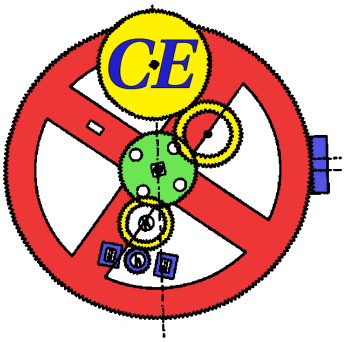


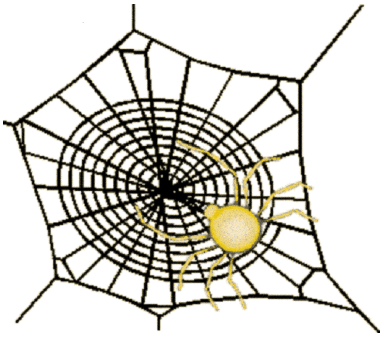
Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2010

MSc THESIS

Exploring and implementing a User Datagram Protocol/Internet Protocol stack within a small Field Programmable Gate Array

M.J.M. Bieleveld



CE-MS-2010-06

Abstract

This thesis is part of the Arachne project which focusses on novel processor architectures that enable an ubiquitous and unobtrusive communication environment. Nowadays, the Internet provides many services of which some are envisioned to be utilized by stand alone devices. Access to those services requires an Internet Protocol stack (IP) implementation. Current solutions with IP functionality in reconfigurable hardware focus on high-end members of FPGA families and often require an embedded RISC processor. The major goal of this project is to implement a design on a low cost FPGA, while leaving space available to run an application alongside. The system is designed around the Xilinx Picoblaze, where additional modules are implemented to improve the performance and provide the required functionality. The experimental results show that such a design is feasible.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Exploring and implementing a User Datagram Protocol/Internet Protocol stack within a small Field Programmable Gate Array

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

M.J.M. Bieleveld
born in Gouda, The Netherlands

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

Exploring and implementing a User Datagram Protocol/Internet Protocol stack within a small Field Programmable Gate Array

by M.J.M. Bieleveld

Abstract

This thesis is part of the Arachne project which focusses on novel processor architectures that enable an ubiquitous and unobtrusive communication environment. Nowadays, the Internet provides many services of which some are envisioned to be utilized by stand alone devices. Access to those services requires an Internet Protocol stack (IP) implementation. Current solutions with IP functionality in reconfigurable hardware focus on high-end members of FPGA families and often require an embedded RISC processor. The major goal of this project is to implement a design on a low cost FPGA, while leaving space available to run an application alongside. The system is designed around the Xilinx Picoblaze, where additional modules are implemented to improve the performance and provide the required functionality. The experimental results show that such a design is feasible.

Laboratory : Computer Engineering
Codenumber : CE-MS-2010-06

Committee Members :

Advisor:	dr. ir. J.S.S.M. Wong, CE, TU Delft
Member:	dr. ir. K. Bertels, CE, TU Delft
Member:	dr. ir. A. van Genderen, CE, TU Delft
Member:	dr. ir. R. van Leuken, CAS, TU Delft

"Perhaps the most valuable result of all education is the ability to make yourself do the thing you have to do, when it ought to be done, whether you like it or not; it is the first lesson that ought to be learned;
and however early a man's training begins, it is probably the last lesson that he learns thoroughly."

- Thomas H. Huxley

English biologist (1825 - 1895)

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

I would like to express my deepest gratitude to all those who stimulated me to complete my thesis. First and foremost, I would like to thank my advisor dr. ir. Stephan Wong for his support and advice. I am grateful for his willingness to advice me during many years, even after a year of absence and allowing me to finish the thesis.

To my girlfriend Lygia, who has put up with a lot of lost weekends, nights and vacations. I thank you for your continuous support and motivation. To my four parents; Gerard, Margreet, Marina and Piet. Thank you for your understanding and your love. Arnoud, thank you for your advice and proof reading this thesis and most importantly just being my friend. I'll miss the comfortable couch to write on.

Finally, I thank Paul and my colleagues at Tele2 for constantly reminding me to finish this thesis. It costed me some beers loosing bet after bet since I had not reached my own deadlines, but in the end it did help me to complete this task.

M.J.M. Bieleveld
Delft, The Netherlands
February 28, 2010

1 Introduction

This Master of Science project entails analyzing an UDP/IP stack and implementing such a stack in a Field Programmable Gate Array (FPGA). The thesis is part of the Arachne project, started at the faculty of Electrical Engineering at Delft University of Technology. The Arachne project focuses on novel processor architectures that enable an ubiquitous and unobtrusive communication environment. The first section of this chapter starts by motivating the project. Section 1.2 lists the main requirements set for this project. Section 1.3 lists the main goals and finally the framework of the thesis is presented in Section 1.4.

1.1 Motivation

In the 1950s networks consisted of a central mainframe connected through leased lines to several terminals. Those early networks were not interconnected and hardly standardized. The standardization of networks began with ARPANET, a project initiated to provide and interconnect different networking systems. The first ARPANET link was created between the University of California, Los Angeles and the Stanford Research Institute in 1969 and soon more followed. Later, in 1981 this network had grown to about 213 hosts. During that time a further unification of network methods was needed that ultimately resulted in the nowadays well known network protocol stack TCP/IP. The newly adopted TCP/IP standard was implemented on the ARPANET network and the Internet was born and its size and use has grown enormously during the last decades.

In October 1990 the RIPE organization, one of five Regional Internet Registries, started a project to measure the number of hosts on the Internet. The host counting project began to measure in nineteen countries and summarized the amount of live hosts [15]. Nowadays the host count runs every month and includes over a hundred top level domains. A historic graphical representation of the RIPE data set is depicted in Figure 1.1 [5]. The graph shows a significant increase in the number of hosts and ends up with approximately 28 million hosts in January 2005.

Not surprisingly, the amount of traffic going through the Internet's backbone has been growing accordingly. A trend also supported by a recent study published by Guo-Qing Zhang, et al. which is based on the routing data of six-month intervals from December 2001 till December 2006. The study states that Moore's law, often applied to computing and storage capabilities, also holds for Internet traffic. The Internet traffic is therefore likely to continue to double each year during the coming decade. Thus, it can be said that the Internet is growing significantly in both bandwidth and the number of devices.

Expected or not, this increase in Internet penetration is not reflected in the amount of freely available FPGA IP stacks. A well known site such as www.opencores.org does

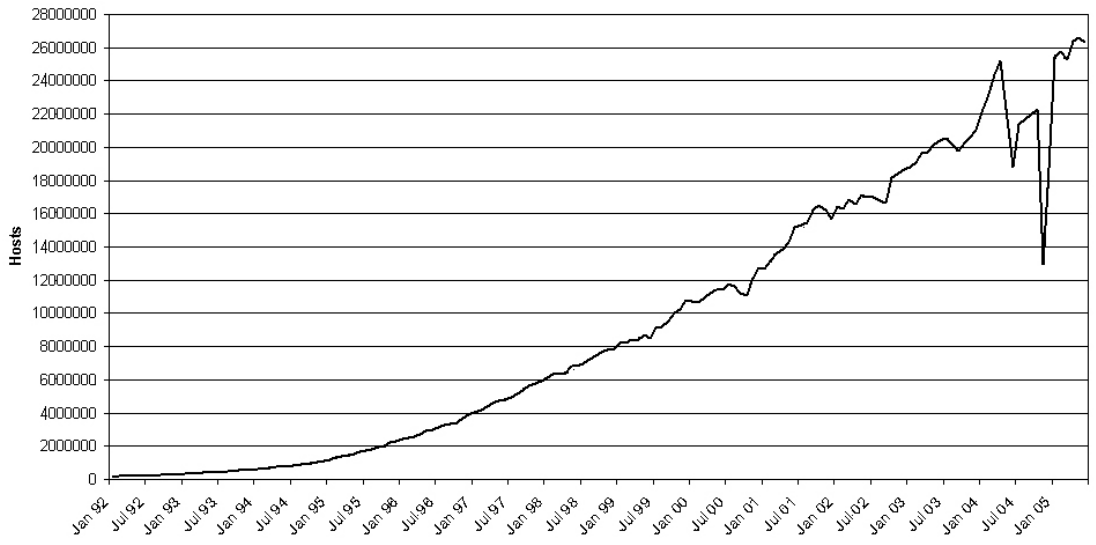


Figure 1.1: The RIPE hostcount

not even list a single implementation. Xess, a major manufacturer of Xilinx prototype boards, lists an UDP/IP stack implementation in VHDL [22]. Other implementations exists and are available, but for a price and usually those intellectual property blocks are pre-compiled. The source code of such an implementation is, as often is the case with proprietary software, not available to the general public.

A free UDP/IP or TCP/IP stack implemented within an FPGA is a good start, as it is allows others to adapt and improve the functioning of the stack. This is made possible due to the fact that the code is published under the GNU GPL license, keeping future improvements free for all to use and learn according to the philosophy of the GNU GPL. This to the contrary to the earlier discussed proprietary IP stacks. Therefore, the goal of this project is to create a small UDP/IP stack, usable in even the smallest of Spartan 3E FPGAs, with enough area remaining on the FPGA to run an application alongside and publishing it under the GNU GPL.

1.2 Project definition

The aim of this project is to design and implement an Internet Protocol stack, which supports the UDP protocol on reconfigurable hardware. Most of the stack is implemented in assembly on a small embedded microcontroller, while some compute-intensive operations are implemented using reconfigurable hardware. Using specialized blocks for compute-intensive operations does not only improve the overall speed, but also reduces the amount of assembly code required on the microcontroller, which is important because the embedded microcontroller's architecture only allows addressing up to 1024 instructions. The choice of implementing an UDP/IP stack instead of a full TCP/IP stack is solely based on the time constraints set for this thesis.

The requirements of the UDP/IP stack are summarized in the following sentence; *The UDP/IP stack should be completely implemented in reconfigurable hardware and be capable of transmitting and receiving valid UDP packets to other hosts, occupying the least amount of area while aiming at 100Mbit/s throughput.*

1.3 Project goals and main contributions

The main goal is to create a small, minimal and freely available core that interacts with other UDP/IP implementations. This core is targeted towards Xilinx FPGAs, in particular to the Xilinx Spartan 3E prototype board. The prototype board contains reconfigurable hardware in the form of an FPGA and has an onboard 10/100 PHY as one of its many peripherals, more details on the board and the reason of selecting it is found in Section 3.1.

A part of this research consists of selecting suitable software functions for hardware speedup. This selection should be based on three factors; the expected increase in obtainable bandwidth throughout the stack, the expected reduction in the amount of needed code space for the microcontroller and finally the amount of area that is required on the FPGA itself. Combining these three factors should result in a fast and small UDP/IP core.

Reconfigurable hardware is used to create an embedded microcontroller together with the accelerated hardware modules. Combining an embedded general purpose microcontroller and several application specific modules is particularly suited for this project considering the earlier set area requirements, due to the constant size of the microcontroller and the complex algorithms it can handle. In addition a speed improvement over a software only solution is expected as a result of the hardware acceleration.

The main aspects of this project are;

- Researching a comparable stack.
- Providing a base IP platform for future projects with low-cost, small FPGAs.
- Determining the computation intensive parts and parts that need large code space.
- Creation of an UDP/IP stack in Picoblaze assembly.
- Creation of several computation intensive blocks in Hardware Description Language (HDL).
- Creation of a demo to show the interaction between a third party application and the stack.

1.4 Thesis overview

This thesis is organized in the following manner. Chapter 2 describes the embedded microcontroller and is preceded by an brief overview of the implemented protocols, such as ARP, ICMP, IP and UDP. In order to interact with those protocols and their implementations an interface needs to be defined. A de facto interface standard exists and is discussed in Chapter 3. Furthermore, that Chapter 3 describes the hardware implementation of the stack. The demo setup and with it the achieved experimental results are found in Chapter 4. Finally, Chapter 5 summarizes the work and provides a few recommendations for further research.

2 Background

This UDP/IP stack implementation requires two functionalities in order to provide basic UDP connectivity between two or more systems. First, it requires a programmable embedded microcontroller that is extended with hardware accelerated modules, second, several protocols need to be implemented in a combination of hardware or software modules. Further discussion about the first requirement is found in Section 2.2, whereas the selected microcontroller is described in Section 2.2.1. The protocols that form the second requirement are discussed in the second half of this chapter.

Section 2.1 introduces other known studies on IP stacks in combination with reconfigurable hardware and/or microcontrollers. Whereas, Section 2.2 describes microcontroller and the reason behind using a microcontroller in an FPGA. Section 2.3 describes the Ethernet standard, a message format standard used between hosts in a network. Section 2.4 describes the Internet Protocol, a protocol used to send packets from one network to another. Section 2.5 describes the Address Resolution Protocol (ARP) which translates IP addresses to Ethernet addresses. An error reporting protocol is discussed in Section 2.6. Finally, the User Datagram Protocol (UDP) is discussed in Section 2.7. The chapter concludes with a summary.

2.1 Related work

Due to the great success of the Internet, the TCP/IP protocol suite has become the standard of communication between computer systems. The suite is utilized by many applications that run on top of the Internet. Application such as; web pages, file transfers, voice over ip calls and emails. Due to the great number of possible applications and accessible data sources there is interest in running such as protocol suite on microcontrollers and embedded processors.

Historically TCP/IP required a relatively large amount of resources on a microcontroller in terms of code size and memory usage. A software implementation that functions well with a very limited set of resources is discussed in [8]. Here two small TCP/IP implementations are introduced that were afterwards ported to many other platforms. The two implementations are *lwIP* and *uIP*. *lwIP* is a full-scale TCP/IP implementation with full support for IP, ICMP, UDP and TCP. Whereas, *uIP* is designed to be a minimal stack with just the necessary functions to have a microcontroller friendly TCP/IP stack. Another software based implementation is discussed in *Networking and Internetworking with Microcontrollers* written by Eady [9]. His book deals with a more hands-on approach in writing a stack for a microcontroller. The main difference between these implementations and this research is that this thesis deals with reconfigurable logic and the optimization of code blocks in order to achieve a higher throughput.

Other research focusses on accelerated solutions. The solutions that are described in this line of research makes use of an FPGA with an on-chip microprocessor, e.g. a Xilinx FPGA with a PowerPC core, on top of which an operating system runs with a standard software based TCP/IP stack [14][23][6]. The TCP/IP stack is then accelerated by implementing some functionalities in reconfigurable hardware. Other configurations exist where a microprocessor is connected to an FPGA. This is seen in [11] which makes use of a prototype board developed at the Swiss Federal Institute of Technology Zurich [16]. In this case there are two FPGAs, one functions as a microprocessor and the other as stand-alone IP stack. A big difference between this thesis and [11] is that the achieved throughput is about a fifty fold higher for this implementation and the focus of this research is to utilize cheap reconfigurable hardware and not high end FPGAs with on-chip cores.

Yet, other implementations focus on designs contained within a single FPGA. For instance, the VHDL IP stack designed at the University of Queensland [22]. This design is quite similar to the design of this thesis, both are targeted towards full duplex networks and lacking TCP support. Differences are that the implementation of [22] is without an embedded processor and perhaps due to this fact can not run on high clock frequencies. As a result the performance of the stack discussed in [22] is limited to 10 Mbit/s. Both designs are targeted as stand-alone applications although [22] does not include an on chip interface and therefore has no running demo application on the FPGA itself to prove its working, but uses software running on a PC to test its functionality.

A high speed design achieving a throughput of 100Mbit/s and a theoretical throughput of 700Mbit/s is described in [6]. The main difference between that design and the implementation described in this thesis is the targeted FPGA. [6] uses about 70% of the total resource available on the largest Spartan 3E FPGA making it unsuitable for low end FPGAs. These high-end Xilinx FPGAs with an embedded PowerPC has a cost of $165 USD, while the targeted FPGA for this project costs less then $30 USD[1]. In case of [6] the largest Spartan 3E FPGA is used, which has a cost of $68 USD [2]. Mainly it does not make sense to develop an FPGA based product requiring large amount of resource just for the stack since standard off the shelf network interface cards are bought for a fraction of the price.

The Request For Comments document *RFC1122 - Requirements for internet hosts - communication layers* [3] forms the basis for all previous implementations and provides a good deal of information about IP stack implementations. The document sets requirements to which any stack has to adhere to in order to comply with most, if not all, other implementations and is used as a guide throughout this design.

[1] Price sampled at 20-02-2009 on Avnet.com. A PowerPC component is XC4VFX12-10FF668C and its price is for quantities >100, targeted component is XC3S500E-4FT256C price is for a Prototype quantity.

[2] Price sampled at 20-02-2009 on Avnet.com. Price is for component XC3S1600E-4FG320CPROTO

2.2 The embedded microcontroller

A first attempt at implementing an IP stack in reconfigurable hardware worked without an embedded microcontroller.[3] This design had several issues; for one, the area constraints could not be met. A big portion of the area was needed to multiplex and route the data to and from the different interacting modules. Secondly, due to the size of the multiplexer the design could not achieve a high operating speed. A higher operating speed could probably be obtained by introducing registers, but since this would increase the footprint and complexity even more it did not seem like a feasible option.

Another issue that became apparent during the first exercise was that debugging internal circuits is a difficult and time consuming job. Although most of the design could be divided up into smaller design blocks which could easily be simulated and debugged in modeling software, it became more troublesome when it came to debugging blocks that had interfaces with some external components, such as the network chip. Significant improvements in debugging were made with the Chipscope software package developed by Xilinx. With Chipscope one inserts a logical analyzer, a bus analyzer and virtual I/O directly into the design [28]. Still debugging and correcting a HDL design was found to be more complex and certainly more time consuming then initially thought.

This guided the design towards the tried solution of implementing a microcontroller within the reconfigurable logic. One of the biggest benefits is that one uses the already existing development environments for that particular microcontroller and reuse it in the design. Another benefit is that the microcontroller's peripherals are completely customizable to whatever needs one has. The amount of area occupied on the FPGA can be reduced by customizing the set of peripherals implemented in reconfigurable hardware. A lot of excess functionality can potentially be removed from the design. Last but not least, state machines with many inputs and outputs and complex dependencies between them are programmed relatively well in a structured fashion in assembly language. This certainly became an important benefit when the protocols, that are discussed at the end of this chapter, were implemented. Those protocols often operate on many bytes inside the packet header and lead to complex and large state machines. Large state machines are not only difficult to understand and maintain, but also require more resources on the FPGA.

