RxEther, ARP…) are triggered directly by RX_CLK in order to read the correct data on the rising edge of the clock. In reverse TxEther module must be triggered by the falling edge of RX_CLK in order to change the transmitted data and signals in the falling edge. Consequently TxRAM, CRC-32 and checksum modules must also be triggered on the falling edge of the clock in order to read the right data on the TxRAM bus which is written by the protocol modules (see figure 5.1b).

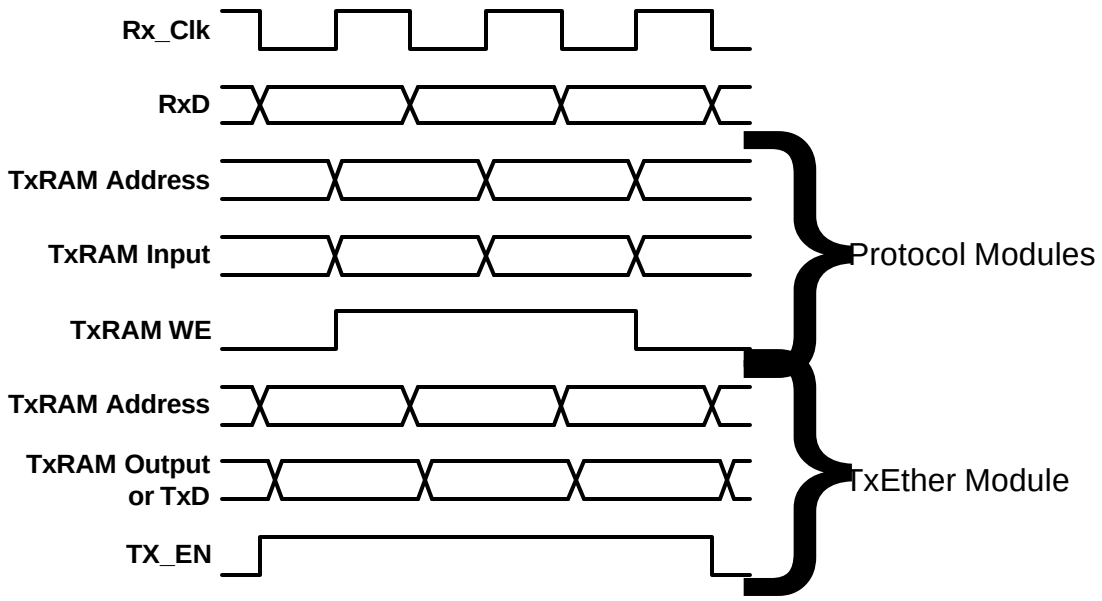


Figure 5.1b Timing waveforms

Another important note is that some Ethernet transceivers support the *preamble suppression mode*. If the Station Management Entity (i.e. MAC or other management controller) determines that all transceivers in the network support preamble suppression then the Station Management Entity need not generate preamble for each management transaction. This is a very important topic because if we choose to enable preamble suppression mode then we must change the RxEther fsm in order to omit the preamble states when receiving and transmitting a packet. In the first case we must start decoding Ethernet header immediately and in the second case we must start writing the MAC address to TxRAM without writing the preamble field first.

A typical application of the National's transceiver (DP83843) is shown below (see figure 5.1c). The RJ-45 module is connected to the transceiver through magnetic modules that provide EMI filtering for the network applications. Our implementation utilizes the MII interface in order to communicate with the transceiver. The status leds provide information concerning the network status, such as collision, the type of the network (10 or 100Mbit and half or full duplex), **f** it has sensed the network (link), and if it is currently transmitting or receiving. A 25MHz module provides the appropriate clock to the transceiver in order to be able functional either at 10Mbits or 100Mbits Ethernet

networks. The transceiver must be clocked by a 25MHz clock since it provides a 4-bit data bus, thus the 100MHz clock of the network is reduced to ¼=25MHz.

