Enabling Gigabit IP for Embedded Systems

Nicholas Tsakiris

B. Eng. (Computer Systems) (Honours)

Flinders University of South Australia

A Thesis Submitted for the Degree of Masters by Research

Flinders University

School of Computer Science, Engineering and Mathematics

Adelaide, South Australia

2009

(Submitted 17th February 2009)

Dedicated to my parents for their support and love.

Abstract

For any practical implementation of chip design, there needs to be a hardware platform available for the purpose of prototyping and implementation of FPGA-based programs, whether they are written in VHDL or Verilog. Communication between the platform and a computer is a useful feature of many hardware solutions as it allows for the capability of regular data transmission between the two devices. Furthermore, the ability to communicate between the platform and a computer at high-speeds requires a specially constructed interface, one that can be modified by the designer at their choosing.

There are a number of commercial packages which provide a hardware platform to perform this task, however there are drawbacks to many of the available options. Some may require special hardware to connect to a computer using proprietary connectors or boards, which increases the cost and reduces the flexibility of any solution. Other options may have limited access to the internal structure of the interface, limiting the ability of the developer to modify the interface to suit their needs. There may be an extra cost to provide the code to the interface, separate from the board, which can also tax design budgets.

This dissertation provides a solution in the form of a Gigabit Ethernet connection with a custom IP/network layer written in VHDL to facilitate the connection. With an increasing number of IP-enabled devices available such as IPTV and set top boxes, the ability to link hardware using Ethernet is very useful and so the development of a lean and capable network layer was considered a suitable focus for the project. The overall goal has been to provide an interface which is cheap, open, robust and efficient, retaining the flexibility a developer might require to modify the code to their needs.

After covering some basic background information about the project, the dissertation looks at the requirements of the board and interface, as well as the alternative interface solutions which were looked at before deciding on Gigabit Ethernet. The protocols used in Ethernet are then covered, with both an explanation of the structure of each and their relevance to the implementation. The Finite State Machines which control operation of the interface are covered in depth, with an explanation of their inter-connectivity to each other and how they fit in the data-flow between the computer and the board. Error correction and reliability is discussed, as well as any remaining components critical to the operation of the interface.

Pipelining, the method of design which provides the speed required for Gigabit Ethernet, is covered along with the extra speed optimisation techniques used in the design such as RAM swinging buffers. Testing and synthesis are covered which ensure the design is as robust as possible, both in simulations and in real-world applications. The final design was implemented on a Xilinx Spartan 3 FPGA (XC3S5000-5FG900C) and capable of a maximum speed of 128.287 MHz, which is more than enough to satisfy the requirements of Gigabit Ethernet under a variety of network conditions. The interface code occupies 1,166 slices of logic on the FPGA (3% of the total amount of logic available), making it sufficiently compact to run large projects on the same chip. The core was tested on physical hardware and performed correctly at real line Gigabit speeds. Configuration of the computer along with the method of connecting to the board and transferring data is mentioned, with explanation of the code run on the computer to make this possible. Finally, the dissertation provides an example application through the use of JPEG2000 image compression/decompression.

Acknowledgements

I would like to thank my supervisor, Professor Greg Knowles, for his invaluable help in getting me through my candidature. To my colleagues at the Flinders University of South Australia, School of Informatics and Engineering, I would like to say a grateful thank you. In particular, to Paul Gardner-Stephen for his help in understanding the intricacies of networking protocols, as well as Geoff Cottrell, Craig Peacock and Terry MacKenzie for their assistance. I would also like to thank the academics and staff of that school for their continual support over the past few years. Finally, I would like to thank my friends and family who have helped me during the period of my candidature.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

FPGAs (Field-Programmable Gate Arrays) are a useful tool in the electronics industry for constructing prototype designs before mass fabrication onto dedicated hardware and are often used themselves as part of the final design. They are reprogrammable, flexible and extendable, with the capability to run several programs at once and at different speeds. For many designs, interfacing with a computer may be required for data I/O, programming and debugging. If the requirements of the design call for high-speed data transfer with a computer, it is preferable to find some way to accomplish this using an existing interface on the computer, to maximise portability and reduce the dependency on specially-designed hardware.

A particularly common interface on many computers is the Ethernet port, normally used for wired network connections to LANs and WANs. The commonality of this interface makes it ideal for interfacing with an FPGA prototyping board, particularly if both network adaptors involved are capable of Gigabit speeds. However, to actually receive and transmit data using Gigabit Ethernet and have that data available for other programs that reside on the FPGAs is not necessarily straightforward, particularly if one wishes to customise aspects of the interface. There are IP cores available for purchase from several vendors which can provide Gigabit Ethernet functionality for FPGAs, but these generally reside as black-boxes and due to them being distributed as encrypted netlists, do not provide the designer with anything but the inputs and outputs of the core, which makes them unsuitable for the designer who wishes to modify the interface code directly. Sometimes it is possible to obtain the source code for these black-boxes, but the extra cost of the code can add substantially to the overall cost of the IP core. For example, Alcatel provides a fully-featured Gigabit Ethernet core for Altera FPGAs,[2] however costs start at $30,000 for an encrypted netlist without code. The source code can be purchased, but for an additional cost.

The purpose of this dissertation is to cover the design of a custom IP/network layer, one which has low cost, high reliability and an open structure for easy manipulation. The primary focus was to find an efficient engineering solution to a practical problem, the problem being how to develop the layer to work with low-power devices. Efficient engineering would solve this problem and provide the ability to use low cost hardware to support Gigabit Ethernet line speeds. The design of the core makes it streamlined for typical FPGAs and does not require higher-end hardware.[3,4] The base platform for its design was a Xilinx Spartan 3 FPGA, but the core can be implemented on other FPGAs so long as the base clocking speed of 125 MHz can be obtained. It is not just FPGAs which would benefit from such a design; there are also an increasing number[5] of IP-enabled devices (eg. IPTV, set-top boxes, fridges) which would benefit from a fast and lean network layer without the bloat of extra protocols and functionality which would not be needed in these highly-specialised devices. For this to be achieved, certain features which are available with commercial solutions are not present, but the benefits of a simpler core are evident once the designer has to put the solution to use. The dissertation also covers the physical implementation of the core on real-world hardware as well as the tests performed to validate the core's accuracy and reliability.

Achieving these requirements and solving the problem of an efficient design required some compromises. ARP support was not implemented due to lack of time. TCP support was not implemented due to the fact that the protocol is never implemented entirely in hardware but rather a software/hardware combination using an embedded CPU, which was not available with the sole Spartan 3 FPGA. The Treck TCP/IP core for Xilinx FPGAs is an example of a core which could perform as an offload engine for processing TCP packets, when run on an embedded or soft processor on an FPGA such as MicroBlaze or a PowerPC CPU.[6] However, even with an embedded CPU the size of the core would increase in size and complexity to a level that was not desirable for achieving the lean and clean architecture, which were part of the goals of the design. The lack of packet error detection/correction that is an inherent part of TCP was still provided through the use of tags. The issue of achieving full Gigabit speeds on the base hardware (the Spartan 3) was ultimate the main factor in determining how to construct the core and still satisfy the requirements of the problem.

This chapter introduces several important concepts and ideas which are needed to fully understand the issues raised in this dissertation. Section 1.1 provides a brief introduction into FPGAs, what they are and how they can be used. Section 1.2 explains what VHDL is and what its purpose is with regards to chip design. Finally, Section 1.3 provides a short introduction to Ethernet extending to Gigabit Ethernet and its purpose for this design.

1.1 FPGAs

A *field-programmable gate array* is a semiconductor device which contains logic components (also known as logic blocks) which are programmable. By selectively programming the device these logic blocks can function as basic logic gates such as AND, OR, XOR, NOT, or can be extended into more complex combinational functions such as encoders, decoders or simple mathematical functions. Modern FPGAs also contain special logic designed to act as memory elements such as RAMs or FIFOs and depending on the type of FPGA the memory elements may be constructed from flip-flops or dedicated memory blocks on the chip. The key function of an FPGA is to provide the ability to run logic programs with the advantage that the FPGA can be re-programmed multiple times, whereas a regular integrated circuit with support for logic gates would have a fixed design, permanently selected and unable to be altered. Despite being slower than a dedicated chip with a permanent design, FPGAs have a much greater level of flexibility and coupled with the ability to easily be reprogrammed, are ideal for running prototype designs and also for performing multiple tasks with the same hardware.

FPGAs have existed since the mid 1980's when Xilinx released the XC2064, the first FPGA. Despite only supporting a size of 1,000 gates, compared to sizes 10,000 times greater in 2004, this initial form of the FPGA proved very popular.[7] The ability to program the same chip over and over again provided cost-effective design development and increased the development of chip design theory and application. FPGAs have a wide range of applications, from digital signal processors (DSPs) to cryptography and beyond.

1.2 VHDL

To program an FPGA, a design-entry language suitable for specifying how the logic blocks interconnect together to perform their tasks is used. For this we use a Hardware Description Language (HDL), which encompasses any computer language specifically designed to formally describe electronic circuits. There are two main languages for this purpose: VHDL and Verilog. VHDL (VHSIC Hardware Description Language, fully expanded as Very-High-Speed Integrated Circuit Hardware Description Language)[8] is the language used by the Gigabit Ethernet project in this dissertation. It is capable of rendering the entire structure of the FPGA including logic, connections and ports and also allows easy simulation capability due to the construction of a testbench. Verilog[9] is another widely-used HDL, but although Verilog is somewhat simpler and easier to code, VHDL was chosen for this design for reasons of familiarity.

1.3 Ethernet

1.3.1 History

Ethernet is the most common technology used on Local Area Networks (LANs) today. Developed in the 1970s by Xerox Corporation, the experimental version of Ethernet ran at 3 Mbit/s, but the first widespread standard of Ethernet ran at a speed of 10 Mbps in 1985 and later at 100 Mbps (sometimes referred to as *Fast Ethernet*) in 1995, at which point Ethernet had become the regular network system for most computers. The first Ethernet networks, 10BASE5, used thick yellow cable with vampire taps as a medium. Later versions of Ethernet (10BASE2) used thinner coaxial cable with BNC connectors as the connection medium. Currently Ethernet has many varieties that vary both in speed and physical medium used. The most common forms used currently are 10BASE-T, 100BASE-TX and 1000BASE-T. All three utilise twisted pair cables and 8P8C modular connectors, more commonly known as RJ45 (Registered Jack 45) connectors.[10] These forms run at 10 Mbit/s, 100 Mbit/s and 1 Gbit/s speeds respectively.[11]

The RJ45 medium is made from copper cabling, which is suitable for 10BASE-T and 100BASE-TX but can sometimes cause problems with the higher 1000BASE-T form of Ethernet. Due to the significant increase in speed and bandwidth requirements, 1000BASE-T is less tolerable of imperfections in the network cabling than previous standards and electrical noise can potentially degrade a Gigabit connection severely when used with poor-quality or inappropriately specified copper cabling. Most modern Ethernet cabling can support 1000BASE-T satisfactorily, but signal degradation becomes more of a problem the longer the cable becomes. Fibre optic variants of Ethernet are commonly seen connecting buildings or network cabinets in different parts of a building but are rarely seen connected to end systems for cost reasons. Their advantages lie in performance, electrical isolation and distance, up to tens of kilometres with some versions. Fibre cabling is therefore a lot more desirable when dealing with super-fast Ethernet connections such as 1000BASE-SX in a large environment, but is not required in most small networks due to the quality of regular copper cabling.[12]

1.3.2 Gigabit Ethernet

Gigabit Ethernet is a form of the Ethernet standard which allows for high-speed transfers up to one Gigabit per second. The standard was approved by the IEEE in 1998 and later adopted by ISO. The initial standard for Gigabit Ethernet was known as IEEE802.3z, however the most commonly implemented form of Gigabit Ethernet (IEEE 802.3ab) was ratified a year later by the IEEE and uses unshielded twisted pair cabling as opposed to fibre cabling in the initial standard. The reason for the latter standard being more useful is because it allows existing copper cabling infrastructure, used for 10/100 MBit Ethernet, to remain in place without having to be replaced by fibre optic.[13] The fibre version of Gigabit Ethernet is known as 1000BASE-SX and can transmit along a single fibre line at a distance of 500m or more with modern cabling before requiring an endpoint. The unshielded twisted pair variant, 1000BASE-T, generally has a maximum length of 100m. The medium chosen however does not affect the operation of the network layer and is up to the requirements of the environment as to which medium to choose. The same core can be used for either.

Gigabit is the logical successor for 10/100 MBit connections found in virtually all NICs (Network Interface Card) and has stood as a standard for some time and support has become very common, with most motherboards with integrated Ethernet supporting Gigabit, as well as new cards generally supporting it as well. The increase in speed is not only due to the higher clocking speeds (125MHz as opposed to 25MHz in 100 MBit), but also double the transmission bits (8 bits instead of 4). This results in a potential 10 times increase in available bandwidth, which makes it useful for high-speed transfers to and from an FPGA. Furthermore, since Gigabit Ethernet is a common standard, it is trivial to find hardware which can support this standard at a reasonable cost without having to resort to other, more exotic forms of data transfer between an FPGA and a PC.

Chapter 2

Requirements

This chapter of the dissertation covers the necessary requirements that precipitated the need for Gigabit Ethernet project. Section 2.1 discusses the purpose of the Xilinx prototyping board and the importance of the high-speed/high-bandwidth requirement. Section 2.2 covers several ideas that were considered before finally settling on the Gigabit Ethernet option.

2.1 High Speed/Bandwidth

The origins of this project were in the Embedded Systems Lab in the Engineering department of Flinders University. A characteristic of the work performed by this lab is the data-heavy nature of the designs used, which can range from DNA sorting algorithms to JPEG2000 wavelet transforms to general compression/decompression setups. Speed and bandwidth are essential for performing these tasks; a digital camera, for example, needs to have a fast hardware implementation when performing image compression for storing images to avoid lag time between shots. High clocking speeds allow the circuitry to perform their tasks faster, plus high bandwidth allows a greater amount of data to be processed per clock cycle. Combining the two advantages result in a huge improvement in the capabilities of hardware implementations compared to their software equivalents, which is what makes hardware designs so attractive for performing specific tasks.

2.2 Proposed Solutions

Since data transfer between the FPGA development board and a PC had to satisfy the requirements of high-speed and high-bandwidth, there needed to be a useful way to establish the connection, both at the hardware and software level. Modern computers have a wide variety of different connection standards and the choice to use a specific connection option depends on several factors (some more important than others):

- Performance (the raw speed of the connection, plus maximum bandwidth the connection is capable of)
- Functionality (is any error-correction inherit in the connection standard? is the connection full-duplex? etc.)
- Flexibility (the ability to adjust the way the connection functions, whether through hardware or software)
- Scalability (how capable is the connection method for expanding? can it support newer revisions without changing hardware?)
- Lifespan (how long can we expect to have the connection standard available to us as technology progresses? how quickly will it become obsolete?)
- Commonality (is the connection ubiquitous? can it be found on the majority of computers, or does it require exotic systems?)
- Physical implementation of the connector on the FPGA board (getting the relevant connector working on the FPGA development board must not be par-

ticularly difficult, otherwise the design begins to go beyond the requirements of cost and simplicity)

2.2.1 USB

One option proposed was USB (Universal Serial Bus). USB is extremely common among most modern computers, with the Hi-Speed (2.0) standard supporting a maximum data rate of 480 Mbit/s.[14] However, the two main issues with USB are that the use of high-speed USB requires a maximum cable length of only five meters,[15] which meant the board would have to be linked to the PC fairly closely and though the data rate was fast, there were still faster options available.

2.2.2 PCI-Express

In searching for the fastest connections available to and from a PC, the computer's mainboard is the most logical place to look. Since hardware directly connected to the mainboard will obtain primary access to the system's resources, this results in the highest speed and bandwidth connections available on a PC. A novel idea briefly examined was to incorporate the use of a PCI-Express computer expansion card, a standard created by Intel in 2004.[16] The proposal was to specifically use a PCI-Express video card and utilise the dual-DVI (Digital Visual Interface) connectors on the card for raw data transfers. This could be achieved by accessing the video frame buffer, bypassing video-related functions and directly accessing the card's memory to transfer data quickly through the high-speed PCI-Express slot. Using current hardware supporting version 3.0 of the standard, the maximum potential transfer rate for a single PCI-Express slot can be as high as 8 GB/s[17]However, future developments would be hindered somewhat as the DVI standard has stagnated at version 1.0. The Digital Display Working Group which developed the DVI standard has since disbanded after the initial release of DVI and so there is no guaranteed upgrade path using the connector, which is why several new standards such as DisplayPort are being established to provide a future connection standard to replace DVI.[18] It was therefore decided that the nature of designing a communications system using a video card's frame buffer would not make it feasible for this design and instead to design this project around a more established standard which could be upgraded in the future with little extra work.

2.2.3 HyperTransport

HyperTransport is an interesting concept in the development of high-bandwidth, low-latency connectivity solutions. The purpose of the interface is to provide a direct link to the computer's CPU with a focus on bandwidth/latency reduction.[19] Key to the connection is support from the CPU and motherboard chipset to allow this connection to work properly. The primary connection for the HyperTransport interface is known as *HyperTransport eXpansion* (HTX) which utilises uses the same mechanical connector as a typical PCI-Express slot. Further functionality can be obtained through the use of plug-in cards which support direct access to a CPU and DMA access to the system RAM.

The primary advantage of HyperTransport is that it provides a flexible data rate ranging from 200 MHz to 2.6 GHz depending on the HT version used and how it is configured.[20] When run using the full-sized, full-speed 32-bit interconnect, the maximum transfer rate is 41.6 GB/s per link. Obtaining mainboards which support HyperTransport is not difficult, although as of yet only AMD-based systems are capable of supporting the standard; Intel has a competing standard to HyperTransport called *Intel QuickPath Interconnect*, although a practical implementation of *Quick-*

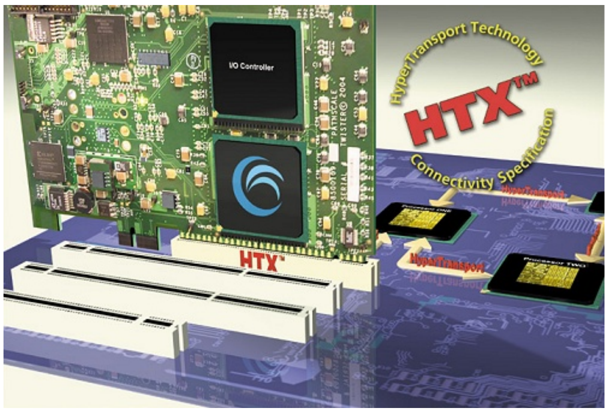

Figure 2.1: HyperTransport plug-in card concept

Path as it is known is not expected to be released until late 2008, too late for the requirements of the project.[21]

An important distinction between other forms of communication with the computer is that by using HyperTransport, the FPGA and relevant hardware must exist on the plug-in card itself. This is unlike other options which use cabling and as such the extra requirements on the design of the FPGA board mean it has to have a physical profile that can fit inside the computer. There is also the issue of limited physical access to the board when the computer case is on, which could cause problems when attempting to view on-board displays or press switches.

There are other ways in which HyperTransport can be used. Instead of plugging a card into an expansion slot, HT modules can be installed directly into a free CPU slot of any compatible multiprocessor motherboard. DRC is a company which supplies plug-in modules supporting this very option; the RPU100 supports a single HT bus along with the Xilinx Vertex 4 LX 60 FPGA. It is inserted directly into a free 940 socket located on a standard AMD Opteron microprocessor motherboard.[22] Multiprocessor motherboards are a requirement when using this direct-access method since the motherboard still requires at least one CPU to run the computer and the HT modules occupy an entire CPU slot without providing the ability for a CPU to piggyback. This would produce the fastest possible bandwidth capabilities of HyperTransport, but would also further limit physical access to the board and require even more specific (and potentially quite expensive) computer hardware. Given other alternatives available, HyperTransport did not satisfy the needs of the project.

2.2.4 InfiniBand

InfiniBand is a communications technology designed for high-performance architectures, which has made it popular with supercomputers and data centres. It is currently the de facto standard for interconnections with these computing systems.[23] InfiniBand has not made much impact with regular desktop computers in part due to cost, but also the significantly larger bandwidth it provides compared to bandwidth requirements a regular desktop system requires. There are some efforts to adapt InfiniBand as a standard or semi-standard interconnect between low-cost machines as well, plus desktop implementations do already exist in a way which could be used by the FPGA project.

The signalling rate for a single InfiniBand connection is 2.5 GBs in each direction and this can be extended further since InfiniBand supports double (DDR) and quad data (QDR) speeds, for 5 GBs or 10 GBs respectively, at the same data clock rate. However, the standard implements 8B/10B encoding which means that for every 10 bits sent, only 8 bits carry data, so the amount of *useful* data and hence the practical transmission rate is reduced per clock cycle. So, the practical data rates become 2, 4 and 8 GBs for single, double and quad data rates respectively. To increase available bandwidth, links can be combined in banks of InfiniBand connectors, most often together in sets of 4 or 12, called 4X or 12X. A quad-rate 12X link therefore carries 120 Gbit/s raw, or 96 Gbit/s of useful data. Most systems today use either a 4X 2.5Gb/s (SDR) or 5Gb/s (DDR) connection.[24]

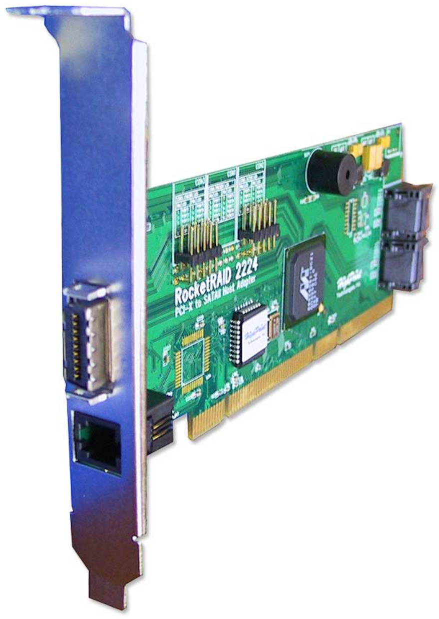

Figure 2.2: External InfiniBand connector (latch type)[1]

InfiniBand provides a massive amount of bandwidth, much more than could be expected from any other available solution, however the costs involved and the complexity of implementation would not offset this benefit. Furthermore, for a simple desktop connection it's unlikely that an FPGA would be able to process the massive amount of data at the rates required. Unless several banks of FPGAs processed the data in parallel, the advantages would never be seen. There is also no established standard programming interface for InfiniBand, as the standard only specifies a set of functions that must exist in any implementation, rather than the syntax of these functions which is left to the vendors. The most common to date has been the syntax developed by the *OpenFabrics Alliance*,[25] which has been adopted by most of the InfiniBand vendors and is available for both Linux and Windows (although at the time of writing, not OS X). Also, despite being less costly than some other ultra-high-speed interfaces, InfiniBand is still very expensive and so ultimately, the bandwidth advantages did not sufficiently outweigh the costs, practical implementation problems and lack of platform compatibility that would be useful for the system.

Table 2.1: Comparison of various communication options

	Bandwidth	Cable Limit
USB (2.0)	480 Mbit/s	5 metres
PCI-Express (DVI medium)	8 GB/s (Dual Link DVI)	4.5 metres (theoretical)
HyperTransport	41.6 GB/s	N/A
InfiniBand	96 GB/s (Quad rate, 12X)	8 metres
Ethernet (1000BASE-T)	10 GB/s	100 metres per segment

2.2.5 Ethernet

Ethernet became the most appropriate solution at this point, since Ethernet is a very common networking platform and the specifications are easy to source. The RJ45 connectors are also very common, cheap and easy to implement, particularly when using integrated magnetics to avoid adding extra passive components. Gigabit Ethernet provides very high speeds with a maximum data rate of 1 GBit/s, plus it's a very ordered method of packet generation that a state machine written in VHDL would be able to parse without much fuss. A useful feature of Ethernet is that the RJ45 connectors are highly ubiquitous with an established standard which is unlikely to be changed anytime soon. This is important in "future proofing" the core to allow for newer forms of Ethernet to connect to the core, particularly since 10 GBit/s Ethernet has started to increase in uptake among developers of networking hardware.[26] The core can also be reused when 10 G Ethernet becomes widely implemented as the interface used in twisted-pair forms of Ethernet in identical between 1 Gb and 10 Gb. The structure of Ethernet and the method for how it is used is documented in the next chapter.

Table 2.1 shows the bandwidth capabilities and maximum cable limits of all the covered communication options.

Chapter 3

Protocols

Communication via Ethernet requires the use of an appropriate *protocol* to facilitate the transfer of data. The choice of protocol is very important, as certain protocols are more suitable than others for specific tasks. Some protocols provide extra error correction capabilities which allow for high reliability data transfer, but at the cost of greater complexity in the packet construction/deconstruction. Most protocols have a particular use for them; some are used for testing the capabilities of the data link, some are just for general data transfer, some are used as utility protocols to facilitate communication in others ways. There's quite a lot of flexibility in how to implement communication between two devices and since implementation using FPGAs requires careful use of look-up tables to minimise wasted space and ensure speed requirements for Gigabit speeds, it is vital that only the necessary protocols are implemented, so choosing the appropriate protocols to use is an important part of any implementation. This chapter provides an overview of the common protocols used in the Internet protocol suite and their relevance to the project.

Table 3.1: Structure of a Gigabit Ethernet packet

7 Bytes	**1 Byte**	**6 Bytes**	**6 Bytes**	**2 Bytes**	**0-9000 Bytes**	**0-46 Bytes**	**4 Bytes**
Preamble	Start Frame Delimiter	Destination Address	Source Address	Length	Data	Pad	Frame Check Sequence

3.1 Ethernet

All the protocols used by the system are encapsulated within an Ethernet packet governed by its own protocol known as the Ethernet protocol. This protocol is important since it holds information that is used to verify whether the packet was transmitted without corrupted or not, as well as where the packet is suppose to be transmitted to in a network. Table 3.1 presents the structure of the Gigabit Ethernet packet. The only difference between the structure of a typical Ethernet packet and a Gigabit Ethernet packet is the size of the Data field - a non-gigabit packet has a maximum size of 1500 bytes, however Gigabit Ethernet can allow for jumbo packets with a maximum size of 9000 bytes. Jumbo packets are covered in Section 7.1.1.

The various fields of the protocol are:

Preamble

A sequence of values used to allow the receiver's clock to be synchronised with the sender. For the purpose of the system however, since the synchronisation is controlled using the physical layer and a Digital Clock Manger on the FPGA, the preamble serves the function of alerting the system to the existence of a new packet. All seven bytes are of the form "10101010".

Start Frame Delimiter

Designates the beginning of the frame and is always “10101011”.

Destination Address

The MAC address of the destination device for the packet.

Source Address

The MAC address of the source device for the packet.

Length

The length of the data field.

Data

The information sent by the packet. For all packets processed by the core, the data field contains the IP protocol, along with an ICMP, UDP or UDP Lite protocol, and then the payload data.