Using an embedded microprocessor also eradicates microcontroller obsolescence and preserving legacy code. Although this argument is more applicable in the context of a production process it is does give this implementation a potential longer shelf life. Obsolescence of electronic components concerns mostly equipment involved in safety critical applications, in particular in the automotive, avionics, and military fields. The desired life time for those systems is many times longer than the obsolescence cycle of the components used in the systems [1]. A typical solution for this problem is to keep stocks of components used in the design, or to find parts in the secondary markets. However, a

[3] Please note that the terms microcontroller and microprocessor are used interchangeably in this text. Usually the term microcontroller refers to a processor core plus additional components, such as UARTs and RAM blocks, together in one package, while microprocessors refer to just the processor core. The distinction between the two is almost none existing within an FPGA since a microprocessor implemented in reconfigurable logic can make use of UARTs and RAM blocks within that same FPGA fabric and thus also operates within a single chip.

different and future proof solution is to emulate the components within the FPGA itself. Instead of using an external microprocessor with a limited availability, a soft core is used that provides the same functionality. Even though the FPGA itself becomes obsolete, the implemented design is still available in a Hardware Description Language (HDL) and retargeted towards future platforms. Now that the core architecture remains the same there is no further need to port the code running on the microcontroller towards a different architecture, which could be rather difficult depending on the differences between the old and new architecture.

2.2.1 The Picoblaze

Currently several parties provide soft core processors; the Altera's Nios II and Xilinx Microblaze soft processors are two of the more widely known implementations. Both have 32-bit processor architectures and are geared towards their respective vendors FPGAs. Open source alternatives are found on the opencores.org web site [2]. The main objective of the Opencores.org project is to design and publish core designs under the Lesser General Public License (LGPL). Currently they list more than thirty soft processors that range from just newly submitted projects to completely finished implementations. Most of the processors listed on the web site are not primarily designed towards resource efficiency. As an example, a 8051 8-bit processor available on the opencores.org web site requires approximately 1500 slices of logic, while it is mentioned that a smaller version is worked on, occupying around 1000 slices. To put these numbers into perspective one must compare these numbers to the available resources available in the reconfigurable logic. In fact, the 8051 processor will use more than half of the available resource in the second but smallest FPGA of the Xilinx Spartan 3E family, while it will not even fit in the smallest family member. Considering that the embedded processor is only a small part of the total design it is easily seen that such a processor does not fulfill the requirements set in the first chapter.

Resource efficient processors like the Nios II/e "economy" processor use about 14% of the resources in the smallest FPGA of Altera's low cost Cyclone III family. The same holds for the Xilinx Picoblaze controller, which only needs 96 slices of logic or 12.5% of an XC3S50 FPGA, the smallest member of the Xilinx low cost Spartan 3E family [26].

Due to vendor lock-in and the focus on area reduction the Picoblaze microcontroller was chosen for this project as the central processor unit. The Picoblaze microcontroller is provided as a free, source-level VHDL file with a royalty-free re-use within Xilinx FPGAs. A big advantage of using an established microcontroller that is supported by a large company is the excellent documentation that comes along with it. An aspect that is often lacking in open source software. Even though this particular controller has been chosen for this design it is generally easy to port the code and design towards a different eight bit controller.

The Picoblaze microcontroller features sixteen byte-wide general purpose data registers and allows 1024 instructions out of the box in programmable on-chip program store, which is automatically loaded during the configuration of the FPGA. It also features an eight bits ALU and the microcontroller has a 64-byte internal RAM which is used as a

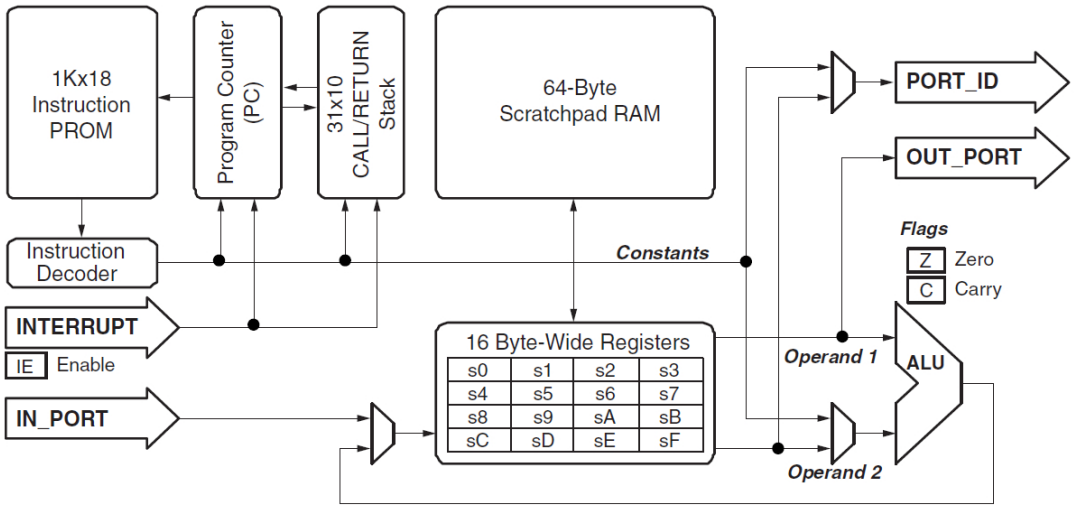


Figure 2.1: The Picoblaze architecture

scratch pad. Its architecture is depicted in Figure 2.1.

The input and output ports are used to extend the controller and act as an interface between the controller and the custom made peripherals implemented in reconfigurable logic. The Picoblaze supports up to 256 eight bits input ports and 256 eight bits output ports, or a combination thereof depending on the addressing scheme. Two instructions are used to interface with the outside world. The *INPUT* instruction reads the data into a specified register, while an *OUTPUT* instruction presents the content of a register on the OUT_PORT.

2.2.2 The KCPSM3 module

The Picoblaze processor is included in the design by instantiating a VHDL module named KCPSM3. The KCPSM3 module contains the ALU, the register file, and all other functions with the exception of the instruction store. The complete pin out of the module is depicted in Figure 2.2. There are several options as to where to store the instructions; The most effective way is to use one dedicated Block RAM (BRAM) inside the FPGA. There are twenty of those RAM blocks available inside this projects FPGA. Each block has a capacity of 18Kbits and all blocks are configured for dual-port access.

There are several HDL library components defined for the BRAMs. Some components define the RAM as a single port memory while others configure it as a dual port memory. The dual-port memory is chosen in the design because it has the advantage of connecting one port of the instruction store to the Picoblaze, while the other port is connected to a JTAG module. The JTAG module provides an interface between the BRAM contents and a PC, enabling the upload of a Picoblaze program directly into the memory during runtime. Normally, without the use of the JTAG interface, a change in the assembly code would require a complete recompilation of the entire design, which could take up

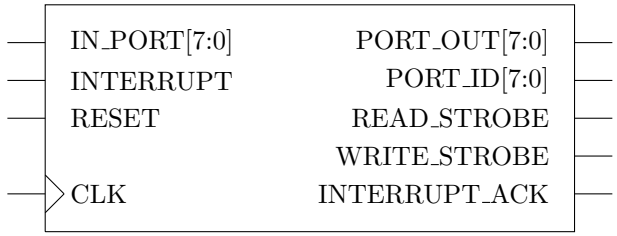


Figure 2.2: The KCPSM3 module

to ten minutes. On the contrary, when the JTAG interface is used, the process is just a matter of seconds, since only the Picoblaze code needs to be assembled and uploaded.

Various tools exist for the Picoblaze environment. A development tool used in this project is the free pBlaze IDE [12]. The IDE contains a simulator that allows for easy debugging of the simulated code with single-stepping, breakpoints, interactive access to registers, flags and memory values. Another tool used for debugging the Picoblaze, and for debugging HDL in general, is ModelSim. ModelSim is used as a digital simulator to simulate the entire environment from network traffic to microcontroller, instead of only the Picoblaze as is the case with the pBlaze IDE. Finally, the tool of all tools is Chipscope. With Chipscope certain signals are brought outside the FPGA and monitored with a software based scope/logic analyzer running on a PC and does so without the need of expensive hardware solutions.

2.3 The Ethernet standard

Since the beginning of the computer era many people have found ways to network computers together so that they may be able to exchange digital data. Some of these methods have become very prominent in our current society and have changed the way we live our daily lives. Nowadays almost every household with an Internet connection makes use of at least one of these methods, namely the Ethernet standard. The Ethernet standard describes a network protocol to connect several devices in a Local Area Network (LAN) together.

Throughout history many efforts have been made to improve the reliability and speed of networking protocols. These efforts lead to the Ethernet standard that is found in almost every house where computer devices are networked together as introduced in Section 2.3.1. Section 2.3.2 explains the data packet format as used in the Ethernet standard. The final section, Section 2.3.3, discusses the control of those packets.

2.3.1 Introduction to computer networks

A long time ago Bob Metcalfe was working for Xerox's Palo Alto Research Center (PARC) when he read a paper about the ALOHA network, or Alohanet, from the university of Hawaii. ALOHA was a new computer networking system and was first deployed in 1970. The key idea was to use cheap amateur radio-like systems to create a

packet-switched radio network linking several geographically dispersed campuses of the university together. Two distinct frequencies were used as to create a so called hub/star configuration. A hub/star configuration is a configuration where a hub, placed on the center of an imaginary star, e.g. *, broadcasts data to all the end-points on the star in the outbound channel working on a frequency of 413MHz. This while the various client machines, which formed the end-points, send all their data to the central hub location on the inbound channel operating on a frequency of 407MHz. The hub, or the base station, listens to the packets transmitted by the end-nodes and retransmits all incoming packets to all end-nodes. Clients could check whether their data was correctly transmitted by simply comparing the broadcasted data sent by the hub, as a response on their own transmitted packet, and the original data.

One important feature of the ALOHA network is the method called random access. The ALOHA network is comprised of a shared medium, so it is certainly possible that several clients send their data simultaneously to the hub. To reduce the occurrence of this problem the clients had to wait for the confirmation of the hub that the packets were received, while a new transmission could be started at any time after the confirmation. When multiple stations simultaneously tried to send data over the shared medium, no confirmation, or a garbled up message was received and thus a so called collision occurred. In case of a collision the transmitting stations wait for a random period of time before retransmitting their data to the hub. The introduction of the random period of time reduced the occurrence of a new collision, since not every host will try to retransmit at the same time.

Years later, Bob Metcalfe was given the task by Xerox PARC to design a way to interconnect several Xerox Altos machines, which were Xerox workstations with a graphical interface. He then modified the Alohanet to incorporate the method of varying the random access interval time based on the traffic load and he also used cables instead of a radio link. This first experimental network was named the Alto Aloha Network. Later the name was changed to "Ethernet", a combination of the word "ether" indicating a physical medium to carry the data and the word "net" as in networking, to make it clear that the system could support not only Altos machines, but in fact any computer system.

2.3.2 The Ethernet frame

The Ethernet frame, a message on the data link layer, is defined in the IEEE802.3 standard and is the format by which all Ethernet implementations communicate. An Ethernet frame is depicted in Figure 2.3. It is shown that data is first encapsulated in a container before it is actually send on the wire. Historically two types of these frame formats existed; one defined in the 802.3 framing standard, where there is a Length field used after the source address, and another type named Ethernet II Framing, where there is a Type field after the source address. Both frame types are now defined and supported within the IEEE802.3 standard. These two types are used interchangeably by the convention that values between 64 and 1522 indicate the use of the new 802.3 frame format, while values higher than 1535 indicate the use of the Ethernet II frame format

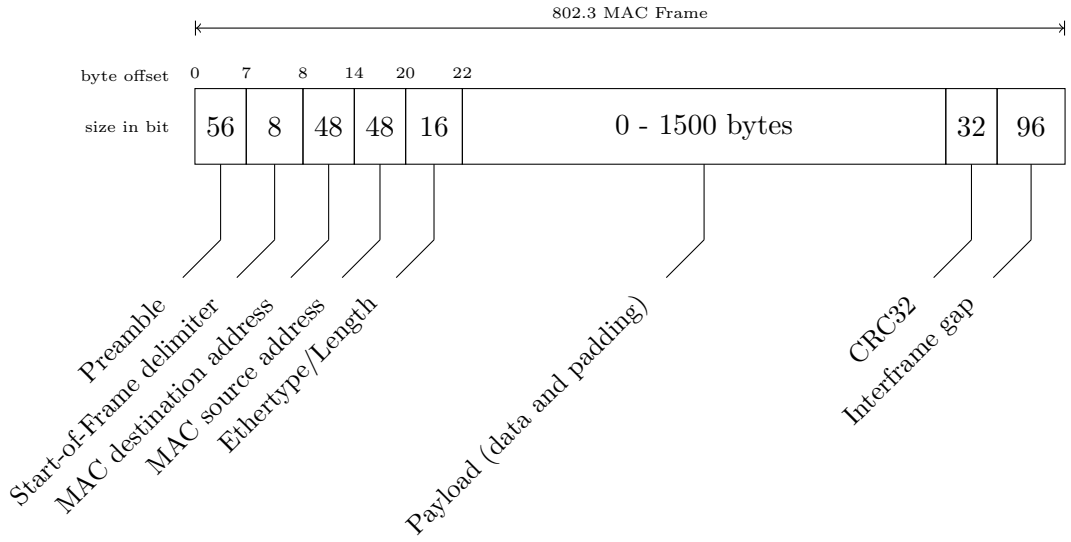


Figure 2.3: The Ethernet frame

in which the identifier refers to an EtherType sub-protocol identifier.

An Ethernet frame varies in size ranging from 64 to 1518 bytes. Every frame starts with a preamble, a repeating 1010 bit pattern needed to synchronize some PHYs, a common abbreviation for a device in the physical layer of the OSI model. The preamble is followed by a Start of Frame Delimiter (SFD) marking the byte boundary for the MAC, meaning that the information that is received from now on is passed on to the MAC layer. This is the reason why one would normally not see the preamble and the SFD in a software based network analyzer, such as Wireshark [4], since it is consumed by the network interface card.

The first two fields after the preamble are station addresses. Station addresses are 48 bits wide and must be unique on the LAN. This requirement is needed, because the Ethernet standard uses a flat hierarchy within a shared medium. Normally unique addresses are guaranteed by the manufacturer during the manufacturing process. Each manufacturer is assigned a three byte Organization Unique Identifier (OUI) used as the first three bytes of the address, whereas the lower part is uniquely assigned by the manufacturer. However in some cases the addresses is locally administered, or even dynamically changed by the device itself as part of a fail-over feature of a firewall. The Source Address (SA) in the Ethernet frame is always the address of the transmitting station. On the contrary, a destination address can also point to an address where the most significant bit is set to one, refering to a range of addresses used for multicast and broadcast purposes. For example, multiple stations can receive simultaneously a packet when it has been broadcasted to multicast address 0xFFFFFFFFFFFF.

A padding field is found after the encapsulated data to pad the frame up to a minimum frame size of 64 bytes. A CRC-32 checksum is appended to the frame for error checking and encompasses the entire frame. Between frames there is a minimum idle

[4] http://www.wireshark.org

time of 96 bits, a so called *interframe gap*, used to maintain continuous synchronization between the NICs at each end of the link.

2.3.3 The Media Access Control layer

Like with the earlier mentioned ALOHA network, there is a need to regulate the transmission and reception of Ethernet frames. This regulation is controlled by a layer called the Media Access Control (MAC) layer. The IEEE 802.3 standard specifies a common medium access control layer that provides several high level functions by which it creates an abstraction layer for the physical medium. In addition, the MAC layer itself does not need to know about the physical medium as it interacts with a PHY, which in turn provides an abstraction layer to the physical medium.

The MAC provides a packet-based, connectionless data transfer between devices and has three main functions. First, it provides data encapsulation; data to be send is encapsulated within a frame before transmission by adding the earlier discussed fields, such as the preamble and the start-of-frame delimiter. At the same time it decapsulates the received frames by passing only the encapsulated data to the a higher level protocol. Another function is to control the media access, thus initiating frame transmissions and allowing recovery from transmission failures. The Ethernet MAC operates in either half or full duplex mode, both are explained in the next section, which is dependent on support from the physical layer and the desired operating mode. Currently the IEEE 802.3 standard requires that all Ethernet MACs can operate in half-duplex operation, while full-duplex is an optional feature. The third main function listed here is addressing. All Ethernet network adapter cards are uniquely identified by a hardware address, also known as the MAC address, which is pre-assigned by the manufacturer. The concept of a unique addresses, or universal addressing, is based on the idea that all devices that are going to network together need a unique identifier when they make use of a shared medium. The advantage of such a unique address is that any device can be attached to any LAN in the world with the assurance that the address is unique and communication is possible. The MAC filters frames received from the shared medium by comparing the destination address in the Ethernet frame with its own unique address.

2.3.3.1 Half duplex

The media access control layer operates in two modes; a half duplex mode and a full duplex mode. The Ethernet half duplex mode is derived from the slotted version of the Aloha protocol. Where the pure Aloha protocol had no slots and stations could transmit at any given time the slotted version only allows the start of a transmission on certain moments in time. These time slots were chosen to be at least of the same duration as it would take a frame to be transmitted. As a result, there is no collision when two devices send in different time slots. A collision now only occurs when two or more transmissions collide with one and another on the shared medium resulting in a corrupted data transmission.

Ethernet uses Carrier Sense Multiple Access (CSMA), with CSMA a device that is about to start a transmission first 'listens' to check if there is already another transmission in progress. A device 'listens' by monitoring, for example, the current flowing

through the cable. The medium is considered idle when no signal is heard and then the transmission starts, while all other devices remain listening. Listening alone does not prevent all collisions, as two devices can still initiate a transmission at the same time, or within a time period in which the signal is still propagating. To detect this collision the Ethernet standard defines an Collision Detection mechanism.

A host needs to determine whether its frame is send out without a collision before it can send out another frame, therefore it needs to be able to detect a collision before it finishes transmitting the frame. To illustrate; when host A starts transmitting it will take a certain propagation time before another host, host B, can see this transmission. Host B determined by listening that the medium is idle. Now assuming that host B starts transmitting its frame just before host B received the frame of host A, it takes the same propagation time for host A to see host B his frame. As a result, in order for host A to determine that there is in fact a collision, while sending its frame, it still needs to be busy with sending that frame at that moment when it sees the frame of host B on the medium. In other words it requires a transmission time greater than two times the propagation time for a host to detect a collision. This time period of approximately two times the propagation time is also known as the collision window or the slot time.

In case of a collision each transmitting device starts sending out a 32 bit jam sequence. The jam sequence should have a different value than a valid checksum for the frame; Some Ethernet cards just send 32 ones while some others use an alternating pattern of ones and zeros. The reason behind the jam sequence is to prevent devices from starting a transmission and to ensure that other transmitting devices recognize that a collision has occurred. Following the jam sequence the transmission is rescheduled for transmission by a controlled randomization process called "truncated binary exponential back off" [10]. The delay is an integer multiple of the slot time. The number of slot times to delay before the n^{th} retransmission attempt is chosen from the range of $0 \leq r < 2^k$, where $k = min(n, 10)$. This algorithm greatly reduces the chance on a new collision.