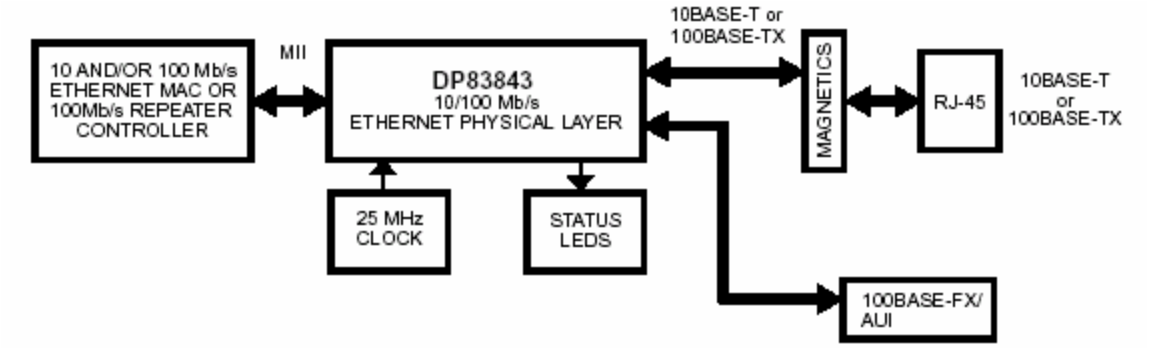


Figure 5.1c Typical transceiver application

5.2 PCI Pammete

PCI Pammete is a generic interface board based on reconfigurable logic manufactured by Compaq [32]. It contains a PCI interface FPGA which has a relatively fixed configuration, and four FPGAs which are programmed with application specific configurations. The board has also two banks of 16-bit wide 64k SRAM, and connectors for industry standard 72-pin SIMM DRAM modules which permit from 4MB to 256MB of DRAM to be attached. The specific board that we used contains a Xilinx 4010-E FPGA for the PCI interface and four Xilinx 4044-XL FPGAs for custom designs. The overall architecture of Pammete is shown below.

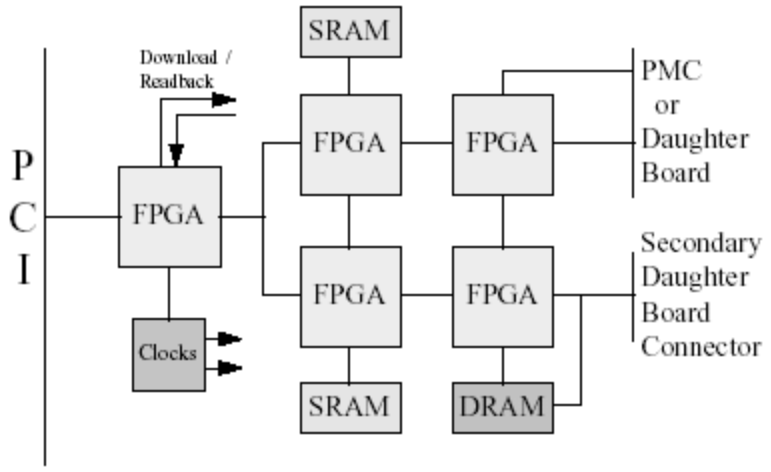


Figure 5.2a Pammete overall architecture

Using the Xilinx Foundation Environment the design, written in VHDL code, was converted to an appropriate netlist in order to be downloaded to Pamette. Compaq provides all the necessary libraries in order to download our design to Pamette using simple C functions.

The implemented design is compact enough to fit in only one 4044-XL FPGA and moreover only one of the two 64k SRAM banks was used, since the maximum TCP packet size is 64Kbytes.

Extensively, the design consists of a total of 1372 CLBs. The specific number of CLBs used by each module is described below:

Component	# CLBs	% CLBs	Speed (MHz)
RxEther module	174	12.7	23.1
ARP module	177	12.9	22.2
IP module	102	7.4	25.9
ICMP module	83	6	38.4
UDP module	73	5.3	37.4
TCP module	368	26.8	15.9
TxEther module	49	3.6	39.3
TxRAM	128	9.3	-
Checksum module	2x40	5.8	22.8
CRC-32 module	2x69	10	23.7
Total design	1372	100	10.1

For the specific FPGA (4040XL -3) the total maximum frequency is 10.1 MHz which is above the required one for 10Mbits Ethernet networks (1.25MHz) and slightly below the required frequency for 100Mbits networks (12.5MHz) with 8-bit data bus. If we use the same family with different speed option (-9) we get the desirable speed. In practice we know that these tools are rather pessimistic and the design can function properly in speeds above the tool estimated limit of 10.1 MHZ in the specific family. Actually the current implementation functions properly in 12.5MHz (80ns) in Pamette although the conservative speed in only 10.1MHz.

Using different Xilinx FPGA families we obtain the following results.

Family	#CLBs	Speed (MHz)
XC4044-XL -3	1372 CLBs	10.1
XC4044-XL -9	1372 CLBs	13.0
Virtex-V1000BG -6	1361* slices	13.7**

* without TxRAM module

** minimum (without timing constraints)

At this point we must note that module's fsms have been implemented into reconfigurable logic using one-hot encoding [9]. Xilinx FPGAs are particularly well-suited for one-hot encoded state machines, because flip-flop resources in the devices are abundant. In a one-hot encoded state machine, a single flip-flop represents each state. Therefore, an n-state state machine requires n flip-flops to implement. One-hot encoded state machines also tend to require less combinational logic because a smaller amount of decoding logic is required.

The external ports of the FPGA are divided into 3 groups. The first group provides all the necessary I/O pins that MII interface supports, the second group provides all the necessary I/O pins for the SRAM control and the third one uses all the spare pins for the debugging functions (5 bits indicates the current RxEther state).