Pad

The Ethernet frame requires a minimum size of 64 bytes. Hence, the data field must be at least 46 bytes long to compensate. If the data field is not large enough, the Pad field can be filled with zeros to bring the size up to the minimum required length. No padding is required for the IP core since the combination of protocols used makes the overall packet large enough to satisfy the minimum length.

Frame Check Sequence

A CRC used to verify the integrity of the frame.

3.2 IP

The Internet Protocol (IP) (RFC 791)[27] is the primary protocol used to send data through a network interface. On its own, IP cannot perform much in the way of

Table 3.2: Structure of the IPv4 header

	Bits 0 - 3	**4 - 7**	**8 - 15**	**16 - 18**	**19 - 31**
0	Version	IHL	TOS/DSCP/ECN	Total Length	
32	Identification			Flags	Fragment Offset
64	Time to Live		Protocol	IP Header Checksum	
96	Source Address				
128	Destination Address				
160+	Data				

data transfer as it is more of a organisation protocol, controlling how devices in a network accept the packets which are sent and received. To actually perform useful communication, the protocol is used in conjunction with another network protocol which a specific task in mind. In this respect, IP forms the backbone of much of the communication used in Ethernet. The IP header is of importance in understanding the nature of IP, as it is the part of the protocol that is added to the Ethernet protocol of each packet before any of the other protocols are seen. Table 3.2 presents the structure of the IPv4 header. (Note: there is an optional *options* field which exists at location 160 and run for 32 bits. Options is rarely used by systems and so is not included in the table.)

The Internet Protocol supports a feature called *packet fragmentation* which allows for packets to be broken up and sent in smaller pieces if the MTU (see Section 7.1.1) is too small to hold the entire packet. The core does not support packet fragmentation and as such is not capable of reassembly of fragmented packets. This is not critical because the MTU can be modified to be as large as required by the software run on the computer to interface with the core, so that all packets can be sent without fragmentation. In particular, the core is designed to support *jumbo packets* with a very large MTU (up to 9000 bytes), so fragmentation can be offset by increasing the MTU to the maximum allowed by the network adaptor and ensuring the packets are jumbo sized, but still limited to a size below the MTU to prevent fragmentation. Features related to packet fragmentation such as dealing with out-of-order packets and dropped packets are handled by the tags system as described in 4.4.1.

The various fields of the IPv4 header are:

Version

The version of the Internet Protocol. For the purposes of the core this will always be set to 4 (see 3.2.1)

IHL (Internet Header Length)

The length of the IP packet header. For all intents and purposes this value is the same between packets. There are occasions where it might be useful to drop particular fields from the header which would result in a shorter header length, but this raises extra complexity for very little gain, so the value here remains constant between packets.

Type of Service or DSCP

Used by network hardware such as routers to determine how packets should be transported and queued. Set to 0 by default.

Total Length

The length of the header *and* the data. Encompasses the whole packet.

Identification, Flags and Fragment Offset

These are used during the process of packet fragmentation. The Identification is a 16-bit number which can be used to identify the packet for reassembly, Flags is used to set whether routers can fragment a packet or not and whether there are more fragments for the particular datagram, and Fragment Offset is set by the router performing the fragmentation to assist in recovery of a fragmented packet. Since the core does not use packet fragmentation, these values are simply copied from incoming packets sent by the computer in all return packets.

Time to live (TTL)

The number of routing hops the packet may endure. Each time the packet passes through a router/switch, this value is reduced by one. If it ever reaches zero, the packet is discarded. When the board is connected directly to the computer as is a common situation, the TTL is considered almost irrelevant due to the direct link and lack of routing.

Protocol

The protocol value is an identifier which specifies what kind of protocol is contained within the packet. For ICMP, UDP and UDP Lite protocols, the value here would be 1, 11 or 88 respectively.

IP Header Checksum

Used for error checking.

Source Address

The IP address of the sender of the packet

Destination Address

The IP address of the destination of the packet

Data

Contains the network protocol and payload which is being encapsulated by IP.

3.2.1 IPv4 vs IPv6

There are two versions of the Internet Protocol in use: IPv4 (Internet Protocol version 4) and IPv6 (Internet Protocol version 6). Version 4 is supported by all networking hardware and software, with version 6 slowly being taken up by various groups such as ISPs as the demand for the newer protocol becomes apparent. However, for the

Table 3.3: Structure of the IPv6 header

	Bits 0 - 3	**4 - 7**	**8 - 11**	**12 - 15**	**16-23**	**24 - 31**
0	Version	Traffic Class		Flow Label		
32	Payload Length				Next Header	Hop Limit
64	Source Address					
192	Destination Address					
320+	Data					

purposes of this design, the only currently implementation version of the protocol is IPv4 due to the commonality and also the simplicity of implementation. If required however, the core can be enhanced by adding support for IPv6. The changes would involve modifying the first FSM to check the version field of the incoming packet and its value will determine whether to process the packet as a IPv4 or IPv6 packet, and further FSMs can be modified to process IPv6 packets and create new ones in the same manner as other supported protocols. Table 3.3 presents the structure of the IPv6 header.

The various fields of the IPv6 header are:

Version

The version of the Internet Protocol. Value of 6 in IPv6.

Traffic Class

Classifies the packet's priority when there is network congestion.

Flow Label

Quality of Service management. Currently unused.

Payload Length

The size of the payload in bytes. If this is set to zero, the packet is identified as having a *jumbo payload* with the packet being known as a *jumbogram*.

Next Header

Specifies the next encapsulated protocol. This field uses the same values as those specified specified in the IPv4 protocol field.

Hop Limit

Replaces the *time to live* field used by IPv4.

Source Address

The IP address of the sender of the packet

Destination Address

The IP address of the destination of the packet

Data

Contains the network protocol and payload which is being encapsulated by IP.

3.3 ICMP

The Internet Control Message Protocol (ICMP) (RFC 792)[28] is a protocol used to determine the operating status of a network. Used predominately by the *ping* computer program, ICMP packets are sent between machines and can be used to quantify aspects of the quality of the network connection such as latency, robustness, stability and whether a host is reachable. The segment structure of an ICMP packet is presented in Table 3.4 and shows the header information only - actual payload data follows after the header.

Table 3.4: Structure of an ICMP packet

	Bits 0 - 3	**4 - 7**	**8 - 15**	**16 - 18**	**19 - 31**
0	Source Address				
32	Destination Address				
64	Type		Code	Checksum	
96	ID			Sequence	

The various fields of an ICMP packet's header are:

Source Address

The source IP address from whom the packet was sent.

Destination Address

The destination IP address of the packet.

Type

The ICMP type. Despite there being a fairly large number of available types, only two are used by the ping command and so are supported by the system: 8 (Echo Request, often referred to as just *ping*) and 0 (Echo Reply, also known as a *pong*). When a ping packet is sent by the computer to the board, this is a request for the host to respond so the ICMP packet has a type of 8. When the board replies (assuming the connection is alive), it is replying to the ping which was sent to it, so the responding packet has a type of 0.

Code

Further specification of the ICMP type. Varies depending on the ICMP type.

Checksum

This field contains error checking data calculated from the ICMP header and data, with value 0 for this field.

ID

This field contains an ID value which should be returned in case of a pong.

Sequence

This field contains a sequence value which should be returned in case of a pong. Following the sequence, the ICMP payload data is present.

When testing the link between a computer and the FPGA board, a typical use of ICMP is to ping the board to ensure the link is live. Valid replies from the board indicates all aspects of the connection are satisfactory, which includes a correct ARP entry, correct programming of the board and valid interface code on the FPGAs. A more interesting test can involve a *flood ping*, which is when the ping tool is instructed to ping non-stop at the target host the moment it receives a reply, instead of waiting for a pre-determined amount of time before sending another ping. By flooding the host, in this case the FPGA board, a test of stability/reliability can be achieved. A flood ping can be achieved in UNIX-like operating systems with the command (must be run as a super-user): `ping -f <IP_ADDRESS>`

The protocol is not used for general-purpose data transfer with the exception of the ping tool - its purpose by design is solely to provide information about a network. For this reason it was vital that it be included in the design, particularly for debugging.

3.4 ARP

The Address Resolution Protocol (ARP) (RFC 826)[29] is a utility protocol used to translate IP addresses to Ethernet MAC addresses. When a program makes a request to send data to a particular machine, it specifies an IP address as the destination, however the destination network adapter providing the appropriate link does not have

Table 3.5: Structure of an ARP packet

	Bits 0 - 7	**8 - 15**	**16 - 31**
0	Hardware type (HTYPE)		Protocol type (PTYPE)
32	Hardware length (HLEN)	Protocol length (PLEN)	Operator (OPER)
64	Sender hardware address (SHA)		
*	Sender protocol address (SPA)		
*	Target hardware address (THA)		
*****	Target protocol address (TPA)		

a native IP address, but rather a MAC address (Media Access Control). Typically, when a request for a particular IP address is made, assuming there are no previous records of the nature of the machines on the link the computer will send an ARP packet requesting a MAC address resolution for the IP address. Any machine on the link which matches the IP address will send a return ARP packet specifying what their MAC address is, so the computer can correctly send the initial request for data transfer to the right machine.

The packet structure of an ARP packet is presented in Table 3.5 (minus the IP header which was covered in Section 3.2).

(Note that the lengths of the SHA, SPA, THA, & TPA fields are determined by the hardware & protocol length fields, so they cannot be explicitly mentioned in the table as they may vary.)

As an example, let's say a host with an IP address of 192.168.0.1 and MAC address of 00:01:02:03:04:05 wants to send a packet to another host at 192.168.0.50, but does not know the MAC address. To determine the MAC address, an ARP request is sent to discover the address. If the host at 192.168.0.50 is running and available then it would receive the ARP request and send the appropriate reply containing its MAC address (which in this example will be 00:40:12:E8:44:A1). This example assumes either a scratch or previously flushed ARP table. The header fields are covered as follows, with the above example being used in the descriptions:

Hardware type (HTYPE)

Each data link layer protocol is assigned a number used in this field. The system uses Ethernet, which has a value of **1**.

Protocol type (PTYPE)

Each protocol is assigned a number used in this field. The system uses IPv4, which has a value of **0x0800**.

Hardware length (HLEN)

Length in bytes of a hardware address. Ethernet addresses are **6** bytes long.

Protocol length (PLEN)

Length in bytes of a logical address. IPv4 addresses are **4** bytes long.

Operation

Specifies the operation the sender is performing: **1** for a request and **2** for a reply. Hence, the first ARP packet would have a value of **1** and the responding packet would have a value of **2**.

Sender hardware address (SHA)

Hardware address of the sender

- For the ARP request - **0x000102030405** (this the MAC address 00:01:02:03:04:05, represented in HEX and encoded 6 bytes long, as per the hardware length)
- For the ARP reply - **0x004012E844A1** (this the MAC address 00:40:12:E8:44:A1, represented in HEX and encoded 6 bytes long, as per the

hardware length. This is the MAC address from the unknown host, which is what the system uses to determine the routing.)

Sender protocol address (SPA)

Protocol address of the sender

- For the ARP request - **0xC0A80001** (this is the IP address 192.168.0.1, represented in HEX and encoded 4 bytes long, as per the protocol length)
- For the ARP reply - **0xC0A80032** (this is the IP address 192.168.0.50, represented in HEX and encoded 4 bytes long, as per the protocol length)

Target hardware address (THA)

Hardware address of the intended receiver

- For the ARP request - **0x000000000000** (Since the purpose of the ARP request is to find out this value, ARP request packets have this field set to zero)
- For the ARP reply - **0x000102030405** (this the MAC address 00:01:02:03:04:05, represented in HEX and encoded 6 bytes long, as per the hardware length)

Target protocol address (TPA)

Protocol address of the intended receiver

- For the ARP request - **0xC0A80032** (this is the IP address 192.168.0.50, represented in HEX and encoded 4 bytes long, as per the protocol length)
- For the ARP reply - **0xC0A80001** (this is the IP address 192.168.0.1, represented in HEX and encoded 4 bytes long, as per the protocol length)

ARP is not particularly difficult to implement, but since FPGA logic needed to satisfy the requirements of both running as fast as possible and retaining the smallest profile so as to allow other programs to access the space on the FPGAs, the decision was made to avoid implementing ARP on the FPGAs by utilising a bypass on the computer. As the system's primary setting will be in a direct connection between the board and a single computer, manually adding an ARP entry on the computer side is easy to do and satisfies the requirements of the system without requiring extra logic. First however, it is necessary to explain how the bypass works.

3.4.1 Manual ARP Entries

Operating systems keep track of the various MAC addresses and their corresponding IP addresses through the use of an *address translation table*. These tables are used by the operating system to work out where a packet should go, since packets only specify the destination in the form of an IP address and not as a MAC address. When a new IP address is used for the first time, the OS does not have any knowledge about the relevant MAC address it refers to, so the OS sends an ARP request to all machines on the IP's subnet and waits for a reply from the appropriate machine. Once it receives the ARP reply, the translation table is updated to make note of the IP-to-MAC relationship and so all future uses of that particular IP will been seen in the table and automatically use the relevant MAC address without requiring a second use of ARP. If an ARP request is made and no machine responds with a relevant ARP reply within the timeout period, the requested IP is not written into the table as there are no available hosts.[30]

This system does not currently support ARP, which means that when connected directly to a computer, the computer will know a link has been established (because the physical components on the board achieve this independently of the VHDL code), but the computer will not be able to understand what's at the other end because ARP requests will go unnoticed. This means that any attempts to ping the board or send UDP transmissions will fail because every attempt will automatically be preceded by an ARP request. The operating system will only stop sending ARP requests and allow regular network traffic once it has an entry for the board in the address translation table and since the board does not achieve this on its own, we must perform the addition of the entry ourselves.

Adding an entry to the ARP table is performed using the *arp* command available on most operating systems.[31] For example, in both Windows and Linux an ARP entry can be manually set using the following command: `arp -s <IP_ADDRESS> <MAC_ADDRESS>`

The IP_ADDRESS (which can also be a hostname) is the address we want the board to be known as and the MAC address is the physical address of the board. Unlike most other networked devices, the board does not have a hardwired MAC address to begin with, so the MAC address specified can be virtually anything, so long as it doesn't match any other network devices already connected to the computer. The other requirement is to pick an IP address that exists on the subnet of the network adapter's IP address, the adaptor which connects to the board. For example, if the network adaptor has an IP address of 192.168.0.1 and the subnet mask is 255.255.255.0, then an IP address of 192.168.0.50 is valid for the board since it exists within the allocated subnet. After executing the arp command with valid arguments, the board is then accessible using the chosen IP and can be validated with using the standard ping command. A full example of how to use the arp command along with the other necessary steps involved to connect the board to a computer is covered in Section 7.2.

Setting the ARP for the FPGA board manually allows access to the board with a specified IP without having to implement ARP construction/deconstruction code in the FPGA. This increases available space for other programs on the FPGA and also makes the routing easier for the interface, ensuring an easier chance of matching full gigabit speeds. The main disadvantage is that in more complicated networking situations, if the board is not directly connected to the end of the networking jack of a computer but somewhere down a chain of networked devices, such as on a router/switch, the board will not be understood by the devices it's directly connected to unless that device itself has an ARP entry added for the board. This may or may not even be possible, particularly if connected to a router/switch that doesn't support manual addition of ARP entries. The implementation of ARP construction/deconstruction on the FPGA can be considered a future improvement to the design, but can still be worked around provided the networking requirements are simple.

3.5 TCP

The Transmission Control Protocol (TCP) (RFC 793)[32] is a data transfer protocol, considered one of the core protocols of the Internet protocol suite. It provides reliable delivery of packets and acts as the major backbone of most Internet communications, such as with file transfer, email and web browsing. The segment structure of a TCP packet is shown in Table 3.6 (contains header and data information - IP header is not included in the diagram as it was covered in Section 3.2):

The fields are summarised as follows:

Source port

Identifies the sending port

Destination port

Identifies the receiving port

Table 3.6: Structure of a TCP packet

	Bits 0 - 3	**4 - 7**	**8 - 15**	**16 - 31**
0	Source port			Destination port
32	Sequence number			
64	Acknowledgment number			
96	Data offset	Reserved	Flags	Windows
128	Checksum		Urgent pointer	
160	Options (optional)			
160/192+	– Data –			

Sequence number

The sequence number designates the position of the current packet's first data byte in the data stream. If the SYN flag is present, the sequence number is the position plus one, to allow for the SYN flag in the payload. For example, if the first data byte of the current packet is the 100th byte in the data stream so far, the sequence number will be 100 if the SYN flag is absent, or 101 if the SYN flag is present.

Acknowledgment number

If the ACK flag is present, then the value of this field is the sequence number that the sender of the acknowledgment expects next.

Data offset

Specifies the size of the TCP header in 32-bit words. The minimum size header is 5 words and the maximum is 15 words thus giving the minimum size of 20 bytes and maximum of 60 bytes. This field gets its name from the fact that it is also the offset from the start of the TCP packet to the data.

Reserved

For future use and should be set to zero

Flags (aka Control bits)

Contains 8 bit flags

- CWR – Congestion Window Reduced (CWR) flag is set by the sending host to indicate that it received a TCP segment with the ECE flag set
- ECE (ECN-Echo) – indicate that the TCP peer is ECN capable during 3-way handshake
- URG – indicates that the URGent pointer field is significant
- ACK – indicates that the ACKnowledgment field is significant
- PSH – Push function
- RST – Reset the connection
- SYN – Synchronise sequence numbers
- FIN – No more data from sender

Window

The number of data bytes which the sender of this segment is willing to accept, beginning with the byte indicated in the acknowledgment field.

Checksum

The 16-bit checksum field is used for error-checking of the header and data

Urgent pointer

If the URG flag is set, then this 16-bit field is an offset from the sequence number indicating the last urgent data byte

Options

The total length of the option field must be a multiple of a 32-bit word and the data offset field adjusted appropriately

Despite TCP being a significant component of the Internet Protocol, it is not implemented at all in this design. There are several reasons for this:

1. TCP is a complex protocol. This complexity translates into extra logic required on the FPGA, which limits the potential speeds and room for extra programs on the FPGA. Justification for such implementation would only work if there were no realistic alternatives.

2. UDP (covered below) is a simpler protocol which has similar properties to TCP but without much of the reliability architecture. However, reliability can be effectively added to UDP without much overhead, so this negates the issue of using UDP in preference to TCP.

3. TCP requires extra overhead to cover the error-correction/reliability it provides. As can be seen in the header diagram, a lot of the header fields exist just for this protection, which makes it less efficient than the UDP standard which uses far less header per packet. Since the intention of the design is to supply a high-speed, high-bandwidth device, removal of overheads such as this are desirable.

4. The extra reliability of the packet transmission provided by TCP is just not necessary for this design. For the purposes of this core there was no intention to have it embedded within devices connected to the Internet; it is mostly intended for local networks and/or direct connection to a computer. Ultimately the extra effort to implement TCP would not be justified compared to that of UDP.

TCP was originally designed for unreliable low-speed networks. However, as network speeds have increased the hardware requirements for the network adaptors to support TCP have had to increase as well. If an embedded CPU on an FPGA intended to process TCP packets, Gigabit networking may result in issues with the CPU keeping up with the line speed. As a general rule of thumb, 1 MHz of CPU is required for every 1 Mbits of TCP traffic, so if the CPU was solely responsible for processing

TCP packets and their payloads, a Gigabit link would require at least a 1 GHz CPU to keep up with the line speed.[33] Embedded processors on FPGAs generally aren't able to run at such speeds; for example, the popular MicroBlaze soft processor core for Xilinx FPGAs has a maximum clock frequency of 115 MHZ on a Spartan 3, and even on a Vertex 5 the maximum possible speed is only 235 MHz.[34] This has been the reason for development in TCP offload engines which transfer processing of the TCP stack to the network controller logic instead of solely on a CPU. In many commercial applications however, TCP is rarely coded entirely in hardware; for the situations where a device requires the use of TCP, a common strategy is to provide a combination solution, by way of an embedded CPU to run the state machines in software using code and perform the checksum calculations in hardware using logic.[35] This provides a more efficient and balanced implementation than by using just logic. An example of such an implementation is used by Treck Inc. and their embedded Internet protocol stacks.[6]

3.6 UDP

The User Datagram Protocol (UDP) (RFC 768)[36] provides the bulk of the data transfer capability for the project. It is connectionless, which means no handshaking is required between the two devices. This along with several other features means the protocol is simple to implement in hardware, but does not guarantee reliability or ordering in the way TCP does. The segment structure of a UDP packet is shown in Table 3.7; note that it contains header and data information - IP header is not included in the diagram as it was covered in Section 3.2.

Table 3.7: Structure of a UDP packet

	Bits 0 - 15	**16 - 31**
0	Source port	Destination port
32	Length	Checksum
64	Data	

The UDP header consists of four fields:

Source port

This field identifies the sending port. When a packet is sent by the computer this port is generally assigned a random number. Any packets sent by the board back to the computer utilise this port number for the packet's destination port.

Destination port

This field identifies the destination port.

Length

A 16-bit field that specifies the length in bytes of the entire datagram: header and data. The minimum length is 8 bytes since that's the length of the header. The field size sets a theoretical limit of 65, 527 bytes for the data carried by a single UDP datagram. The practical limit for the data length which is imposed by the underlying IPv4 protocol is 65, 507 bytes.

Checksum

The 16-bit checksum field is used for error-checking of the header and data. Implementing the UDP Checksum is optional, as the standard allows for the UDP checksum field to be zeroed out to denote no checksum and hence for devices to ignore the checksum entirely. However, it is still fully implemented in the design for two main reasons:

Table 3.8: Structure of the UDP pseudo-header with remaining UDP packet

	Bits 0 - 7	**8 - 15**	**16 - 23**	**24 - 31**
0	*Source address*			
32	*Destination address*			
64	*Zeroes*	*Protocol*	*UDP length*	
96	Source Port		Destination Port	
128	Length		Checksum	
160	Data			

1. Although the zeroed-checksum UDP packets will pass through to the computer unhindered, any programs which utilise the Berkley socket interface for communication will not accept the packets unless they have a filled in (and valid) UDP checksum. As this constitutes the majority of the system's communication strategy, UDP checksums must be implemented fully to allow for sockets to be used.

2. UDP has less inbuilt error-correction capabilities than TCP, so enabling UDP checksums ensures the received packets are uncorrupted and provide at least some form of protection from unintentionally accepting corrupted packets.

An important distinction between the checksums used in ICMP and UDP is that unlike in ICMP, the checksum for UDP does not just cover the UDP header and data but also requires a *pseudo-header* to be attached to the beginning of the UDP header. This pseudo-header is never sent with the packet; it's created internally in the system and used solely for determining the UDP checksum. Note that the pseudo-header is different depending on whether UDP is run over IPv4 or IPv6; since the core only uses IPv4, only one form of pseudo-header will be covered in dissertation. The structure of the pseudo-header is listed in Table 3.8; the pseudo-header component is attached to the beginning of the UDP packet header and shown in *running text.*

One specific difference of UDP over TCP is that it has very little in the way of extra error-correction, which means that any system using UDP needs to implement its own error-correction method or none at all, depending on the nature of the application. However, since this project has potential applications in video encoding/decoding and many other areas where the occasional lost packet is irrelevant, the choice of UDP is a benefit since it has less overhead than TCP. In the event of a need for extra reliability, there are simple ways to add error-correction such as adding a small tag to the beginning of each data payload and verifying the order as each packet is transmitted/received. Implementing UDP in hardware is very popular due to its simplicity and it has had many other commercial successes which makes the decision to use it here clear.[37]

3.7 UDP Lite

UDP Lite (RFC 3828)[38] is a special variant of UDP. Both protocols share the majority of header fields, sizes and features, however UDP Lite has an advantage in flexibility by allowing modification to how the UDP checksum operates. The structure of the UDL Lite packet is shown in Table 3.9.

Usually, UDP datagrams are always discarded when their UDP checksum does not pass. This means that the corruption of just one byte leads to the whole packet being dropped. If the packet is large, this can have a major impact on the efficiency of the system, particularly if the occasional corrupted packet is irrelevant (eg. such as in multimedia protocols, video/audio, VoIP, etc). UDP Lite makes it possible to limit the scope of the checksum so that it only needs to cover what the system wants it to cover, rather than the entire packet. Limiting the scope of the checksum also helps simply the logic necessary to determine the checksum.

Table 3.9: Structure of a UDP Lite packet

	Bits 0 - 15	**16 - 31**
0	Source port	Destination port
32	Checksum Coverage	Checksum
64	Data	

The only difference between UDP and UDP Lite as far as the headers are concerned is that instead of a UDP length field, UDP Lite has in its place a "Checksum Coverage" field, the coverage measured in bytes. The smallest value which can be specified for this field is 8 bytes, covering just the header and no data.

UDP Lite was originally supported only for the Linux platform, however since then Windows XP/Vista and Mac OS X have incorporated support for the protocol. A major advantage of using UDP Lite is that due to the flexibility of the checksum used in the protocol, the implementation is simpler in hardware and hence more efficient than regular UDP. Further explanation of the protocol and its significance to the project is covered in Section 4.5

Chapter 4

Implementation

The primary aspect of this chapter is to cover the implementation of the IP/network layer, with particular focus on the coding structure and hardware components which form the design.

4.1 Functionality

The networking code provides the following capabilities:

- It can accept and understand ICMP, UDP and UDP Lite packets,
- It will cleanly reject all packets which do not match the protocols listed above,
- It can send back mirror versions of UDP packets for testing purposes,
- It can handle several packets being processed at once due to the pipelined nature of the design,
- It is robust and very efficient

4.2 Data Flow

Implementation of the IP/network layer is separated into three distinct Finite State Machines (FSMs). FSMs are blocks of code which step through a specific series of known states, each state performing one or more tasks. Hence, each collection of states forms an FSM and an FSM is generally designed for a specific task. An FSM can be thought of as a module, consisting of individual task handlers which internally process the inputs and outputs given. By writing the code in such as way as to separate the important parts of the design to individual FSMs, the system can be made much easier to write, execute, debug and understand.

The core uses three FSMs. *fsm_read* takes a packet sent by the computer and received by the board, processes it by placing the header of the packet into RAM and the payload (application data) into a FIFO, and then depending on the packet and mode the system is in (see below), may trigger the next FSM or simply wait for a new packet to be sent. *fsm_packgen* is used to prepare packets for transmission, and does so by being triggered from either *fsm_read* or *toplevel*. It primarily performs copy operations between various RAMs and FIFOs as well as calculates checksum and CRC values necessary for the packet to be accepted by the computer. *fsm_send* takes the packet constructed from *fsm_packgen* and transmits it to the computer, ordering fields as necessary.