The slot time is directly related to the frame size. Short packets use less time to be transmitted and thus this transmitting host has less time to discover a collision. This implicates that there is a minimum frame length for CSMA/CD networks. Longer minimum frame lengths would lead to longer slot times, which in turn would mean that a larger distance, or diameter of the network, can be covered. A shorter minimum frame length corresponds to shorter slot times and therefore to smaller network diameters. There is a trade-off between the maximum size of the network diameter and the need to reduce the impact of a collision recovery. In the IEEE802.3 specs this trade off was settled by allowing an Ethernet network diameter of 2500 meters and the minimum frame length was set accordingly.

With the development of high speed Ethernets having a data-rate of 100Mbps or Gigabit Ethernets with a data-rate of 1000Mbps, the time required to transmit a frame is greatly reduced. In a Fast Ethernet network a frame is transmitted in approximately one tenth of the time it would require in an Ethernet network. Fast Ethernet was designed to be backwards compatible to the Ethernet standard and thus this speedup translated to a reduction of the maximum network diameter by a factor of ten. As a result the maximum diameter for FastEthernet is 200 meters.

The problem of reducing the maximum diameter even further was addressed by the

Gigabit Ethernet standard, because a maximum network diameter of around 20 meters is not practical. Instead of reducing the maximum network diameter the minimum packet size was altered. The solution was kept backwards compatible by adding a variable length data field to frames which are shorter than the minimum required size of 520 bytes. The variable length data field is filled with random data and is automatically removed when the frame is received. Consequently, a Gigabit Ethernet network has the same maximum network diameter as a Fast Ethernet network. Note that the minimum frame size is only increased for half duplex links. Full duplex links still allow a minimum frame size of 64 bytes.

2.3.3.2 Topology

The first Ethernet networks used a shared bus topology, as depicted in Figure 2.4, which had some obvious problems. A shared bus topology consists out of a continuous coaxial cable, that is directly connected to all the devices in the network. Both ends of the cable are terminated by special terminators to prevent signal reflections. A break in the coaxial cable, a common problem, creates two disjoint networks that are not able to communicate with each other. A ground fault, another problem, in the cable could even disrupt all transmissions for every device. Although the most common issue with this topology is in fact not a problem at all but simply the action of adding and removing devices within the network, since this would disrupt the entire network too.

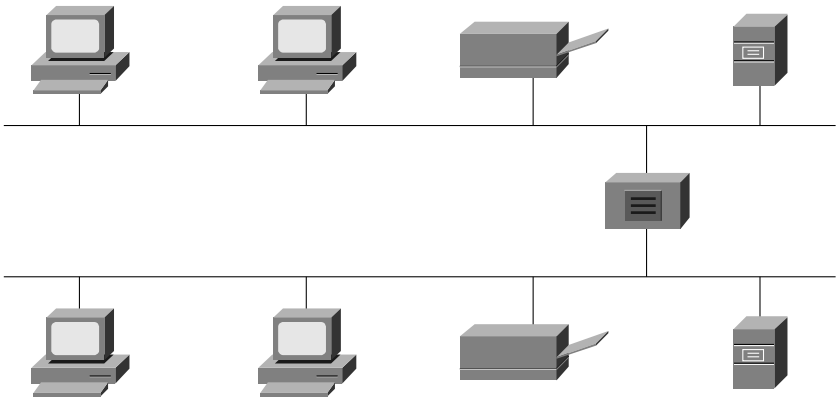


Figure 2.4: A bus topology

Networks started to switch over to a star topology, depicted in Figure 2.5, to reduce the impact of the cabling issues. A star topology consists out of a central device, called a hub, and several other devices that are directly connected to the hub. The big advantage of the star topology is that the cabling now runs only from one single device to the hub. A faulty cable in this topology will solely impact the device it connects to. A hub works at OSI layer one and does not need to understand the data it repeats, it simply repeats the incoming signal to all other ports. Like with a single coaxial cable, there is still only one collision domain and thus a collision occurs between any of the segments connected

to its ports.

The maximum network diameter for an Ethernet network is 2500 meters; however, the maximum cable length is only 200 meters. In order to extend an Ethernet network several hubs, or repeaters are used together. However, the performance of a network with one single collision domain suffered as more and more devices started to transmit large amounts of data. As a response, switches became popular. Even more so because the cabling used for switches, Unshielded Twisted Pair (UTP) cabling, was cheaper than coaxial cabling. A switch buffers frames which works since it work on second OSI layer and thus 'understands' the frame formats. A network is divided into several collision domains by buffering the frames and so increases the performance of large networks.

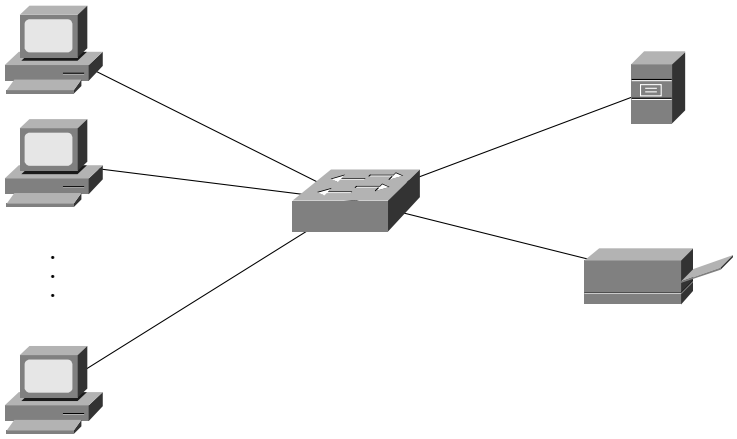


Figure 2.5: A star topology

2.3.3.3 Full duplex

In the last decade the dominant topology has evolved from a bus topology to a star topology. With a star topology the medium is no longer required to be a shared medium. In fact even the transmit channel and receive channel are separated with UTP cabling. Unlike with coax, where simultaneous transmitting and receiving was not possible, a device now transmits and receives data simultaneously without data corruption. A connection in which data is transported in both directions at the same time is considered to be *full duplex*. A full duplex point-to-point connection does not need CSMA/CD, because each device transmits on one UTP pair and receives on another. As a result, there is no media contention, no collisions and thus also no need to schedule retransmissions. Making full duplex connections much simpler and easier to implement.

2.4 The Internet Protocol

The Ethernet standard defines messages, so called frames, that are send from one host to another. Conventionally, two hosts need to be on the same local area network or to be more precise within the same broadcast domain to receive each other frames. To send packets to another network a different standard is used. A protocol in the network layer provides this delivery of data between hosts outside the LAN. This section discusses such a protocol; the Internet Protocol.

The Internet Protocol provides several functions to the higher OSI layers. The addressing scheme the protocol provides makes it possible to distinguish one device from another. All public IP addresses on the Internet are unique. IP addresses are organized in a hierarchical way, on the contrary to MAC addresses which are organized as a flat address space. IP addresses were originally directly handed out by the Internet Assigned Numbers Authority (IANA) to the organizations that needed them. Organizations would receive Class A, B or C blocks, see Table 2.1, however due to the limited amount of IPv4 addresses and to reduce the growth of routing tables across the Internet the community has switched over to classless addressing. With Classless Inter-Domain Routing (CIDR) there are no fixed boundaries that a network needs to adhere to. Instead, arbitrary length prefixes are allowed where multiple contiguous prefixes are aggregated to a larger network. Nowadays the IANA does not assign IP addresses themselves but hands them out to five Regional Internet Registries (RIRs); AfriNIC, APNIC, ARIN, LACNIC and RIPE NCC. Each RIR administers a range of IP addresses, which in turn is handed out to National Internet Registers (NIRs) or Local Internet Registries (LIRs). In the end Internet Service Providers (ISPs) obtain a range of IP address that they allocate to their customers thus guaranteeing a worldwide unique address.

The main benefit of an hierarchical addressing structure is illustrated with the following example; a mailman in country X does not need to know anything about addresses and places within country Y, he only needs to know the main postal office in Y, which in turn will distribute the mail within Y. The same holds for computer networks, where one device does not need to know the exact location of hosts within another network. It only needs to know one host that knows another host, that knows another host, until there is a host that is directly connected to the destination host.

The Internet Protocol also provides data encapsulation. Data encapsulation is a direct result of using an architecture that consists out of layers and where data is passed through the layers. By means of data encapsulation the transport layer protocols utilize the IP protocol while performing higher level functions themselves; separating the functionality and complexity of the network layer from providing, in case of TCP, a reliable network connection.

A fragmentation function is defined in IP, because when data is encapsulated and passed through to the network layer it makes sense that the network layer should take into account the maximum frame size of the data link layer. This is important when the IP packet is too large to fit in, for example, one Ethernet frame. In this case the network layer needs to split up the packet to fit within the constraint set by the data link layer, thus fragmenting the data. The opposite process of recovering the IP packet

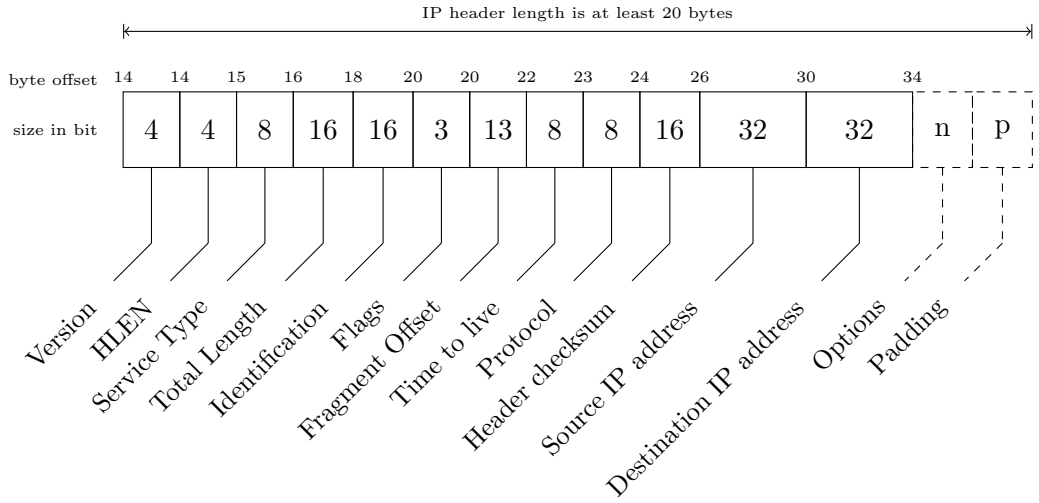


Figure 2.6: The IP packet

out of several frames is named reassembly.

2.4.1 Routing

A host needs to determine wether a packet should be send directly or indirectly before it delivers it to the data link layer. Directly means that the packet is destined to a host on the same Ethernet network and means that the packet can just be send down to the data link layer. In the data link layer the IP address is resolved by the ARP protocol to a data link layer address after which the packet is send over the Ethernet. In case of an indirect delivery of a packet the host needs to route the packet to an intermediate host, this is the case when the hosts belongs to different networks.

In *RFC791 Internet Protocol* several classes of networks are defined, see Table 2.1. By using the definition of these classes the host determines the class of an address simply by matching the high order bits of the class with the high order bits of the address. After the class is determined the format of the address is known and the network part is extracted. The network part is matched with the sending host's network when the network part is known to see whether routing is necessary.

Table 2.1: Network classes

High order bits	Network (bits)	Host (bits)	Class
0	7	24	A
10	14	16	B
110	21	8	C
111	Escape to extended addressing mode		

However soon it became apparent that the classful addressing scheme was not scal-

able. Medium sized companies had to use class B networks since a class C network only allows for the allocation of a maximum of 254 IP addresses, or hosts. This led to an exhaustion of the class B network address space. A solution was presented in the standard RFC1519 Classless Inter-Domain Routing (CIDR) where the network and host separation was no longer fixed to 8, 16 or 24-bit boundaries, but became variable. In CIDR an IP network is represented by a prefix, which is an IP address and the length of the mask. The length of the mask can be written as $/n$ where n is the number of network bits. For example, the network 192.168.1.0 255.255.255.0 can be written as 192.168.1.0/24. CIDR allows for a further subdivision of larger networks into smaller ones, thus giving ISPs the tool to hand out an adequately spaced smaller network to companies that want to be connected to the Internet.

With classless routing a packet needs to be forwarded when the network prefix of the destination host and sending host do not match. Whether they match or not is determined by applying a bitmask, which starts out with a number of 1s equal to the prefix length with the remaining bits set to zero, to an IP address. The so obtained network part needs to differ with the destination network in order for forwarding to occur.

Now, although routing became more complex for ISPs for almost every other PC connected to the Internet routing boils down to sending everything that is not destined for the local network to the default gateway, which is usually the modem/router located in the premises.

2.5 The ARP protocol

In the previous section, Section 2.4, it was shown that an IP address consists out of four octets. However, an Ethernet address consists out of six octets as discussed in Section 2.3. Obviously some kind of translation is needed between addresses of the network layer and the data link layer. This is where the Address Resolution Protocol comes into play. The Address Resolution Protocol is defined in RFC 826 an Ethernet Address Resolution Protocol and was originally designed for the Ethernet as implied by its name [17]. The protocol is however general applicable and can be used to resolve any two addresses in any two lengths.

The need for address resolution stems from the fact that network addresses say nothing about the physical connection and proximity between devices, but more about where a device is placed in a hierarchical layer three network. The addresses in the data link layer are used for data exchange between hosts that are geographically close to each other, while the network addresses are used to create virtual networks that span the entire world.

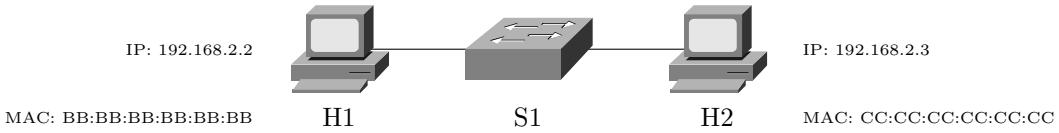


Figure 2.7: An ARP example

It is important to understand that as devices are communicating on the network layer all frame transmissions are still done through the data link layer. This is easier to visualize with Figure 2.7. A packet from host H1 destined for H2 is forwarded first to S1 before it reaches H2, even though both hosts are directly connected in the network layer since they are within the same 192.168.2.0/24 network.

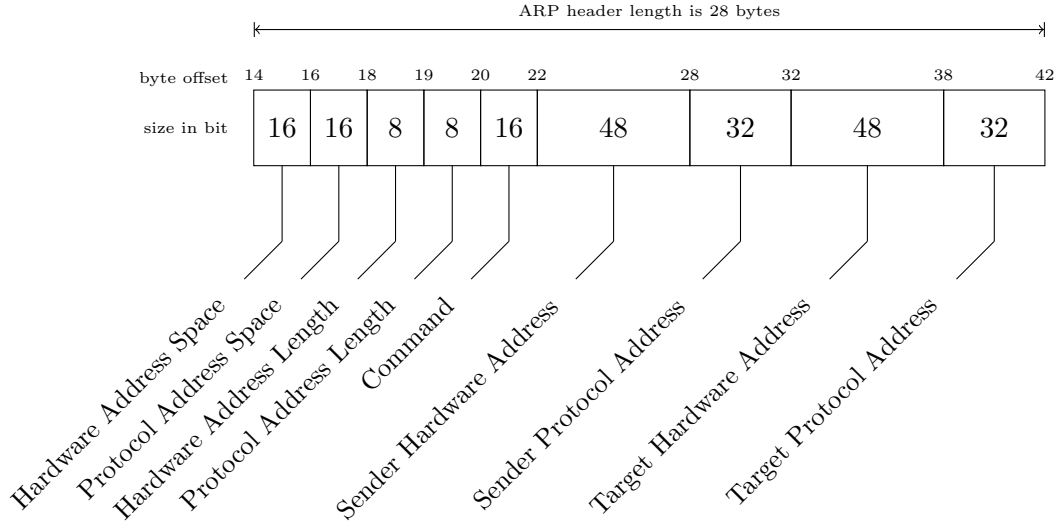


Figure 2.8: The ARP packet format

An ARP packet used for translating IP addresses to Ethernet addresses is depicted in Figure 2.8. In the packet four bytes are used for the protocol address fields and six bytes for the hardware address. The protocol defines two types of messages; a request and a reply. The message type of a packet is defined by the value set in the opcode field. The length of both the hardware and protocol addresses is specified in their respective fields to support addresses of an unfixed length. The length of the protocol address is also implicitly defined by the protocol type (the protocol address space field). The same holds for the hardware address length which is derived from the hardware type indicated by the hardware address space. This redundancy is according to the specifications included for consistency checking, network monitoring and debugging. The known hardware and protocol addresses of sender and receiver are found in their respective fields.

A typical ARP sequence is depicted in Figure 2.9 where host H1, see Figure 2.7, is about to send an IP packet to host H2 by Ethernet, but currently does not yet know what the hardware address is of host H2. To simplify the situation the switch is

assumed to be replaced by a direct link.

To conclude the discussion of the address resolution protocol the following list shows a typical step-by-step interaction between two hosts trying to resolve an address;

1. Host H1 has queued up an IP packet and passes it on to the data link layer.
2. Host H1 tries to lookup the IP address in its cache. If the address is cached it is used and the frame is send out.
3. If the address was not cached host H1 will generate an ARP request. The hardware destination address field is not set since that is what needs to be determined.
4. The ARP request is broadcasted on the Ethernet and now host H1 either receives a reply or the request will time out.
5. Host H2 receives the ARP broadcast, as it is in the same broadcast domain, and determines it is the target for the ARP request by matching the configured IP address with the destination protocol address field.
6. Host H2 generates a reply in which the source address and destination address are swapped and the MAC address is inserted in the source hardware address field.
7. It is very likely that in the near future H2 needs to send packets to H1, therefore host H2 inserts host H1 protocol address and the matching hardware address in its cache for future use.
8. The ARP reply is send to host H1, although this time it is not as broadcast but send as a unicast frame.
9. Host H1 receives the generated ARP reply and inserts the correct destination MAC address for the IP packet.
10. Host H1 caches the MAC address for future lookups and to prevent a flood of ARP request on the network for every IP packet that needs to be transmitted.