MII Pins

- rx_clk : input - receiving clock
- rx_dv : input - receiving valid data
- rx_er : input - receiving error
- rxd : input - receiving data (8 bits)
- col : input - collision detected
- crs : input - carrier sense
- tx_en : output - transmit enable
- tx_er : output - transmit error
- txd : output - transmit data (8 bits)

SRAM Pins

- sram_oe : output - SRAM output enable
- sram_cs : output - SRAM chip select (2 bits)
- sram_we : output - SRAM write enable (2 bits; low or high bits)
- sram_a : output - SRAM address (15 bits; 32 K * 16 bits = 64 Kbytes)
- sram_dq : bidirectional - SRAM input and output (8 bits)

General Pins

- sram_r : input - SRAM read enable (start transmitting SRAM data to host)
- state : output - RxEther module's state

Chapter 6
Test and Verification

A failure will not appear till a
unit has passed final inspection.
Murphy's Law

Since our design is not targeted at any specific application there is no program that can be used to test the whole design. Instead there are many network utilities that can be used in order to examine the correct replies of the implementation to specific protocols. For example, ping is a quite useful program that can help us test our design in ARP and ICMP packets requests.

The flow of our verification starts with the execution of a network analyzer program. Network analyzers help us capture and analyze packets transmitted on the network. Upon starting this memory-resident program all incoming and outgoing packets are captured and saved in log files. Then we execute a network program such as ping, Internet browser or any ftp utility. At the end of execution every packet that this program has send to and received from the network is recorded in log files. These packets are transformed to a proper data file in order to be sent to Pamette through the PCI interface, using a custom C++ program. Using the same C++ program we save the packets that Pamette returns after processing the received packets. Finally, the packets received by network are compared to the packets received by Pamette. If the packets are the same, except some random fields such as identification numbers, then the implementation has met the required specifications.

The verification steps can be summarized in the following statements and are illustrated in the diagram below (see figure 6):

1. Start the execution of one network analyzer
2. Execute any network program (ping, ftp, chat)
3. Save the captured packets
4. Transform the packets to appropriate files (MII compatible)
5. Start the Visual C++ program in order to download the implementation (bit file) to Pamette
6. Send the new files to Pamette through PCI using the Visual C++ program (request)
7. Save the data that come from Pamette to log files (replies)
8. Compare the data received by network with the data received by Pamette