The flow of data between the FSMs and the other layers of the design is represented in Figure 4.1. Data begins at the PHY (Physical Layer) which is connected to the *toplevel* component, the external interface of the IP layer and the component which programs wanting to utilise the core connect to. Depending on the settings in the

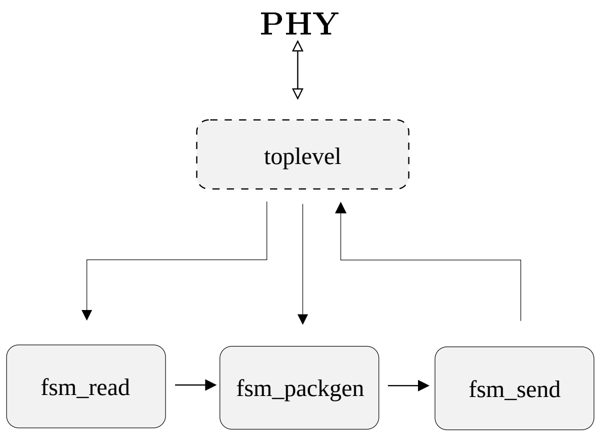

Figure 4.1: Flow of data between the various FSMs

---

toplevel, as well as the protocol of the incoming packet, there are three potential paths for the data to take. From the *toplevel*:

1. [Mirror] - The payload data sent to the PHY is identical to the packet received. Data progresses from *fsm_read... fsm_packgen... fsm_send... toplevel* and streams the output back to the PHY. This path is guaranteed with ICMP packets since they are used by the ping tool and require a response, hence the full path from reading the packet to generating a response. The path can also occur with UDP packets if toplevel is set to mirror all incoming UDP packets (mainly for debugging purposes).

2. [Typical] - The payload data is entered into the fsm_packgen component by the external FPGA program and therefore the payload content is constructed as necessary. Data progresses from *fsm_packgen... fsm_send... toplevel* and streams the output back to the PHY. In this path the fsm_read component is bypassed and in fact the PHY is not used to trigger the FSMs at all, but rather the external FPGA program triggers the FSMs. This path is used when there is a need to create a packet from scratch instead of responding to an incoming

packet and is only ever used with UDP or UDP Lite. This is the typical use of the core for the majority of data transfers.

3. [Storage] - The latter two FSMs are totally bypassed and no return packet is sent. Data is read using *fsm_read* and progresses no further. This path is used for storing payload data for extraction by another FPGA program.

4.3 Finite State Machines

The design uses three FSMs when processing packets - one to read an incoming packet (fsm_read), one to build a packet for transmission (fsm_packgen) and one to actually perform the packet transmission (fsm_send). These three FSMs are documented in Sections 4.3.3, 4.3.4 and 4.3.5. To completely understand how the layer is implemented, components such as the RAMs and FIFOs will now be covered along with an explanation of the state machines used in the design.

4.3.1 RAM

RAM (Random Access Memory) is a storage component used to hold data for a finite amount of time (ie. as long as power is applied). There are two blocks of FPGA-based RAM used in the design, one of them embedded in fsm_read (which shall be referred to as RAM1) and the other in fsm_packgen (RAM2). When an incoming packet is received, the header is stored in RAM1 by fsm_read. When a packet is being generated for transmission, the header is stored in RAM2 by fsm_packgen and read from RAM2 during the operation of fsm_send. The location of the RAMs and interconnections between them and the other FSMs is shown in Figure 4.2. The RAMs have a linked data path as shown by the arrow between them, as RAM2 will copy fields from RAM1 for any returned packet regardless of protocol.

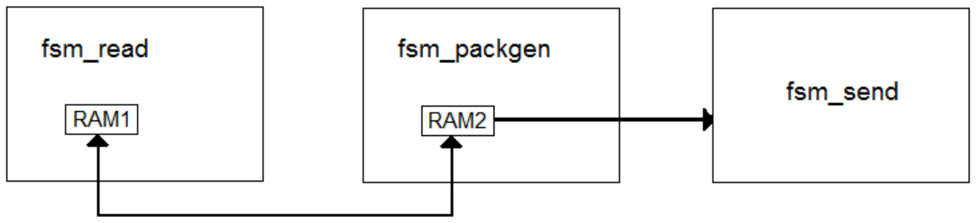

Figure 4.2: RAMs used by the interface along with their locations and data paths

RAM is *random access* in the sense that its elements can be accessed non-sequentially. By enabling the read and/or write control signals on a RAM and setting the appropriate address location, any field can be accessed as required and in any location, which provides great flexibility and is essential when recombining header fields for transmission in fsm_send. The header for the sent packet uses many of the fields from the previously received packet, and due to the copy operations that occur in fsm_packgen, fields such as source and destination IP addresses need to be swapped during sending. RAMs make this easy because they allow for reading and writing with addresses which are out of order.

4.3.2 FIFOs

A FIFO (First-In, First-Out) is a block of memory which provides a structured way of adding and extracting data to and from it. Like RAM, FIFOs are used for data storage, but unlike RAM they do not allow for easy access of data elements in different areas of the FIFO. As the name suggests, a FIFO's method of operation is that the most recently written element is the first to be retrieved in a read call. FIFOs are popular containers for data storage because of the simplicity in reading and writing values that are in a sequence or stream. Given the usefulness of FIFOs there is continuous research into the most efficient ways to synthesise these components.[39]

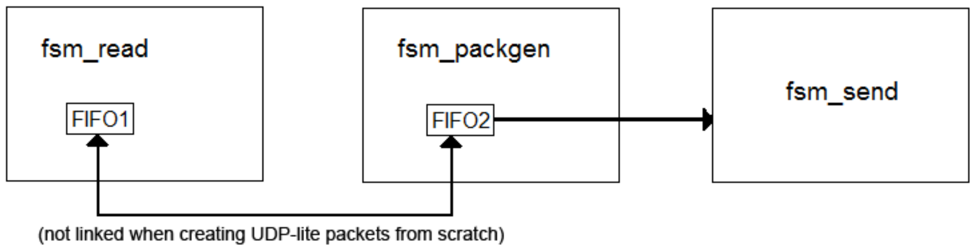

Figure 4.3: FIFOs used by the interface along with their locations and data paths

---

Much like the RAMs, there are two FIFOs used in the design, one of them embedded in fsm_read (which shall be referred to as FIFO1) and the other in fsm_packgen (FIFO2). When an incoming packet is received, the payload data is stored in FIFO1 by fsm_read. When a packet is being generated for transmission, the payload data is stored in FIFO2 by fsm_packgen and read from FIFO2 during the operation of fsm_send. The location of the FIFOs and interconnections between them and the other FSMs is represented in Figure 4.3. Note that the arrow between the FIFOs represents the flow of data when the system is running in mirror mode (bouncing UDP packets) or returning ICMP packets; when creating UDP Lite packets from scratch, there is no need for FIFO1 to access FIFO2.

The FIFOs are used in this design for storing the payload data of a packet. The payload is a contiguous memory block, one which does not require access to various parts of it while inside the MAC layer. Therefore the act of reading and writing with a FIFO is made easy due to the contiguous nature of the data.

4.3.3 fsm_read

The purpose of fsm_read is to parse and process an incoming packet. The direction of the data flow, for the purpose of clarifying the context of an incoming packet, is from the *server* to the *core*. As we are dealing with only a small selection of protocols and virtually anything could be transmitted down the wire, the FSM needs to be able to correctly interpret the packet, deal with it appropriately and also deal with any packet which does not match the required protocol set.

When the interface begins execution, the fsm_read code will initialise and hold in a waiting state. The FSM waits for the RX_DV signal to go high before proceeding any further. This signal is controlled by the physical layer of the network and is raised when a packet is received by the layer. At this point the FSM will begin checking for the Ethernet preamble, shown below:

```
55 55 55 55 55 55 55 D5
```

Once a packet's preamble is verified, the FSM can begin processing the packet contents. The first stage is to analyse the header information of the packet. All header data is saved to the RAM1 component; this data can then be quickly and easily accessed by the various components of the code. All of the RAMs used in the design are dual-port (simultaneous read and write) Xilinx Block RAMs. At a certain point in the header the IP protocol will be specified. If the protocol number matches any of the implemented protocols the system can handle, the state continues processing as normal. Otherwise, the packet is rejected by changing to a waiting state which will only advance once RX_DV goes low. This bypasses the rest of the packet since it's of a form which does not need to be dealt with.

The FSM does not check for a correct MAC or IP address because it is not necessary. The core is designed to be implemented on a board which is directly connected to a computer. There aren't any routers/switches/hubs which are required to determine where the packets should be sent. For this reason the FSM simply receives packets and passes them on without concerning itself as to whether the packets match the

MAC/IP addresses of the board running the core. This is also the case for the ports used by the packets.

After reading the entirety of the packet header, the system begins to read the payload (application) data, which is where the useful content of the packet is located. The data is saved to the FIFO1 component. After the payload has been processed, a CRC is calculated against the packet and matched to the FCS (Frame Check Sequence) supplied by the packet; if these two are identical, the packet was sent uncorrupted and so the content is considered sane. If the values are different however, a control signal is raised to represent a packet failure.

The states in fsm_read are commented below:

init

This state has the purpose of providing a clean reset to the FSM. It is executed upon power-on of the system and also executed when a packet has completed being processed by fsm_read. It sets important control signals to their default values so that future packets are not corrupted by unknown signal states.

waitforpacket

A holding state which remains dormant until both the RXDV and dcmlock signals go high. When RXDV goes high, the Ethernet layer has received a new packet containing "valid" data which can be processed by the FSM. Note that valid in this case means a packet able to be understood by the Ethernet layer; a corrupted packet which passed the Ethernet layer unhindered would fail checksum calculations later on in the FSM. The dcmlock signal is connected to the DCM (Digital Clock Manager) which controls the clocking of all the components on the core. When this signal goes high, the system is allowed to begin operating as a steady clock cycle has been obtained.

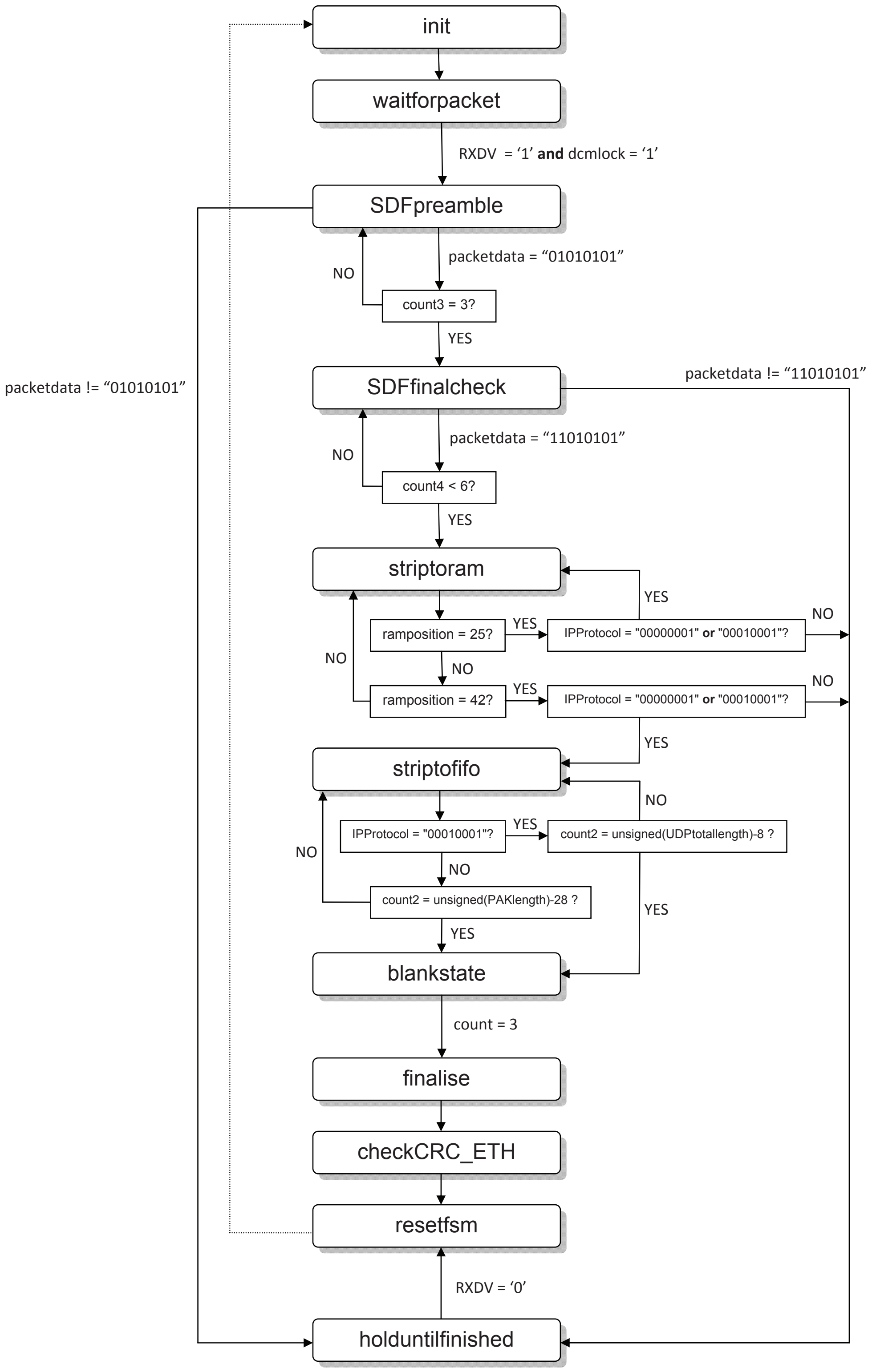

Figure 4.4: fsm_read state flowchart

SDFpreamble

Preceding the actual packet content is the preamble and SDF (Start Frame Delimiter). This is represented by the 55 55 55 55 55 55 55 D5 sequence which exists in every packet, the 55s acting as the preamble and the D5 as the SDF. This state checks for THREE of the 55 bytes in a row. If something else interrupts this sequence or cannot be finished for whatever reason, the packet is considered damaged and control is moved to the **holduntilfinished** state.

SDFfinalcheck

Assuming the previous state ran with success, the system keeps counting through the necessary number of bytes in the packet until it reaches the point where the SDF (D5) should exist. If it does, the system proceeds as normal, otherwise the packet is considered damaged and control is moved to the **holduntilfinished** state.

striptoram

Satisfied the packet is readable enough to pass initial testing, the system uses this state to store the packet's header information (both IP plus the protocol's header) into RAM1. Several fields are also copied from the packet into registers for use later, such as the specific packet protocol, the length of the data, any checksums present depending on the protocol and any protocol-specific fields which are important. Another check is performed in this state as well - if the scanned protocol number does not match one implemented by the system (eg. the packet might be TCP which the network code has no handler), the packet is ignored and control is moved to the **holduntilfinished** state.

striptofifo

Once RAM1 has stored the packet header data, control now moves to the striptofifo state which has the task of storing the payload data into FIFO1.

blankstate

Modifies the Ethernet checksum stream by zeroing out the existing checksum data. This is required when calculating the received Ethernet checksum - the checksum already embedded in the packet must be cleared during calculation.

finalise

A single clock cycle holding state.

checkCRC_ETH

Performs a check of the Ethernet checksum by comparing signals *crc_output_ETH* and *crc_output_ETH_FCS*. If they are identical, this means the calculated Ethernet checksum is identical to the embedded Ethernet checksum and so the packet was received without any corruption. If the checksums are different, the packet is corrupted somehow and cannot be trusted to hold correct data. In this case the FSM resets without triggering fsm_packgen.

resetfsm

This state calls a special register which orders the FSM to execute a reset. Control restarts at init.

holduntilfinished

If the packet failed any of the preamble, SDF or valid protocol tests, control is moved here. This state simply waits until the packet has run through the system, but does not capture any of its data. Once the packet has cleared and RXDV goes low, the system is reset.

4.3.4 fsm_packgen

This FSM is responsible for generating packets to be sent back to the computer. Depending on the nature of the packets to be sent, it will have to copy the contents of RAM1 and FIFO1 from fsm_read to its internal component buffers, RAM2 and FIFO2.

When the interface begins execution, the fsm_packgen code will initialise and hold in a waiting state. The FSM waits for an activation signal from either fsm_read or toplevel (the interface to the external FPGA code) - if it comes from the former, execution requires a mirrored packet to be sent; if it comes from the latter, the FIFO in packgen will already be filled with the required payload so no extra copying needs to be performed. There are two paths of execution the system can take for that matter:

1. ICMP and UDP (mirror mode) - copy RAM1 to RAM2 and FIFO1 to FIFO2
2. UDP (regular mode) and UDP Lite - copy RAM1 to RAM2 but leave FIFO2 untouched since it will have the necessary payload installed from an external source, such as a program on the FPGA.

Once these tasks are completed the FSM progresses to completing the calculation of the CRCs for each packet and injecting them in the relevant fields of the packet. There are two cases where this does not happen - UDP regular or UDP Lite, for reasons which are outlined in Section 4.5. The overall task for the FSM is then completed and while fairly simple it's an important part of the conjoined FSMs used to transmit data back to a computer. The FSM is then reset and waits for another trigger to begin operation.

The states in fsm_packgen are commented below:

init

This state has the purpose of providing a clean reset to the FSM. It is executed upon power-on of the system and also executed when a packet has completed being created

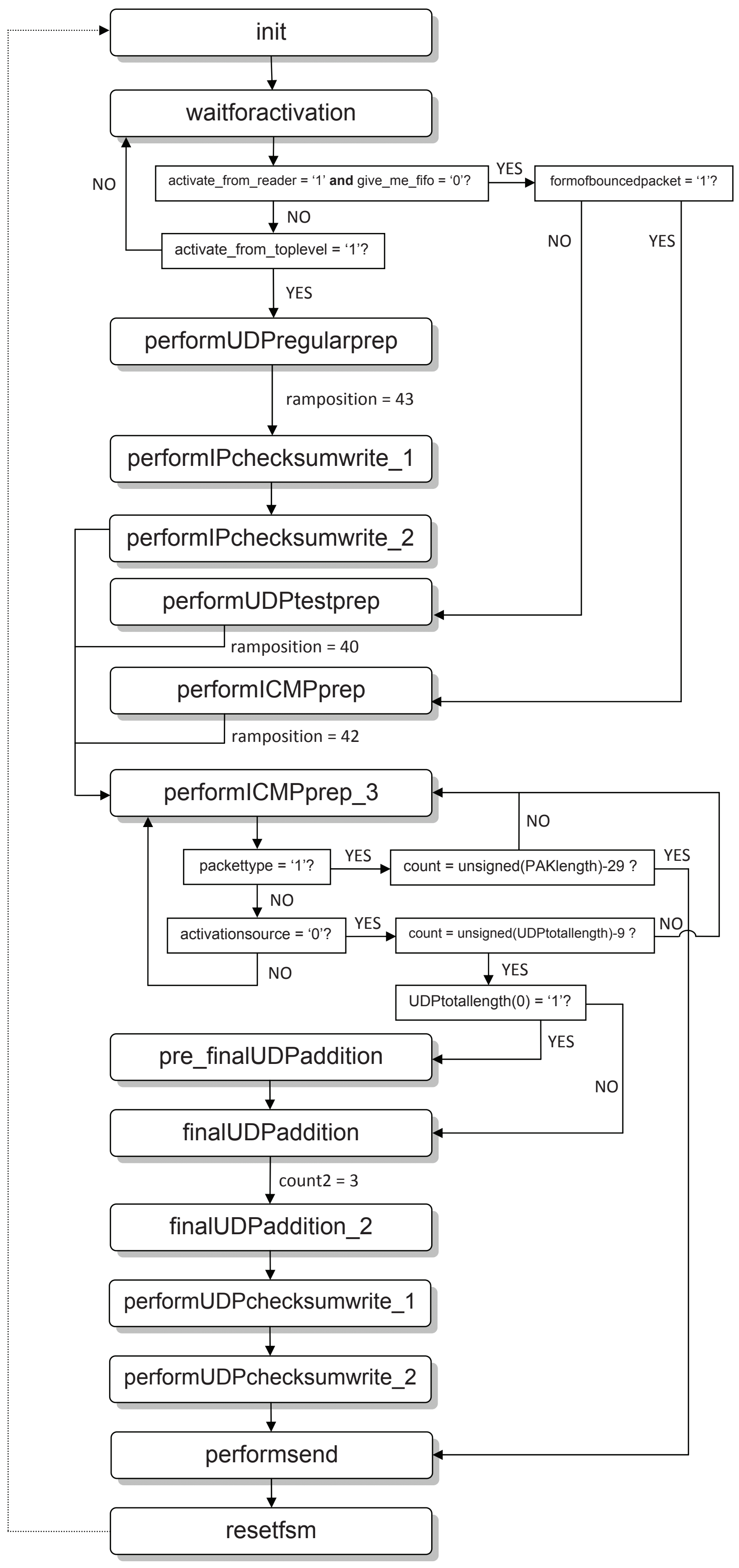

Figure 4.5: fsm_packgen state flowchart

by fsm_packgen. It sets important control signals to their default values so that future packets are not corrupted by unknown signal states.

waitforactivation

This state holds execution until it receives one of two sets of triggers: *activate_from_reader* is high and *give_me_fifo* is low, or *active_from_toplevel* is high. The first trigger set is activated from fsm_read once an ICMP or UDP (when in mirror mode) packet is received and the second trigger is activated manually by another design which is connected to the top level of the IP/network code - presumably, the design which actually needs to use the Gigabit Ethernet interface. *give_me_fifo* is a special control signal that is used to override access of the FIFOs which hold the payload data. If held high this ensures the system cannot accidentally overwrite payload data by an incoming packet, such as a ping for example, as it will stop the FSM from continuing. Depending on which trigger set is activated, the FSM will move to the necessary state required to perform the task.

performUDPregularprep

This state is used to create a UDP/UDP Lite packet from scratch. The assumption is that RAM1 already contains header information from a previous UDP packet sent by the computer, which is necessary to supply the fields with the necessary template information for the outgoing packet. It is also assumed that FIFO2 has been preloaded with the required payload data. With these assumptions in place, the state copies the header from RAM1 to RAM2 and replaces key header fields with the correct values pertaining to the new packet. The IP checksum is also calculated during this process.

performIPchecksumwrite_1 - performIPchecksumwrite_2

The IP header requires a correct IP checksum, which was calculated in the previous state. These two states write the checksum into the relevant areas of RAM2.

performUDPtestprep

This state is used to create a UDP packet which is a mirror copy of the just received packet from fsm_read. The tasks performed in this state are far simpler than those in **performUDPregularprep**, since the majority of the state simply involves copying from RAM1 to RAM2 with no actual substitution of fields or their values.

performICMPprep

This state is used to create an ICMP response packet. It also is fairly simple and involves copying from RAM1 to RAM2.

performICMPprep_3

Despite the name, this state is reached by all of the preparatory states once they have completed their RAM copy procedures. This state performs the FIFO1 to FIFO2 copy procedures for the ICMP and UDP bounced packets.

pre_finalUDPaddition

A waiting state used to synchronise control signals with the rest of the data.

finalUDPaddition - finalUDPaddition2

Sends any remaining values to the UDP checksum calculator

performUDPchecksumwrite_1 - performUDPchecksumwrite_2

Writes the correct UDP checksum value to the relevant area in RAM2.

performsend

Triggers fsm_send with the appropriate control signals so that it knows what kind of packet to transmit. fsm_packgen has done its job by this point and is ready to reset.

resetfsm

This state calls a special register which orders the FSM to execute a reset. Control restarts at init.

4.3.5 fsm_send

This FSM is responsible for the actual transmission of packets from the core to the computer which were prepared by fsm_packgen. Both of these FSMs work together - fsm_send is always executed after fsm_packgen and fsm_send cannot be triggered from any other source other than fsm_packgen. Once enabled, fsm_send holds TXEN high for the duration of the FSM execution, which tells the PHY that data is being sent to it. The preamble is also sent at this moment, then the relevant fields of the packet's header stored in RAM2. The specific fields are sometimes different between the various protocols in use, but the header lengths are identical to all packets which makes the FSM simpler to implement. After the header has been sent, the FSM connects to FIFO2 and sends the required payload data up to the necessary length. Once the payload has been sent, the Ethernet checksum is sent to finalise the packet and TXEN is pulled low. The FSM is reset and waits for another trigger by fsm_packgen.

The structure of the FSM can be described as an ordering system. Although containing the largest number of states in any of the three FSMs, most of the states perform the same task but with different data. Each *send* state transmits a particular component of the packet, whether it is a header field or the payload. Some content is taken from RAM2 and FIFO2, with the rest sent from values set in registers.

The states in fsm_send are commented below:

init

This state has the purpose of providing a clean reset to the FSM. It is executed upon power-on of the system and also executed when a packet has completed being sent by fsm_send. It sets important control signals to their default values so that future packets are not corrupted by unknown signal states.