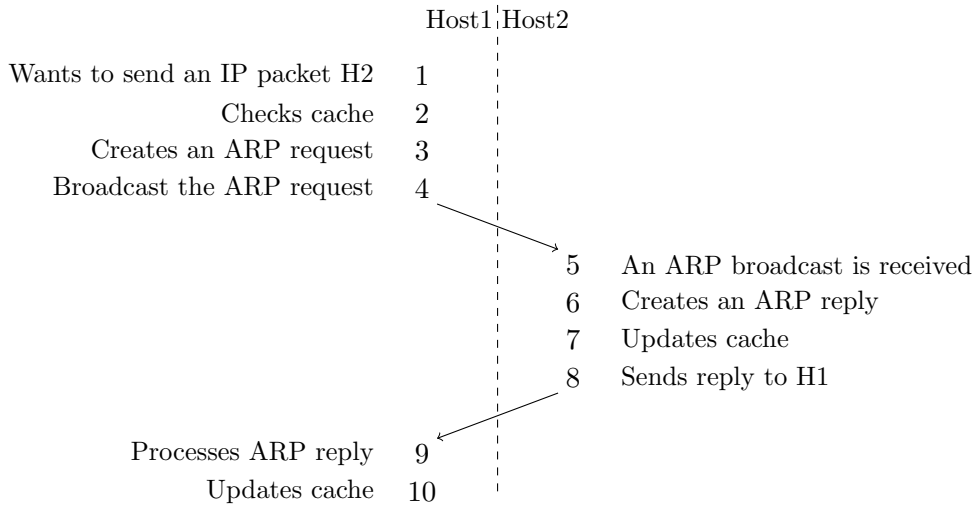


Figure 2.9: A typical ARP sequence

2.6 The ICMP protocol

When a host sends out IP packets there is no guarantee that all or even some of these packets will eventually arrive at their destination. There can be no guarantees, because there is neither a prior connection setup nor a confirmation of the arrival of the IP packet. There are several reasons why a packet does not arrive at its destination. Some of those reasons might be solvable or preventable by the Internet Protocol, or a higher level protocol running on top of it, if there would be a mechanism in place that could notify a transmitting host of errors that happened along the way.

This mechanism exists and is defined in *RFC 792 titled Internet Control Message Protocol* [19]. RFC792 defines the ICMP protocol which is designed to be a general purpose messaging system and is extendible with other message types. A few other message types are defined in RFC1256 ICMP Router Discovery Messages, RFC 1393 Traceroute using and IP Option and RFC 1812 Requirements for IP Version 4 routers. A new version, which is an IPv6 version, of the ICMP protocol is defined in RFC1885 and revised in RFC2463.

ICMP is an integral part of the IP protocol and must be implemented by every internet protocol stack. ICMP messages can be generated by a fault trigger routine somewhere along the path of a traveling packet. Such a message is itself an encapsulated Internet Protocol message and transmitted like any other ordinary IP packet, thus ICMP does not receive any special treatment from the Internet Protocol.

ICMP does not only define error messages, but also messages that are more informational in nature. A clear distinction is made for ICMP IPv6 in RFC2463, where error messages have a message type form 0 to 127 and informational messages have a message type from 128 to 255. Typically, informational messages are used for testing purposes. The mechanism to trigger the generation of an informational message is by specific queries, either by the end user or a device, this in contrast to error messages

which are triggered by errors.

An ICMP message is classified by the value set in the type field in the ICMP message format. The type field is eight bits wide and allows for the definition of up to 256 different message types. Each message type is further divided by a subtype value named code. Just like the type field the code field is eight bits wide and thus allows for up to 256 subdivisions of the message. The values assigned to the message types and subtypes are assigned by the Internet Assigned Numbers Authority (IANA) and the currently assigned numbers for ICMP are found in RFC1700.

The ICMP only defines the message format and the exchange of its messages. The protocol does not perform any specific action on any of the messages. All ICMP messages, independent of their purpose, have a few fields in common. There are three of those common fields; a type field, a code field and a checksum field as depicted in Figure 2.10. The checksum field is an 16-bit checksum and is calculated over the entire ICMP message to detect errors. The format and contents of the message body depends on the type of the ICMP message. Two examples of messages that are supported by this project are discussed in the following two section.

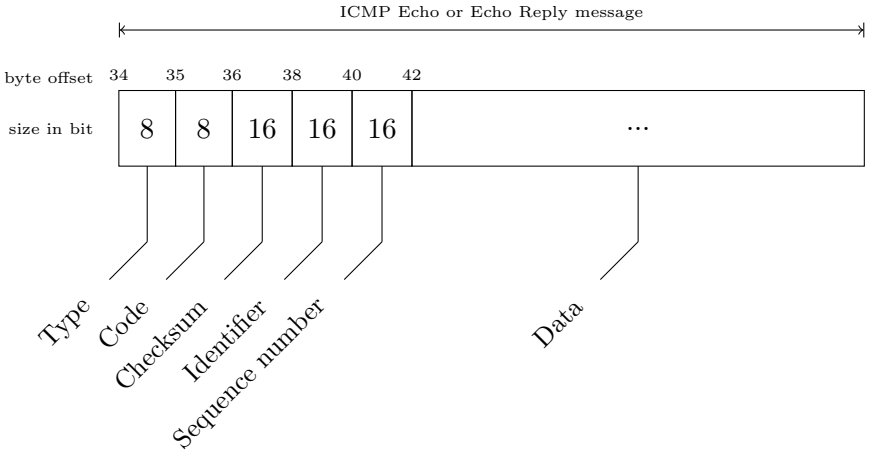


Figure 2.10: The ICMP message format

2.6.0.1 Destination unreachable

As discussed before, the Internet Protocol does not guarantee the arrival of a packet at its destination. This is usually not a problem since higher level protocols like TCP run on top of IP and create a virtual connection with error correction and rate limiting and by using a handshake protocol they provide a reliable network connection. However, a situation can occur where it is impossible to reach the destination. Possible reasons for such an occurrence are routing problems or simply that the destination host is not connected to the network. In those cases it is more efficient to notify the host about the fact that the destination is unreachable then to let the transmitting host continue to send packets which will never arrive.

The ICMP defines a destination unreachable message type that is used by the IP protocol to notify a packet sending host of a failure somewhere along the way. When the sending host receives one of those messages it knows that there is a problem and can decide to undertake an action. To help the stack to diagnose the problem the destination unreachable message includes a portion of the original packet that caused the failure, in particular the IP header and a minimum of eight bytes of the original data datagram.

The Destination unreachable message type is further divided into twelve subtypes through the code field. For example, when the code field is set to a value of one it means that the destination host, as defined in the IP header in the message body, is not reachable. More importantly however it indicates that the intervening communications infrastructure up to the last host is working correctly. The last router has even send an ARP request but it simply was unanswered. Possibly, a network administrator needs to check some cables or perhaps the host is simply not connected or turned on. When the code field is set to a value of three it means that the destination port is unreachable, the sending host might decide in this case to take action and contact the destination on a different port.

2.6.0.2 Echo or Echo Reply Message

A frequently used tool to debug computer networks is the ping utility. The ping utility is used to test the reachability of a host and can be used to measure the latency between two hosts. Besides the default values there are a range of other parameters that are set; for example, the size of the ping message, the source IP address and the amount of ping messages send. The utility gives a lot of information about the network, for example the round trip delay of a packet, calculated by measuring the amount of time elapsed between sending a ping and receiving the echo, or the amount of successful received echo compared to the amount of send pings.

The ping utility works by transmitting informational ICMP Echo or Echo Reply messages. The code field in this ICMP packet is set to a value of zero, since there are no subtypes defined for this specific message type. Three unique fields for this message type are; an identifier field, a sequence number field and an optional data field. The identifier and sequence number fields are used by the echo message sender to determine which echo replies match with which echo requests. A host can determine which of the Echo messages was successful by matching the identifier and sequence number of the received packets. This is important to know, since IP itself does not even guarantee that packets are received in the same order as they were send and usually the first Echo Message is dropped if the last hop needs to ARP to the destination. On the contrary, a missing Echo Reply somewhere after the first Echo message might indicate connection problems.

Concluding, the ICMP gives a user or protocol a wealth of information about a failure or a triggered query. ICMP can be used to notify an end user of connection problems and can be used to diagnose network problems.

2.7 The UDP protocol

The User Datagram Protocol (UDP) is defined in a very short request for comments document named RFC768 - User Datagram Protocol [18]. The User Datagram Protocol is a best effort datagram service in that it is an unreliable and connectionless protocol. Although unreliable does not sound like a feature it can actually be one depending on the requirements set by the application layer.

The unreliability stems from the fact that there is no guarantee that the receiver will actually receive the datagrams in the right order, or even receive them at all as was the case with the Internet Protocol. The only guarantee given by the UDP protocol is that the datagram contents will arrive without any data corruption. This does not mean that the packets never have any data corruption, but merely that when a cycle redundancy check fails the datagram is dropped and not send to the UDP layer. Due to the fact that there is no additional overhead for error-checking above the packet level, as there is with TCP, UDP data can be much faster than TCP, which becomes apparent when considering that the minimum transaction time for a UDP request-reply is the round trip time plus the processing time needed by the server. With TCP however the minimum transaction time, with assuming that a connection has been setup, is two times the round trip time plus the processing time needed by the server.

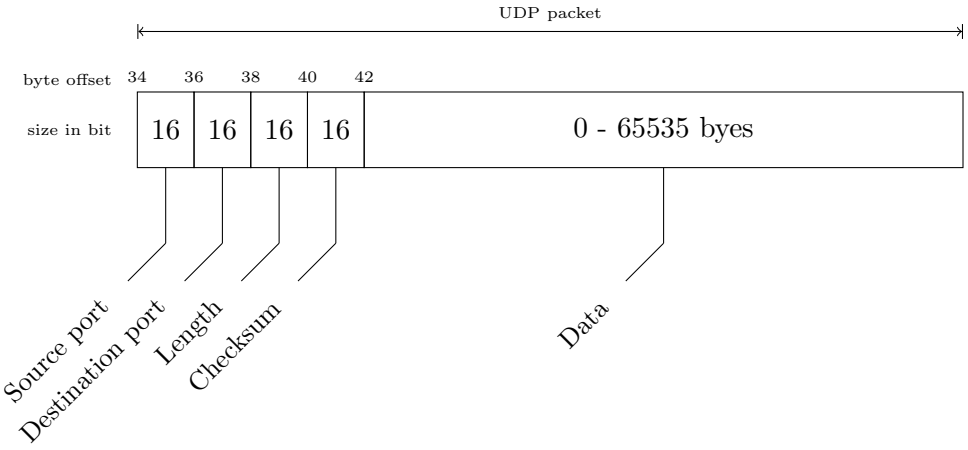


Figure 2.11: The UDP message format

In multimedia applications, such as Voice over IP, real-time video conferencing and audio/video streaming there is no need for congestion control and retransmissions. Retransmissions are not useful since by the time the lost packet has been resend several round trip delays have been passed and by the real-time nature of the application there is no longer need for the packet. In addition, the reliable service that TCP provides can introduce an unacceptable jitter due to the fact that an application needs to wait for a retransmitted packet, while the next packets could already have queued up.

The UDP has one function that IP itself does not have, namely its multiplexing feature. A UDP header consists out of four fields each sixteen bits wide, as depicted in

Figure 2.11. The source port and checksum fields are optional fields in that they do not have to be filled in with meaningful data. The port fields are used to distinguish and multiplex UDP packet flows. Generally, applications request a specific port number or are assigned one by the operating system.

The source port is used by the destination to determine to which port it needs to forward its answer. The destination port is usually set to a specific port number of a well known application. For example, if one wants to use the DNS protocol, which runs on top of UDP, the destination port is set to 53. Like with ICMP the IANA keeps a list of officially assigned port numbers and their respective applications. The length field specifies the length in bytes of the entire datagram; the length of the header and data combined. As a result the minimum length is eight bytes, since that is what it takes to send the four fields of the UDP packet.

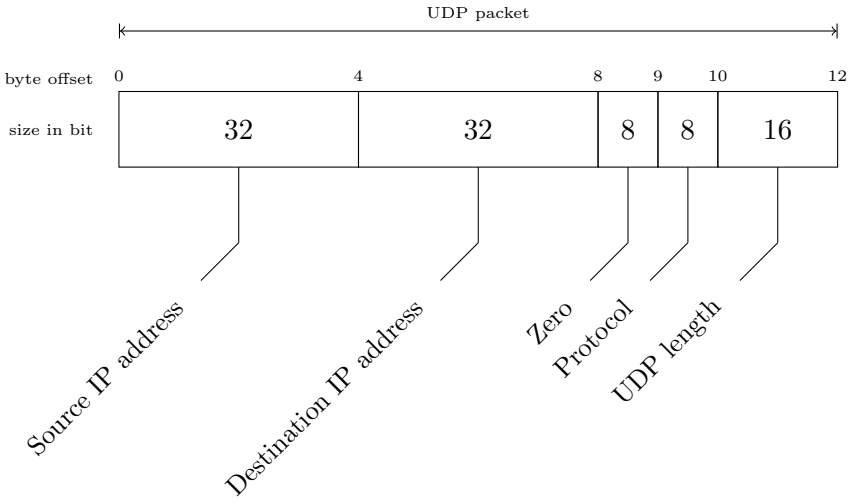


Figure 2.12: UDP Checksum header

A value in the checksum field is optional. The field should be filled with zero to indicate it is empty when not used. To distinguish between a zero value and an empty value a calculated checksum of zero should be set as a negative zero. The exact method of calculating the checksum, as specified in the RFC, is the 16-bit one's complement of the one's complement sum of the UDP header, the data which is padded with zero octets at the end to finish the packet on a word boundary and a pseudo IP header. The pseudo header, depicted in Figure 2.12, contains the source address, the destination address, the protocol and the UDP length. Although the pseudo header is used for the checksum calculation it is not actually transmitted along with the data. The inclusion of an IP address within the pseudo header is a layer violation and will make it difficult to run UDP on top of any other protocol then IP. Another implication is that systems, such as NAT devices, which modify the IP address need to modify the UDP-layer checksum.

2.8 Conclusions

This chapter presented the basic background knowledge to understand the context of the overall design. In order to fulfill the requirements set out in Chapter 1 and keep the design as resource efficient as possible an embedded processor needs to be selected and implemented. An embedded processor has several advantages over creating a complex state machine, for instance; it provides an easy debugging interface, makes it possible to emulate complex state machines in easy to understand assembly language and keeps the design small, since the complexity of the program does not alter the physical implementation of the processor.

Several publicly available embedded processors are found on the Internet. A major drawback of most of these microcontrollers is that they are not designed with resource efficiency in mind, but are designed to be backwards compatible with some existing microcontroller. Two alternatives are given which are designed towards resource efficiency; one made by Xilinx the other by Altera. The development board for this board was selected before the microcontroller and thus the Xilinx implementation was chosen due to vendor lock in.

In addition to the microcontroller, this chapter also discussed a number of protocols that are implemented in the design so that UDP packets can be received from and transmitted to other hosts. The first standard discussed was the Ethernet standard. A history of the Ethernet was given to show the natural evolution of the Ethernet towards full duplex links. For this reason, this project is designed for solely full duplex links, keeping the design small and simple, since no collision detection or sorts need to be implemented.

Other protocols discussed in the chapter are the Internet Protocol, the Address Resolution Protocol, the Internet Control Message Protocol and the User Datagram Protocol. The Internet Protocol provides a basic datagram delivery service. In particular the addressing scheme is discussed that determines where to forward packets to. The Address Resolution Protocol provides in essence a lookup table between four byte IP addresses and six byte Ethernet addresses. A lookup is necessary since the addressing scheme of higher level protocols, like IP, differ from lower level hardware schemes like MAC addresses. To support a few well known network diagnostic utilities a small subset of the Internet Control Message Protocol is discussed and implemented in the design. Finally, the User Datagram Protocol is discussed, which provides access to the Internet Protocol to the application layer and adds a multiplexing feature. By using the multiplexing feature a great number of applications can run and make concurrent use of the UDP/IP stack. The UDP protocol is implemented in this design, because it is belongs to the minimum set of protocols needed to transmit data from an FPGA towards another host on the Internet or on a local area network.

3 Implementation details

This chapter describes the general architecture as well as the specialized hardware blocks. The first section of this chapter presents the hardware platform together with the relevant modules on the platform. The second section, Section 3.2, gives a global overview of the software written for the Picoblaze and the interface presented to the outside world. Section 3.3 describes the hardware implementation and gives an overview of the resources required in the reconfigurable logic. Finally, Section 3.4 summarizes the chapter.

3.1 Platform specifications

Many different FPGA demo boards are available. Prices for these boards range from as high as $70.000 to as low as $50[1]. The board selected for this project is the Spartan 3E development kit, which is manufactured by Digilent and sold by Xilinx at a price of $149. The board is positioned by Xilinx as the preferred development kit for the Xilinx Spartan 3E family. The board is depicted in Figure 3.1.

The development platform is build around a Xilinx Spartan-3E FPGA. The kit is designed to provide a prototyping platform to a wide range of embedded systems [27]. The Xilinx Webpack software package is used in combination with this platform to write, compile and upload the HDL to the board and is freely available at the Xilinx web site [29]. The following list shows a complete overview of the components present on the board;

- Xilinx XC3S500E Spartan-3E FPGA
- Switches, buttons and a knob
- Clock source
- Character LCD Screen
- VGA Display Port
- RS-232 Serial Ports
- PS/2 Mouse or Keyboard Port
- Digital to Analog Converter
- Analog Capture Circuit
- Intel StrataFlash Parallel Flash PROM

[1] source: http://www.fpga-faq.com/FPGA_Boards.shtm

- SPI Serial Flash
- DDR SDRAM
- 10/100 Ethernet Physical Interface
- Expansion Connectors

The relevant components for this project are the serial ports (Section 3.1.1), the character LCD screen (Section 3.1.2), the 10/100 Ethernet physical interface (Section 3.1.3 and the FPGA itself (Section 3.1.4). The Ethernet component consists of the physical layer (PHY) interface and a RJ-45 connector, the Media Access Controller (MAC) is implemented in the FPGA. An Ethernet connector is used to connect the board and a PC together in order to form a small computer network. Serial interfaces are used as a debugging interface between the PC and the UDP/IP stack. The following subsections give a brief overview of the components, a more detailed description of all components is found in [27].