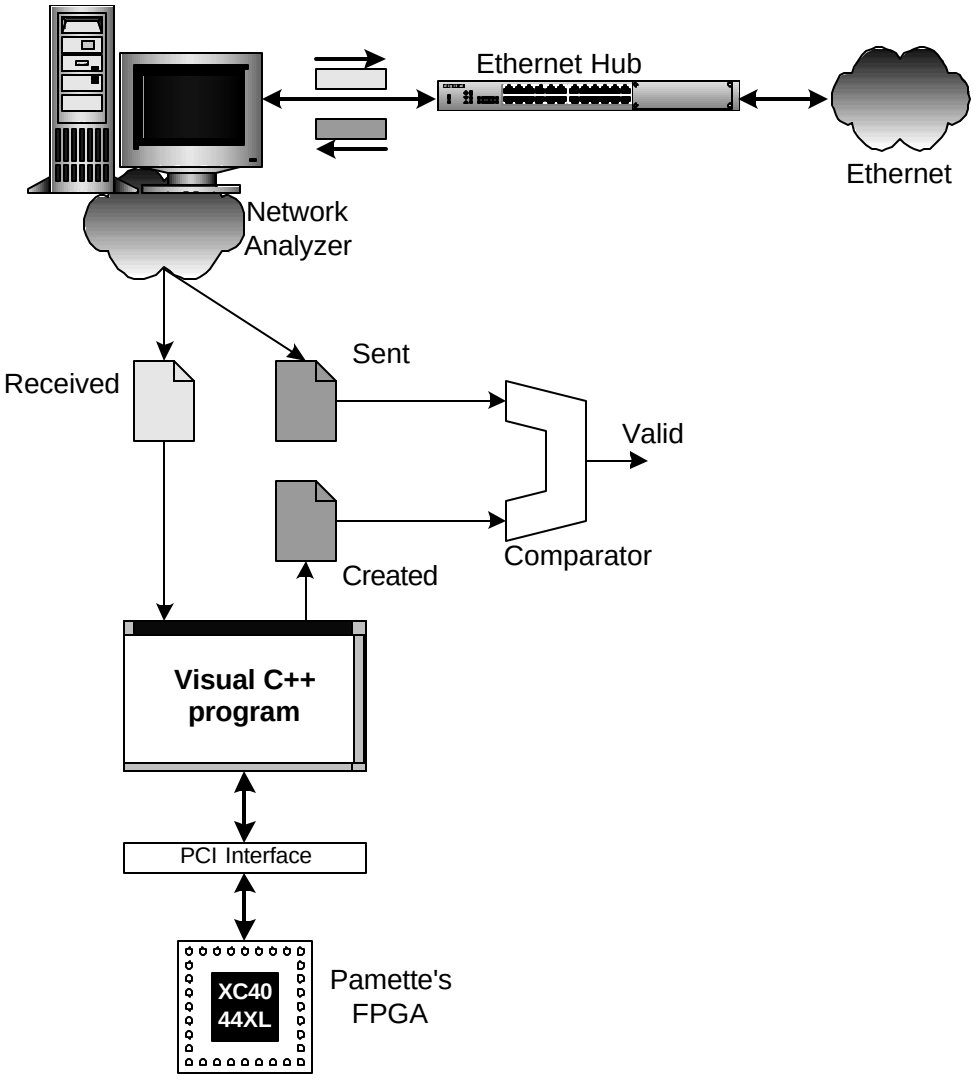


Figure 6 Verification block diagram

This chapter describes all the programs required to test and verify that the implementation meets the required specifications. The first section describes the network analyzer programs. The second section contains information regarding the networks programs that were executed in order to record the incoming and outgoing packets. This way the design can be stimulated by the same packets and compare the outgoing packets. The last section describes the program that was developed in Microsoft Visual C++ environment, which is responsible for sending the appropriate packets to Pamette through the PCI interface and recording the incoming packets from Pamette to a log file.

6.1 Network analyzers

Network analyzers, also known as packet sniffers, are programs used for capturing the incoming and outgoing packets on a network. These programs are able to capture not only packets that are destined for or come from the host computer but also every packet in the network. Network analyzer programs are usually freeware or shareware with many capabilities. The current programs are very sophisticated and most of them provide many functions such as selected capture depending on the protocol (TCP, UDP…), the port or the IP address, statistics and other useful options.

One mainstream packet sniffer that was used in the current thesis is CommView (TamoSoft Inc.). CommView [55] is a program for monitoring network activity capable of capturing and analyzing packets on any Ethernet network. It gathers information about data flowing on a LAN and decodes the analyzed data.

CommView allows us to see the list of network connections, vital IP statistics, and examine individual packets. IP packets are decoded down to the lowest layer with full analysis of the main IP protocols: TCP, UDP, and ICMP. Full access to raw data is also provided. Captured packets can be saved to log files for future analysis, as well as exported to other formats. A flexible system of filters makes it possible to drop unwanted packets and capture only packets with specific features such as IP address, TCP port or MAC address.

The main functions (tabs) of the last version of CommView (v2.3, see figure 6.1a) are listed below:

- IP statistics

  This tab is used for displaying detailed information about your computer's network connections (IP protocol only).

- Packets

  This tab is used for listing all captured network packets and displaying detailed information about a selected packet.

- Logging

  This tab is used for saving captured packets to a file on the disk. CommView saves packets in its own format with the .CCF (CommView Capture Files) extension.

- Log Viewer

  Log Viewer is a tool for viewing and exploring captures files created by CommView and several other packet analyzers. It has the functionality of the Packets tab of the main program window, but unlike the Packets tab, Log Viewer displays packets loaded from the files on the disk rather than the packets captured in real time.

- Rules

  This tab allows you to set rules for capturing packets. If one or more rules are set, the program filters packets based on these rules and displays only the packets that comply with the rules.

- Packet Generator

  This tool allows you to edit and send packets via your network card.

By using this program we were able to capture all the packets that networks utilities send to the network. The specific network utilities that we used in this thesis are described in the next section.