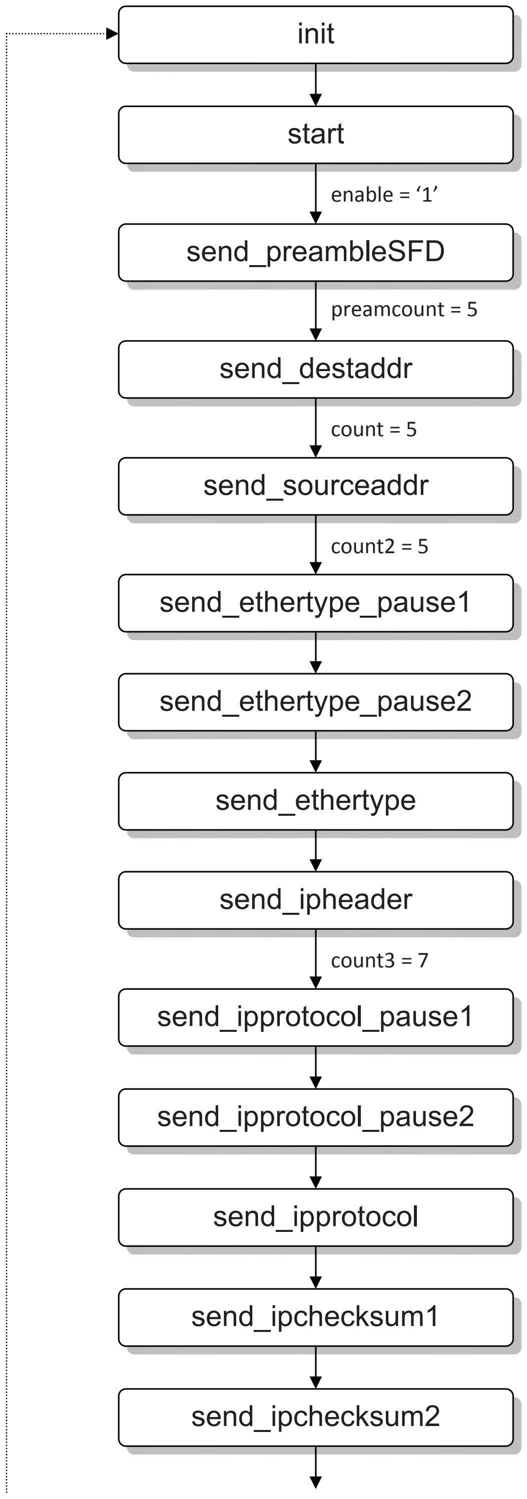

Figure 4.6: fsm_send state flowchart (page 1)

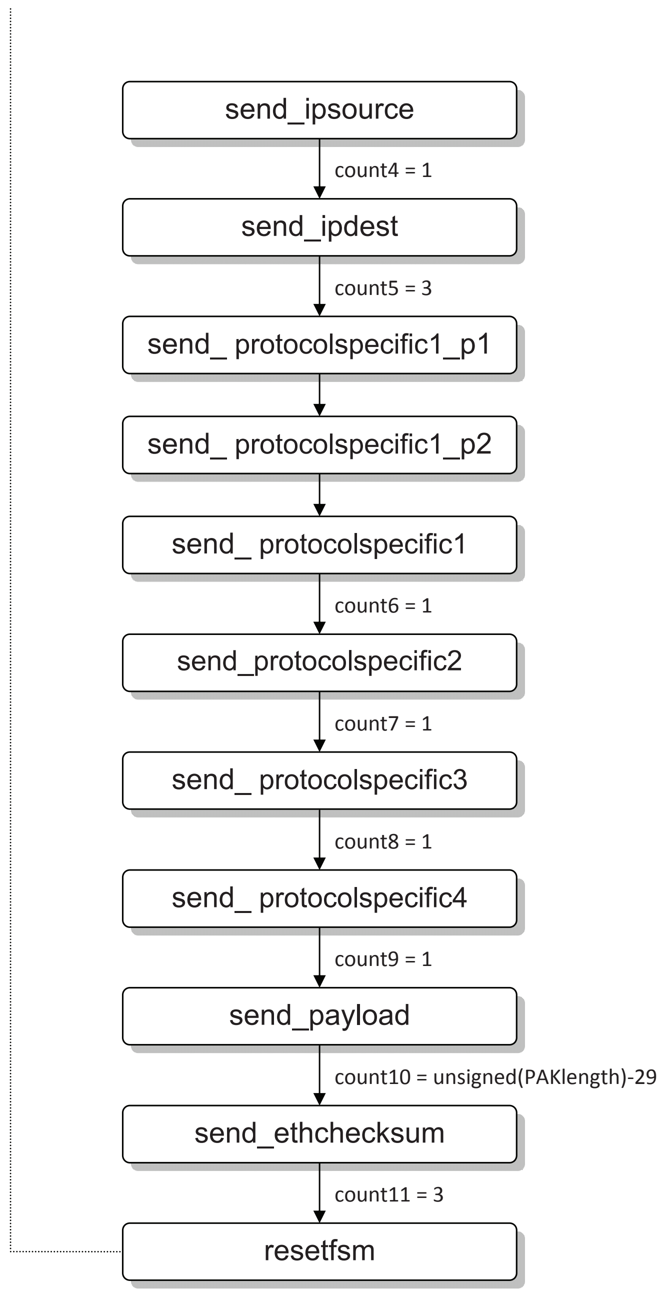

Figure 4.7: fsm_send state flowchart (page 2)

waitforactivation

This state waits for an enable trigger from fsm_packgen.

send_preambleSFD

The Ethernet preamble is transmitted during this state. Due to the pipelining delays in the design, the SFD actually gets sent in the next state.

send_destaddr - send_sourceaddr - send_ethertype_pause1 - send_ethertype_pause2 - send_ethertype - send_ipheader - send_ipprotocol_pause1 - send_ipprotocol_pause2 - send_ipprotocol - send_ipchecksum1 - send_ipchecksum2 - send_ipsource - send_ipdest - send_protocolspecific1_p1 - send_protocolspecific1_p2 - send_protocolspecific1 - send_protocolspecific2 - send_protocolspecific3 - send_protocolspecific4

All these states perform similar tasks and are run in sequence. They send the required header fields in the correct order as necessary, some of the values taken from the RAM and others with fixed values depending on the form of packet being sent. There are some occasional pause states to allow for propagation delays that exist in the RAM and pipelining architecture. The four *protocolspecific* states process the 32-bits of protocol-specific header data that are present in the packet and their tasks differ between protocols.

send_payload

Transmits the payload (application data) of the packet.

sendethchecksum

Transmits the Ethernet checksum, which is calculated on the fly. This checksum is essential for the receiver to verify the integrity of the packet.

resetfsm

Executed after the Ethernet checksum has been sent. Control restarts at init.

4.4 Reliability/Errors

Reliability, in the context of computer network protocols, is a measure of how capable a protocol is in ensuring data is delivered correctly to the intended recipient(s).[40] The system utilises UDP and UDP Lite for most of its communication, but these are considered to be unreliable protocols because they do not guarantee that a packet will be correctly sent and received. They do allow for checksumming of the header and payload, but do not support any additional features for ensuring a packet goes where it's supposed to. This is unlike TCP which uses flags and additional packets to confirm receipt of packets. For this reason, there existed a need to create effective reliability support for the system, such that if packets were missed or sent/received out of order, the system can recover and deal with the situation in a graceful manner. There is also the scenario that the system might suffer an unexpected failure, such as an incomplete/corrupted packet sent by the computer for whatever reason, or that the board might intercept a packet with an unknown or unsupported protocol. This Section of the dissertation covers the ways in which the system provides reliability and error correction. Section 4.4.1 covers the tagging method which keeps track of packets passing through the system, while Section 4.4.2 explains the quality-control methods for individual packets.

4.4.1 Tags

Tags are a simple but effective way of providing the ability to detect missing packets from a data stream, as well as determine if packets were sent out of order in the

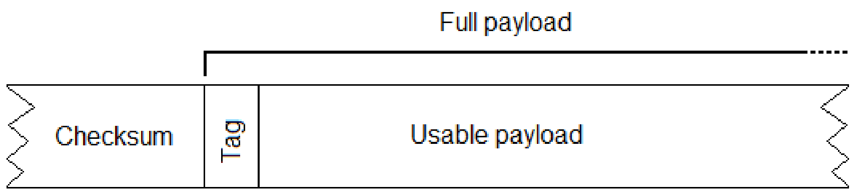

Figure 4.8: Fragment of a packet with tag added to beginning of payload

steam. A tag is a small piece of data inserted into the beginning (or end) of the payload of each packet - the tag consists of a number which increments in subsequent packets. In a particular steam, the first packet sent or received would have a tag of zero, the second packet a value of one and so on. These tags would not be controlled by the operating system but rather the program which sends/receives the packets and so it's up to the program to determine how to deal with missing packets or out of order packets. With regards to the board, the FSMs are not responsible for tags, only the programs which use the packet data. Tags are effectively transparent and considered part of the regular payload data, which provides the flexibility to use tags or not without requiring extra logic to process them.

Figure 4.8 shows an example of part of a packet which has been configured by the system to use tags. The tag shown here is located at the beginning of the payload immediately after the UDP checksum field, but it is also possible to embed the tag at the end - the tag can be anywhere so long as the location remains consistent between packets and the VHDL/computer program(s) utilising the tags are correctly programmed to process them. Although the tag takes the appearance of a regular header field, it's not part of the standard UDP structure and so can only be added as part of the payload. The tags can be of any length, but to ensure maximum packet efficiency they should be as small as possible since they take up a portion of the payload and reduce the *usable* payload, the area which can actually contain useful data. It is also up to the VHDL/computer program(s) to determine what to do if a packet is received with a missing or out of order tag. Depending on the nature of the program, it can either be discarded, relocated later once the full data stream is completed, or re-requested.

One example of where the tag system is implemented is in ATA over Ethernet, a network protocol developed by the Brantley Coile Company, designed for accessing ATA storage devices over Ethernet networks. AoE does not rely on network layers above Ethernet, such as IP, UDP, TCP, etc, but it does implement the tag system on packets and provides a look-up table on both sides of the system to determine where all the packets are located and can effect a resolution if a particular packet is missing or corrupted. This means the tag system for reliability and error correction has already proved itself in a commercial setting, making it a good choice for use with this system.[41]

4.4.2 Corrupted/Unsupported Packets

Several key requirements of the interface are for it to be robust and stable. This means the interface must be able to withstand any level of network activity without becoming caught in a state that it cannot easily recover from. Such a scenario would cause the interface to lock up and become unresponsive, forcing a reset of the entire board and potentially losing data stored on the board. To avoid this, the interface was built with several protection mechanisms designed to gracefully deal with packets and data that do not conform to expected behaviours.

As was explained in Section 4.3.3, fsm_read has the capability to determine if an incoming packet is corrupted, malformed or using an unsupported protocol such as TCP. In these cases, control of fsm_read is sent to a waiting state and execution of the FSM remains here until the current packet has finished being sent by the computer. Note that both fsm_packgen and fsm_send are totally unaffected by bad packets being received by the board and will continue to operate on packets even as fsm_read is forced to stall. Without the holding state, fsm_read may become stuck in a writing state which may not be completed due to a failure to correctly parse the packet, which is undesired for the system.

4.4.3 Checksums and CRCs

Checksum values and CRCs (Cyclic Redundancy Checks) are used heavily within the interface.[42] All packets passing through the system will contain at least two checksums and a major checksum known as the FCS (Frame Check Sequence). These checksums have the purpose of verifying the integrity of the packet, ensuring the data contained in the packet was transmitted without loss or damage to the packet. Once all the data has been processed, a CRC is calculated against the packet (known by the code as crc_output_ETH) and matched to the FCS supplied by the packet (crc_output_ETH_FCS); if these two are identical, the packet was sent uncorrupted and so the content is valid. If the values are different however, a control signal is raised to represent a packet failure.

4.5 UDP Lite

UDP Lite was mentioned earlier in this dissertation, but the specifics of the protocol and how it is implemented in the system were not covered until now. UDP Lite is special because it allows for a simpler hardware implementation that avoids several paths of logic which UDP would normally have to enter. This is due to the relationship between the checksum calculations required by the protocols and what happens to a FIFO when it is read.

Every packet that's sent needs to have correct checksums in their checksum fields. If the checksums are incorrect or missing, the recipient will generally consider the packet to be corrupted and discard it. Some protocols may use checksum fields which don't exist in others, but regardless of the protocol in use there are always at least three checksums that need to be calculated - two of those are always the Ethernet checksum and the IP checksum. The Ethernet checksum represents the contents of the entire packet (excluding the preamble), while the IP checksum covers only the IP pseudo-header of a packet, but not the payload. For UDP packets there is a third checksum field, the UDP checksum, which covers the UDP header plus the payload. For ICMP packets the third checksum field is the ICMP checksum, with similar specifications. Correct operation of the board requires all checksums for transmitted packets to be correct, otherwise the packet won't be accepted by the computer and data will be lost.

For ICMP and UDP (mirrored) checksums, the IP and ICMP/UDP checksums for the transmitted packet are easy to calculate since they are identical to those from the received packets (the order of the headers are often switched around but the data contents are identical, since they're just copies of the same payloads). Ethernet checksums are calculated during the sending of a packet, working the same regardless of protocol or how packgen was triggered (either externally or internally) and so do not pose any problems. However, when sending a UDP packet on its own without being triggered from fsm_read, all necessary checksums have to be calculated because the payload data won't match the previously-copied checksums.

To calculate the UDP checksum the UDP payload data needs to be read, however whenever a FIFO is accessed the last element read is permanently removed from the FIFO. Unlike RAM, it's not possible to simply parse the payload for checksum information without removing the entire payload itself. To address this issue, there are two possible approaches. One solution would be to replace the FIFO with RAM which would allow reading without data destruction, but this would require adding address control signals and possibly slow down the design with the increased logic. Another solution, the one which was decided upon, is to use a new protocol called *UDP Lite*.

UDP Lite is very similar to regular UDP but with one difference - the UDP checksum's "coverage" (the amount of the UDP header the checksum has to correctly match) can be varied. This means it's possible to set a coverage such that the UDP checksum only covers the header component and NOT the payload, which simplifies things enormously since the FIFO can remain untouched and only the RAM needs to be scanned. As this protocol has gained support in most modern operating systems, there are no significant disadvantages in using it compared to standard UDP.

4.6 Physical Implementation

The core is designed to run on a Xilinx Spartan 3 FPGA, and is considered the base level FPGA for the core. It is not recommended to attempt to install the core on anything less capable than the Spartan 3. We used the following development board to test our IP core:

- Two Xilinx Spartan 3 FPGAs
- Two National Semiconductor DP83865 10/100/1000 Gigabit Ethernet Physical Layer transceivers
- Two PulseJack Gigjack T12 RJ45 network jacks

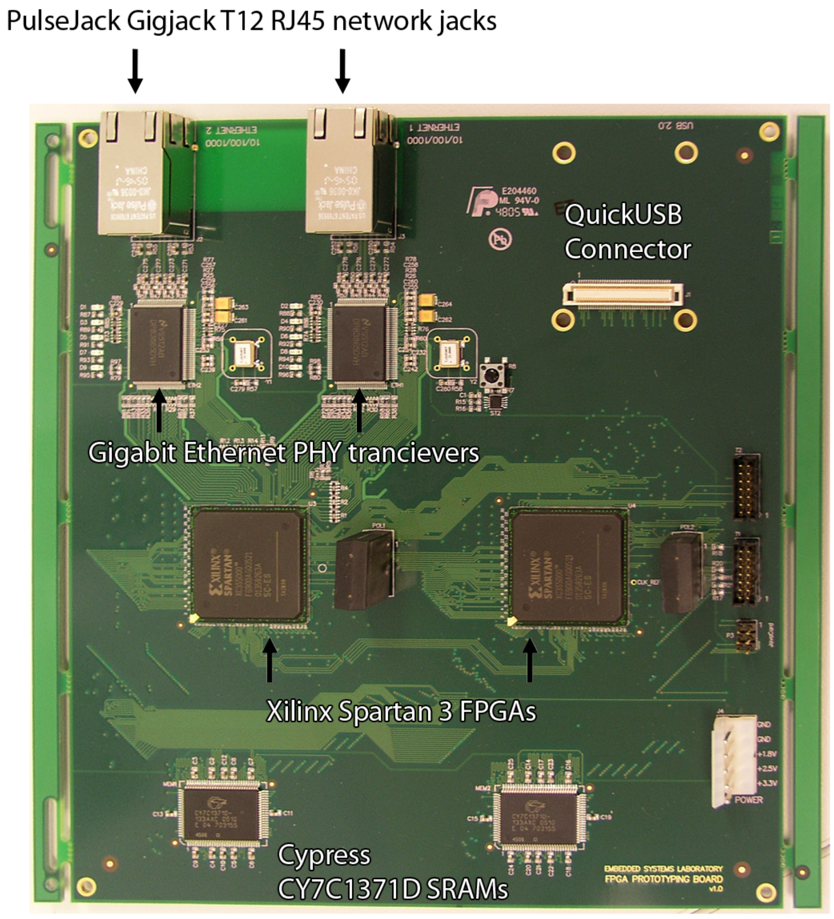
PulseJack Gigjack T12 RJ45 network jacks
QuickUSB Connector
Gigabit Ethernet PHY trancievers
Xilinx Spartan 3 FPGAs
Cypress CY7C1371D SRAMs

Figure 4.9: Prototyping board used for testing

- Two Cypress CY7C1371D SRAM memory modules, providing a combined total of 18 Mbits of storage space for programs on the FPGAs to use

- A QuickUSB daughter board connector, providing a means of utilising the QuickUSB Plug-In module for easy USB access

- Various extra connectors such as JTAG, FPGA parallel connector, JavaCard connector and board power

Figure 4.9 shows the board with major components highlighted:

For each network jack, the interface between the physical layer (the DP83865 transceivers) and the Media Access Control device (the FPGA running the IP core) is established using the standard Gigabit Media Independent Interface (GMII).[43] GMII is a very commonly implemented standard for establishing Gigabit Ethernet connectivity, and specifies the following attributes:

- Network speeds operate up to 1000 Mbit/s
- The data interface consists of an eight bit channel clocked at 125 MHz
- The interface is backwards compatible with the Media Independent Interface (MII) specification, which as a result can support speeds of 10/100 MBit/s as a fall-back

In the case of the IP core, backwards compatibility with MII was not implemented because the core is designed for Gigabit connections only. Supporting MII would be trivial if it were only a speed difference (the system can operate at 10/100 MBit/s speeds without modification), however MII uses a data channel of four bits instead of eight, and so it was decided not to accommodate the older standard because there was no desire to use slower connections. The core is specifically intended for high-speed, high-bandwidth applications, something that MII-based connections would not be at all appropriate for.

The core was physically tested on the board by uploading the core to each Spartan 3 and verifying the system linked correctly and processed test packets. The process of converting the source code into a form which can be uploaded to the FPGAs is called *synthesis* and is covered in Section 6.2. By performing the synthesis, statistics were obtained about the technical specifications of the core, in particular how well it ran and how much logic it occupied on a Spartan 3. The exact specifications of the FPGA model used in the design is a Xilinx Spartan 3 XC3S5000, package FG900 (ie.

900-pin), speed grade of -5 and the software packages used to synthesise the design and achieve the following results were Xilinx ISE 10.1 and Synplify 8.8.0.4:

```
Logic Utilization:
  Number of Slice Flip Flops:           970 out of  66,560    1%
  Number of 4 input LUTs:             1,920 out of  66,560    2%
Logic Distribution:
  Number of occupied Slices:          1,166 out of  33,280    3%
    Number of Slices containing only related logic:
                                      1,166 out of   1,166  100%
    Number of Slices containing unrelated logic:
                                          0 out of   1,166    0%
  Total Number of 4 input LUTs:       2,020 out of  66,560    3%
    Number used as logic:             1,664
    Number used as a route-thru:        100
    Number used for Dual Port RAMs:     256
      (Two LUTs used per Dual Port RAM)
  Number of bonded IOBs:                 40 out of     633    6%
  Number of RAMB16s:                     16 out of     104   15%
  Number of BUFGMUXs:                     1 out of       8   12%
Constraints cover 44747 paths, 0 nets and 8533 connections
Design statistics:
   Minimum period:   7.795ns   (Maximum frequency: 128.287MHz)
```

Chapter 5

Pipelining

Pipelining, as defined in computer architecture, is a design technique in which code and/or data used by a system can be restructured in such as way as to reduce the amount of time necessary to process it. In the current context, pipelining involves running the various Finite State Machines in *parallel*, stacked one on top of another, instead of a conventional *serial* execution. Effective use of pipelining can be extremely beneficial to applications such as cryptography, data processing[44] or computer architecture, where a variation in encryption/decryption technique can cause a significant change in the time needed to perform the operation.[44]

The decision to use a pipelined architecture in the core was performed out of necessity. There was no other way to obtain full Gigabit speeds and full speed packet delivery on a Xilinx Spartan 3 FPGA, which is a fairly low-powered FPGA at the time of writing and also the base FPGA for the design of the core, without resorting to pipelining. Pipelining was used not only for processing of the packets but also with regards to all memory access in a move to further increase the speed of the core. Also of importance is that the pipelined architecture allows for packets to be processed one after the other without requiring any waiting period. Since the finite state machines act in parallel, this allows fsm_read to process packets as they appear and does not have to wait for a further FSM to complete.

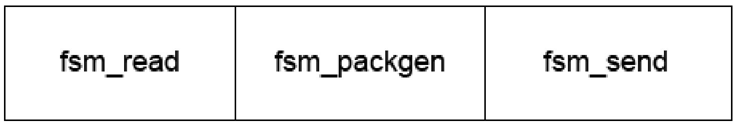

Figure 5.1: The three FSMs the single ICMP packet will be processed with (in order)

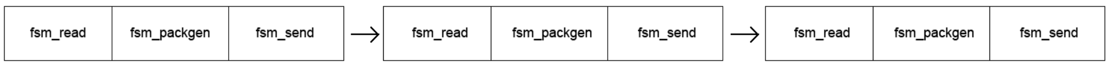

Figure 5.2: The time-line of three ICMP packets processed in serial

---

To better describe the operation and effectiveness of a pipelined architecture, observe Figure 5.1 which shows the flow of the system for a single ICMP packet received by the board. The overall block represents the passage of time for each received packet (from the left flowing to the right) and each segment of the block represents the passage of time for that particular FSM. Every packet has to be parsed by fsm_read, a return formulated by fsm_packgen and finally transmitted fsm_send. Let us assume that the time taken for each FSM is roughly similar, since the received and returned packets have identical lengths and data. Let us also assume that instead of waiting a predetermined amount of time before sending ping packets, the pings are configured to send a new ping as soon as it is considered safe to do so. What is considered “safe” depends on the execution method - the aim is to send packets as quickly as possible without there being any loss or congestion in the system and this rate will be shown to vary between serial and parallel (pipelined) execution.

Figure 5.2 shows the time-line of three sequential ICMP packets under serial execution. Once the first packet is received and returned to the computer, the next one is sent immediately and once that is returned to the computer the final packet can

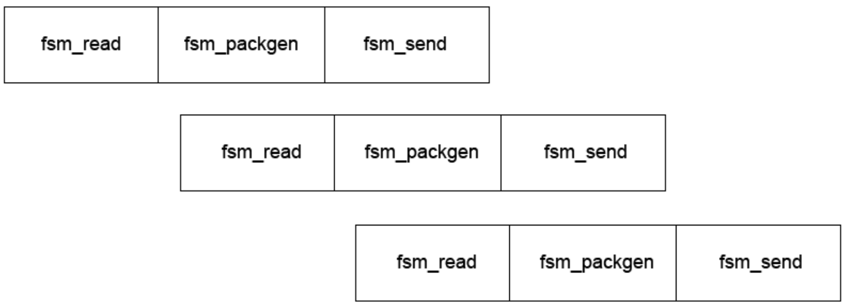

Figure 5.3: The time-line of three ICMP packets processed in parallel (pipelined)

be sent. Serial execution is simple to understand and apply, but the rate of packet transmission is very slow. Only one FSM is in operation at any given time, but this need not be the case, as they are capable of working independently on another packet separate from the first.

Figure 5.3 shows the same three sequential ICMP packets now running under a parallel (pipelined) execution. In this scenario, the computer is sending ping packets one after the other as fast as it can without waiting for a responding pong from the board. This is achievable with a pipelined architecture because once the first packet is read by the board, execution moves to fsm_packgen AND fsm_read - packgen works on creating a return for the first packet while fsm_read busies itself with reading the second packet. What is of note is that when the first packet has reached the fsm_send part for transmission, the second packet is working with fsm_packgen and the system will have begun sending the third packet for reading by fsm_read. At this point all three FSMs are working separate from each other on separate packets. Over time the return packets will be sent in sequence until the third and final packet is sent.

With serial execution, assuming roughly the same length of time per FSM, it takes 9 units of time to process three ping packets. With pipelined execution, it takes only 5 units of time. So, by ensuring the design is pipelined instead of serial, we are able to cut the total time (ping and pong) of three ICMP packets by nearly half. With this implementation, pipelining does not increase the speed at which the FSMs operate, but rather organises the data such that several FSMs can operate on different data chunks at the same time. For this reason the pipelining architecture in this design is referred to as Data Pipelining, rather than Instructional Pipelining.

5.1 RAM Swinging Buffers

An issue can arise when dealing with pipelined data - data corruption by subsequent packets. If the computer sends a packet to the board and while the board is in one of the latter FSMs the computer sends another packet (as is allowed under a pipelined architecture), there is the potential for packet data from the first packet to be overwritten by that of the second packet.

For example, let's say the computer sends an ICMP packet along with a UDP packet. For the first packet, the board will read the header & payload data, store them into RAM1 and FIFO1 respectively, then move onto fsm_packgen to perform a RAM1-RAM2 and FIFO1-FIFO2 copy. During the copy, the UDP packet is received by the board and so fsm_read performs its duty and reads the incoming UDP packet. Now, the information in this packet is totally different to the prior ICMP packet - the headers are different and the payload data is most certainly different. Hence, we run into several problems:

1. If fsm_packgen is performing a RAM1 to RAM2 copy and at the same time a new packet header is being read into RAM1, the header data of the new packet might be copied into RAM2 along with that of the original packet. This would result in the transmitted packet having a combination of ICMP and UDP

header fields, which would be seen as garbage by the receiving computer and discarded.

2. If fsm_packgen is performing a FIFO1 to FIFO2 copy while a new packet is being read, if the header data of the new packet specifies a payload length that's different to the previous packet (as a result of the RAM copying scenario in point 1), issues might result with the payload. For example, the payload might not be copied entirely or too much data may try to be copied regardless of whether there's anything in FIFO1 or not. This would result in the transmitted packet having not only an incorrect amount of payload data as well as a corrupted header.

3. If fsm_packgen encompasses both RAM and FIFO copy stages, the resultant packet will be corrupted and worse, the system is now considered unstable. The RAM will have corrupted data which can't be relied upon for future packets created from scratch and the FIFO will almost certainly be useless since the state of the data stored in it will not be known. At this point all future packets are likely to be transmitted corrupted and so the only solution is to reset the board.

To combat the issue of data corruption by packets overloading each other, there are three possible solutions:

1. Take extra control over the way the computer sends out packets, so that there is sufficient time between subsequent packets to allow for smooth operation without packet crowding. This solution is undesirable however - it would be the slowest, since no pipelining would be allowed. It would also involve extra work in configuring the computer to allow for this brief pause between packets, which may not be easy to accomplish without modifying the kernel of the operating system.

2. Block fsm_read from accepting new packets if it is not safe to do so. For example, fsm_read would not begin reading new packets if fsm_packgen was running, because the potential for packgen to corrupt the transmitted packet would exist if the data contained in fsm_read was being modified at the same time as the copying. This solution again is undesirable, since it would mean the board would sometimes miss packets being sent by the computer.

3. Double the size of the RAMs so that each RAM can store the header data of *two* packets instead of just one and switch the addressing between them as necessary. It isn't possible to do the same with FIFOs since they do not have an accessible addressing mechanism, but this is not an issue if the headers remain correct, since the payload data can be added to the end of the FIFO even during a copy process without damage, as each header defines the appropriate payload data length. This is the option which has been implemented, as it allows for the pipelined architecture without packet corruption.

5.2 Timing Diagram

The diagram in Figure 5.4 shows the path of three sequential packets, all of them in UDP mirrored mode so that they are sent back immediately. Once one packet has been sent, the next one is fired off straight away. Each row in the diagram represents the time-line of a packet, the first row being the first packet and so on. Each row is split into three Sections - the first represents the activity for FSM_READ, the second for FSM_PACKGEN and the third for FSM_SEND. The purpose of the diagram is to show the operation of the RAMs and FIFOs and the RAM swinging buffers and what components are running during the relevant stages of each packet. The diagram shows why the system is not vulnerable to packet corruption:

- There never occurs a situation where the same storage area in a RAM is used more than once at any given time. There are times where the same RAM is both read and written at the same time, but due to the swinging buffer, this occurs on opposite halves of the RAM, which are isolated and so do not corrupt each other. There is a requirement that the RAMs used are dual-port RAMs, otherwise the read/write operations could not work simultaneously. We will see later that switching the RAM buffer only requires changing the MSB (Most Significant Bit) of the RAM address.

- Although there are many cases where the FIFOs are being read and written at the same time, this is not a problem. The nature of the FIFO (**F**irst **I**n, **F**irst **O**ut) means that whatever data is being written, will only be read once the data in "front" of it has been read. In other words, simply reading at the same time as writing won't damage the integrity of the data, since the IO operations occur at *opposite sides* of the FIFO's internal storage.