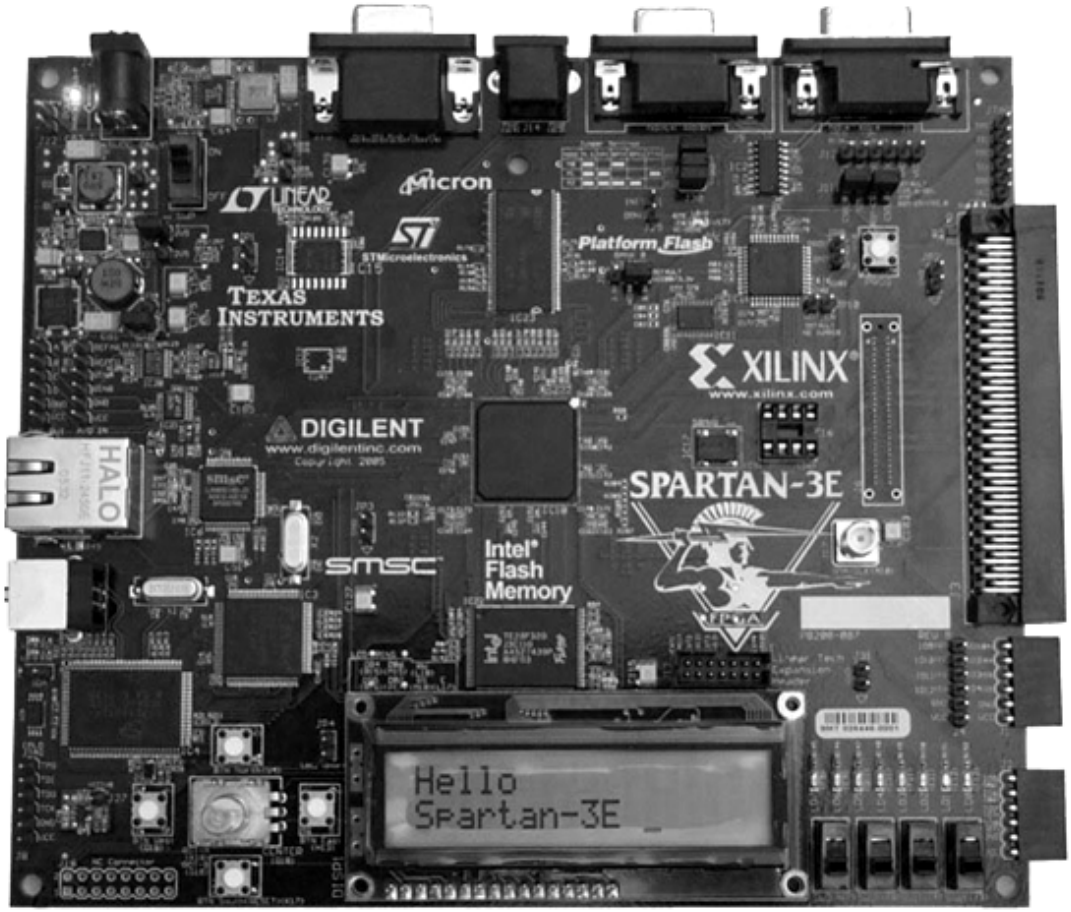


Figure 3.1: The Xilinx Spartan-3E development board

3.1.1 The serial ports

The board contains two serial ports; a female DB9 DCE connector and a male DTE connector. The DCE-style port is used for debugging purposes and is connected to a PC. The DCE-style serial interface consists of a Max3232 line driver connected to a few of the FPGA pins. The function of the line driver is to transform the low voltage outputs of the FPGA to the higher voltage levels needed for the RS-232 standard, a standard used by devices for serial data transmissions. The voltage transformation is done by charging external capacitors, so called charge-pump capacitors, and using the stored energy to create the needed higher voltage levels. For this design there are no handshaking signals and only the Rx and Tx signals are used.

3.1.2 The LCD

The LCD interface on the Spartan-3E Starter Kit board is used for the demo application provided with this project's UDP/IP stack. It features a 2-line by 16 character display that is controlled via the 4-bit data interface with the FPGA. The character LCD has an internal ST7066U graphics controller. The graphics controller has three internal memory regions, the Display Data RAM (DDRAM), the Character Generator ROM (CG ROM) and the Character Generator RAM (CGRAM). The CG RAM and ROM provide space for 5-dot by 8-line character bitmaps and are referenced by their respective character codes. The DDRAM stores the character codes to be displayed on the screen, see Figure 3.2. The demo application only interacts with the DDRAM by writing the character codes on the character display addresses that are mapped to the display. For example writing the hexadecimal character code 0x53, which is the character code of the letter 'S', to the character display address 0x41 results in the letter 'S' showing up on the second position of the second line.

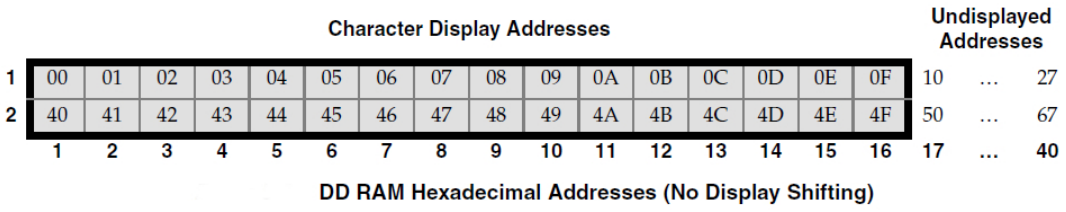


Figure 3.2: The LCD character display addresses

3.1.3 10/100 Ethernet physical layer

The development board includes a Standard Microsystems LAN83C185 10/100 Ethernet Physical layer (PHY) interface and a RJ-45 connector [20]. The LAN83C185 is a fully IEEE 802.3/802.3u compliant analog interface IC and allows embedded Ethernet applications on the FPGA. It contains a full-duplex 10BASE-T/100BASE-TX transceiver and supports 10 Mbps (10BASE-T) and 100Mbps operation with unshielded twisted-pair cables. The FPGA connects to the LAN83C185 using a standard Media Independent

Interface (MII) and combined with an Ethernet Media Access Controller the board provides an Ethernet connection to the network.

3.1.4 Xilinx FPGA

Xilinx is the market leader in programmable logic, where the reconfigurable logic is implemented in Field Programmable Gate Arrays. FPGAs are "off-the-shelf" chips that a developer programs to perform a specific function, this in contrast to chips "programmed" by the manufacturer during the manufacturing process, so called Application-Specific Integrated Circuits. Where historically FPGA chips were only used as a replacement for discrete logic chips, they are now more and more used for design integration. The main reason for this change is that FPGA chip costs are approaching the costs of the equivalent ASIC. Using readily available FPGAs increases the product design flexibility and gives a faster time-to-market.

For this project a XC3S500E FPGA is used. This chip is part of the Xilinx Spartan-3E FPGA family, a low cost series suitable for high volume applications. Densities in the family range from 100,000 to 1.6 million system gates, which should be big enough to leave a significant amount of free gates after the UDP/IP stack is implemented. The Spartan-3E family does not come with an on chip microprocessor making it relatively easy to retarget the design to a different Xilinx family or to an FPGA of a different vendor.

3.2 Software implementation

The Berkeley Socket Interface (BSI) was part of the 4.2 release of the original Berkeley distribution of the UNIX operating system that contained the TCP/IP protocol stack and was released in 1983 [21]. It was not until 1989 however that the University of California, Berkeley could release its operating system together with the networking library completely free from the constraints set by AT&T on its copyright protected UNIX. Since that time, the Berkeley Socket API forms the de facto standard for abstraction of network sockets and is implemented in this design.

3.2.1 UDP sockets

Berkeley sockets, also known as the Berkeley Software Distribution (BSD) Socket API, is an Application Programming Interface (API) that provides a library of functions for performing inter-process communications through the use of C code. Nowadays many other programming languages use a similar interface as provided by the C API and all modern operating systems have some kind of Berkeley based socket interface implementation. The API provides a high level abstraction of many different kinds of I/O devices and their drivers by wrapping them up in an abstraction layer called *internet sockets*.

A socket is the BSD method for Inter-Process Communication (IPC), which allows concurrently running processes to exchange data through sockets. Some IPC mechanisms only support data exchange between local processes, while others allow data exchange

between geographically dispersed machines. Sockets are of the latter type, providing both local and remote data exchange and were designed as a standard way to support many different kinds of communication protocols and data streams.

Originally, sockets were treated as files. The UNIX system calls *open()*, *read()*, *write()* and *close()* are used on both files and sockets. An example of this similarity is that when a file is being created by calling the *open()* function an integer is returned. This integer, referred to as the file descriptor, is then used to change the file. The same holds for sockets where a *socket()* call returns a socket descriptor that gives access to the socket. A *socket()* call opens a socket and takes two key arguments; a domain parameter to select the protocol family, and a type parameter to select the protocol.

There are currently about ten supported protocol families. The one used in this implementation is the IP_INET family, which is the Internet family for IPv4. The widely popular protocols TCP, UDP and IP belong to this domain. Internet sockets provide five different types of access. The five defined type are SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, SOCK_SEQPACKET and SOCK_RDM. SOCK_RDM is not yet implemented and SOCK_SEQPACKET is only used for PF_NS. A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams and is implemented on top of TCP. On the other hand a SOCK_DGRAM socket supports datagrams, which are connectionless, unreliable messages of a fixed maximum length. Finally, the most powerful, versatile and low level interface, the raw socket SOCK_RAW, enables a process to read and write IPv4 datagrams with an IPv4 protocol field that is normally not supported by the operating system. An example, the Internet Group Management Protocol (IGMP), a communication protocol used to manage the membership of multicast groups does not use TCP or UDP but uses the IP layer directly through raw sockets.

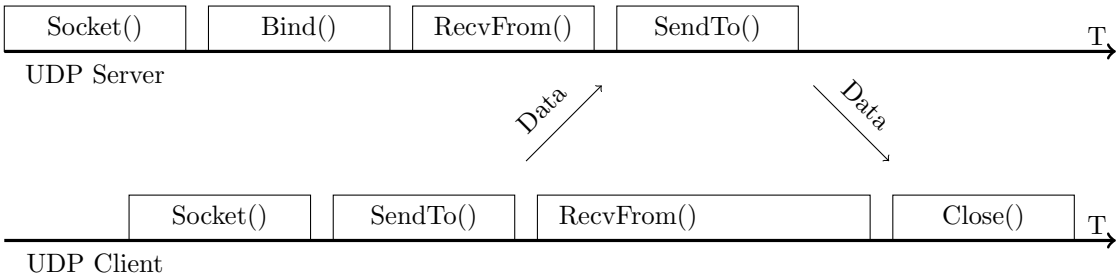


Figure 3.3: An UDP data exchange between client and server

A typical UDP setup looks like Figure 3.3, in which a client tries to exchange data with a server. The server started by opening up a socket with a *socket()* call specifying the domain, PF_INET, and the type SOCK_DGRAM and it sets the protocol to IPPROTO_UDP. Then it binds the socket to an address with a *bind()* call specifying the socket descriptor, an address structure and the length of the address structure. The server calls the function *RecvFrom()*, which blocks until a datagram is received and returns the length of the received datagram. After receiving the datagram it is likely that the server wants to send some data back and does so with a *SendTo()* call. From the clients perspective it is all quite similar except that it will start by sending the data,

since it initiated the communication, and will wait for possible data before eventually closing down the socket with a *close()* call.

3.2.2 Signals

In UNIX, signals are used to notify running applications that a certain event has occurred. Usually they interrupt the running process and expect to be handled immediately. Each signal is identified by a unique integer number and a symbolic name. A signal can have a signal handler, a function that handles the received event. A signal is in principle similar to a hardware interrupt, except that the operating system sends the signal instead of a hardware component.

Although in UNIX many different signals are defined only one is used in this implementation. The SIGIO signal is used in combination with non-blocking socket calls. A common problem associated with asynchronous IO is that there is no way of knowing when to expect data. At the same time the default behavior of a socket call is to stop the program until the requested function is completed. One can expect that a lot of time is wasted on waiting for data with a *RecvFrom()* call. An alternative way of retrieving data is to configure the socket for non-blocking IO so that all calls made to the socket are non-blocking. A non-blocking function will return immediately with a value indicating either a success or a failure. Instead of having to wait for the return of the function, the function is now polled periodically until the function returns successfully. The CPU can now used between the polls for more useful work than just idling. An even better solution is to do no polling at all but instead listen for the SIGIO call with a signal trap. This frees up the CPU completely from waiting for data and allows other tasks to run, while it is still ready to receive any data at any time.

3.2.3 Sockets and signals implemented

Concluding, the Berkeley Socket Interface provides an abstraction for network sockets and allows client/server communication with just five different functions. Combined with a signal the interface provides an easy access to the UDP/IP stack, where received data is indicated by an interrupt. This functionality is implemented in order for an easy access to the UDP/IP stack, without inventing a complete new interface.

The physical interface consists of two sets of register banks. Each bank consisting out of sixteen eight bits registers, as depicted in Figure 3.4. One register bank, the IN bank, is only written by the application running alongside the stack, while the other bank, the OUT bank, is only written by the UDP/IP stack. Both banks are read by both the application and the stack.

There are three types of registers defined. The first type is *Command ID* and is filled with any arbitrary number, as long as it is not the same number as used in the previous command. The *Command ID* is used by the UDP/IP stack to check whether there is a new command present in the registers, while the application reads the OUT *Command ID* to check if the command is completed. For both cases this is determined by checking the current value with the previous value. The stack is busy as long as the IN and OUT *Command IDs* differ in value. The second type is *Command* and dictates

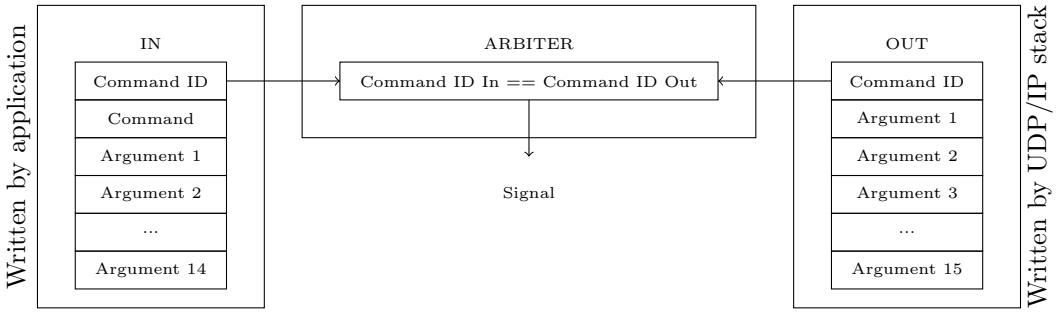


Figure 3.4: The UDP/IP stack interface

which command is executed on the UDP/IP stack, since only the application can request actions there is no *Command* register in the OUT bank. The third and last type is a *Argument* type and can have any value and its meaning is determined by the *Command* that is to be executed. A list of commands similar to the Berkeley Sockets is shown in Table 3.1 and a list of additional commands supported by the stack is shown in Table 3.2.

In principle all functions are blocking functions, with the exception of the *RecvFrom()* call. The advantage of having a non-blocking *RecvFrom()* is that it allows multiple applications to poll the stack and request an incoming packet. With a blocking function it would not have been reasonably possible to use the stack concurrently, for the simple reason that one application could block the stack forever by requesting a packet that might never arrive. Instead of polling the stack one could also use the earlier mentioned method of signalling the application.

3.2.4 The program flow

A high level program flow for this UDP/IP stack is depicted in Figure 3.5. The program starts by initializing several variables, such as the OpenSockets variable and the LastMessageSize variable. After the initialization of the stack the MAC module is resetted and the first frame is requested. The MAC module will now continuously check if it has received a valid new frame. If it did, the frame is copied to a separate RX buffer and a bit in a special function register is set to indicate that the copying is completed. Periodically the stack will check if there is a new packet present by checking the value of the just mentioned bit.

If there is no new frame present in the receive buffer, the stack will check the command registers for a new command, as is discussed in 3.2.3. A change in the *Command ID* register indicates an UDP/IP stack function call by one of the applications. The value of the *Command* register is loaded and compared in sequence with the constants attached to the functions, as depicted in the command value column in Table 3.1. The functions *SendTo()* and *RecvFrom()* are positioned at the top of the command list to reduce the inherent delay of checking each constant.

Figure 3.5 has been simplified by leaving out the handling of the commands and

Table 3.1: UDP/IP stack commands

Command value	Command	Arguments IN	Arguments OUT
07	Socket()		Result (FF=Error, !FF=Ok) SocketId
08	Bind()	SocketId PortH PortL	Result (FF=Error, !FF=Ok)
0B	RecvFrom()	SocketId BufferAddressH BufferAddressL	Result (!00=Error 00=Ok) SourcePortH SourcePortL SourceIPAddress[3] SourceIPAddress[2] SourceIPAddress[1] SourceIPAddress[0] LengthH LengthL
0A	SendTo()	SocketId LengthH LengthL BufferAddressH BufferAddressL UDPDestinationPortH UDPDestinationPortL IPDestinationAddress[3] IPDestinationAddress[2] IPDestinationAddress[1] IPDestinationAddress[0]	
09	Close()	SocketId	

protocols. The handling of the commands such as *getIP()*, *setIP* or *getNetMask()* is relatively easy. The value of the currently assigned IP address, gateway address and the netmask are stored in the scratchpad of the Picoblaze microcontroller. The commands simply call a copy function with a pointer and a length as arguments. The copy function then copies the contents of the scratchpad to the argument registers in the OUT register bank, thus returning the requested value.

A more complex command is *SendTo()*, which is responsible for sending out UDP traffic. Normally only one packet is placed in the transmit buffer at any given time. This works well for larger packets where the amount of data is large compared to the overhead caused by processing the packet. For large packets the achievable throughput is close to 100% of the theoretical throughput. However small frames with a size of around 64 bytes suffered heavily of having only one buffer, mainly because a packet can

Table 3.2: Additional UDP/IP stack commands

Command value	Command	Arguments IN	Arguments OUT
0C	GetHardwareAddress()		MACAddress[5] MACAddress[4] MACAddress[3] MACAddress[2] MACAddress[1] MACAddress[0]
01	GetIPAddress()		CurrentIPAddress[3] CurrentIPAddress[2] CurrentIPAddress[1] CurrentIPAddress[0]
02	SetIPAddress()	NewIPAddress[3] NewIPAddress[2] NewIPAddress[1] NewIPAddress[0]	
03	GetGateway()		CurrentGatewayIP[3] CurrentGatewayIP[2] CurrentGatewayIP[1] CurrentGatewayIP[0]
04	SetGateway()	NewGatewayIP[3] NewGatewayIP[2] NewGatewayIP[1] NewGatewayIP[0]	
05	GetNetMask()		CurrentNetMask[3] CurrentNetMask[2] CurrentNetMask[1] CurrentNetMask[0]
06	SetNetMask()	NewNetMask[3] NewNetMask[2] NewNetMask[1] NewNetMask[0]	

not be copied to the transmit buffer while there is still another packet stored in it. To improve the throughput the buffer is split up in two equal sized parts. The buffer acts as a single large buffer for packets with a size larger than 768 bytes, however if there are two consecutive small packets, then each packet is placed in only one half of the buffer. In this way one packet stays in the buffer accessible by the MAC module, while the stack works on the second packet in the other half. After the data is copied from the application buffer to the internal transmit buffer a pseudo header is placed before

it, as discussed in Section 2.7. Furthermore an internal checksum calculator build in reconfigurable logic is activated. The result of the calculation is placed inside the UDP header and the pseudo header is dropped. Now the UDP packet is encapsulated in an IP packet and the destination IP address is copied from the IN register bank. After the IP header is formed the stack calls an internal send function which checks if the destination IP address is a local address or a remote address, for definitions of local or remot address see Section 2.4.1. If the address is a local address it is immediately looked up in the ARP table. However, if it is a remote address, the default gateway is used and looked up in the ARP table. The MAC address belonging to the IP address is copied directly to the transmit buffer if the address is found in the table. Alternatively, the address is not found, which leads to dropping of the packet and the generation of an ARP request in the transmit buffer. Finally, whatever is currently placed in the transmit buffer is send away.