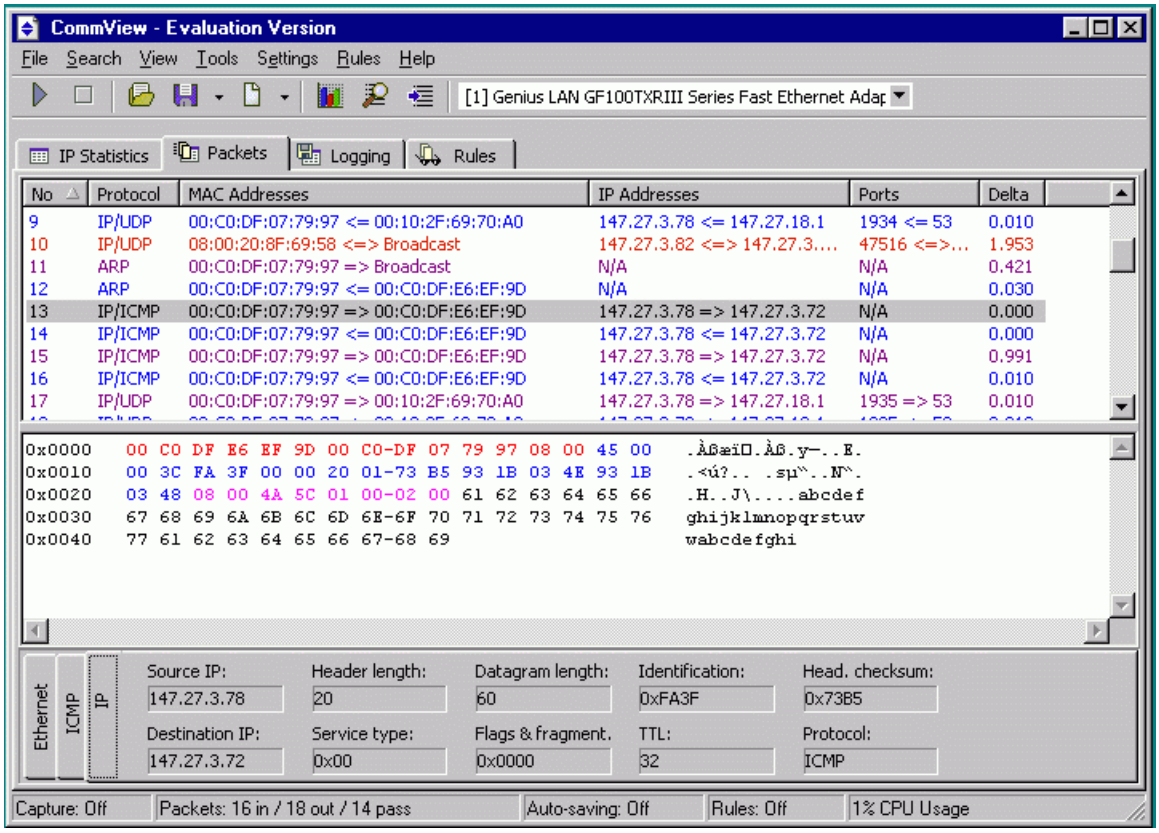


Figure 6.1a

6.2 Network programs

The networks programs that we used to test our design are divided into two groups: off-the-shelf programs and custom programs. Off-the-shelf programs are widely used programs such as ping, telnet and ftp utilities. Custom programs are utilities that were developed in Microsoft Visual Basic Environment and were used to test the trivial communication of TCP and UDP protocols in simple tasks such as chat communication or simple file transfer.

6.2.1 Off-the-shelf programs

Ping

Ping is the most widely used utility to check a valid connection. The ping utility verifies connections to remote computer or computers, by sending ICMP echo packets to the computer and listening for echo reply packets. Ping waits for up to 1 second for each packet sent and prints the number of packets transmitted and received. Each received packet is validated against the transmitted message. By default, four echo packets containing 64 bytes of data (a periodic uppercase sequence of alphabetic characters) are transmitted. We can use the ping utility to test both the computer name and the IP address of the computer.

Ping can be run using many parameters. The main parameters are:

- *-n count*
  Sends the number of ECHO packets specified by count. The default is 4.
- *-l length*
  Sends ECHO packets containing the amount of data specified by length. The default is 64 bytes; the maximum is 8192.
- *-i ttl*
  Sets the Time to live field to the value specified by ttl.
- *-w timeout*
  Specifies a timeout interval in milliseconds.

Before sending an ICMP echo request, ping sends an ARP request in order to obtain the physical address of the network card by advertising the corresponding IP. The ping utility was used to obtain both ARP and ICMP packets in order to test the design, while custom programs were used for testing the other protocols.

6.2.2 Custom Programs

The custom programs were developed using Microsoft Visual Basic. Visual Basic provides us with a complete set of tools to simplify rapid application development. The main advantage of Visual Basic is the ActiveX controls. ActiveX controls, formerly called OLE controls, are standard user interface elements that allow us to assemble forms and dialog boxes rapidly. For the purposes of this thesis the WinSock ActiveX control was used. WinSock control allows us to connect to a remote machine and exchange data using either the User Datagram Protocol (UDP) or the Transmission Control Protocol (TCP). Both protocols can be used to create client and server applications. By using this module two utilities were created; one for each protocol. The UDP utility implements a simple chat application without the need for a server and a client program. In contrast, the TCP utility is divided into two applications; the server and the client utility.

UDP chat utility

UDP chat utility consists of two text boxes and three options (see figure 6.2.2a). These options determine the name of the remote host we want to chat with, the remote port and the local port. After initiating the program we can chat with the other host and in the same time we can capture the incoming and outgoing UDP packets by our network analyzer, CommView. Every character typed in the text box is sent to the network encapsulated into a UDP packet.