5.3 Implementation

The RAM component for holding the header data is called *headram* and its black-box is defined in VHDL as follows:

```
component headram
    port (
        A:    IN std_logic_VECTOR(6 downto 0);
        CLK:  IN std_logic;
        D:    IN std_logic_VECTOR(7 downto 0);
        WE:   IN std_logic;
        DPRA: IN std_logic_VECTOR(6 downto 0);
        QDPO: OUT std_logic_VECTOR(7 downto 0);
        QSPO: OUT std_logic_VECTOR(7 downto 0));
end component;
```

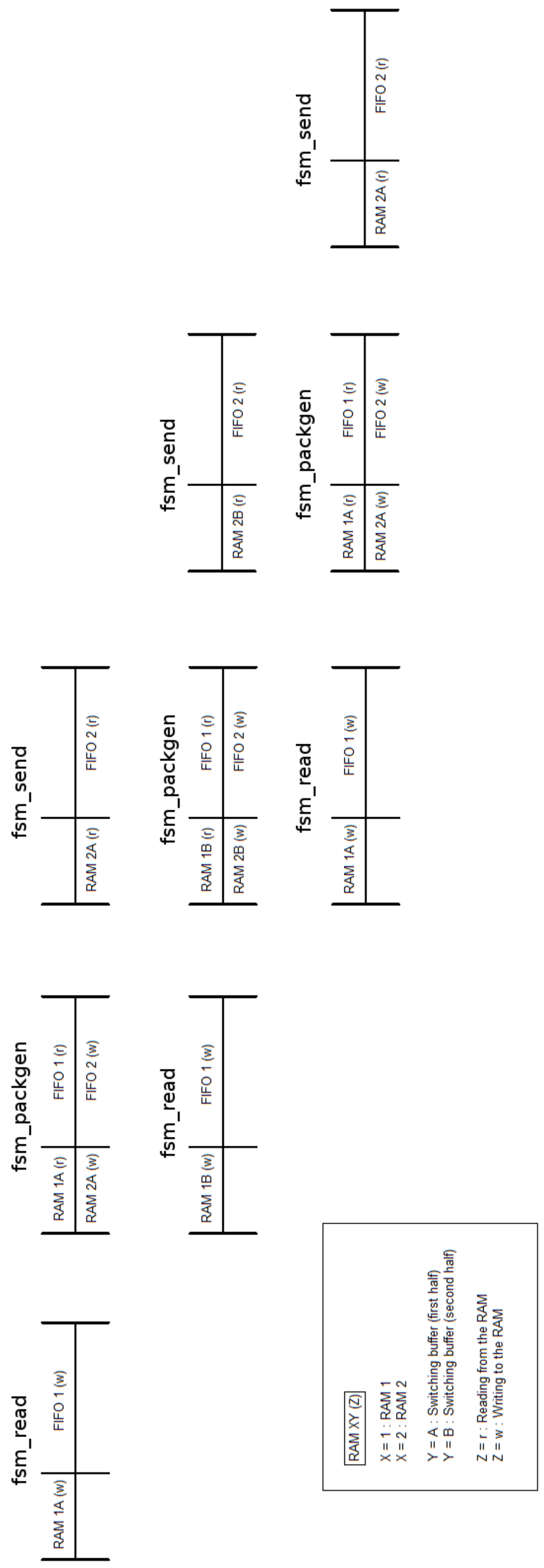

Figure 5.4: Timing diagram of three UDP packets in mirrored mode

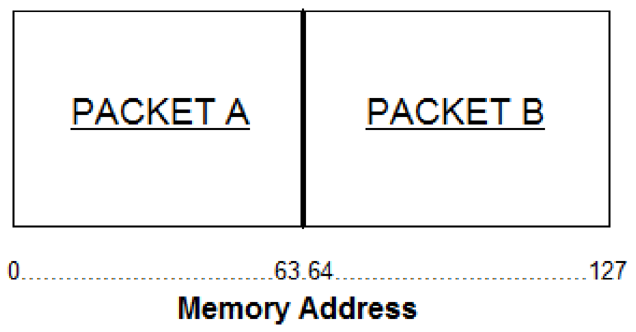

Figure 5.5: The locations of two packets in RAM

The address (A port) is 7 bits wide, which means a total of $2^7 = 128$ address locations, with each address holding a byte of data (as D port is 8 bits wide). As the RAM only holds the packet header and as no header used by the system will ever be greater than 42 bytes long, 128 memory addresses may seem excessively large. Indeed, an address size of 6 bits resulting in a total of $2^6 = 64$ address locations would still have been large enough for the packet header and given the importance of culling excessive resources to ensure maximum speed and routing efficiency, using the address size of 128 for all RAMs may not appear optimal. However, the reason for selecting 7 bits instead of 6 is to encompass the *swinging buffer* which provides the protection the system needs from overlapping packets and for full pipelined capability.

Figure 5.5 shows how the packets are arrange in RAM. Each RAM is divided into half - each half holds the header of a packet. Address locations 0...63 are used to access the first half and 64...127 for the latter half. The swinging buffer is used to store the header contents of two packets at the same time - packet A and packet B. When the board is initialised after a reset or power-on, the RAM is empty. The computer sends a packet (which is designated here as packet A), the board receives the packet and fills the RAM from address 0 onwards until the end of the header, as normal. When the packet has finished being read, a pulse is trigger from the statement

```
SWITCH <= '1';
```

When SWITCH is enabled high, a toggle flip-flop is triggered which inverts the value of the *switchbufpos* signal. After being enabled SWITCH is turned off during the subsequent state, completing the one-cycle pulse which occurs at the end of every packet read. switchbufpos is used in the following statements:

```
writeaddress_usedonRAM(6) <= switchbufpos;
writeddress_usedonRAM(5 downto 0) <= writeaddress_requested;
readaddress_usedonRAM(6) <= not switchbufpos;
readaddress_usedonRAM(5 downto 0) <= readaddress_requested;
```

writeaddress_ usedonRAM and *readaddress_ usedonRAM* are the address signals used to connect to the RAM - the former specifies the location for writing data and the latter for reading data. Any addresses requested by the system are specified with the *writeaddress_ requested* and *readaddress_ requested* signals. The "usedonRAM" and "requested" versions of the addresses are identical except for the MSB (Most Significant Bit) specified with the former batch of signals. The MSB is determined by the *switchbufpos* signal and hence will cause the RAM to address entirely separate areas of the memory depending on its value. When the MSB is zero, the RAM will only be accessing the first half of its available space (address range 0000000 [0] to 0111111 [63]), but when the MSB is one, the RAM uses the second half (address range 1000000 [64] to 1111111 [127]). By providing the ability to switch between the two halves, two whole packets can be held in the one RAM and thus the packets are isolated from one another when subsequent packets are being processed.

The design of the swinging buffer is such that simultaneous read and write memory operations must address *opposite* memory banks. An example is shown in Figure

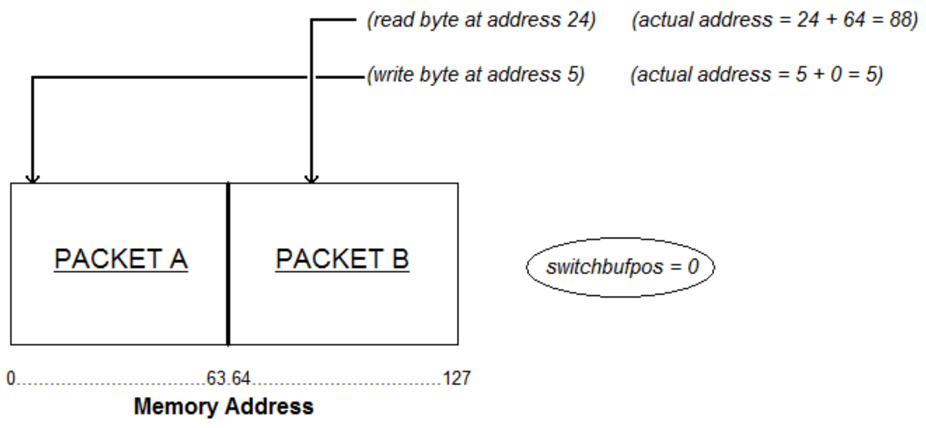
(read byte at address 24)
(actual address = 24 + 64 = 88)
(write byte at address 5)
(actual address = 5 + 0 = 5)
PACKET A
PACKET B
switchbufpos = 0
0
63 64
127
Memory Address

Figure 5.6: Two RAM operations operating at the same time via swinging buffer

5.6. In this example the system both reading and writing data on a particular RAM. With switchbufpos having a value of zero, this means the RAM is writing a byte to address position 5, which since the MSB is zero represents the absolute (real) address 5. At the same time the RAM is reading a byte at address 24, which since the MSB is the NOT of switchbufpos, has an offset of 64 positions. Hence, although the read address is specified as 24, the absolute address is actually 88. With this design it is impossible for a packet to be overwritten by a subsequent packet. Although the buffer only allows for two packets in the RAM, there's no need for more packets since by the time the third packet arrives, the initial packet would have been either fully processed or moved entirely to a new RAM via packgen. As a result, the space originally occupied by the first packet would become available for the third, the space for the second packet would become available for the fourth and so on.

Chapter 6

Testing/Synthesis

Performing testing on a system as large and intricate as this design required a great deal of planning. When the design was in its early stages, it contained fewer components with less functionality and as such had simpler testing requirements. As the system grew and became more complete however, simple tests were no-longer sufficient. For example, initial tests were run on just one test packet sent through a simulator, observing the flow of data as the packet travelled through the system. This was sufficient at testing various protocols in a structured way, but a key aspect of the system is its robustness, its ability to withstand any and all possible situations or *boundary conditions* it may face. For this, the testing had to become more rigorous and extend towards automation of testing rather than simple hand-crafted packets. A further explanation of the process is covered later in this chapter.

6.1 Testing

6.1.1 Single Packet Tests

The first protocol to be implemented in the interface was ICMP (ping). This protocol is very useful for testing the majority of the system's functionality, because a successful test requires both a received packet and a transmitted packet to properly use the protocol. This therefore makes use of all FSMs in the design and hence provides a more thorough test than just with receiving a packet from the computer. When advancing to single packet testing of UDP and UDP Lite, these protocols do not have any requirement to send back response packets as part of their design. For testing and debugging purposes, the interface code can be configured to automatically create return packets when a packet matching one of these protocols is received. This form of single packet testing for non-ICMP packets uses all the FSMs and provides a good test of the system's ability to handle single packets belonging to the supported protocol set.

Single packet testing of this sort was accomplished by constructing a testbench in VHDL and simulating it in Modelsim which would supply the contents of a packet stored in a text file. The testbench loads a file called "numbers-icmp.txt" which contains a sequence of binary values strung together to create a single chain of values. This forms the data for a pre-built packet stored in the text file, which is passed off to the main interface and by using a testbench, as far as the interface is concerned this is data being sent by a computer. Necessary signals to facilitate this are also simulated by the testbench, such as RXDV.

The testbenches used in single packet testing do not address the output of the system. Verification of packets send back to the "computer" as part of how ICMP operates are made from visual inspection of the *wave table* the simulator generates. Figure 6.1 shows an example of such a wave table for a single ICMP packet test.

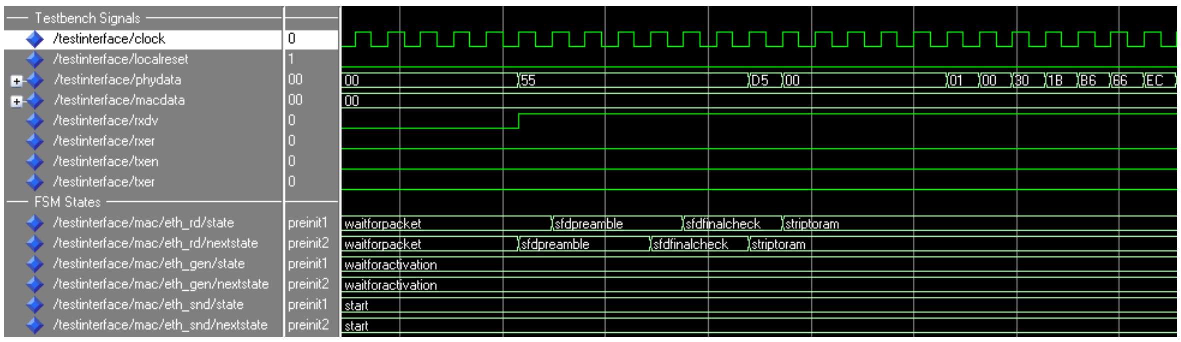

Figure 6.1: Partial wave table for the beginning of an ICMP packet

6.1.2 Multiple Packet Tests

Once a single packet has been verified to run successfully through the interface code, it's time to start performing in-depth testing with a large number of packets sent one after the other. This form of testing achieves several goals:

- Mass numbers of packets provide a realistic scenario for use of the interface. In a real-world application, it is highly unlikely for the system to be passing solitary packets with significant time between them, but rather a large block of data containing packets strung close to one another.

- Boundary conditions can be tested. There are many different combinations of data in the fields of a packet and it is counter productive to hand test each situation to determine if a flaw exists in the code. By running differing packets through the simulator, each with a different block of data and several different (but valid) changes to the headers of each, it is possible to determine if the system's code is functioning properly.

- The ability of the system to sustain continuous operation under heavy load can be measured. Robust operation is only satisfied if the system operates perfectly under continuous, unrelenting packet transmission with all forms of packets.

- The tag method of error detection covered in Section 4.4.1 can be tested.

The primary method for performing the automated testing was by creating a testbench to perform automated packet construction. There are two sets of data used in the formation of these packets - the header and the payload data. The following shows the steps the testbench takes when performing automated testing:

1. A "template" packet, either ICMP or UDP depending on the form of packets to be tested, is loaded from the contents of a text file, which consists of base header information for the packet. The majority of this header template is used for each packet generated by the testbench, however several of the fields are modified per packet later on in the testbench to reflect changes from packet to packet (eg. checksums, sequence identifiers, etc).

2. A "payload" packet is loaded from the contents of a text file, which consists of the payload data to be used in the testing. The packet protocol to be employed has no bearing on the form of the payload data, so virtually anything can be used so long as it can be loaded by the testbench code. Each generated packet contains a chunk of this payload data, so testing the integrity of a large amount of data processed by the interface code is very easy to do using the multiple packet testing method.

3. The testbench starts creating and sending a packet on the fly, with Ethernet checksum, IP/UDP/ICMP CRCs generated when necessary. It's important to ensure all of these values are correct, otherwise they will be rejected by the error detection mechanisms in later FSMs. Tags are also embedded as part of the testing. Once a packet is sent in its entirety, the testbench waits for a responding packet sent back by fsm_send (it is assumed that UDP has been put into mirror mode if UDP is the protocol being testing).

4. Verification of the incoming packet is performed in three ways:

- Tag values from incoming packets are stripped out and written to a file called "tags.txt" on the computer running the simulation. The testbench does not check whether the tags are in order or have missing packets; this task is left to the user, as tags aren't a required part of the interface and their implementation is optional.

- For each packet sent and received by the testbench, two Ethernet checksums are calculated. When a packet is being passed to fsm_read, the testbench uses this packet information to calculate a prediction of the Ethernet checksum for the mirrored packet, which is stored inside the testbench as *crctocheck*. When fsm_send produces the actual mirrored packet, it sends a copy of its Ethernet checksum back to the testbench via toplevel and is stored inside the testbench as *top_crceth2*. As *crctocheck* is the predicted checksum, it is assumed to always be correct for the particular packet being sent by fsm_read. When the incoming packet from fsm_send is received, the two checksum values are compared; if they are identical, this means all the header values and payload data in the received packet are as they should be, which means all FSMs are operating as expected. If they are different however, something has gone wrong in one or more of the FSMs and the simulation is automatically stopped via a VHDL assertion call in the testbench. The following code fragment shows the process which controls the assertion:

```
process (TXEN)
   begin
        if falling_edge(TXEN) then
            -- Transmit Enable (TXEN) has gone low; fsm_send has finished
            -- top_crceth2 contains valid data
            -- Check if Ethernet checksums are identical
            assert top_crceth2 = crctocheck
                report "SIMULATION FAILURE: Ethernet checksums different"
                severity Failure;
        end if;
end process;
```

- Payload data from the incoming packets is stripped out and written to a file called “output.txt”. If the system has worked properly, the output.txt file and the file used to send the payload data should be exactly the same in both size and content. A simple application of the *diff* command can verify this.

The testbench for multiple packet testing requires its own state machine to control its operation, which is as follows:

```
process (State, udpcount, TXEN, crc_output_ip, ipcount,
         subject3, crc_output_eth16)
    begin
        NextState <= init;

        Case State is
            when init =>
               -- Initialisation state
               crcclock <= '0';
               crc_enable_ETH16 <= '0';
               finish_ETH16 <= '0';
               crc_enable_IP <= '0';
               finish_IP <= '0';
               resetcrc <= '1';
               -- 0 = UDP, 1 = ICMP
               packetformat <= '0';
               NextState <= finalint;
           when finalint =>
               if packetformat = '0' then
                  datatosend <= filedata0;
               else
                  datatosend <= filedata1;
               end if;
               resetcrc <= '1';
               NextState <= doCRCcount;
            when doCRCcount =>
               resetcrc <= '0';
               crc_enable_ETH16 <= '1';
               crc_enable_IP <= '1';
               if ipcount >= 10 and ipcount <= 11 then
                  crcdataIP <= "00000000";
               elsif ipcount = 20 then
                  crcdataIP <= "00000000";
               elsif ipcount = 21 then
                  finish_IP <= '1';
               elsif ipcount = 22 then
                  datatosend((8*(31)+1) to (8*(31)+8))
                        <= crc_output_ip(15 downto 8);
```

```
 elsif ipcount = 23 then
    datatosend((8*(32)+1) to (8*(32)+8))

          <= crc_output_ip(7 downto 0);

 else
    if ipcount < 200 then
       if packetformat = '0' then
          crcdataIP <= filedata0((8*(ipcount+21)+1)

                       to (8*(ipcount+21)+8));

       else
          crcdataIP <= filedata1((8*(ipcount+21)+1)

                       to (8*(ipcount+21)+8));

       end if;
    end if;
 end if;
 if udpcount = 0 then
    crcdata16 <= "00000000";
 elsif udpcount = 1 then
    crcdata16 <= "00010001";
 elsif udpcount = 17 then
     control3 <= '1';
  if packetformat = '0' then
    crcdata16 <= filedata0((8*(udpcount+31)+1)

                 to (8*(udpcount+31)+8));

    else
    crcdata16 <= filedata1((8*(udpcount+31)+1)

                 to (8*(udpcount+31)+8));

end if;
 elsif udpcount >= 18 and udpcount <= fullsize+17 then
    crcdata16 <= subject3;
 if udpcount = fullsize+17 then
     control3 <= '0';
 else
     control3 <= '1';
 end if;
 elsif udpcount = fullsize+18 then
     if packetformat = '0' then
        crcdata16 <= filedata0((8*(45)+1) to (8*(45)+8));
     else
        crcdata16 <= filedata1((8*(45)+1) to (8*(45)+8));
     end if;
 elsif udpcount = fullsize+19 then
     if packetformat = '0' then
        crcdata16 <= filedata0((8*(46)+1) to (8*(46)+8));
     else
        crcdata16 <= filedata1((8*(46)+1) to (8*(46)+8));
     end if;
```

```
                elsif udpcount = fullsize+20 then
                   crcdata16 <= "00000000";
                elsif udpcount = fullsize+21 then
                   finish_ETH16 <= '1';
                elsif udpcount = fullsize+22 then
                   datatosend((8*(47)+1) to (8*(47)+8))

                         <= crc_output_eth16(15 downto 8);

                elsif udpcount = fullsize+23 then
                   datatosend((8*(48)+1) to (8*(48)+8))

                         <= crc_output_eth16(7 downto 0);

                else
                   if packetformat = '0' then
                      crcdata16 <= filedata0((8*(udpcount+31)+1)

                                   to (8*(udpcount+31)+8));

                   else
                      crcdata16 <= filedata1((8*(udpcount+31)+1)

                                   to (8*(udpcount+31)+8));

                   end if;
                end if;
                if udpcount = fullsize+23 then
                   NextState <= hold;
                else
                   NextState <= doCRCcount;
                end if;
             when hold =>
                 begincycle <= '1';
                 NextState <= holdbeforereset;
             when holdbeforereset =>
                 begincycle <= '1';
                 initial <= '0';
                 crc_enable_ETH16 <= '0';
                 crc_enable_IP <= '0';
                 finish_ETH16 <= '0';
                 finish_IP <= '0';
                 if falling_edge(TXEN) then
                    NextState <= init;
                 else
                    NextState <= holdbeforereset;
                 end if;
             end case;
     end process;
```

The purpose of this portion of the testbench is to prepare the data stream for transmission to the IP core as well as send appropriate control signals to the checksum and CRC calculators. The initial stages of the FSM begin by setting the *datatosend* signal to the template packet which is to be transmitted later by the testbench. Two different template packets are available, one for UDP packets and one for ICMP packets. Both templates are stored as text files along with the testbench and are loaded by the testbench upon execution. These template files contain a fairly generic example of a packet with the protocol the particular template is referring to, minus a payload which is sourced from a separate file. Depending on the value of the *packetformat* signal (0 for UDP, 1 for ICMP), the *datatosend* signal is assigned the relevant template packet data. The testbench then proceeds to run through the entire packet, including the payload which is assigned to the CRC calculator as signal *subject3*, analysing the contents and assigning the required IP/ICMP/UDP checksums and CRCs where appropriate in the packet. After the packet has been prepared to look like a correct, genuine packet of the designed protocol, the FSM triggers another FSM in the testbench which actually starts sending the packet to the IP core. The FSM resets itself once the transmission has completed (via monitoring the TXEN signal) and begins sending another packet with a new chunk of payload data. The same template is used, just modified as appropriate to support each new chunk of payload data. The same file of payload data is always used, however since only a small amount is actually sent per packet (eg. 6,000 bytes depending on the settings in the testbench), the testbench is designed to advance through the full payload data, taking small segments from it to use as individual packet payloads when creating each new packet.

A key distinction between single and multiple packet testing is that verifying the integrity of packets in single packet testing is performed solely by using the wave window in Modelsim and a packet's integrity cannot be determined any other way. Multiple/automated packet testing solves this problem by only requiring use of the wave window in situations where the simulation has stopped due to an Ethernet checksum mismatch, or other debugging requirements for specific areas of interest. The checksum/CRC elements are the key to this, which is why the above code exists

- it determines what the checksum and CRC values of the returned packet (from fsm_send) are supposed to look like *before* the packet is actually returned. A comparison can be done at the end of the packet to ascertain if the FSMs are working correctly. So long as the interface is working fine, the entire Modelsim GUI can be bypassed entirely and most operations can be setup using scripts to verify things like tags and payload data which the testbench does not perform itself.

6.1.3 Malformed Packet Testing

The system is designed to handle packets that are malformed or otherwise damaged and cannot be validated. In these cases, single and multiple packet tests were performed by adjusting the test packets in various ways, such as changing a byte in the payload so that the checksums in the headers would be incorrect, increasing the size of certain fields such as the payload size, etc. In these cases, the results were such that for all cases of malformed packets, the Ethernet CRC will **always** fail to match, and the system is designed to simply let the packet pass through without allowing it to trigger further FSMs. Often the IP checksum would fail first and so the system would wait until the packet passed through before waiting for another packet. Since the checks for malformed packets are mostly based on checksum and CRC values, this simplified the various areas that needed to be tested and so the tests involving actual malformed packets were of less importance than testing known good packets.

6.2 Synthesis

Synthesis is the process by which any HDL code (VHDL or Verilog) can be turned into a form suitable for uploading to and running on an FPGA. It is analogous to the act of compiling source code to run software on a computer. Many times during the development of the interface, the code would be synthesised to test the current build of the design in a real-world scenario, as the operation of a design during simulation will not necessarily be identical to its operation on actual FPGAs connected to additional hardware.

The following statistics show the physical resources the core uses on a Spartan 3 after synthesis. No additional programs were installed onto the FPGA, so these statistics are entirely from what the IP stack utilises on its own. The exact specifications of the FPGA model used in the design is a Xilinx Spartan 3 XC3S5000, package FG900 (ie. 900-pin), speed grade of -5 and the software packages used to synthesise the design and achieve the following results were Xilinx ISE 10.1 and Synplify 8.8.0.4:

```
Logic Utilization:
  Number of Slice Flip Flops:           970 out of  66,560    1%
  Number of 4 input LUTs:             1,920 out of  66,560    2%
Logic Distribution:
  Number of occupied Slices:          1,166 out of  33,280    3%
    Number of Slices containing only related logic:
                                      1,166 out of   1,166  100%
    Number of Slices containing unrelated logic:
                                          0 out of   1,166    0%
  Total Number of 4 input LUTs:       2,020 out of  66,560    3%
    Number used as logic:             1,664
    Number used as a route-thru:        100
    Number used for Dual Port RAMs:     256
      (Two LUTs used per Dual Port RAM)
  Number of bonded IOBs:                 40 out of     633    6%
  Number of RAMB16s:                     16 out of     104   15%
  Number of BUFGMUXs:                     1 out of       8   12%
Constraints cover 44747 paths, 0 nets and 8533 connections
Design statistics:
   Minimum period:   7.795ns   (Maximum frequency: 128.287MHz)
```

After successful synthesis, the core was downloaded onto a physical FPGA and verified to operate correctly under flood pings and regular ICMP/UDP activity. As covered in the Section on ICMP, the flood-ping test can be used to bombard a target machine with multiple ICMP (ping) packets to see how the network holds up under stress. Flood pings are very useful when testing the physical capability of the interface code, as it can show stresses from physical conditions in a way which cannot be easily simulated. Although it is easy to send a continuous stream of ICMP packets in a simulator, the results cannot be construed as "real world" because there are few variables to deal with. Even if the simulated code is found to support demanding situations such as flood pings appropriately, the code running on the FPGA might break down due to several factors:

- Attributes are added to the code so that various pins on the FPGA are matched with the relevant signals in the code. If any of these attributes are incorrect, the interface will malfunction.
- Computers can provide widely-different forms of packets with a wide range of flexibility. It is useful to test these in case certain conditions have not been addressed in simulations (eg. out-of-bound conditions, unusual payload/header information, etc)
- Any unforeseen circumstances

In the next Section, this dissertation will cover the means by which these tests were achieved, though the configuration of the computer to interface with the board as well as some C code used to send UDP data to the board.

Chapter 7

Computer-Side Operation

In this Section we assume the Gigabit Ethernet core has been loaded onto an FPGA demonstration board. Working with the core requires a reliable connection between the board and a computer. Although the board is technically capable of being connected to a switch or router, the lack of support for ARP means it will not be automatically detected by the rest of the network. We will not address the editing of a router's configuration to include manual ARP tables, but rather the simpler simpler situation of a direct connection between the core and a computer's network adaptor, which is ultimately the most likely use for the core.

7.1 Requirements

The computer needs to contain a network adaptor which supports Gigabit Ethernet, although this is a standard for virtually all current adaptors. The host operating system also needs to support UDP Lite, which is supported in Linux, Windows and OS X with the latest service packs/kernel. There is also a specification which, although not necessary for the board to work properly, is certainly desired in order to make maximum use of the board and its capabilities. Before it can be covered though, a brief explanation of MTUs is required.