Furthermore the stack itself handles all ARP and ICMP traffic. Whenever a packet is placed inside the RX buffer the stack checks first if the packet is an ARP request. If so, the stack waits for the complete transmission of an outgoing packet and will then start building an ARP reply in the transmit buffer with its own IP address and MAC address filled in. The stack will store the remote host's MAC and IP address for both the ARP reply and the ARP request. The stack also builds automatic replies for received ICMP echo request, where the building entails copying the received frame to the transmit buffer, change the message type to a reply and adjust the checksum.

3.3 Hardware implementation

This section will discuss the overall system as implemented in reconfigurable logic. Before the actual implementation, several decisions had to be made in terms of compatibility, speed and area. One such a decision is when to comply with the RFC documents applicable to this design and when not. For example, it is understandable that half duplex Ethernet is required for backwards compatibility by the RFC documents. However, virtually all switches and routers support full duplex nowadays. The tradeoff here is between adhering to the standard and increasing the size of the design or optimizing the design and loosing a functionality, that will most likely never be used.

3.3.1 Accelerating modules

The stack needs to perform certain actions and it needs to perform these actions with a real-time constraint to be able to send out frames at linespeed. The amount of time that can be spend on all actions combined is the same time as it takes one frame to be transmitted for a pipelined design. This results in that maximum 275 instructions can be executed per frame, taking into account the line speed for 64 byte frame and the frequency that the Picoblaze runs on. Simply fetching the bytes from the application buffer and placing it in the transmit buffer, as happens when the frame is build, already takes 80 instructions. A more complicated task as performing an ARP lookup takes 275 instructions by itself. One can only conclude that some of these actions need to be

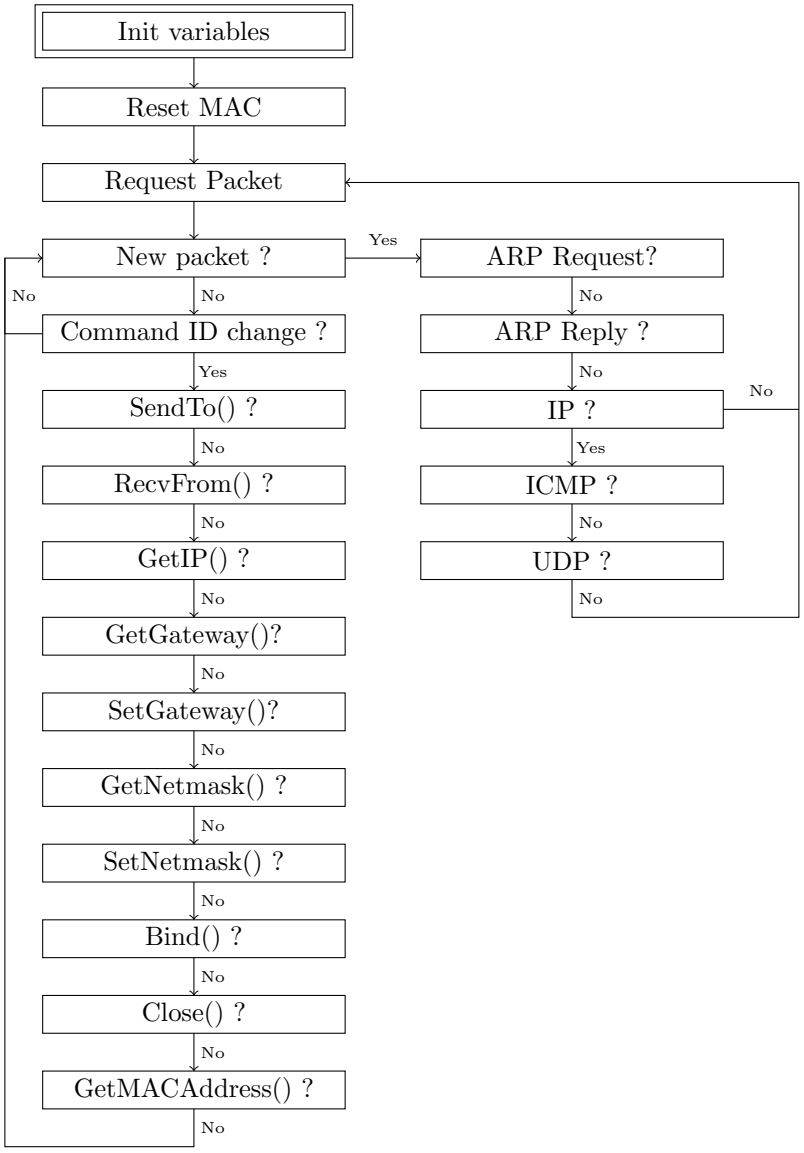


Figure 3.5: The program flow

accelerated in order to send out frames at linespeed.

One of the main problems while designing the stack was to select the functions that are implemented in hardware. The following factors are considered when this choice was made, as illustrated in Figure 3.6. First, while throughput is not set as a hard requirement it is still a factor that is taken into account, because a stack that sends out a single byte per minute is not useful for many applications.

Another important factor is the amount, and complexity, of the data that needs to be exchanged between the module and the software versus the achievable speedup. If the costs of exchanging the data are for some reason higher then the speedup of

Figure 3.6: Increase in area and speed vs increase of instructions and decrease of area

the calculation itself then this function is not a good candidate for implementation in reconfigurable logic. The costs of the data exchange is expressed in the amount of instructions it costs to perform the data exchange.

The amount of area required on the FPGA by the hardware implementation is another factor. Since one of the requirements for this implementation is that the stack can be implemented on the smaller versions of the Xilinx Spartan 3E family not every function would be suitable for hardware implementation. In some cases it might be desirable to separate a function into smaller functions in order to achieve a partial speedup while still fulfilling the requirement set for the utilized area.

At the same time not every function is implemented in software too since the Picoblaze architecture supports only up to 1024 instruction out-of-the-box and implementing all functions in software would take more than 1024 instructions. There is not only a limit to the amount of instructions, but also in the amount of available scratchpad memory.

Therefore, a good candidate function is a function which requires big chunks of Picoblaze memory and needs a large amount of instructions to be implemented. Combining all these factors it can be said that in order for a function to be implemented in reconfigurable hardware it must;

- Use a lot of memory.
- Use a lot of instructions.
- Use a lot of processing time.

Several functions come to mind with these requirements. The first one would be the ARP function, since every ARP table entry would take up four bytes for the IP address, six bytes for the MAC address and another byte for the status, adding up to a total of eleven bytes. Considering that there are only 64 bytes of memory available and one would want to store several entries in the ARP table it becomes clear that the ARP function needs a separate memory bank. Another reason to select the ARP function is the amount of time it would take to perform the function in software. As an example, a typical implementation would look like the code shown in Listing 3.1. Here it is assumed that registers R1,R2,R3 and R4 contain the IP address that needs to be searched and R4 points to the first byte of the ARP table structure. The code tries to match the

```
start:
            SUB     R4, 0B
next_entry:
            ADD     R4, 0B
            LOAD    R5, R4
loop:       FETCH   R6, (R5)
            COMPARE R6, R0
            JUMP    NZ, next_entry
            ADD     R5, 01
            FETCH   R6, (R5)
            COMPARE R6, R1
            JUMP    NZ, next_entry
            ADD     R5, 01
            FETCH   R6, (R5)
            COMPARE R6, R2
            JUMP    NZ, next_entry
            ADD     R5, 01
            FETCH   R6, (R5)
            COMPARE R6, R3
            JUMP    NZ, next_entry
found:
```

Listing 3.1: An ARP lookup routine

IP address byte by byte with a specific ARP table entry and if there is a difference the routine jumps back to try the next entry by adding the eleven bytes to the pointer.

In reconfigurable hardware the lookup is much easier. The ARP table is stored in a BRAM with a width equal to the combined width of the IP address, MAC address and status byte. A lookup now entails increasing an address counter and matching the IP address portion of the BRAM's output with a comparator. If there is a match then the MAC address is found immediately at the MAC address portion of the same output. The maximum speedup of accelerating this function in reconfigurable logic is calculated as in Equation 3.1. Here it is assumed that it is possible to hold sixteen entries in the Picoblaze memory, as is the case with the BRAM and that every instruction cost two clocks cycles to process.

$$S_{max} = \frac{T_{code}}{T_{accelerated}} = \frac{entries \cdot instructions \cdot \frac{clock\_cycles}{instruction}}{entries} = \frac{16 \cdot 17 \cdot 2}{16} = 34 \qquad (3.1)$$

Other functions that are selected for implementation in reconfigurable logic are functions that need to perform 16 bit operations and which need to process most, if not all, of the bytes in a frame. Two functions that have these characteristics are the CRC function and the copy function. The CRC function calculates the 16 bits CRC over all bytes in the frame. The copy function copies every byte from the application buffer to

```
next_byte:
            OUTPUT  R1, address_pointer_high
            INPUT   R5, (R0)
            OUTPUT  R3, address_pointer_high
            OUTPUT  R5, (R2)
            ADD     R0, 01
            ADDC    R1, 00
            ADD     R2, 01
            ADDC    R3, 00
            COMPARE R0, R6
            JUMP    NZ, next_byte
            COMPARE R1, R7
            JUMP    NZ, next_byte
finished:
```

Listing 3.2: A copy routine in Picoblaze assembly

a temporary buffer and then to the transmit buffer. Listing 3.2 shows a possible implementation of the copy function as it would be written for the Picoblaze. The function needs 10 instructions to copy a single byte.

In contrast to the software version, a hardware version can perform at a much higher speed, completing the task in the equivalent of half an instruction. For instance, with an 80Mhz clock signal it is possible to do $40 \cdot 10^6$ instructions per second (Equation 3.2), when this result is combined with the amount of bytes that need to be processed to achieve a 100Mbit/s throughput (Equation 3.3) it can be seen that a copy action should take no longer than 1 instruction per byte. This is much less than the 10 instruction per byte the Picoblaze would require. Implementing the copy function in software would by itself decrease the maximum throughput from 100Mbit/s to 10Mbit/s.

$$\frac{80MHz}{2\frac{instruction}{clock}} = 40 \cdot 10^6 \frac{instructions}{second} \tag{3.2}$$

$$\frac{100\frac{Mb}{s}}{8bits} = 12,5 \cdot 10^6 \frac{bytes}{second} \tag{3.3}$$

$$floor(\frac{40 \cdot 10^6}{2 \cdot 12,5 \cdot 10^6}) = 1 \frac{instruction}{byte} \tag{3.4}$$

3.3.2 The interface with the reconfigurable logic

Now that the accelerated modules have been chosen, the communication method between the modules needs to be selected. Several methods are used to interface the processor with the modules. One of the most important criteria for the method is the additional overhead it would cost to implement the method. The additional overhead is not only

expressed in the time it would take to transmit the actual data, but also in terms of resources required on the FPGA to implement the method.

Several options are explored that could facilitate the communication between the Picoblaze and the reconfigurable hardware modules. One way is to make use of a Direct Memory Access (DMA) controller inside the FPGA and use a large external memory. The benefits of this approach are that several modules can operate independently on the packets, that are stored in a central memory. Ideally the ARP module could look-up the IP address, while at the same time a different module concurrently calculates the checksum of the packet and updates the respective checksum field. This entire process then works with minimal input from the microcontroller. However, one could envision many applications that have no need for a large packet buffer and do not utilize a large memory for their own functioning. For those cases, adding an external memory of relative large size, compared to the size of the BRAMs, and a DMA controller would be completely undesirable.

The use of a bus topology is another option. Several bus architectures are readily available for connecting system-on-chip modules. Xilinx uses IBM's CoreConnect with the embedded PowerPC microprocessor. Altera on the other hand uses ARM's Advanced Microcontroller Bus Architecture (AMBA) to connect their embedded microprocessor to the rest of the FPGA. Another initiative is the Wishbone System-on-Chip Interconnect Architecture which is maintained by Opencores.org and is not copyrighted and in the public domain.

All these busses operate in more or less the same way. They comprise of some switching fabric, an arbiter that selects the device that can access the bus and a protocol for using the bus. Implementing one of these buses greatly benefit the re-usability of the design when the system is used in a different setup. However, the additional overhead incurred by a bus topology compared to the next option is significant. Another reason not to choose this option is that the design would not benefit much of using a standardized bus within its boundaries, since no external system would access any of its modules directly.

The simplest option for interfacing with the reconfigurable logic is using LUT look-up tables in Distributed RAM (DRAM) as register banks [25]. Instead of using one of a handful BRAMs within the FPGA, a logic block within the FPGA is used as distributed RAM. The main reason for choosing DRAM over BRAM is the limited amount of BRAM blocks available in the smaller FPGAs, four in the smallest FPGA of the Spartan-3E family and . Distributed RAM can be configured as a single-port RAM with synchronous write and asynchronous read or as dual-port RAM with one synchronous write and two asynchronous read ports. Coupling the register banks with the input/output ports of the Picoblaze results in a bus structure where the microcontroller acts as an arbiter and the switching fabric.

There are several limitations with this option. First, the microcontroller needs to participate in every communication since it needs to transfer the data between the modules and second, no direct communication between the modules is possible. Perhaps the most important drawback is that in this design the data needs to be exchanged byte wise, since the Picoblaze is an eight bit processor. The main benefits of this option are however the minimum amount of resources needed on the FPGA to implement it and the

simplicity of the solution. The microcontroller now accesses a module by just reading or writing from a register with the *input* and *output* commands.

3.3.3 UDP/IP Stack hardware implementation

The general overview of the entire design is depicted in Figure 3.7. The system consists out of the designed IP module, depicted as the inner block within the figure, and possibly several application running alongside. The applications interact with the UDP/IP stack through the host interface, which consists out of several registers and a memory interface. The general idea is that the applications and the stack share a common buffer and exchange pointers through the host interface. A function that provides an abstract interface to the UDP/IP stack, such as *read()*, *write()* and *open()*, is 'called' by setting specific registers. The results of those functions is then obtained by either polling or acting on a generate interrupt.

3.3.3.1 The *read()* function

When a packet is received it is placed within the receive buffer contained in the MAC module. Within this buffer several packets are stored before they are processed. However, if the buffer is full all new frames that arrive are dropped. After the packet is completely received and stored in the buffer a signal is generated that is read by the Picoblaze. The Picoblaze polls this signal every time as part of its main loop. The Picoblaze eventually notices the signal and processes the frame.

The first thing the Picoblaze checks is whether the frame is destined to this system by comparing the destination MAC address with its own MAC address. If it is a match, or the frame is broadcasted, then the frame is acted on. In case it is an ARP request, the Picoblaze will check if the request involves its own IP address. If it does, a reply is generated and the IP address of the sending host is stored in the ARP table for future use.

If the packet is an ICMP echo request, the Picoblaze will copy the entire packet directly towards transmit buffer, swap some fields, change the type to a reply and transmit the echo reply. The applications connected to the UDP/IP stack are not involved with handling the ICMP packets.

The packet can also be a non ICMP IP packet, in this case the IP address is checked to match with the internal IP address. If there is a match, the headers of the packet are stripped off and the data is copied to the common buffer. A register is set and an interrupt is created to notify the applications running alongside the stack. Depending on the protocol that delivered the packet, eg. UDP, several other registers may be set to indicate the destination and source port.

The receiving buffer is only about 2000 bytes in size. This is large enough to handle even the largest Ethernet frames of 1518 bytes, excluding jumbo frames. When such a large frame is received the buffer is not large enough for a second frame. It is therefore the application's responsibility to process the incoming packets as fast as possible to make space for incoming frames by calling the *read()* function.

3.3.3.2 The *send()* function

The transmission of a packet is more complex than the reception of a packet because a new header must be created, rather than just stripping one. When an application wants to transmit data, it has already stored the data without any headers in the buffer shared with the UDP/IP stack. The application exchanged the pointer to this data with the UDP/IP stack and sets the destination IP address, source port and destination port with a *send()* call.

The Picoblaze prepares the required headers in the transmit buffer and copies the data from the common buffer to the transmit buffer through a dedicated cyclic redundancy check (CRC) module after a change is noticed in the command register by polling it. This module will copy the data and calculate the CRC at the same time. A separate CRC module is necessary, since calculating the CRC within software is certainly possible, but the obtainable bandwidth will come nowhere near the Ethernet line speed.

Now that the headers are created, the destination IP address of the packet needs to be looked up in the ARP table. The lookup procedure depends on whether the destination address is located within the same network as the UDP/IP stack or not. If the IP address is a local address and thus in the same network, the address is directly looked up in the ARP table. However, if the address is found to be within a different network, then the default gateway IP address will be looked up and not the destination address. In both the local and remote case it so that if the IP address is found in the ARP table, the MAC address is copied to the destination MAC address field of the Ethernet header. However, if the address is not found, then the current packet will be dropped. This is common behavior and accepted by the RFC. Instead of the dropped packet, an ARP request is send to broadcasted the MAC address belonging to the IP address.

3.3.4 The arp module

The ARP module, depicted in Figure 3.8, provides a fast way to store and lookup MAC and IP addresses. The interface consists out of a set of registers, which are mapped onto an address space of sixteen bytes accessible by the Picoblaze. The output of each register is combined to form a wide bus. The bus presents the value of the IP address and the MAC address to the BRAM. The BRAM is configured to have two independent ports, one read and one read/write port. The bus is connected to both the ports.

The program running on the Picoblaze stores an IP address and MAC address combination by first writing their values to the appropriate registers. After the registers have been set the Picoblaze sets the first bit of the status register, which is assigned the first address allocated to the ARP module. This triggers the finite statemachine to move to the write state. The ARP module increases the ARP table's address counter in the write state, so that new entries are stored in a circular fashion, and at the same time writes the values to the current address into the ARP table.