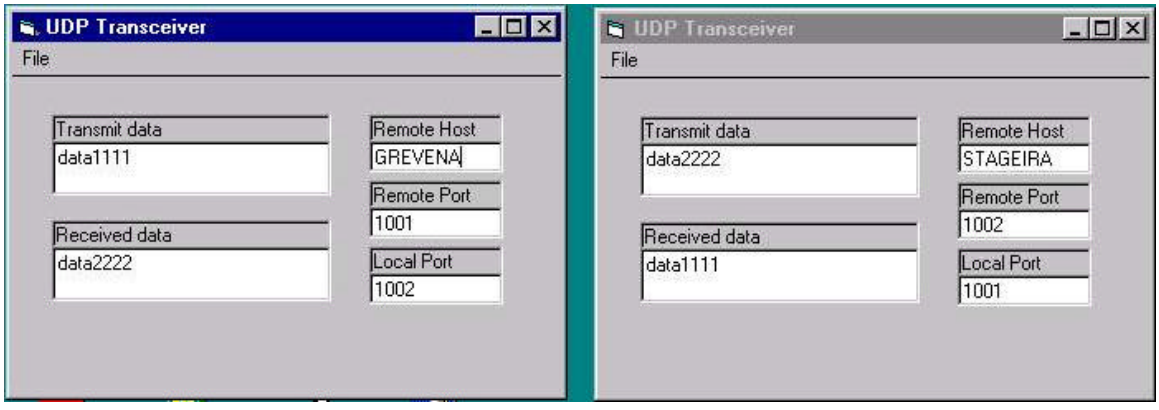


Figure 6.2.2a The UDP chat application

TCP chat utility

The TCP chat utility is divided into two sub-utilities. The first one is the server and the second the client utility.

The server utility is pretty simple since its graphical user interface (GUI) consists of two text boxes representing the transmitted and received characters. The server port number is set to 1001 and is actually a passive open port. This means that this port only listens to the network and does not initiate a connection.

The client utility is more sophisticated. The GUI consists of two text boxes, two options boxes and three commands. The text boxes represent the transmitted and received characters while the options boxes let the user specify the remote host's name and the remote host's port number. In order this utility to communicate with the server utility the remote host name must be the computer's name that the server utility is running and the remote port number must be set to 1001. To initiate a connection we must run the *Init* command first and then the *Connect* command. To end properly a connection the *Exit* command must be executed.

In contrast to the UDP utility, the TCP connection is not established immediately according to the TCP protocol. First the TCP client sends a packet stating that it wants to initiate a connection. The TCP server responds to this packet by sending an acknowledgment packet and then the TCP client replies with its acknowledgment packet. This way the connection is established. Each time a character is typed in any utility, it is sent as a packet to the other utility. By executing the *Exit* command, the client utility sends a packet stating it wants to close the connection. The Server utility sends an acknowledgment packet, a packet also stating the end of connection and then waits to receive the acknowledgment packet. Upon receiving this packet the connection has been totally closed.

These utilities enable us to capture all the TCP packets needed to stimulate our design and verify the proper response of the implementation. Since the implementation on Pamette does not have any packets to send, it corresponds to the TCP server utility, while the computer on which Pamette is installed acts as the TCP client. Pamette has a passive open port, set to 1001, and responds to all packets that refer to that port.

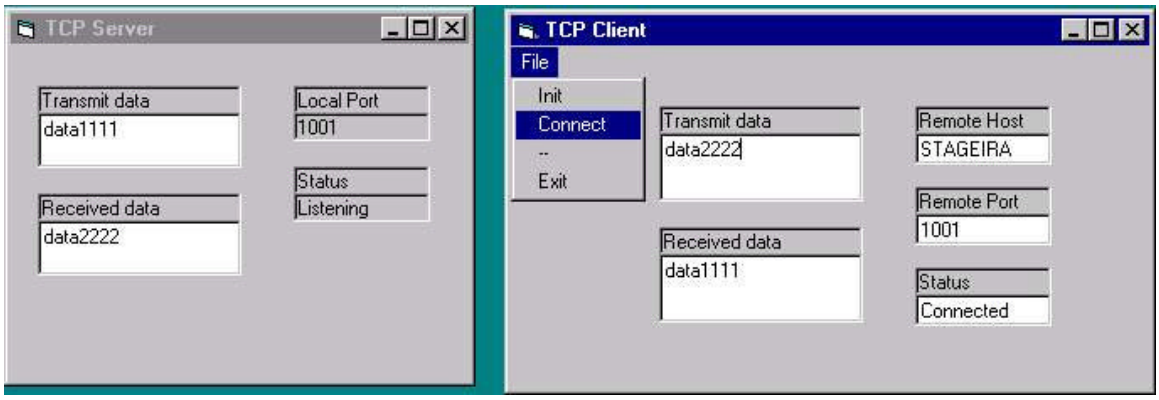


Figure 6.2.2b The TCP chat application

The next section describes the program that was developed in Visual C++ in order to stimulate Pamette through the PCI interface and record the responses in UDP and TCP packets referring to a specific port.

6.3 H/W – S/W co-simulation

Although Pammete has many different interfaces modes to communicate with the host computer through the PCI interface the static mode interface is used for this implementation. This mode is a simple low-performance interface that provides statically configured 16-bit paths to and from the custom FPGAs (see figure 6.3). Extensively,

from the host side it consists of a single 32 bit *link* register (address 0x38 in struct PamRegs of PamRT library) that can be read or written. The high 16 bits of write data are ignored. From the user-area side it consists of a 16 bit input port driven by the low 16 bits of the link register and a 16- bit output port which loads the high 16 bits of the link register on each Clksys cycle.

The user-area input port is EBus<15:0> and the output port is EBus<31:16>.