7.1.1 MTU

The Maximum Transmission Unit (MTU) specifies, in bytes, the size of the largest packet that a network interface can support.[45] MTUs are important because they ensure that large chunks of data being transmitted can be cut down into more manageable units. They also ensure that if there is a lot of activity on a network, packets from different sources can still travel quickly to their destination without waiting for a particularly large packet to finish. MTUs are also extremely important for the success of any protocol which performs integrity checking such as TCP or UDP with custom error checking (eg. tags).

MTUs are generally specified alongside the specifications of a communications interface, in particular network adaptors. For fixed situations such as Ethernet, the MTU has a standard size normally set to a default by the interface, but can be modified to suit the requirements of the connection. The higher the MTU, the fewer packets needed to be sent for a given amount of data, which increases bandwidth efficiency and hence reduces the time required to shift a specific amount of data. However, large packets have the potential to congest the interface and reduce the flow of other packets in the system, which makes larger MTUs only suitable for connections with high bandwidth requirements.

When an MTU is specified, it is generally considered to be a *Framed MTU*. A framed MTU means the size only takes into account the payload data and not the header and checksum that are attached by the data link layer. For example, Windows will by default set an internal network to use an MTU of 1500 bytes. This is framed, because only the payload is 1500 bytes. The actual size of the transmitted packet (or the maximum frame length) when the header and checksum attached, is 1518 bytes. Coincidentally this default MTU is also the largest MTU possible for a regular Ethernet LAN, however larger MTUs are available if the LAN is Gigabit capable and supports *jumbo frames* (also known as *jumbo packets*). One of the advantages of Gigabit Ethernet is that it allows for much larger MTUs and hence payloads, per packet. The definition of a jumbo frame is any Ethernet frame with more than 1500 bytes of payload and can be increased to carry up to 9000 bytes by changing the network interface's MTU to 9000 bytes via the relevant method on the computer's operating system the card is installed under.[46] Jumbo frames are only possible with Gigabit Ethernet, but not all Gigabit Ethernet network interface cards support jumbo frames. The choice of network adaptor should be based on whether it supports jumbo frame and not just gigabit speed, as the benefits of having both are significant.

To show how MTUs affect the performance of a system, let's take the situation of a simple file copy operation between two computers over a network. Let's also assume the file is fairly large, around one gigabyte, to convincingly demonstrate show the significance of MTUs. Now, while the file is in the middle of copying, let's assume the user attempts to browse the Internet using a shared Internet connection originating from the other computer (ie. the same network adaptor is being used for both the copying and Internet operations). The following scenarios show the result of the operations depending on the MTU:

MTU SIZE = 1500 (DEFAULT FOR MOST SYSTEMS)

- File is transmitted using a huge number of relatively small packets
- Significant header & checksum information added to the total amount of data sent
- Web browsing is unaffected because the MTU is small enough to ensure any packets related to Internet requests are not held up by the major file copy operation. There is no noticeable lag.

MTU SIZE = 9000 (HIGHEST POSSIBLE SIZE FOR JUMBO PACKETS)

- File is transmitted using a low number of relatively large packets
- Header and checksum information is significantly lower in total quantity when compared to the previous scenario
- Since there are less header/checksum bytes as part of the overall transmission, the file copy will be completed faster than the previous scenario
- Web browsing is slow to download pages and slow to respond to links. Pages generally consist of a large number of small files with a significant number of small packets. The high MTU means the system is optimised for large continuous chunks of data, in this case, the copied file, but all the packets required for Internet communications are held up waiting for the current packet to clear and even then only a small number of packets can be processed before the next, very large packet for file copying resumes. The net affect can be so negative as to stall or timeout attempts to view sites.

A more quantitative example is shown as follows: assume a computer user wants to download a 1 megabyte file over the Internet. The effect that each of the above MTUs has on the download is as follows:

MTU SIZE = 1500

- Packet's payload size = 1500 Bytes
- Size of file to be downloaded = 1 MB = 1,048,576 Bytes
- Therefore, a minimum of 700 packets required to send data (1048576 / 1500 = 699.0507)
- Header component sent with each packet = 18 Bytes

- Therefore, 12600 additional bytes for all the headers are transmitted along with the payload (700 * 18)

MTU SIZE = 9000

- Packet's payload size = 9000 Bytes
- Size of file to be downloaded = 1 MB = 1,048,576 Bytes
- Therefore, a minimum of 117 packets required to send data (1048576 / 9000 = 116.5084)
- Header component sent with each packet = 18 Bytes
- Therefore, 2106 additional bytes for all the headers are transmitted along with the payload (117 * 18)

Moving from an MTU of 1500 to 9000 results in an 83% reduction in the amount of header data sent along with the downloaded file. This percentage remains the same regardless of how much data is transmitted, however the effect of less header becomes significant when shifted huge amounts of data. This is why it is important to use an MTU that is beneficial for the context of the data being processed.

The purpose of using and selecting an appropriate MTU in networking is to allow for multiple activities on the network without any significant congestion. However, given the exclusive access the interface expects when connecting the FPGA board and the computer, there is no need to worry about running various sizes of packets at once when communicating with the board, since only one activity would be in progress at any given time. So, by enabling jumbo packets and setting a high MTU, the interface is much more capable of dealing with the potentially large quantities of data it may need to process, as opposed to the default MTU assigned to it by the operating system.

One thing to make aware

7.2 Configuration

There are several steps involved before the board can be used by the computer. These include the mechanics of setting up a manual ARP entry which were covered in ARP protocol section but are elaborated further here. Step-by-step instructions are as follows:

1. If the network card supports jumbo frames, set the MTU to as high a value as is allowed. The method for doing so varies between operating systems. An MTU of at least 6000 is desirable.

2. Configure the host network card with a static IP address and subnet mask. Ensure the IP address is unique and doesn't conflict with any other devices on the computer. An example would be: IP - 192.168.0.1, Subnet Mask - 255.255.255.0

3. To ensure the UDP stack can handle the larger bandwidth requirements of the system, it is necessary to edit the operating system's buffer sizes to accommodate many UDP packets queued in the system. In Linux, the buffer sizes are specified in the files /proc/sys/net/core/rmem_default and /proc/sys/net/core/rmem_max. Sometimes editing these files directly is not possible due to the files being locked, so an effective solution to change the sizes is as follows: create a file called "size" which contains the value of the buffers desired, and then run the following commands in a terminal:

   ```
   cat size >> /proc/sys/net/core/rmem_default
   cat size >> /proc/sys/net/core/rmem_max
   ```

4. Connect the FPGA board to the computer via a suitable cable. Apply power to the board and observe the LED activity on both the computer's and board's network adaptors - if they flicker initially, this is a sign of a good physical connection. If possible, query the computer for the state of the connection before continuing - it should report a Gigabit link-up.

5. The ARP table needs to be configured before any communication is possible. Running the following command in a terminal will work:

   ```
   arp -s 192.168.0.10 00:00:00:00:00:01
   ```

   This command sets an ARP entry to link the device at IP 192.168.0.10 to the MAC address 00:00:00:00:00:01. The IP was chosen because it exists on the same subnet as the adaptor and the MAC address was chosen to be unique and not match the MAC address of any other network adaptors. Now, when data is sent to 192.168.0.10, the computer will automatically send the packets to the board without future ARP requests because it knows which network adaptor to use.

6. Attempt to ping the board:

   ```
   ping 192.168.0.10
   ```

   If the pings work correctly, then so does the board.

These steps constitute a thorough configuration of the computer in preparation for the board. For future use of the board on the same computer, most of these steps can be skipped or optimised. For example, steps 1/2 are normally stored in the operating system's configuration files so they don't need to be set later, step 4 can be entered into the system boot scripts so the ARP table is configured every time the computer starts, while the ping is not necessary for the most part, as its purpose is for testing rather than general use.

7.3 Operation

Sending data from the computer to the board can be accomplished by using a custom program to establish a socket connection between the two devices. Any program which supports UDP is capable of interfacing with the board, however for the purposes of the system it is a lot more prudent to use software which has been designed to work well with the board and so reacts in a way which is expected. There is also the added benefit of providing tags and other error-correction features into the code, rather than relying on off-the-shelf tools which may not understand how the core will operate.

To test the UDP and UDP Lite support of the core, a C program was written to send data to the demonstration board and wait for the board to send the same data back (ie. mirror mode). Running a *diff* on the input and output files would show how successful the board was in processing and returning the data and whether there were any issues with the connection. This Section covers important areas of operation though the use of code fragments and output, while the entire code of the program is listed in Appendix A.

- There are four important #define statements used at the beginning of the code:

```
#define SOCKBUFFSIZE  16777216
#define BUFFERSIZE    6144
#define PROTOPORT     4950
#define MILLSECS      5000
```

SOCKBUFFSIZE

The requested size of the receive/send socket buffers (bytes). Socket buffers used in the system need to be significantly large to avoid issues with lost packets. Unlike TCP which can queue packets when under congestion, UDP does not support such capabilities and if packets fill up system buffers without being cleared, the system is likely to simply drop them without reporting this. The importance of having a large socket buffer is shown when using the code to transmit several hundred gigabytes of data to the board. If the SOCKBUFFSIZE is too small, large packet loss will be reported and the socket is likely to stall and fail to respond until reaching the timeout. The size chosen above has been shown to support a 3 GB test data file with tagging enabled without failure. The SOCKBUFFSIZE value should be set as high as reasonably possible; there are no disadvantages to having a high buffer size as long as there is sufficient memory on the computer.

BUFFERSIZE

The size of the payload for each packet (bytes). Each packet will have exactly this amount of payload data; if the final packet does not contain enough data to fill the entire payload, zeros are padded to bring the payload to the required length. The BUFFERSIZE is dependent on the MTU of the network adaptor the board is connected to; the above example of 6144 bytes assumes jumbo frames are available and for testing purposes the MTU was set to 6200 bytes, as both the payload AND the header data need to fit in the MTU. The BUFFERSIZE value should be set as high as reasonably possible, provided the network adaptor can support it.

PROTOPORT

The destination port number. Packets require two port numbers for transmission - a source port and a destination port. The source port is determined by the operating system while the destination port is supplied by the program. The chosen port number does not matter when dealing with the board, as the board doesn't differentiate and instead accepts all packets sent to it.

MILLSECS

The receive socket timeout (ms). When the program is listening for incoming packets, it remains in a waiting loop. It will normally exit the loop and cleanly stop the program once all expected packets are received, however if there is a problem and packets have been lost or otherwise are never received, it may never exit the loop as it waits for packets that will never appear. To provide for a clean exit which saves unwritten disk buffers and returns allocated memory, the program needs to be capable of terminating the loop if it appears likely no more packets will be received. The MILLSECS value provides the amount of time the program waits for a packet before giving up.

- The line

```
pstrzWhichProtocol = "udp";
```

specifies which protocol the sockets will use. There are two self-explanatory options - “udp” and “udplite”.

- The program, if run without any parameters, will display the following message:

```
Usage:  client_send <destination> <inputfile> <outputfile>
```

<destination>

The host the program will connect to. For the purposes of the system, the destination will be the IP/name of the board.

<inputfile>

The file to be sent to the board. The content of the file can be anything, in any form (binary, ASCII, etc).

<outputfile>

The file which the program will write the returned data to. Assuming all components in the system are working correctly and all configuration stages have been performed, the output file should be a facsimile of the input file. An easy way to determine the integrity of the output is to run a *diff* on the two files; a zero output means the files are identical and so the board is working perfectly.

- The line

```
if (fork()==0) {
```

begins running a block of code which performs the sending of packets to the board as a threaded process. What this means is that the moment the sender code is instantiated, the main execution of the program moves onto the code after the fork, which happens to be the receiver, so the sender and receiver work simultaneously as required.

- The following code performs the actual sending of packets, located in the forked processes, with tags embedded:

```
while (lDummy > 0) {
   sprintf(tag,"%.3c",iSentp);
   strncpy(tagbuffer, tag, 3);
   tagbuffer += 3;
   memcpy(tagbuffer, lpDummy, sizeof(buf)-3);
   tagbuffer -= 3;
   if (lDummy < sizeof(buf)) {
      padding = sizeof(buf) - lDummy;
      tagbuffer += lDummy+3;
      memcpy(tagbuffer, zerobuf, padding);
      tagbuffer -= lDummy+3;
   }
   iSentp++;
   lSent   += (long)send(sd, tagbuffer, sizeof(buf), 0);
   lpDummy += (long)sizeof(buf)-3;
   lDummy  -= (long)sizeof(buf)-3;
}
```

The sender keeps transmitting packets so long as lDummy (the remaining number of bytes to be sent) is greater than zero. The *sprintf* and *strncpy* instructions at the beginning of the code first convert the value of *iSentp*, which holds the number of packets sent, into a character type and store it into the tag variable, which is then copied into the tagbuffer memory buffer. *iSentp* is used as the basis for the tag value since it starts at zero and increments by one for every iteration of the while loop (ie. every packet sent). It is converted into a character type from an integer, as the character type provides a greater range of possible values to be held in the three bytes allocated for the tag. Assuming the use of jumbo packets with a payload size of 6144 bytes (as defined in the first bullet point) and assuming each byte used for the tags stores a range of values from 0-255, this would result in the maximum value of the tag to be 16,777,215. With the same jumbo-sized packets and payload size, this provides a maximum data size of 103,079,208,960 bytes (more than 100 GB). It's worth noting that the tags are only incremented for each individual file and start from zero once a new file is sent.

After the tag is copied into the tagbuffer, the code makes a copy of the actual payload to the tagbuffer, but performing a shift on the buffer so that the first three bytes are left for the previously-copied tag and the remainder is comprised of the payload. The payload for each packet is comprised of the input file data split into chunks, each chunk BUFFERSIZE bytes large. The "if (lDummy < sizeof(buf))" statement means the following: if the remaining number of bytes to be sent is *less* than the size the payload is required to be, pad the unused payload space with zeroes. This is to ensure each packet has an identical size, for reasons of simplicity. If the padding is necessary, a block of zero values is copied from the end of the remaining packet data to the end of the payload from a pre-zeroed buffer.

Once the payload is prepared for sending, the actual send code is executed. First, the *iSentp++* statement increments the respective variable, for use in keeping track of the number of packets sent. Next the line

```
lSent   += (long)send(sd, tagbuffer, sizeof(buf), 0);
```

performs a socket send to the board of the contents of the tagbuffer, which contains the prepared payload. The next two statements perform some important modifications to variables which track the progress of the data being sent. If at the end of the final statement the value of *lDummy* is zero or a negative number, this means there is no more data to be sent and so the sending process is completed.

- The following code performs the receiving of packets from the board:

```
while (1) {
   n = (long)recv(sd, llbuffer, sizeof(buf), 0);
   if (n < 0) break;
   strncpy(tagrecv, llbuffer, 3);
   llbuffer += 3;
   memcpy(buffertowrite, llbuffer, sizeof(buf)-3);
   llbuffer -= 3;
   sprintf(runningtags, "%.3c", runningtag);

   if (strncmp(tagrecv,runningtags,3)) {
      fprintf(stderr, "\nERROR: Packet received with tag #%s, expected #%s, %d\n",
          tagrecv, runningtags, runningtag);
      free (llbuffer);
      free (lpDummy);
      free (tagbuffer);

      fclose(pFileOut);
      closesocket(sd);
      exit(1);
   }
   runningtag++;
   lEchoed += n;
   fwrite(buffertowrite,1,n-3,pFileOut);
}
```

As the receiver is running at the same time as the forked sender process, it is difficult to know exactly when to stop waiting to receive packets. For this reason, the receiver code is surrounded by an infinite while loop and will only stop waiting for packets once the receive statement reports a timeout. The line

```
    n = (long)recv(sd, llbuffer, sizeof(buf), 0);
```

waits for a specific amount of time for a packet to be sent from the board. The amount of time it allows for this is specified by MILLSECS. If this amount of time expires without a packet being received, then n is set to a value of -1 and the if statement immediately after causes the receiver code to break out of the while loop. So, as long as packets are continually being sent by the board within the time specified by MILLSECS, the while loop ensures the receiver will pick them all up, but the moment this timeout is reached, the receiver will quit.

Chapter 8

JPEG2000 Core

The connectivity of the board serves the purpose of providing a communications link between the computer and an application running on the board. An an example of a real-world application of the system, let us assume the board is to be used as part of an encoding/decoding scheme for the implementation of the JPEG2000 image compression format. Figure 8.1 shows a conceptual diagram of how the interface and core program could reside on the FPGA. As the space required by a core program can vary, the diagram is not a guaranteed representation of the physical profile of the software running on the FPGAs. Instead, it is used to show the comparative sizes between how much space is required by the MAC layer (INTERFACE) and the core program (JPEG2000 WAVELET TRANSFORM). Reducing the profile of the interface increases the maximum size available for the core program, reinforcing the need to have a quick and efficient implementation of the IP layer with adding unnecessary bloat.

To explain how the board can be used in this example, it is first necessary to cover what JPEG2000 is, how it works and how the board can be used to service the image format.

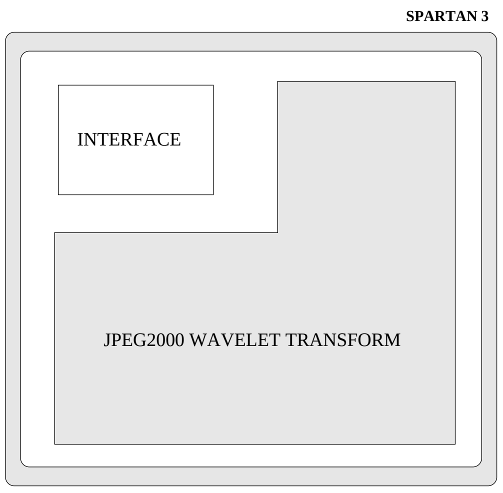

Figure 8.1: Space occupied by HDL programs on the FPGA

8.1 History of JPEG/JPEG2000

Ever since digital images have existed, there has been a need to develop ways to compress digital images into a limited amount of storage space. Quality compression techniques are highly desirable, since they allow for greater numbers of images per storage unit, as well as making better use of available bandwidth if images are sent over a network. In 1992, the Joint Photographic Experts Group introduced an image compression standard called JPEG,[47] which to date is the most commonly used and known compression format, particularly with digital cameras and the Internet. However, although the format can provide reasonable image quality at average levels of compression, the image quality becomes increasingly inferior as the level of compression increases. At low bit rates this would render the JPEG format virtually useless as a compression technique. The JPEG committee saw room for improvement and several years ago a new image compression standard named JPEG2000 was released.[48] While JPEG2000 improves upon JPEG in a number of areas, the most relevant to this project is the fact that is offers significantly better compression than its predecessor. The compression technology used in JPEG2000 is such that even at low bit rates the image quality is still reasonably acceptable, compared to what the original JPEG format could produce.

The JPEG2000 standard, although not as widespread as regular JPEG, has seen use in areas not originally reserved for the format. For example, there has been interest in using JPEG2000 to store films for the movie industry. *Digital Cinema* as it is known, is an example of a idea to encode individual frames of a film using JPEG2000 along with encryption technology to avoid problems with leaks of the electronically-encoded film.[49] The Digital Cinema Initiatives (DCI) has worked to publish a list of specifications for digital cinema, agreed upon by the major studios. The specification uses JPEG2000 for picture encoding and use of the CIE XYZ color space at 12 bits per component encoded with a 2.6 gamma applied at projection.[50] As of October 2007, there are over 5000 DLP-based Digital Cinema Systems installed.[51]

JPEG2000's main advantage over JPEG is the use of the Discrete Wavelet Transform (DWT) in the compression algorithm. JPEG uses the Discrete Cosine Transform (DCT), which does not allow for compression ratios as high as the DWT without significant degradation in image quality. The use of the DWT in JPEG2000 enables the compression algorithm to achieve much better compression than JPEG without any noticeable degradation in image quality than JPEG. The primary disadvantage of JPEG2000 is due to the heavy calculations required to decompress an image, making is noticeably slower than regular JPEG. Since JPEG2000 is an such intensive format to process, software implementations are entirely inappropriate as they would be unable to decode super high resolution images at 1/24 second speeds. There are continuing developments into improving the efficiency of the JPEG2000 wavelet transform code,[52] however since these techniques require either a full or partial hardware solution, the board would be perfect for such an application.

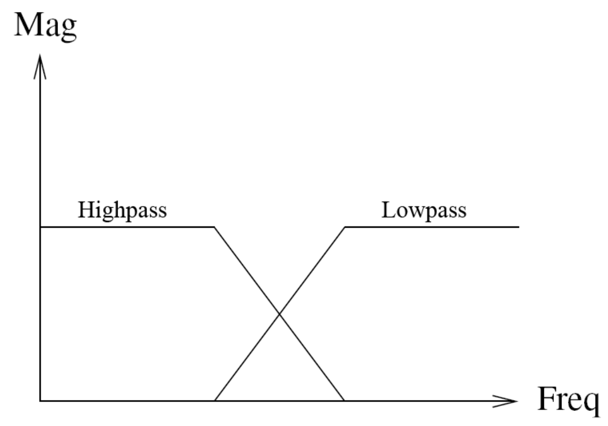

Figure 8.2: High and low pass wavelet filters

8.2 The Wavelet Transform

Core to the operation of JPEG2000 is the wavelet transform. The wavelet transform is a filtering operation in which a sequence of values are high and low pass filtered. The high and low pass filters are derived from the wavelet along with a scaling function for the particular wavelet transform being performed. Different filters are used for different wavelets and since there are an infinite number of wavelets there are an infinite number of wavelet transforms. Not all of them are useful however, but there are two useful wavelets which have been specified in the JPEG2000 image compression standard: one for *lossy* compression and one for *lossless* compression. In general the filters have a frequency response similar to that shown in Figure 8.2.

For one-dimensional signals the transform is performed from the beginning of the signal to the end. However for two-dimensional signals the transform must be performed on both the rows and the columns, with the rows transformed first and then the columns. Digital images are a prime example of such a signal, which is why the JPEG2000 employs the wavelets on image data by separating the pixel information into two sets: a sequence of values to denote the rows (horizontal components) and

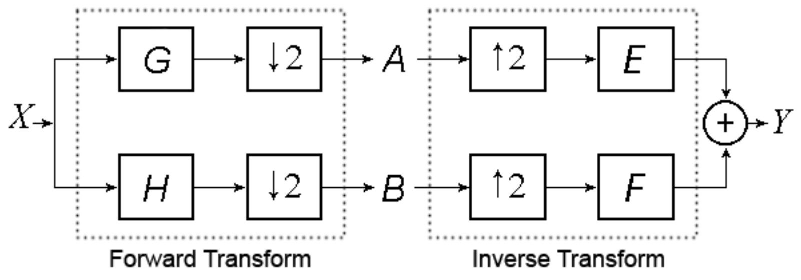

Figure 8.3: One-stage wavelet filter bank

a sequence of values to denote the columns (vertical components) of the image data. After filtering, all the low pass values are placed at the beginning of their row/column and all the high pass values at the end, a process called deinterleaving.

Figure 8.3 shows a one-stage filter bank which presents a simple approach to wavelet signal decomposition and reconstruction. The bank takes a signal X and performs a forward wavelet transform on it by sending it though a high pass filter G and a low pass filter H. The output of filter G is a sequence of high pass values and the output of filter H is a sequence of low pass values. Each set of values is then run through a downsampler which reduces the size of each set of values by two. The downsampling by two is often referred to as *decimation by two*. The processed sets of values are listed as A and B, and if we choose to combine the two outputs into a single signal we can say the bank has produced a *single octave* or *first pass* transform. To show the inverse transform however, the outputs are not combined but instead fed into the next set of filters. The outputs are first sent into a separate upsampler for both (*interpolation* by two), which brings the size of each set of values back to the lengths of the original signal, then the inverse wavelet transform is processed on each set of values by another set of high pass and low pass filters, denoted as E and F respectively. Once processed, each set is recombined with the final signal represented

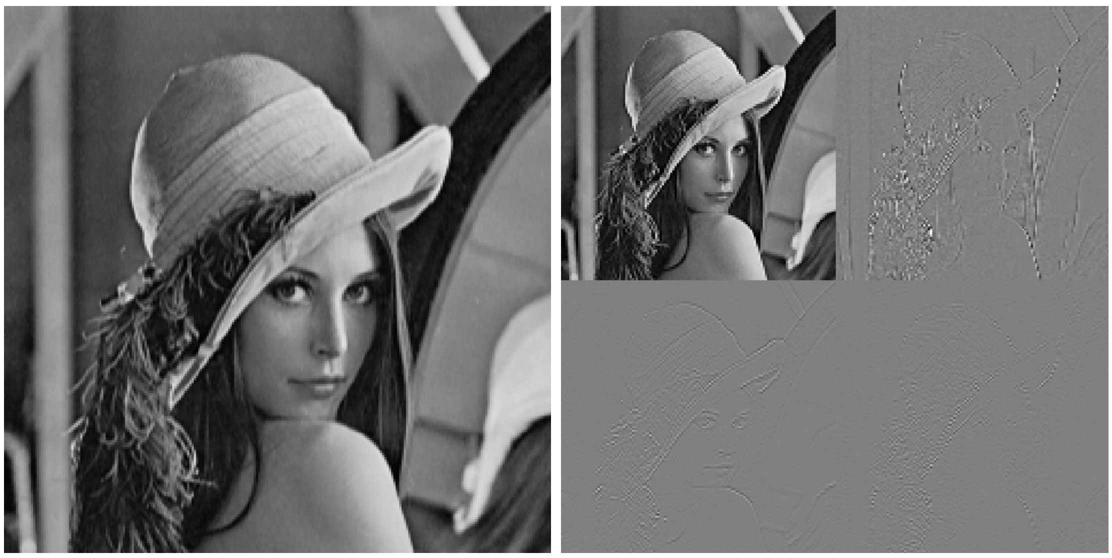

Figure 8.4: Left - Original uncompressed image; Right - One octave wavelet transformed image

as Y. If signal Y is identical to X, then the wavelet transforms used in the filter bank are said to provide *perfect reconstruction*.

As an example of how wavelets operate in JPEG2000, we shall use the traditional Lena image to represent an uncompressed image and send it through a one octave wavelet transform. The process of passing the image through a one octave wavelet transform is shown in Figure 8.4, along with the original untransformed image as a comparison.