The ARP table is also used for lookups. The procedure to lookup an MAC address is quite similar to writing an entry. The program will set the IP address registers and then set the second bit of the status register. A write to second bit triggers the finite statemachine to move to the read state. In the read state an address counter is increased

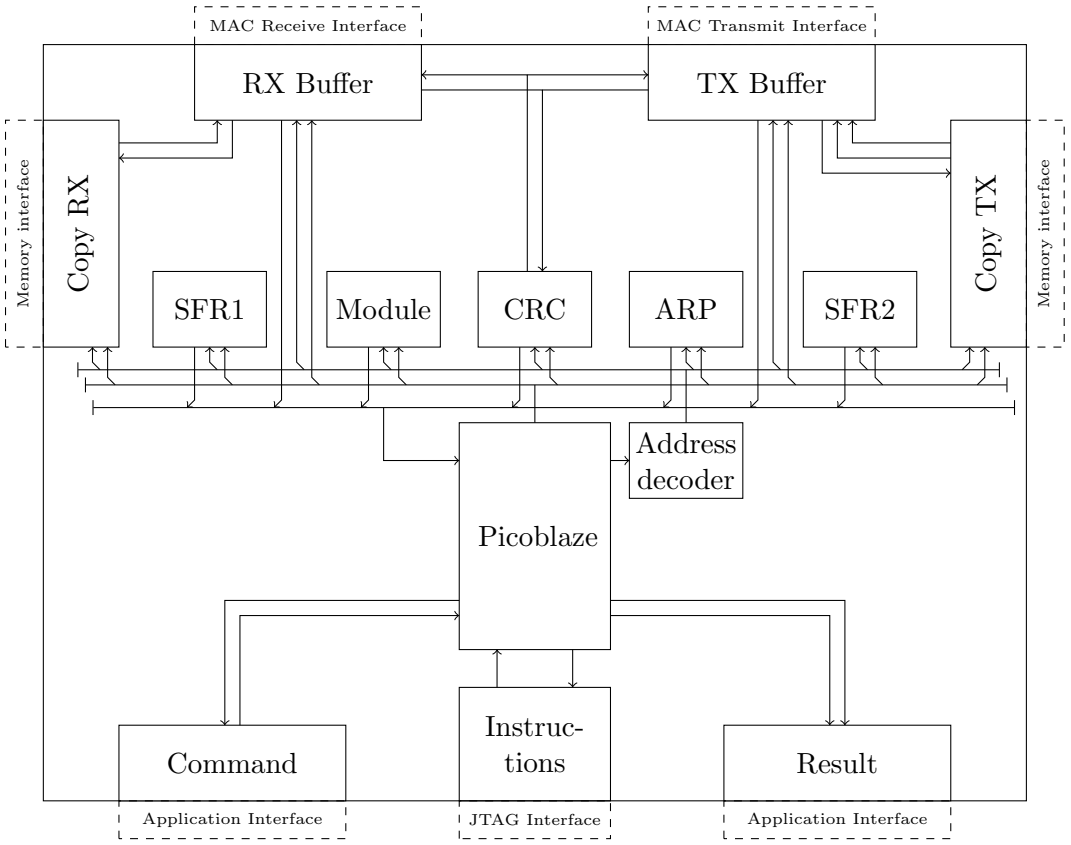


Figure 3.7: UDP/IP IP Core Architecture

step by step. A found signal is generated if the IP address present on the output bus of the ARP table matches the IP address set in the registers. After the address value is corrected for the offset created by pipelining the design the found bit in the status register is updated. If there is no match found in sixteen cycles the statemachine will set the not-found bit in the status register.

3.3.5 The copy module

The copy module looks quite similar to the ARP module design. The copy module exists out of a set registers. The registers hold the sixteen bit address of the first bye to copy, the destination address for the first byte and the amount of bytes to copy. The program running on the Picoblaze sets directly the addresses and length of the block, which is possible because the upper and lower halves of the registers are separately addressable. As with the ARP module, the setting of a bit in the status register triggers the statemachine. The statemachine starts a pipelined copy process. During the process both the address counters are increased, while the amount of bytes register is decreased. The statemachine moves into the finishing states when the amount of bytes to be copied is down to two. The finishing states copy the last few bytes and finally update a status

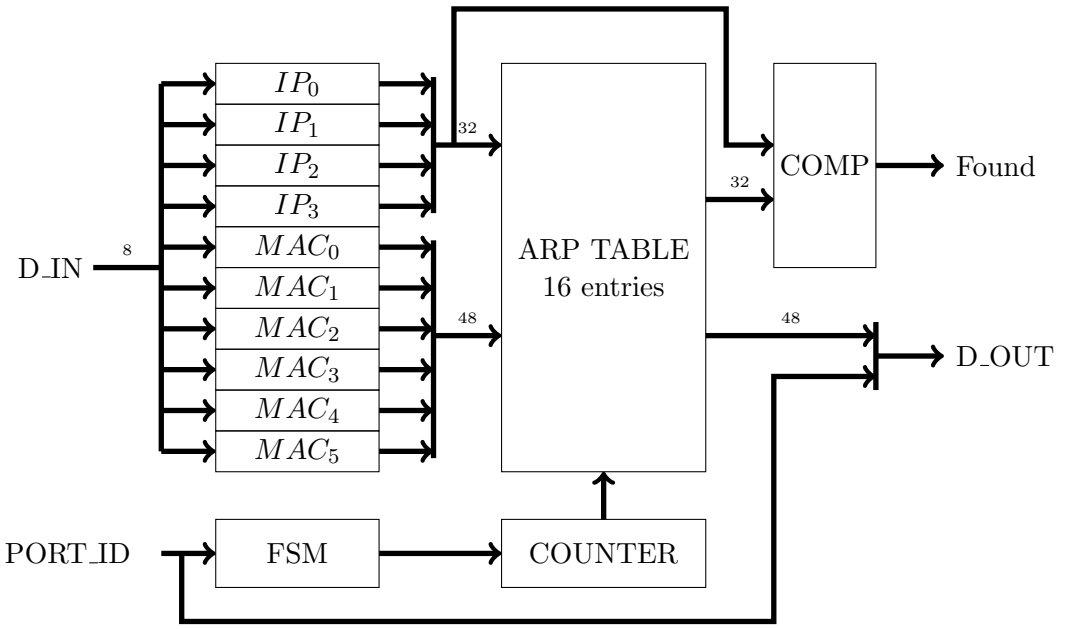


Figure 3.8: High overview of the ARP module

bit to indicate that the process is finished.

3.3.6 Modification of the MAC

This project uses the 10/100/1000 Tri-mode Ethernet MAC module which was published at opencores.org under the LGPL and written by Jon Gaočitegao. The module's functionality is part of the OSI Data Link Layer and provides addressing and channel access control mechanisms to send out and receive Ethernet packets to the physical layer.

Out of the box, the module was not suitable for this project since the data bus used in the MAC module is 32 bits wide, while the Picoblaze's data bus is just eight bits wide. A multiplexer could be used to multiplex the four bytes into the Picoblaze processor and a de-multiplexer could be used to send out the data back to the MAC, but for this project a more elegant solution has been chosen. The frames that are transmitted and received are first buffered in BRAMs. The BRAMs are configured as dual port BRAMs, where each port is configured independently of the other, while still accessing the same data. In this way the memory is addressed byte for byte without regard for word boundaries by the Picoblaze while keeping the interface towards the MAC unchanged.

However decreasing the width of the data bus increases proportionally the width of the address bus, which is also limited to just eight bits. This problem is solved by dividing the buffer into segments. The Picoblaze accesses a particular byte by loading the higher part of the address first into a register that drives the higher part of the address signal. Secondly, the remaining bits of the address are driven directly by the address port of the Picoblaze. Using this technique there is hardly any penalty in addressing the buffer compared to a full width bus, since in this case only one operation out of 128 needs to update the segment address, if the data is accessed sequentially.

3.3.7 Placing the design into the FPGA

The UDP/IP stack provides direct access through the JTAG interface to the memory block where the code executed by the Picoblaze is stored. The JTAG interface is used to update the code without recompiling and uploading the entire FPGA design [24]. By using this technique an enormous amount of time is saved during debugging, since recompiling the entire FPGA design takes several minutes, while uploading the new code to the BRAM only takes seconds. The resource costs for this additional feature are minimal, but the feature does utilize a rare resource; namely the JTAG interface itself. If the JTAG module is needed for a different part of the design, or for debugging the entire design with Xilinx's Chipscope, it will be necessary to comment out the JTAG module in the Verilog module that instantiates the memory to store the instructions.

Table 3.3: Timing constraints

Constraint	Period	Actual Period		Timing Errors	
	Requirement	Direct	Derivative	Direct	Derivative
CLKIN_IBUFG_OUT	20.000ns	N/A	19.890ns	0	0
CLKFX_BUF	12.500ns	12.431ns	N/A	0	0

One Digital Clock Manager (DCM) is used in this design out of the two, four or eight DCMs that are available in the Spartan 3E FPGA family, depending on the device size. The DCM is instantiated within the design by using the DCM primitive. The DCM provides clock-skew elimination, phase shifting and frequency synthesis. The latter being the reason why the DCM is incorporated in the design, since frequency synthesis is used to generate the appropriate clocking signals for the stack. The stack runs on two frequencies, namely at 50MHz for the MAC module and at 80MHz for the remainder of the stack, as is seen in Table 3.3. The two clock domains are separated by asynchronous dual-port BRAMs.

The design has been optimized for higher clock frequencies by pipelining the data path. The most effective strategy for pipelining is to pipeline the port_id output of the Picoblaze. Decoding the port addresses that are used to select the appropriate registers in the different modules causes delays since some of the signal may need several layers of combinatorial logic due to the fact that eight inputs need to be decoded to one bit. The Picoblaze uses two clock cycles for both the read and writes to the ports. This means that the first clock cycle is used to register the signal that selects the module. An additional benefit is that registering the output reduces the fan out of the port_id and out_port signals reducing even further the delay. Using this technique the design could be implemented at a clock frequency of 80MHz, increasing the throughput of the initial design by 60%.

The device utilization of the UDP/IP stack as placed in the Xilinx XC3S500E Spartan-3E FPGA is presented in Table 3.4. The most interesting numbers of the table are the ones at the top, the number of slice flip flops and the number of 4 inputs LUTS. The number of occupied slices is less relevant since the place and route tool does not

necessarily use up all available logic present in a single slice before using another, in other words the design is spread out over the FPGA. Summarizing the table it can be said that the entire design uses 1328 slice flip flops, 1767 4 input LUTs, which is $< 20\%$ of the available resources in this FPGA.

Table 3.4: Device Utilization Summary

Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	1,328	9,312	14%
Number of 4 input LUTs	1,767	9,312	18%
Logic Distribution			
Number of occupied Slices	1,388	4,656	29%
Number of Slices containing only related logic	1,388	1,388	100%
Number of Slices containing unrelated logic	0	1,388	0%
Total Number of 4 input LUTs	1,908	9,312	20%
Number used as logic	1,500		
Number used as a route-thru	141		
Number used for Dual Port RAMs	208		
Number used for 32x1 RAMs	52		
Number used as Shift registers	7		
Number of bonded IOBs	97	232	41%
Number of RAMB16s	5	20	25%
Number of BUFGMUXs	5	24	20%
Number of DCMs	2	4	50%

3.4 Conclusions

This chapter presented the implementation details of the UDP/IP stack. The prototype board was described and the relevant perihperals, such as the serial port, the liquid crystal display and the 10/100 Ethernet physical interface, were discussed.

The introduction of the BSD Socket API, which forms the de facto interface standard for the TCP/IP stack, made it possible to provide a standardized way of interacting with the stack. The supported funcion calls were described as well as the required arguments and the results they returned. A high overview of the program flow described the handling of, in particual, the *send()* and *receive()* function calls. Using the concept of signals as a way of supporting non-blocking function calls makes concurrent use of the stack possible for applications.

Furthermore, the chapter discussed the balance between FPGA area requirements, throughput and the limited amount of supported instructions by the Picoblaze. Concluding it could be said that functions that either operate on every byte of a frame or require large amounts of memory should be implemented as a separate modules in reconfigurable hardware. For this reason, the memory copy function, the arp function as well

as the crc function were chosen for implementation in HDL, showing a tenfold increase in throughput over a software only implementation.

In addition to speeding up modules, an effective bus strategy was discussed. The bus forms the integral component of the interface between the processor and the various modules spread out over the FPGA. It turned out that a simple shared register bank is the most resource efficient option, with the drawback of having all data operations going through the relatively slow Picoblaze.

Finally, the design was implemented in the FPGA. Several timing issues initially limited the clock frequency to 50MHz. Achieving a higher clock frequency is relevant, since an increase in clock frequency results in a linear increase in maximum throughput. By using the Xilinx Timing Analyzer the key problems were identified. Most of the problems could be solved by pipelining the design and removing an additional clock domain that was used for the memory components. In the end the design achieved a clock frequency of 80MHz. The amount of resources on the FPGA required by the implemention is lower than 20% on this FPGA. Unfortunately the number of used BRAMs make the implementation unsuitable for the smallest Spartan 3E FPGA.

4 Experimental results

The previous chapter described the platform specifications, the software implementation and the hardware implementation. This chapter presents the tests performed on the implementation and the obtained results. Section 4.2 discusses the functionality test and the results. Section 4.3 shows the performance tests in which the throughput is measured for several framesizes and in three scenarios; a transmitting stack, a receiving stack and a back-to-back stack. Section 4.4 summarizes the results and concludes this chapter.

4.1 Demo application

A demo application was created and connected to the UDP/IP stack as it would be in a typical situation. Using the stack in a typical situation makes sure that the results are repeatable and meaningful. The demo application is build on top of another Picoblaze processor and embedded in the FPGA next to the stack. The demo application and the UDP/IP stack share a single BRAM block. The BRAM block is configured to have two independent interfaces; where one interface is connected to the stack, and the other to the demo application. The complete setup is depicted in Figure 4.1. In addition to the BRAM, an LCD is attached to the output port of the Picoblaze. The control signals for the LCD are steered by the demo application. Interaction with demo application is made possible through the serial port present on the development board.

The demo application provides full access to the UDP/IP stack command and result register banks through its serial port. The stack supports several functions for debugging purposes in addition to the functions described in Table 3.1 and Table 3.2. These functions are *readPort*, *writePort*, *readMem* and *writeMem*. With just these four functions all modules within the stack itself is controlled and the results reported back to the application. The functions also provide direct access to the scratchpad memory of the Picoblaze inside the stack. The scratchpad memory stores critical information about the assigned IP addresses and opened sockets.

4.2 Functionality tests

The demo supports several functions and commands that are used to test the functionality of the design. The demo incorporates a Dynamic Host Configuration Protocol (DHCP) client. The DHCP provides an automatic network configuration parameters assignment to network devices and is defined in RFC2131 [7]. DHCP has two parts; a server that sends host specific parameters and a method for allocating resources. DHCP is based on a client/server model in which a client requests information from a server.

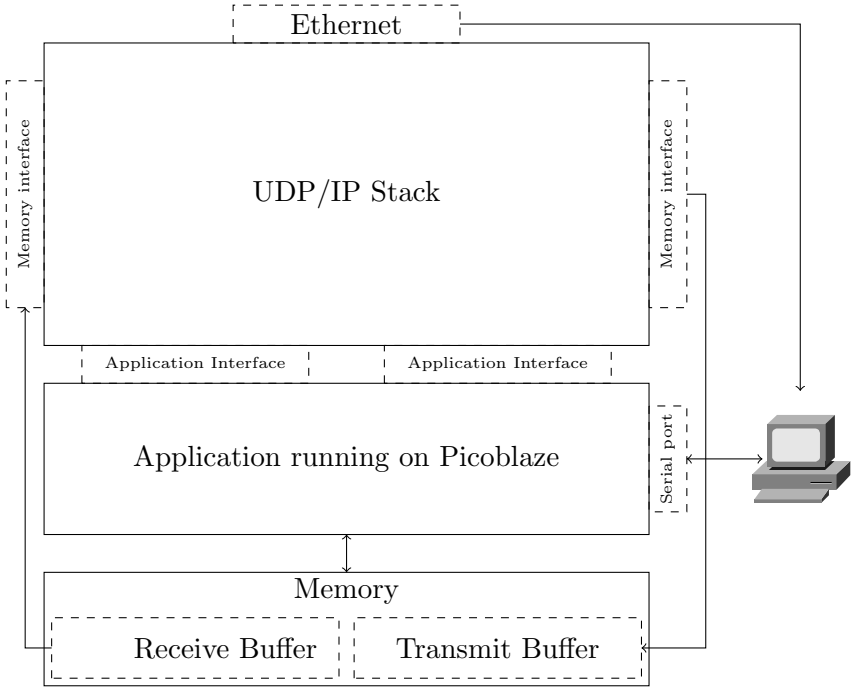


Figure 4.1: Both the demo application and the stack implemented within the FPGA

Typical information that is requested is the default gateway, the domain name servers, the assigned IP address and the subnet mask. The DHCP server maintains a database of resource pools and the assigned resources together with the lease time, the length of time the allocation is valid.

Figure 4.2 depicts the steps taken by the client and server when a DHCP client requests network configuration parameters from a DHCP server. The client starts by broadcasting a DHCPDISCOVER message to locate DHCP servers on the LAN. A DHCP server that receives the broadcast will offer the client configuration parameters in a DHCPOFFER message. After receiving the offers from the DHCP servers the client accepts a single offer by requesting the offered parameters in a DHCPREQUEST message. Finally, the selected DHCP server confirms the allocation of the resources by returning a DHCPACK message.

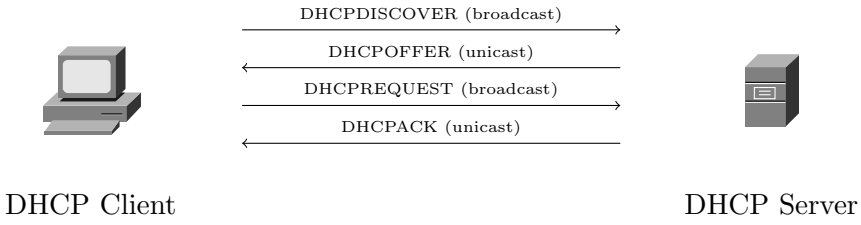


Figure 4.2: A client requesting parameters from a DHCP server

The DHCP client implemented in the demo application tests most functions supported by the stack. The demo is setup by connecting a PC via the serial port to the application and connecting the board to an Ethernet network with a working DHCP server. The DHCP client process is started by entering *s* on the command line. After the demo obtains an IP address it is pinged by the PC to check the ARP and ICMP functionality. The demo application will follow the following steps during the DHCP process;

1. Request a socket from the stack
2. Bind the socket to UDP port 67
3. Generate the DHCP Discover packet
4. Request the stack to transmit the DHCP Discover packet
5. Wait for the stack to notify it has received a new UDP packet on UDP port 67
6. Request the stack to place the packet in the common buffer
7. Check if the received packet is a UDP Offer
8. If so, generate a DHCP Request with the offered IP address
9. Request the stack to transmit the DHCP Request
10. Wait for the stack to notify it has received a new UDP packet on UDP port 67
11. Request the stack to place the packet in the common buffer
12. Check if the received packet is a UDP ACK
13. If so, sets the following parameters
    (a) The IP Address
    (b) The Subnet Mask
    (c) The default gateway
14. Closes the socket

Figure 4.3 shows a screen capture of a Wireshark window running on the PC that runs the DHCP server. Wireshark is a software based network protocol analyzer which offers deep packet inspection and live capturing of frames on off-the-shelf PC hardware components.[4] The screen capture is taken after the completion of the DHCP process and includes the ping from the DHCP server towards the demo application.