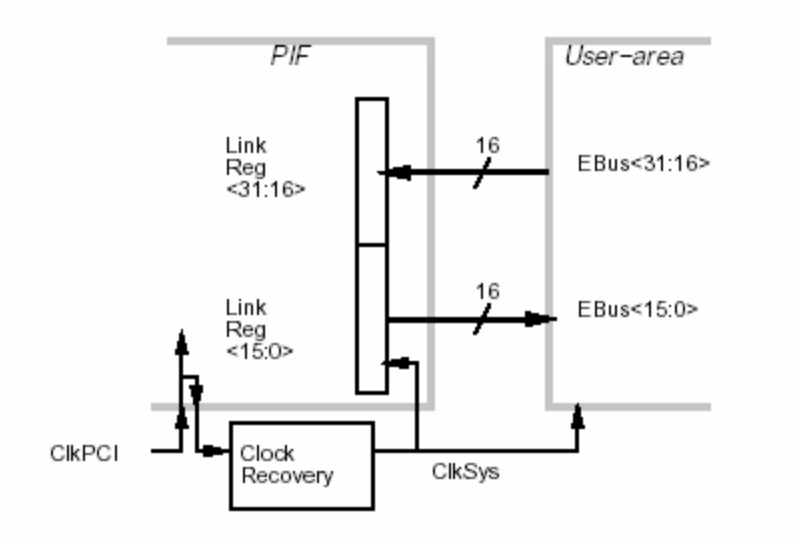


Figure 6.3 Static mode

Even though Pammete supports interfaces that provide 32-bit bidirectional bus the static mode was selected since the I/O pins of the design are less than 16-bit in each direction (input-output).

The program that has been created in C++ uses the static interface mode to send and receive data through the PCI interface. These data has the MII interface structure so that the simulation is as realistic as possible in case our design is connected to an Ethernet transceiver. Our program sends the packet that was captured using the network analyzer and then saves the reply into a log file. The protocols supported by the program are:

- ARP,
- ICMP,
- UDP and
- TCP

The ARP and ICMP packets sent to the Pamette are captured by the ping program. The UDP packets are captured when the custom UDP chat application is running. On the other hand the TCP packets are captured when the custom TCP chat application is running. Packets captured by a running Internet browser are also used as TCP packets for simulation. Using these packets our implementation is tested if it can respond properly to fragmented TCP packets, by saving the data to the appropriate address of the SRAM.

This program supports an additional function, the *read SRAM* option. Running this command we have the ability to save the SRAM contents to a log file through our FPGA and EBus<31:16>. The SRAM contents are saved to the *sram.log* file. This way the TCP data saved in the SRAM can be checked.

A GUI program was created in Visual Basic in order the C++ protocol Simulator to be more user friendly. This program consists of 4 buttons, one for each supported protocol and a button to save SRAM contents to a file. In addition there is command history text box and a button that is used to open the corresponding log file for each protocol. The program's GUI in shown below.

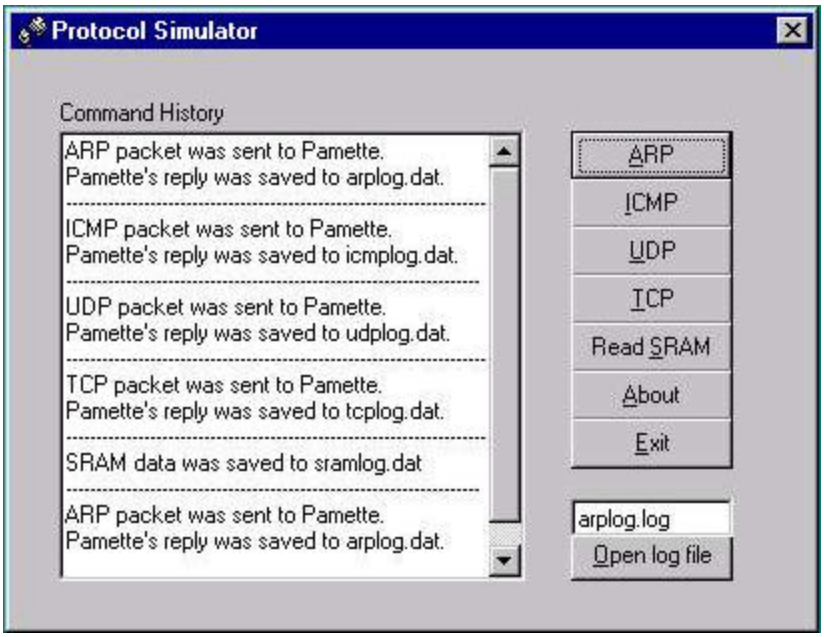


Figure 6.3b Visual Basic's Protocol Simulator

Chapter 7
Conclusions and Future Work

New systems generate new problems.
Murphy's Law

7.1 Conclusion

This document describes all the basic steps, according to the waterfall model [9], that need to be taken in order to go from a problem to a complete system. The early chapters described the requirements and the specifications of the desired IP core. Then we moved on analyzing the specific design for this core and the implementation of this design into reconfigurable logic. Finally we described the various steps of testing and validating this design

As displayed in the implementation chapter, the design is compact enough to fit into many of the current FPGAs. This way the TCP/IP core can be easily embedded in a larger design that will target any application needing Internet connectivity. Thus any future application that will incorporate this core will only need one chip, while the TCP/IP ASIC solutions need at least two chips (one for Internet connectivity and one for the specific application). This implementation is also fast enough to function into fast Ethernet network (100Mbits), in contrast to the microcontroller's library solutions that target to 10Mbits Ethernet networks only.