The process of de-interleaving has placed all of the filtered low pass values at the start of their respective row/column resulting in a low pass-low pass (HH) sub-band in the upper left quadrant of the image. The de-interleaving process also produced an entirely high pass (GG) sub-band in the lower right quadrant and two sub-bands (GH and HG) which contain a combination of the high pass and low pass components. The information in the GH sub-band contains the vertical edges of the image, the

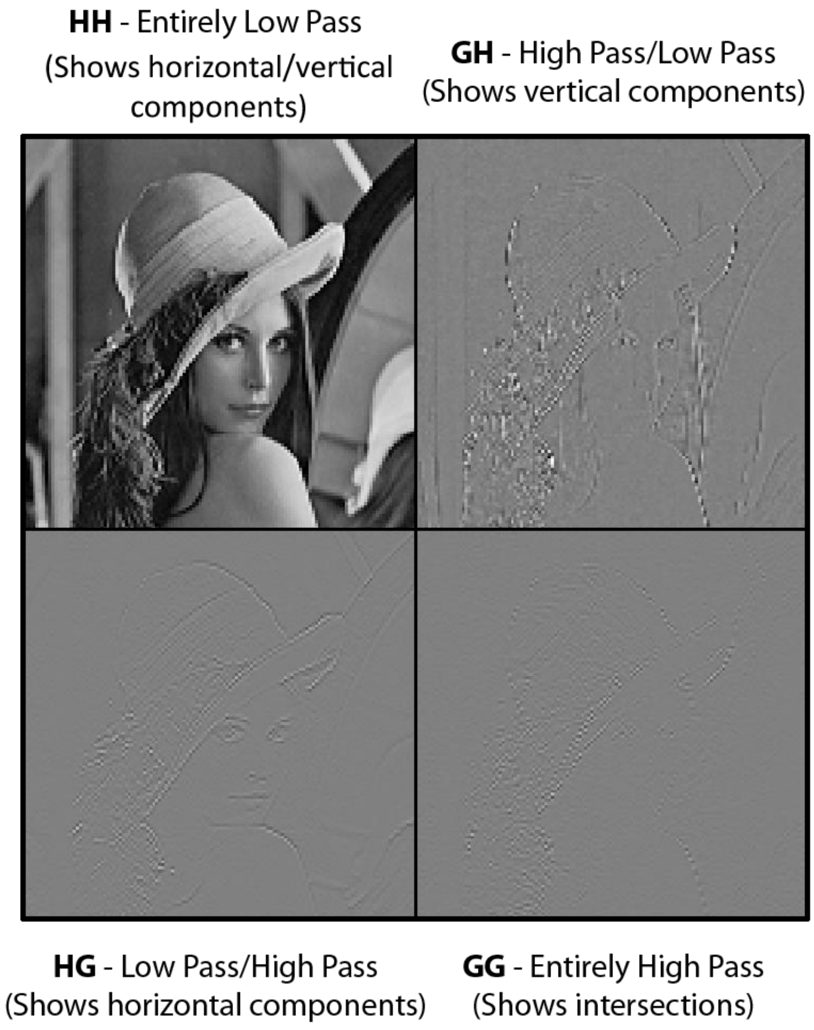
HH - Entirely Low Pass
(Shows horizontal/vertical components)
GH - High Pass/Low Pass
(Shows vertical components)
HG - Low Pass/High Pass
(Shows horizontal components)
GG - Entirely High Pass
(Shows intersections)

Figure 8.5: One octave wavelet transform - quadrant contents

information in the HG sub-band contains the horizontal edges and the GG sub-band is a combination of the two. The specifics of each quadrant are shown in Figure 8.5.

If the transformed output is itself passed through the wavelet transform, it is possible to obtain further compression of the image. Instead of having just a one octave transform as before, we can obtain multiple octaves of the wavelet transform in an almost identical process to performing a one octave wavelet transform. This process is known as *dyadic decomposition*. To perform a second octave of the wavelet transform, the HH sub-band from the first octave is used as the input. The size of the HH sub-band is half the size of the original image, so whatever method is used to perform the wavelet transform must take into account the shortening input lengths as the number of octaves increases. Figure 8.6 shows the Lena image after it has been processed to four octaves. After the wavelet transform has been performed with the desired number of octaves, the final HH sub-band contains all the information required to reconstruct the original image. By performing the equivalent number of octaves of the inverse wavelet transform on this HH sub-band, the original image can be reconstructed. For every octave of the forward transform the image size is halved, so by performing the inverse transform to reconstruction the original image, each octave of the inverse transform results in the image being doubled. Once all octaves of the inverse wavelet transform are processed, the image returns to the same size as the original.

As mentioned before, JPEG2000 employs two wavelet filters to accomplish two forms of compression - lossy and lossless. The JPEG2000 standard has chosen two specific wavelets to support the compression - the lossy compression is achieved using the *Daubechies 9/7* wavelet, while the lossless compression is achieved using the *LeGall 5/3* wavelet. The lifting equations for both of these wavelets are listed in Figures 8.7 and 8.8 respectively (the lifting scheme is covered in the next section). Lossy compression is akin to how regular JPEG works; the compression is achieved mostly by removing data from the digital image, specifically the data that's considered less important to the quality of the image. As the compression level is increased, the quality of a JPEG/JPEG2000 image degrades as the encoder has to work more aggressively, although the degradation is less pronounced in a JPEG2000 file due to improved lossy compression algorithms. Lossless compression however is a method of compression which retains all the data of a digital image without throwing away detail, while still managing to achieve a reasonable level of file size reduction. JPEG does not have a

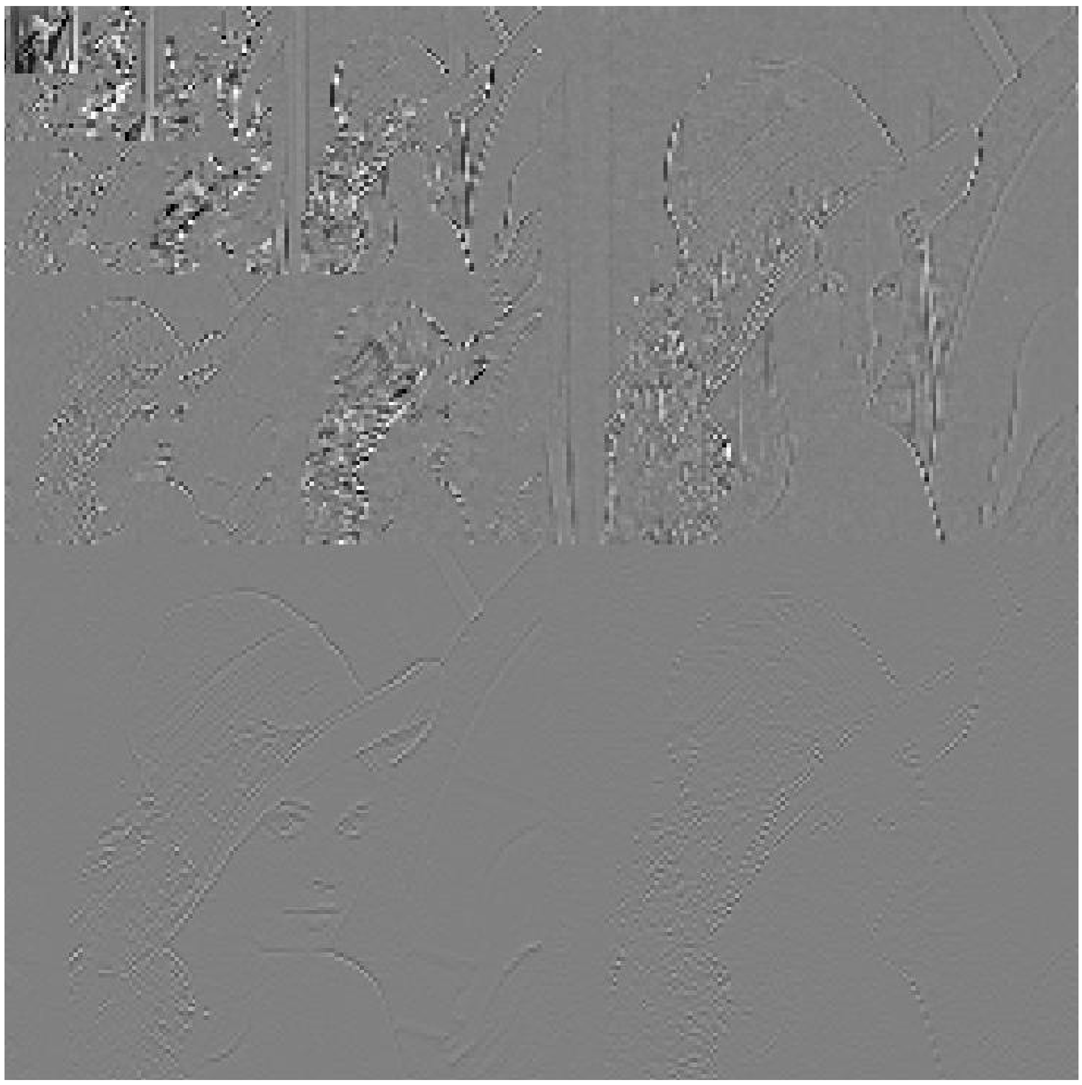

Figure 8.6: Four octave wavelet transform - quadrant contents

true lossless compression technique, although by reducing the compression to minimal levels the effect can be similar, however JPEG2000 has full lossless compression capabilities.

Out of the two wavelets the Daubechies 9/7 lossy compression is the most desirable for practical situations since the compression it yields far outweighs that of the LeGall 5/3 lossless, plus unlike JPEG, the lossy compression is a lot less degrading on the quality of the final image. However, Daubechies 9/7 is a lot more complicated than LeGall 5/3 mathematically. It processes a high and low pass transform on the entire sequence and the multiplication involved results in a slower speed of encoding and decoding. It is for this reason the system provides a good base to construct a hardware implementation of the Daubechies 9/7 wavelet transform, using the computer to supply either raw image data or an existing JPEG2000 file and the board to encode/decode the content respectively and transfer the resultant back to the computer, all communication performed using the Gigabit Ethernet link.[53]

8.2.1 Lifting Scheme

There are two methods for performing wavelet transform calculations. The first is *convolution*, which was the original method for performing the transform. Convolution requires a significant number of calculations and memory for the storage of intermediate values,[54] which does not lend itself well to high speed and/or low powered applications and hardware and this is especially the case with embedded devices where these are often critical requirements. Eventually a new, more efficient method for implementing the wavelet transform was produced, called the *lifting scheme* or just *lifting*. For a sequence of values, the wavelet transform via lifting involves modifying each value based on its neighbouring values. The lifting equations used in JPEG2000 are listed in Figures 8.7 and 8.8.

There are several advantages lifting has over convolution. Lifting is very desirable to use when performing transforms in hardware, because the lifting steps can be implemented very easily and efficiently due to how FPGAs operate. Lifting operations consist of steps which can be naturally implemented in hardware, requiring significantly less gates to implement than any convolution solution could achieve. Another advantage of lifting is that it can be performed *in-place,* meaning no additional memory is required to store intermediate values during the lifting operation; the modified values are simply written back over the old values. Old values can then be recovered using the inverse lifting procedure.[55] Less gates will in general result in a higher maximum clock speed, as well as reducing the profile of the program on the FPGA to allow for more code elsewhere. Less memory also reduces the program's profile on the FPGA if the memory was created on-chip, otherwise external memory is saved for other uses. Overall a faster, lower powered solution can be achieved using lifting when compared to convolution, so we will use it to provide an example of the way in which application code can be implemented along with the IP core on an FPGA.

Figures 8.7 and 8.8 show the forward and inverse equations for the two wavelets used by JPEG2000 along with the scaling factors for the Daubechies 9/7 equations. The *ext* subscript notation indicates symmetric extension must occur at the boundaries of the signal, an explanation of which is in the next paragraph. The brackets that appear without a notch on their tops in Figure 8.8 mean "integer part of". For this reason the LeGall transform can only be performed on integers values, as the brackets mean "always round down to the nearest integer". The integer to integer characteristic of the LeGall transform enable the perfect reconstruction of images using the inverse

Forward

Step 1 $Y(2n+1) \leftarrow X_{ext}(2n+1) + \left(\alpha \times \left[X_{ext}(2n) + X_{ext}(2n+2)\right]\right)$

Step 2 $Y(2n) \leftarrow X_{ext}(2n) + \left(\beta \times \left[Y(2n-1) + Y(2n+1)\right]\right)$

Step 3 $Y(2n+1) \leftarrow Y(2n+1) + \left(\gamma \times \left[Y(2n) + Y(2n+2)\right]\right)$

Step 4 $Y(2n) \leftarrow Y(2n) + \left(\delta \times \left[Y(2n-1) + Y(2n+1)\right]\right)$

Step 5 $Y(2n+1) \leftarrow -K \times Y(2n+1)$

Step 6 $Y(2n) \leftarrow \left(\frac{1}{K}\right) \times Y(2n)$

Inverse

Step 1 $X(2n) \leftarrow K \times Y_{ext}(2n)$

Step 2 $X(2n+1) \leftarrow -\left(\frac{1}{K}\right) \times Y_{ext}(2n+1)$

Step 3 $X(2n) \leftarrow X(2n) + \left(\delta \times \left[X(2n-1) + X(2n+1)\right]\right)$

Step 4 $X(2n+1) \leftarrow X(2n+1) + \left(\gamma \times \left[X(2n) + X(2n+2)\right]\right)$

Step 5 $X(2n) \leftarrow X(2n) + \left(\beta \times \left[X(2n-1) + X(2n+1)\right]\right)$

Step 6 $X(2n+1) \leftarrow X(2n+1) + \left(\alpha \times \left[X(2n) + X(2n+2)\right]\right)$

$$\boxed{\begin{array}{l} \alpha = -1.586134342 \\ \beta = -0.052980118 \\ \gamma = 0.882911075 \\ \delta = 0.443506852 \\ K = 1.230174105 \end{array}}$$

Figure 8.7: Daubechies 9/7 Wavelet Transform Equations (Lossy Compression)

Forward

Odd Step — $Y(2n+1) = X_{ext}(2n+1) - \left\lfloor \frac{X_{ext}(2n) + X_{ext}(2n+2)}{2} \right\rfloor$

Even Step — $Y(2n) = X_{ext}(2n) + \left\lfloor \frac{Y(2n-1) + Y(2n+1) + 2}{4} \right\rfloor$

Inverse

Even Step — $X(2n) = Y_{ext}(2n) - \left\lfloor \frac{Y_{ext}(2n-1) + Y_{ext}(2n+1) + 2}{4} \right\rfloor$

Odd Step — $X(2n+1) = Y_{ext}(2n+1) + \left\lfloor \frac{X(2n) + X(2n+2)}{2} \right\rfloor$

Figure 8.8: LeGall 5/3 Wavelet Transform Equations (Lossless Compression)

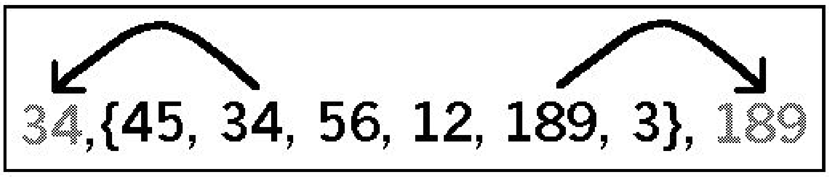

Figure 8.9: Symmetric extension at the boundaries

transform process. This is why the LeGall transform is known as a lossless transform. Daubechies on the other hand does not contain any integer part brackets in the equations and indeed the scaling factors specified to nine decimal places indicates this is instead a floating point transform, which as a result turns it into a lossy transform. By using floating point, computation of values can be increased to a very high degree of accuracy, but at a cost of not being able to completely reproduce all the original values to their original level of precision during the inversion process. This results in a small amount of information being lost due to rounding errors, although in practise the visual impact of such lost information in JPEG2000 is negligible.

The lifting equations involve modifying values of a sequence based on the neighbouring values. That is, for a value in an even (lowpass) position, it is modified based on the values in the neighbouring odd (highpass) positions and vice versa. The only problem with this arrangement is what to do for the values at the start and the end of a sequence, which only have one neighbouring value. The solution is to use the one neighbouring value as both neighbouring values at the beginning and end of a sequence. This is known as symmetric extension at the boundaries, an example of which can be seen in Figure 8.9 and shows how the extra neighbour is obtained for the values at each end of a signal.

Symmetric extension is easy to implement in a hardware design, because control signals are set to tell the lifting blocks when symmetric extension needs to be performed, with those signals read in each lifting block and the equations modified as necessary. Lifting was very important to the development of the wavelet transform, because it provided the ability to implement wavelet transforms in the real world at realistic speeds and so lifting was incorporated into the JPEG2000 standard. Lifting has simplified the implementation of the wavelet transform and made a hardware implementation far more desirable.

8.2.2 Pipelining the Design

In a sequential (ie. non-pipelined) architecture, the lifting would occur in a single logical block, passing transformed values internally until the results appear at the end of its operation. Although the system works, it is inefficient because the lifting equations provide a distinct advantage when implemented in hardware - several equations can be run at the same time, working on separate sets of data. However, this advantage can only be exploited by designing the lifting to work in a pipelined architecture, since a sequential design with the single logical block results in only one lifting stage running at any given time. When pipelined, the lifting can be split into several logical blocks instead, each running as required.

As a practical example of the benefits of pipelining, let us observe the lifting equations working in a pipelined system. The waveform in Figure 8.10 shows a system where there exists an array of eight values to be wavelet transformed via lifting. These values are shown individually in the waveform as **x0** to **x7**. The lifting calculations are split into four lifting blocks (stages) instead of using a single large logical block. Every time the clock signal hits a rising edge (ie. goes from low to high), three of the values are taken and processed by a particular lifting stage, with the output for

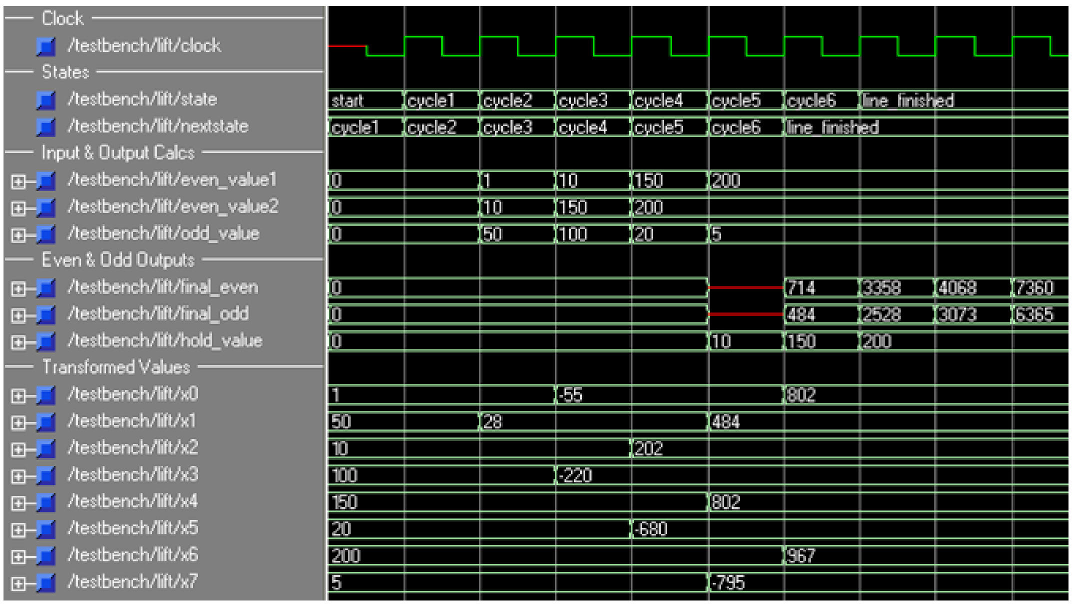

Figure 8.10: Pipelined execution of the lifting system

---

the stage appearing during the next rising edge of the clock. For example, during the first rising edge transition (where state has the value "cycle2"), **x2, x1** and **x0** are taken by a lifting stage called *Odd Step 1*. The activity of this step is shown in Table 8.1 and the equation for Odd Step 1 is shown in Figure 8.7 as **Foward: Step 1**. During this stage the output value of 28 is calculated and written to the existing value of **x1**, but due to the output signal passing through a flip-flop this value is not updated until the next rising edge of the clock, which is seen during cycle2.

It can be observed that once the first lifting stage its output, the two subsequent stages (Odd Step 1 again and now *Even Step 1*, also shown in Figure 8.7 as **Foward: Step 2**) calculate their respective values (-55 and -220) at the same time in the next clock cycle instead of just one stage operating in the cycle, as was the case with the serial execution. Further on it can be seen that three stages produce values in a single clock cycle (484, 802 and -795). After these blocks provide their own return values from separate calculations, the cycle continues to the point where every lifting stage is running at the same time as all others. Once the majority of calculations are

Table 8.1: Timing chart for each lifting stage with a sequence of eight values

Clk Cycle	Odd Step 1	Even Step 1	Odd Step 2	Even Step 2
0	x2, x1, x0 -> x1			
1	x4, x3, x2 -> x3	x1, x0 -> x0		
2	x6, x5, x4 -> x5	x3, x2, x1 -> x2		
3	x8, x7, x6 -> x7	x5, x4, x3 -> x4	x2, x1, x0 -> x1	
4		x7, x6, x5 -> x6	x4, x3, x2 -> x3	x1, x0 -> x0
5			x6, x5, x4 -> x5	x3, x2, x1 -> x2
6			x8, x7, x6 -> x7	x5, x4, x3 -> x4
7				x7, x6, x5 -> x6

performed, the number of simultaneous stages drops off until the system completes its calculations. Due to the number of stages operating in parallel at any given time, significantly fewer clock cycles are required to produce all the necessary results using pipelining than it would if using serial execution. Figure 8.11 shows the hardware block diagram for the lifting stages using pipelining, along with extra components such as the RAM which stores the various output values from the latter stages as well as the Finite State Machine which organises the entire lifting process.

Table 8.1 shows how wavelet transform code processes a sequence of eight values, the situation present in the system. Note that for *Odd Step 1* and *Odd Step 2*, **x8** is assigned the value of ***x6*** for the purpose of symmetric extension and for the initial iteration of *Even Step* 1 and *Even Step 2*, the third value not shown is assigned the value of **x0**, also due to symmetric extension.

Take note that the waveform in Figure 8.10 contains calculations which return integers, despite the fact that the Daubechies 9/7 wavelet is a floating-point transform. In the hardware design of JPEG2000, the decision was made to use fixed-point approximations to the floating point numbers instead of using floating point directly, as

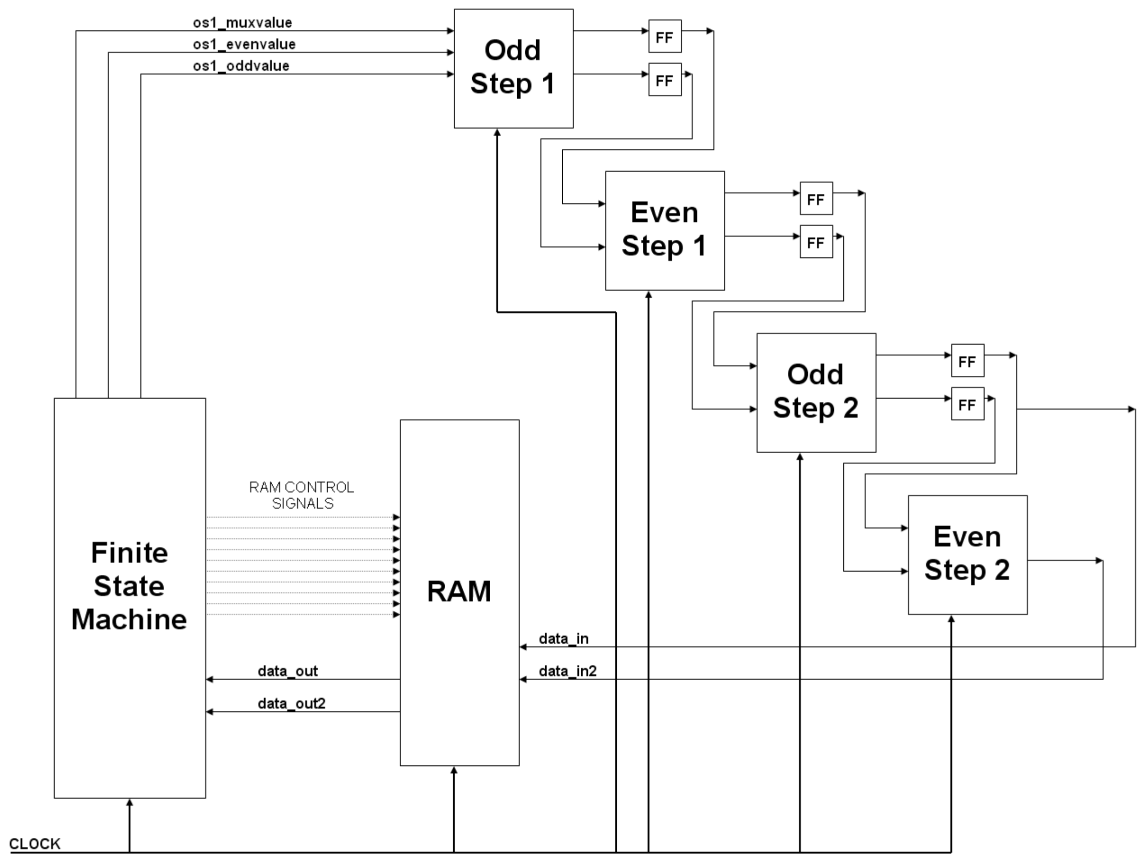

Figure 8.11: Hardware lifting blocks in a pipelined architecture

calculations on the fixed-point approximations would perform significantly faster and require less logic on the FPGA. To accomplish this, the scaling factors were multiplied by 256 to obtain a value which could easily be stored as an integer. For example, Odd Step 1 requires the use of the alpha coefficient which is -1.586134342. This value can't be stored and utilised in hardware, so the coefficient was instead hardwired in the code as -406, a value obtained from the calculation -1.586134342 * 256. After the lifting block produces its result, a rounding down calculation is performed to divide the result by 256 and throw away the values after the floating point so that the result can be stored as a floating-point value. The rounding down calculations utilise signals with a high number of bits to ensure a satisfactory level of accuracy for the final result. Scaling is then performed to finalise the calculations as required by the JPEG2000 standard.

8.2.3 Connectivity

Figure 8.12 shows how the various components of the system interact in this application.

A basic explanation of the operation of the entire system for a practical case is shown below:

- The computer, using UDP Lite packets, sends data in the form of uncompressed raw image data or a JPEG2000 image, for encoding or decoding respectively.
- The network interface takes the packets and layers the data into FIFO1. Once all the data has been sent the FIFO can be accessed by the JPEG2000 wavelet transform application on the board.

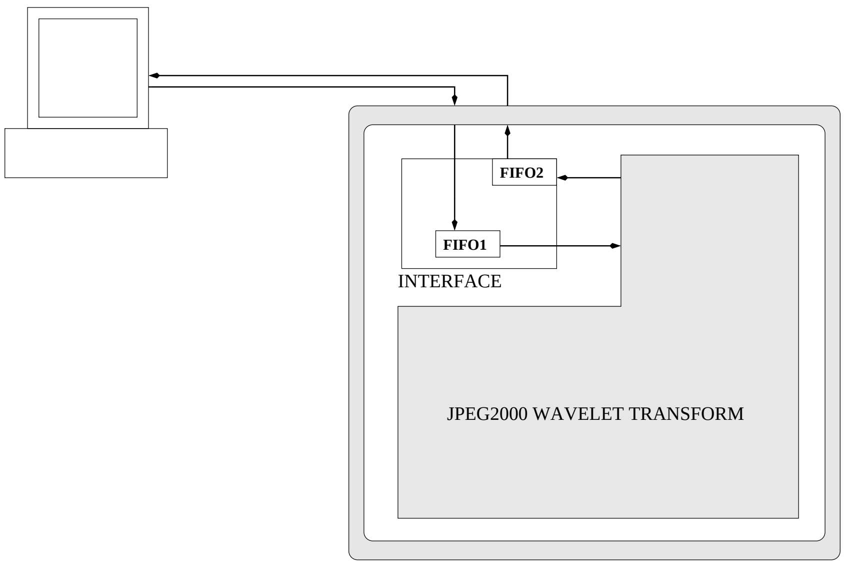

Figure 8.12: Inter-connectivity between system components

- The JPEG2000 wavelet transform performs the encoding or decoding as appropriate. When the program is ready to send the output to the computer, the application and the network interface work in tandem. The JPEG2000 application places individual chunks of data into FIFO2 and activates fsm_packgen to begin preparation for transmission of that particular packet. Once it's transmitted, the application supplies another chunk and activates fsm_packgen for the next packet and so on until all the data has been sent to the computer.