7.2 Future work

The modular structure of this design allows it to be upgraded in many ways. It can be upgraded in the design level by changing the supported protocols or at the application layer by adding various accessories that need Internet connectivity.

A possible upgrade of the design level is the change of the RxEther module to support ATM networks instead of Ethernet networks. The IP module can also be changed in order to support the new IP v.6 protocol (also called IP Next Generation). Another possible upgrade would be the addition of upper OSI's layer modules in order to achieve a total protocol representation to hardware. These modules could implement any upper layer application such as telnet, FTP, TFTP or SMTP. In addition a possible upgrade would be the support of multiple connections in order to serve multiple clients.

In the implementation level the most important upgrade would be the connection of the design with an Ethernet transceiver, in order to examine its proper function in a real network.

In addition this module could be used to specific applications that need Internet functionality. For example the FPGA could be connected to a digital camera in order to implement a web camera without the need of a PC. Furthermore the implementation could be incorporated to any household appliance that needs to be connected to Internet, thus implementing a part of the smart house idea.

This implementation is not restricted to commercial devices. It can also be used, with the appropriate changes, in an active network project such as PLATO as a network switch that forward the packets depending on the header or even the packet's data. It can also be used as a hardware-based packet sniffer which analyze the packets much faster and more efficiently than software-based packet sniffers (hopefully not by Echelon). This TCP/IP core has unlimited applications and can only be restricted by the designer's imagination.

References

Bibliography

[1] ?.??e??p?????, G.?a?????????, *???ep????????e? ?a? ???t?a ?p?????st??*, ISBN: 9602200863
[2] Timothy Parker, *Teach yourself TCP/IP in 14 days*
[3] Gilbert Held, *Ethernet networks*, ISBN: 0471253103
[4] Jean Walrand, *Communication Networks*, ISBN: 025608864 0
[5] Chris Lewis, *Cisco TCP/IP Routing Professional Reference*, ISBN: 0070411301
[6] Peter Ashenden, *The Designer's Guide to VHDL*, ISBN:1558602704
[7] Stefan Sjoholm, *VHDL for designers*, ISBN: 0134734149
[8] Mark Zwolinski, *Digital System Design with VHDL*, ISBN: 0201360632
[9] Apostolos Dollas, *The Art of Microelectronics Systems*

Requests for Comments (RFCs)

http://www.netsys.com/rfc/index.phtml

[20] RFC768 "User Datagram Protocol"
[21] RFC791 "Internet Protocol"
[22] RFC792 "Internet Control Message Protocol"
[23] RFC793 "Transmission Control Protocol"
[24] RFC826 "Ethernet Address Resolution Protocol"
[25] RFC862 "Echo Protocol"
[26] RFC894 "Standard for the Transmission of IP Datagrams over Ethernet networks"

World Wide Web

FPGAs

[30] http://www.xilinx.com
[31] http://www.altera.com

Pamette

[32] http://www.compaq.com
[33] http://www.motorola.com
[34] http://www.ee.princeton.edu/~zhenluo/Pam/pam.html

TCP/IP ASICs and microcontroller's libraries companies

[35] http://world.std.com/~fwhite/ace/
[36] http://ilima.eng.hawaii.edu/XCoNET/XCoNET.htm
[37] http://www.rabbitsemiconductor.com
[38] http://www.iready.com
[39] http://www.seiko.com
[40] http://www.hyundai.com
[41] http://www.connectone.com
[42] http://www.edevice.com

[43] http://www.embeddedpower.com
[44] http://www.livedevices.com
[45] http://www.triscend.com
[46] http://www.cmx.com

Ethernet transceivers
[47] http://www.intel.com
[48] http://www.tdk.com
[49] http://www.ti.com
[50] http://www.national.com
[51] http://www.pulseeng.com

CRC
[52] http://www.ee.mu.oz.au/staff/summer/pl_case_study/vhdlvers/crccalc.html
[53] http://cell-relay.indiana.edu/cell-relay/FAQ/ATM-FAQ/f/f3.htm
[54] http://www.ee.mu.oz.au/staff/summer/pl_case_study/vhdlvers/solution.html

Packet sniffers
[55] http://www.tamos.com
[56] http://www.wildpackets.com

Miscellaneous
[57] http://www.cisco.com