- The computer receives the packets sent by the board, assembles them and performs any remaining tasks, whether they be to save the output to a file and/or send another block of data for processing.

This example assumes the RAW/JPEG2000 images are sent in their entirety before being processed by the wavelet transforming code. Although provided for simplicity in explanation, this not need be the case. Due to the pipelined architecture of the system, several packets can be processed at once, so as long as the core program (in this case the JPEG2000 wavelet transform) is designed appropriately, the JPEG2000 code can read several packets to form an incomplete image and process them there and then. The processed data can then be sent back to the computer while the computer is still sending packets from the original image data. The flexibility of such a task is dependent on the core program being used - some programs might be able to process partial data streams while others might need a data block in its entirety before it can be processed. Either way, the capability is there.

Chapter 9

Conclusion

The purpose of this project was to develop an IP core which could provide high-speed, high-bandwidth computer to FPGA and back communication. This was achieved through the implementation of three Finite State Machines to handle the reception of packets from a computer, processing of new packets, and transmission of these new packets back to the computer. The core was designed to work on a low-power Xilinx Spartan 3 FPGA, which makes it very useful for low-cost designs which require some form of fast connectivity with a computer. A key design choice for the core's architecture has been to pipeline all the data in FIFOs, RAMs and flip-flops to achieve full Gigabit line speeds. The resulting implementation is therefore fast, efficient, robust and stable. As the requirements were strict on speed and efficiency, only protocols and functionality that are strictly necessary to fulfil these requirements were implemented. Several compromises had to be made, such as a lack of TCP and IPv6 support, but these compromises have resulted in the core only occupying a small number of slices on the Spartan 3 FPGA - 1,166 slices out of 33,280, a total utilisation of 3%. This is important since the greater amount of available logic on an FPGA, the larger the programs which perform the practical tasks can be. The core has a maximum operating frequency of 128.287 MHz, which satisfies the nominal Gigabit line speed of 125 MHz and allows for a small amount of flexibility, should there be variations in this speed.

9.1 Future Improvements

9.1.1 ARP Support

One potential improvement would be to incorporate support for ARP, so that manually adding an ARP entry for the system would not be necessary. This would therefore provide a much simpler and foolproof method of using the board, as it could be treated like any other networked device. Implementation of ARP would also improve the support of the board when connected to a larger network, as routers/switches wouldn't have to manually configure ARP entries to identify the board.

9.1.2 IPv6 Support

The IP core currently only supports Internet Protocol Version 4. This is suitable for most situations in which the core may find itself used, however it would be beneficial to support IPv6 for reasons of future proofing. One of the most noted advantages of IPv6 over IPv4 is the larger address space, which means there can be a significantly larger number of devices attached to an IPv6 network than its predecessor. The core is currently more suited to the context of a direct connection between an FPGA and a computer, which makes this advantage not quite so attractive. However, if the core is eventually used on a device which connects to the Internet it will be very important to support IPv6, so that it can process packets sent to it as the uptake of IPv6 on the Internet continues to grow. All modern operating systems now support IPv6 so there are no limitations on the computer side of the implementation.

9.1.3 TCP support

Adding TCP to the system would have limited use with this core, since TCP is most useful for systems which require reliable data transfer over an unreliable medium. Since the core would generally be connected to a local network or directly to a computer, the extra reliability provided by TCP would not normally justify the added complexity and expense of implementing a full TCP stack. To the author's knowledge, there are currently no full custom TCP/IP 1000Base-T cores available. There are commercially available TCP/IP cores which require an embedded processor such as a MicroBlaze or PowerPC, and only offload certain functions to dedicated hardware.[35] This is the case of the Treck TCP/IP core.[6,33] However, at the time of writing this thesis there is no evidence of the Treck core having been implemented fully and at Gigabit line speeds on a Spartan 3. As it is, the core without any current TCP support is very fast and low-cost and it would be preferable to keep these specifications than to expand in a direction which is not required.

Bibliography

[1] Arizona Macintosh Users Group, "AMUG RocketRAID 2224 PCI-X SATA Controller Plus EditBox EB4-ML Review," 2007. `http://www.amug.org/amug-web/html/amug/reviews/articles/highpoint/2224/`.

[2] Anthony Cataldo, EE Times, "Alcatel preps Gigabit Ethernet core for Altera FPGAs," 2001. `http://www.us.design-reuse.com/news/987/alcatel-preps-gigabit-ethernet-core-altera-fpgas.html`.

[3] Nicholas Tsakiris, Greg Knowles, "A Gigabit IP core for Embedded Systems," *International Journal of Computers and Communications, www.universitypress.org.uk (in press)*, 2008.

[4] Nicholas Tsakiris, Greg Knowles, "Enabling Gigabit IP for Intelligent Systems," in *10th International Conference on Mathematical Methods and Computational Techniques in Electrical Engineering (MMACTEE'08), pp 162-166*, 2008.

[5] Loring Wirbel, EE Times, "MMI predicts 215 million IP-enabled consumer devices in the home by 2012," 2008. `http://eetimes.com/news/latest/showArticle.jhtml?articleID=208800177`.

[6] Treck, Inc., "Embedded TCP/IP, Embedded IPv6, and related Internet protocols," `http://www.treck.com/xilinx.html`.

[7] Jerry Kaczynski, "The Challenges of Modern FPGA Design Verification," *FPGA and Structured ASIC Journal*, 2004.

[8] Doulos Ltd., "What is VHDL?," http://www.doulos.com/knowhow/vhdl_designers_guide/what_is_vhdl.

[9] Doulos Ltd., "What is Verilog?," http://www.doulos.com/knowhow/verilog_designers_guide/what_is_verilog.

[10] Pinouts.ru, "8 pin RJ45 (8P8C) male connector diagram and applications," http://pinouts.ru/connector/8_pin_RJ45_8P8C_male_connector.shtml.

[11] Cisco Systems, "History of Ethernet," 2006. http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ethernet.htm.

[12] Cisco, "1000BASE-T - Delivering Gigabit Intelligence on Copper Infrastructure," http://www.cisco.com/warp/public/cc/techno/lnty/etty/ggetty/tech/1000b%5Fsd.htm.

[13] IEEE, "Completion Of 1000BASE-T High-Speed Standard Enables Deployment of 1000 Mb/s Ethernet over both Installed Copper and Fiber Cabling Infrastructure," http://standards.ieee.org/announcements/802.3ab.html.

[14] www.usb.org, "An Introduction to Hi-Speed USB." http://www.usb.org/developers/whitepapers/usb_20t.pdf.

[15] www.usb.org, "USB Cable Limits." http://www.usb.org/about/faq/ans5.

[16] Ajay V. Bhatt, "Creating a third generation i/o interconnect," tech. rep., Intel. http://www.intel.com/technology/pciexpress/devnet/docs/WhatisPCIExpress.pdf.

[17] ExtremeTech.com, "PCI Express 3.0 Bandwidth: 8.0 Gigatransfers/s," http://www.extremetech.com/article2/0,1697,2169018,00.asp.

[18] Bruce Montag, "DisplayPort: Next-Generation Digital Display Interface," 2006. http://www.dell.com/downloads/global/vectors/displayport.pdf.

[19] HyperTransport Consortium, “HyperTransport Technology Overview,” 2005. http://www.hypertransport.org/tech/index.cfm.

[20] HyperTransport Consortium, “HyperTransport Technology FAQs,” 2005. http://www.hypertransport.org/tech/tech_faqs.cfm.

[21] George Ou, ZDNet.com, “Details of Intel CSI QuickPath released,” 2007. http://blogs.zdnet.com/Ou/?p=712.

[22] DRC Computer Corporation, “DRC RPU100-L60 datasheet,” 2007. http://drccomputer.com/pdfs/DRC_RPU100_datasheet.pdf.

[23] Voltaire, “Voltaire Dominates InfiniBand Deployments in Top500 Supercomputer List,” http://www.infinibandta.org/newsroom/articles/Voltaire_Top500_FINAL_06_23_05.pdf.

[24] Odysseas Pentakalos, “An Introduction to the InfiniBand Architecture,” 2002. http://www.oreillynet.com/pub/a/network/2002/02/04/windows.html.

[25] OpenFabrics Alliance. http://www.openfabrics.org.

[26] Rick Merritt, EE Times, “New switch, transceiver and card push 10GE ahead,” 2008. http://eetimes.com/news/latest/showArticle.jhtml?articleID=207400753.

[27] DARPA Internet Program, “RFC 791 - Internet Protocol,” 1981. http://tools.ietf.org/html/rfc791.

[28] DARPA Internet Program, “RFC 792 - Internet Control Message Protocol,” 1981. http://tools.ietf.org/html/rfc792.

[29] DARPA Internet Program, “RFC 826 - Address Resolution Protocol,” 1982. http://tools.ietf.org/html/rfc826.

[30] linux-ip.net, "Address Resolution Protocol (ARP)," `http://linux-ip.net/html/ether-arp.html`.

[31] Linux.about.com, "Linux / Unix Command: arp," `http://linux.about.com/library/cmd/blcmdl8_arp.htm`.

[32] DARPA Internet Program, "RFC 793 - Transmission Control Protocol (Version 4)," 1981. `http://tools.ietf.org/html/rfc793`.

[33] Treck, Inc., "Getting the most TCP/IP from your Embedded Processor," p. 14. `http://www.treck.com/xilinx.pdf`.

[34] Xilinx, Inc., "MicroBlaze Processor Performance," 2008. `http://www.xilinx.com/products/design_resources/proc_central/microblaze_per.htm`.

[35] Zhan Bokai, Yu Chengye, "TCP/IP Offload Engine (TOE) for an SOC System," in *Institute of Computer & Communication Engineering, National Cheng Kung University*, 2005. `http://www.altera.com/literature/dc/3.3-2005_Taiwan_3rd_ChengKungU-web.pdf`.

[36] DARPA Internet Program, "RFC 768 - User Datagram Protocol," 1980. `http://tools.ietf.org/html/rfc768`.

[37] V. Vishwanath, P. Balaji, W. Feng, J. Leigh, D. K. Panda, "A Case for UDP Offload Engines in LambdaGrids," in *Ohio State University*, 2006.

[38] Network Working Group, "RFC 3828 - UDP Lite," 2004. `http://tools.ietf.org/html/rfc3828`.

[39] Clifford E. Cummings, Peter Alfke, "Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons," in *SNUG-2002 San Jose, CA*, 2002.

[40] Wilkov, R., *Analysis and Design of Reliable Computer Networks.* IBM Thomas J. Watson Research Center, Yorktown Heights, 1972.

[41] S. Hopkins, B. Coile, "AoE (ATA over Ethernet)," 2006. `http://www.coraid.com/documents/AoEr10.txt`.

[42] Ross N. Williams, "A PAINLESS GUIDE TO CRC ERROR DETECTION ALGORITHMS," 1993-1996. `http://www.repairfaq.org/filipg/LINK/F_crc_v3.html`.

[43] IEEE, "IEEE Standard 802.3, 2000 Edition," 2000. `http://ieeexplore.ieee.org/xpl/standardstoc.jsp?isnumber=19017`.

[44] G. P. Saggese, A. Mazzeo, N. Mazzocca, A. G. M. Strollo, *An FPGA-Based Performance Analysis of the Unrolling, Tiling, and Pipelining of the AES Algorithm.* Springer Berlin / Heidelberg, 2003.

[45] J. Mogul, Steve Deering, "Path MTU Discovery," 1990. `http://www.ietf.org/rfc/rfc1191.txt`.

[46] Phil Dykstra, "Gigabit Ethernet Jumbo Frames," 1999. `http://sd.wareonearth.com/~phil/jumbo.html`.

[47] CCITT, "ISO/IEC IS 10918-1 | ITU-T Recommendation T.81 (JPEG Standard)," 1992.

[48] The JPEG2000 source, "JPEG2000 info." `http://www.jpeg2000info.com`.

[49] Ali Bilgin, Michael W. Marcellin, "JPEG2000 for Digital Cinema," 2005. `http://www.filmweb.no/nordicproject/template/pdf/teknisk%20jpeg2000.pdf`.

[50] dcimovies.com, "DCI Cinema Specification v1.1," `http://www.dcimovies.com/DCI_DCinema_System_Spec_v1_1.pdf`.

[51] DCinematoday.com, "DLP Cinema Technology Surpasses 5,000 Screen Milestone," 2007. `http://www.dcinematoday.com/dc/pr.aspx?newsID=912`.

[52] Nicholas Tsakiris, Greg Knowles, "Hardware Architectures for the JPEG2000 Wavelet Transform," *ICIS*, 2005.

[53] Deepika Sripathi, "Efficient Implementations of Discrete Wavelet Transforms Using FPGAs," Master's thesis, Florida State University, 2003. `http://etd.lib.fsu.edu/theses/available/etd-11242003-185039`.

[54] The Data Analysis BriefBook, "Convolution," 1998. `http://rkb.home.cern.ch/rkb/AN16pp/node38.html`.

[55] C. Valens, "The Fast Lifting Wavelet Transform," `http://pagesperso-orange.fr/polyvalens/clemens/lifting/lifting.html`.

Appendix A

Socket Code

```
/* client_send.c -- Functional UDP packet sender
/* (c) 2007, Nicholas Tsakiris, Flinders University */
#define closesocket close
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <errno.h>
#include <netdb.h>
#include <unistd.h>
#include <sys/select.h>
#include <time.h>
#include <assert.h>
#define SOCKBUFFSIZE 16777216     /* requested size of the recv/snd socket buffers */
#define BUFFERSIZE   6144         /* size of the payload packet                    */
#define PROTOPORT    4950         /* default protocol port number                  */
#define MILLSECS     100          /* recv() timeout in milliseconds                */
extern int errno;
char localhost[] = "localhost";   /* default host name                             */
main(argc, argv)
int argc;
char *argv[];
{
   struct timeval tv;             /* struct for specifying the recv() timeout      */
   struct hostent *ptrh, *ptrh2;  /* pointers to a host table entry, one           */
                                  /* for detination, one for local                 */
   struct protoent *ptrp;         /* pointer to a protocol table entry             */
   struct sockaddr_in sad;        /* structure to hold an IP address to connect to */
   struct sockaddr_in cad, from;  /* structure to hold an IP address to bind local */
   int sd;                        /* socket descriptor                             */
   int port;                      /* protocol port number                          */
   int iloop;                     /* incrementor for the zero buffer loop          */
   char *host, *localAddr;        /* pointer to host name and localendpointname    */
   int n;                         /* number of characters read                     */
   int runningtag;                /* incrementor for the tag counter               */
   char runningtags[3],tag[5];
   char tagrecv[5];
   int padding;                   /* number of extra characters to pad packet      */
   char buf[BUFFERSIZE + 3];      /* buffer for data from the server               */
   char zerobuf[BUFFERSIZE + 3];  /* zero buffer for padding                       */
   FILE * pFileIn;                /* file pointers for reading and writing files   */
   FILE * pFileOut;
```

```
long lSize, lEchoed, lSent, lDummy;   /* number of data bits echoed and sent    */
int iEchoedp, iSentp;          /* number of packets echoed and sent             */
char * lpDummy;                /* buffer for holding our input file             */
char * buffertowrite;          /* buffer for holding our output file            */
char * tagbuffer;              /* buffer for holding our sent buffer            */
char * llbuffer;               /* buffer for holding our received buffer        */
char * pstrzInputFname;        /* input file filename                           */
char * pstrzOutputFname;       /* output file filename                          */
char * pstrzWhichProtocol;     /* indicate whether we're using tcp or udp       */
uint32_t timeoutval;
socklen_t optlen;
socklen_t buflen;
size_t bufsize;
char * newbufvalue = SOCKBUFFSIZE;
tv.tv_sec = MILLSECS/1000;
tv.tv_usec = (MILLSECS%1000) * 1000;
clock_t start, stop;
double tt = 0.0;
pstrzWhichProtocol = "udp";
localAddr = localhost;
memset((char *) & sad, 0, sizeof(sad));    /* clear sockaddr structure */
memset((char *) & cad, 0, sizeof(sad));    /* clear sockaddr structure */
sad.sin_family = AF_INET;                  /* set family to Internet   */
cad.sin_family = AF_INET;                  /* set family to Internet   */
/* Initializes the zero buffer for use when padding short packets */
for (iloop = 0; iloop < sizeof(buf); iloop++)
   zerobuf[iloop] = '0';
/* Display usage message if no arguments are present */
if (argc == 1) {
   fprintf(stderr,
      "\nUsage: client_send <destination> <inputfile> <outputfile>\n\n");
   exit(1);
}
/* Set port to default port number */
port = PROTOPORT;
if (port > 0)
   sad.sin_port = htons((u_short)port);
else {
   fprintf(stderr, "\nERROR: Bad port number %d\n", port);
   exit(1);
}
printf("\nPort number:  %d\n", port);
/* Check host argument and assign host name */
if (argc > 1 && strcmp(argv[1], "-"))
   host = argv[1];
else {
   fprintf(stderr, "ERROR: Specify a destination host\n");
   exit(1);
}
printf("Destination:  %s\n", host);
/* Check host argument and open input file */
if (argc > 2 && strcmp(argv[2], "-")) {
   pFileIn = fopen (argv[2], "rb");
   if (pFileIn == NULL) {
      fprintf(stderr, "ERROR: Couldn't open file %s\n", argv[2]);
      exit(1);
   }
}
else {
   fprintf(stderr, "ERROR: Specify an input file\n");
   exit(1);
}
printf("Input file:   %s\n", argv[2]);
/* Check host argument and create output file */
if (argc > 3 && strcmp(argv[3], "-")) {
   pFileOut = fopen (argv[3], "w");
   if (pFileOut == NULL) {
      fprintf(stderr, "ERROR: Couldn't create file %s\n", argv[3]);
      fclose(pFileIn);
      exit(1);
   }
}
else {
   fprintf(stderr, "ERROR: Specify an output file\n");
   fclose(pFileIn);
   exit(1);
}
printf("Output file:  %s\n\n", argv[3]);
/* Get the size of the file */
fseek (pFileIn, 0, SEEK_END);
lSize = ftell (pFileIn);
rewind (pFileIn);
printf("%s filesize = %d bytes\n", argv[2], lSize);
/* Allocate memory to write the output buffer */
llbuffer = (char*) malloc (10000);    /* 10000 byte buffer */
if (llbuffer == NULL) {
   fprintf(stderr, "CRITICAL ERROR: Out buffer Memory allocation failed\n");
   exit (2);
}
/* Allocate memory for the sender buffer (input filesize + extra for padding) */
lpDummy = (char*) malloc (lSize + sizeof(buf));
if (lpDummy == NULL) {
   fprintf(stderr, "CRITICAL ERROR: Send Buffer Memory allocation failed\n");
   exit (2);
}

/* Allocate memory for the individual packets */
tagbuffer = (char*) malloc (sizeof(buf));
if (tagbuffer == NULL) {
   fprintf(stderr, "CRITICAL ERROR: Packet buffer Memory allocation failed\n");
   exit (2);
}
/* Allocate memory for the individual packets */
buffertowrite = (char*) malloc (sizeof(buf)-3);
if (buffertowrite == NULL) {
   fprintf(stderr, "CRITICAL ERROR: buffertowrite Memory allocation failed\n");
   exit (2);
}
/* Read the contents into the buffer */
fread(lpDummy, 1, lSize, pFileIn);
fclose(pFileIn);
/* Convert host name to equivalent IP address and copy to sad */
ptrh = gethostbyname(host);
if (((char *)ptrh) == NULL) {
   fprintf(stderr, "ERROR: Invalid host: %s\n", host);
   exit(1);    }    memcpy(&sad.sin_addr, ptrh->h_addr, ptrh->h_length);
/* Convert localendpoint name/address to equivalent IP address and copy to cad */
ptrh2 = gethostbyname(localAddr);
if (((char *)ptrh2) == NULL) {
   fprintf(stderr, "ERROR: Invalid host: %s\n", localAddr);
   exit(1);
}
memcpy(&cad.sin_addr.s_addr, ptrh2->h_addr, ptrh2->h_length);
/* Map UDP transport protocol name to protocol number */
ptrp = getprotobyname(pstrzWhichProtocol);
/* Create a socket which uses datagram (assuming a UDP server) */
sd = socket(PF_INET, SOCK_DGRAM, ptrp->p_proto);
if (sd < 0) {
   fprintf(stderr, "ERROR: Creation of socket failed\n");
}
/* Connect the socket to the specified server. For UDP, this means the socket  */
/* would "remember" where we are sending data to each time we want to transmit */
if (connect(sd, (struct sockaddr *) & sad, sizeof(sad)) < 0) {
   fprintf(stderr, "ERROR: Socket->Server connection failed\n");
   exit(1);
}
else
   printf("Socket->Server connection established\n");
optlen = sizeof(timeoutval);
buflen = sizeof(bufsize);
if (getsockopt(sd, SOL_SOCKET, SO_RCVBUF, &bufsize, &buflen)) {
   printf("getsockopt: %s\n", strerror(errno)); close(sd); exit(1);
}
printf("\nCurrent socket receive buffer size:  %d bytes\n", bufsize);
printf("Requested new receive buffer size:   %d bytes\n", newbufvalue);
if (setsockopt(sd, SOL_SOCKET, SO_RCVBUF, &newbufvalue, sizeof(newbufvalue))) {
   printf("setsockopt: %s\n", strerror(errno)); close(sd); exit(1);
}
   if (getsockopt(sd, SOL_SOCKET, SO_RCVBUF, &bufsize, &buflen)) {
   printf("getsockopt: %s\n", strerror(errno)); close(sd); exit(1);
}
printf("New socket receive buffer size:      %d bytes\n", bufsize);
if (getsockopt(sd, SOL_SOCKET, SO_SNDBUF, &bufsize, &buflen)) {
   printf("getsockopt: %s\n", strerror(errno)); close(sd); exit(1);
}
printf("\nCurrent socket send buffer size:     %d bytes\n", bufsize);
printf("Requesed new send buffer size:       %d bytes\n", newbufvalue);
if (setsockopt(sd, SOL_SOCKET, SO_SNDBUF, &newbufvalue, sizeof(newbufvalue))) {
   printf("setsockopt: %s\n", strerror(errno)); close(sd); exit(1);
}
buflen = sizeof(bufsize);
if (getsockopt(sd, SOL_SOCKET, SO_SNDBUF, &bufsize, &buflen)) {
   printf("getsockopt: %s\n", strerror(errno)); close(sd); exit(1);
}
printf("New socket send buffer size:         %d bytes\n", bufsize);
printf("\nInternal buffer size:  %d bytes  (%d payload size + 3 for tags)\n",

       sizeof(buf), BUFFERSIZE);

if (setsockopt(sd, SOL_SOCKET, SO_RCVTIMEO, (struct timeval *)&tv,

       sizeof(struct timeval))) {

   printf("setsockopt: %s\n", strerror(errno)); close(sd); exit(1);
}
printf("Receive timeout:       %d milliseconds\n", MILLSECS);
lSent = 0l;
iSentp = 0;
if (fork()==0) {
   lDummy = lSize;
   printf("\nSENDING...\n");
   assert((start = clock()) != -1);
   while (lDummy > 0) {
      /* At the end of every packet, add a tag value one higher than the */
      /* previous packet. iSentp holds the number of packets, so it will */
      /* work fine as the tag counter */
      /* this converts iSentp (tag counter) to a string stored in tag */
      sprintf(tag,"%.3c",iSentp);
     /* this injects the tag at the end of the buffer */
      strncpy(tagbuffer, tag, 3);
      tagbuffer += 3;
      memcpy(tagbuffer, lpDummy, sizeof(buf)-3);
      tagbuffer -= 3;
      /* If lDummy is less that the size of buf, we will need to pad it with */
      /* extra characters to match this size */
      if (lDummy < sizeof(buf)) {
         padding = sizeof(buf) - lDummy;
        /* this positions the buffer at the end of the packet data */
         tagbuffer += lDummy+3;
         memcpy(tagbuffer, zerobuf, padding);

        /* this positions the buffer back to where we started */
```

```
            tagbuffer -= lDummy+3;
          }
         iSentp++;
         lSent   += (long)send(sd, tagbuffer, sizeof(buf), 0);
         lpDummy += (long)sizeof(buf)-3;
         lDummy  -= (long)sizeof(buf)-3;
      }
      stop = clock();
      tt = (double)(stop-start)/CLOCKS_PER_SEC;
      printf("\nSEND COMPLETE\n");
      printf("\nNumber of bytes sent:    %d  (%d payload, %d padding, %d tags)\n",

            lSent, lSent - (padding-3) - (iSentp * 3), padding-3, iSentp * 3);

      printf("Number of packets sent:  %d\n", iSentp);
      printf("\nProjected output size:   %d bytes\n\n", lSent - (iSentp * 3));
      printf("Run time: %f seconds\n\n", tt*10);
      closesocket(sd);
      exit(0);
   }
   lEchoed = 0;
   runningtag = 0;
   printf("\nRECEIVING...\n");
   while (1) {
      n = (long)recv(sd, llbuffer, sizeof(buf), 0);
      if (n < 0) break;
      /* Recover tag, verify it's in order */
      strncpy(tagrecv, llbuffer, 3);
      /* this positions the buffer at the beginning of the actual payload */
      llbuffer += 3;
      memcpy(buffertowrite, llbuffer, sizeof(buf)-3);
      /* this positions the buffer back to where we started */
      llbuffer -= 3;
      sprintf(runningtags, "%.3c", runningtag);
      if (strncmp(tagrecv,runningtags,3)) {
         fprintf(stderr, "\nERROR: Packet received with tag #%s, expected #%s, %d \n",

               tagrecv, runningtags, runningtag);

         free (llbuffer);
         free (lpDummy);
         free (tagbuffer);
         fclose(pFileOut);
         closesocket(sd);
         exit(1);
      }
      runningtag++;
      lEchoed += n;
      fwrite(buffertowrite,1,n-3,pFileOut);
   }
   printf("TIMEOUT - shutting down\n\n");
   printf("RECEIVE COMPLETE\n\n");
   /* Close the socket */
   closesocket(sd);
   free (llbuffer);
   free (lpDummy);
   free (tagbuffer);
   fclose(pFileOut);
   exit(0);

}
```