The screen capture shows that the application successfully obtained an IP address with the DHCP protocol. The demo application started by sending a DHCP Discover with a source IP address of 192.168.10.2, shown in the second column of the the first packet. The DHCP server responded with an DHCP Offer, indicated by the second

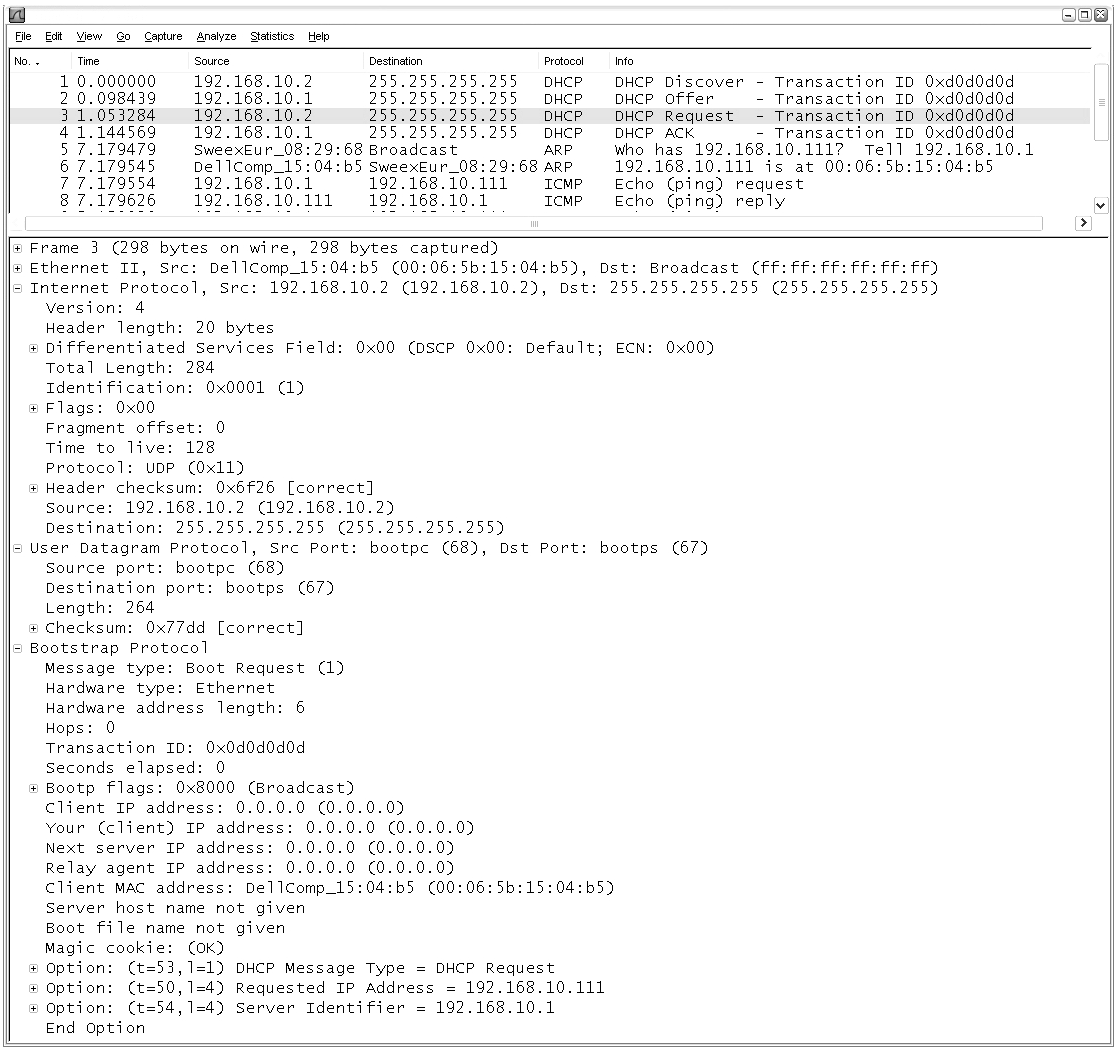


Figure 4.3: A Wireshark capture

packet. After which the demo requested the IP address by using the DHCP Request packet. Finally, a DHCP ACK is received and the IP address, subnet and default gateway are set to their new values.

An ICMP echo request is initiated from the DHCP server by entering the command *ping 192.168.10.111* on the command line interface. The DHCP server does not yet know which MAC address belongs to this specific IP address, therefore it sends out an ARP request to discover the link layer address, which is indicated by packet number six on the screenshot. The stack replies to this discover message by returning its own MAC address with an ARP reply towards the DHCP server. Now that the MAC address is known the DHCP server sends out the ICMP echo request as seen as packet seven. The stack learned the MAC address of the DHCP server when the DHCP server sent out its ARP request and does not need to send out an ARP request itself. The stack replies by sending an ICMP echo reply packet, as seen in packet number eight.

4.3 Performance tests

This section determines how fast the stack can go. It determines the actual usable throughput of the stack compared to the FastEthernet network link limit. How fast a stack is is expressed in the amount of packets that it processes each second. The test is described in RFC2544 Benchmarking Methodology for Network Interconnect Devices, which lists specific frame sizes and reporting formats that should be used for this test. The reason behind this is to rule out "specsmanship" by vendors who could state that a device processes a certain amount of bandwidth per second, while omitting the frame size at which those test where conducted. This could lead to unfair comparisons, since it is much easier to achieve a high bandwidth with big frames than with small frames.

There are several tools available that allow one to perform throughput testing and that runs on standard personal computers. Three of those tools that are widely used are IPerf, Netperf and TTCP. IPerf is developed at the National Laboratory for Applied Network Research (NLANR) and is written in C++. IPerf supports various parameters that can be changed or set to test a network. IPerf, as well as the other two tools, has a client and server. The client and server is used to measure the throughput between the network, either unidirectionally or bi-directionally. Netperf and TTCP work quite similar as IPerf. The main differences between the tools are the operating systems that they support, the presence of a gui and the maximum achievable throughput on a given system, which is often limited by the CPU.

The NT TTCP tool [13] is selected for these throughput tests after a quick evaluation of the three tools mentioned in the previous paragraph. TTCP is one of the first throughput testing tools written and was used by Mike Muuss at the Ballistic Research Lab to compare the performance of TCP stacks for DARPA in order to decide which TCP version to include in the first BSD Unix release. Later several ports were made of the software, including NT TTCP, which is used in this project. The main reason for selecting NT TTCP is the number of frames it can generate at the smallest ethernet frame size, without utilizing the CPU for 100%, compared to the other implementations.

Figure 4.4(a) depicts the UDP transmission test. For this test the demo application was setup to continuously transmit the same UDP data to a single destination. The overhead of transmitting data from the application buffer towards the stack buffer and the communication overhead between application and stack are included in these numbers. The theoretical speed used in the graphs is calculated with Equation 4.1, where N is the frame size in bytes. It can be seen that the stack creates and transmits 94709 packets per second at the minimum frame size of 64 bytes, which is 64% of the theoretical line speed. The main reason that the design can not fill the line for 100% is that the overhead of generating the headers and calculating the checksum takes longer than actually transmitting the frame and thus cannot be compensated wholly by pipelining the creation of the packets. This does not hold for larger packets, as seen in the graphs for frames larger than 128 bytes, where the actual throughput comes very close to the theoretical throughput.

$$PPS_{theoretical} = \frac{100 \cdot 10^6}{64 + 8 \cdot N + 96} \tag{4.1}$$

Figure 4.4(b) depicts the UDP reception test. For this test the demo application was setup to open a socket and listen to incoming packets. The overhead of copying the UDP data of the packet to the application buffer and the communication overhead between application and stack are included in these numbers. The first packet that arrives at the application layer triggers a clock counter that keeps counting until x packets have been received. After the x^{th} packet the value of the clock counter is shown on the terminal and is used to calculate the throughput by using Equation 4.2. The main reason of achieving a higher throughput than in the previous test case is that it takes far less time to strip a packet of its headers than it takes to generate them. The result is that the stack can process frames closes to the theoretical throughput.

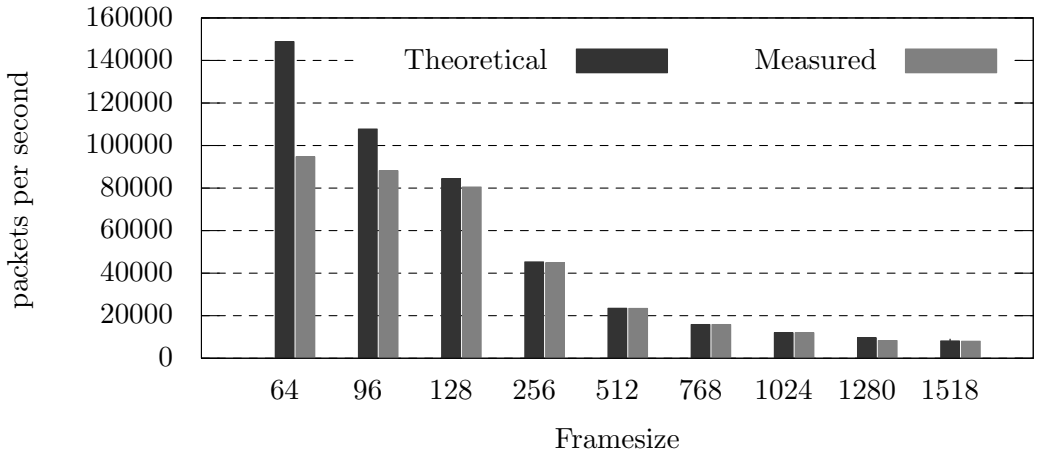
$$PPS_{receive} = \frac{counter}{80 \cdot 10^6} \cdot \frac{1}{x} \tag{4.2}$$

Figure 4.4(c) depicts a test in which packets are transmitted by the PC, as was the case in the previous case, and immediately send them back to the PC, as in the first case. This test is important because it shows the drawback of having a single low performance processor handling both the reception and transmission of frames. The theoretical throughput in this case is calculated by adding the amount of frames that were send to the amount of frames that were received. As seen in the graph the stack does not perform at line speed for the smaller frames. The results for the smaller frames are not simply the addition of the two previous two graphs. The additional overhead incurred by the application layer and the limit processing power of the Picoblaze limit the stack to 143866 packets per second for two symmetric bidirectional traffic streams.

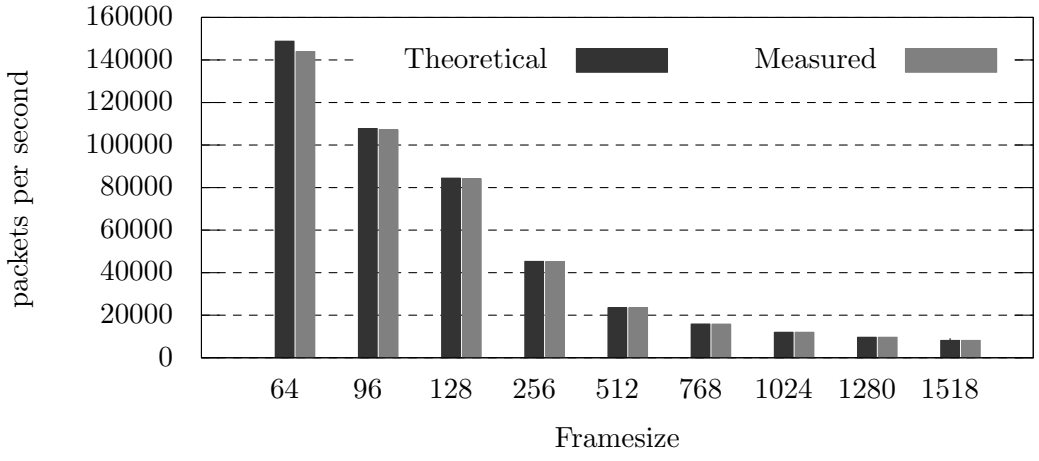
4.4 Conclusions

The demo application provides a means to do both functional as performance tests. The functional tests show that the provided interface and functions perform as required. The DHCP client implemented in the demo's assembly code request and assigns network parameters to the stack. A ping is send to the new IP address after the demo application assigned it to the stack. An echo reply is received, which proofs furthermore the functioning of the stack.

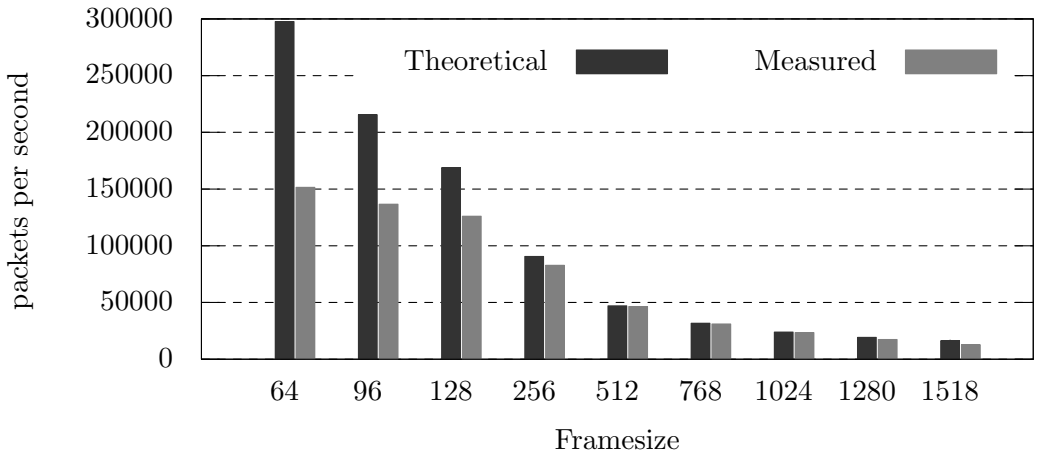
Additionally, the stack has undergone a selection of throughput tests. The tests are performed with several frame sizes, as recommended by RFC2544, in order to rule out specmanship. The stacks main limitation is the transmitting side of the stack. The creation of the headers limits the stack and this means it can not perform at line speed for smaller packets. This is less an issue for larger packets, where the costs of creating the headers is compensated by the transmission of the actual frame due to the stacks pipelined design. The stack receives and processes packets close to line speed. Sending and receiving packets simultaneously is again hindered by the transmission of frames.



(a)



(b)



(c)

Figure 4.4: Maximum throughput, for; (c) is the first

5 Conclusions

In the introduction it was stated that the main goal of this project is to design and implement a freely available UDP/IP stack in reconfigurable logic which could achieve throughput speeds at FastEthernet line speed. These first requirements is met the second only for the receiving side. Another requirement is the amount of resources used and having the possibility to implement this stack on even the smallest member of the Spartan 3E FPGA family. This constraint is partially met due to need of additional buffers in the form of BRAMs.

Section 5.1 summarizes the main conclusions that are drawn from this thesis. The main contributions of this thesis are stated in Section 5.2. Section 5.3 concludes this thesis by recommending topics for future research on the UDP/IP stack.

5.1 Summary

Chapter 2 presents the basic background knowledge for understanding the project's context. As resource efficiency is one of the main goals a small embedded processor is selected and implemented, which will function as the CPU within the stack. The advantage of using an processor is two fold. First, the design size is kept at a constant state, because the complexity is in the code not in the hardware. Second, assembly code is far easier to understand and manage than a very complex state machine with hundreds of states. Most embedded processors that are found on the Internet have the major drawback that they are not designed with resource efficiency in mind, but rather target a specific instruction set. Two alternatives are the embedded processor released by Xilinx and Altera for their respective FPGA brands. Xilinx's Picoblaze 8-bit processor is used in this design, because of vendor lock-in and the fact that the development board was already selected.

Chapter 2 discusses several protocols that are related to the design of the stack. The Internet Protocol provides a basic datagram delivery service. On top of which the User Datagram Protocol delivers a multiplexing feature so that multiple applications can concurrently use the UDP/IP stack. The Address Resolution Protocol is described and is used for translation of four byte IP address to six byte Ethernet Addresses. Finally, the Internet Control Message Protocol is discussed which provides a standardized message format that is used to send error message from one network device to another. The ICMP protocol is the backbone used by several utilities to diagnose network problems.

Chapter 3 presented the implementation details of the UDP/IP stack. The prototype board was described and as well as the relevant peripherals, such as the serial port, the liquid crystal display and the 100/100 Ethernet physical interface. Next, the BSD Socket API is introduced. The API forms the de facto interface standard for the TCP/IP stack. By adhering to the standard applications can use standardized function to interact with

the stack. The supported functions are described together with their respective results. Using the concept of signals, as a way of supporting non-blocking function calls, makes concurrent use of the stack possible.

Furthermore, the chapter discusses the balance between FPGA area requirements, throughput and the limited amount of addressable instructions by the Picoblaze. Concluding, functions that either operate on every byte of a frame or require large amounts of memory should be implemented as separate modules in reconfigurable hardware. For this reason, the memory copy function, the arp function as well as the crc function are implemented in HDL. In addition, an effective bus strategy was discussed. The bus forms the integral component of the interface between the processor and the various modules spread out over the FPGA. It turned out that a simple shared register bank is the most resource efficient option, with the drawback of having all data operations going through the relatively slow Picoblaze.

Finally, Chapter 3 concludes with implementing the design in the FPGA. Several timing issues limited the clock frequency to 50MHz. This resulted in poor throughput performance for small packets. Increasing the clock frequency linearly increases the maximum throughput. By using the Xilinx Timing Analyzer key problem areas were identified. Most of the the timing issues are resolved by pipelining the design and removing an additional clock domain dedicated to a few memory components. In the end the design achieved a clock frequency of 80MHz and the amount of resources on the FPGA required is lower than 20% on this FPGA. Unfortunately the number of used BRAMs make the implementation unsuitable for the smallest Spartan 3E FPGA.

Chapter 4, shows the use of a demo application to provide a means to do both functional as performance testing. The functional tests show that the provided interface and functions perform as required. The DHCP client implemented in the demo's assembly code request and assigns network parameters to the stack. A ping is send to the new IP address after the demo application assigned it to the stack. An echo reply is received, which proofs furthermore the functioning of the stack.

Additionally, the stack has undergone a selection of throughput tests. The tests are performed with several frame sizes, as recommended by RFC2544, in order to rule out specmanship. The stacks main limitation is the transmitting side of the stack. The creation of the headers make that the stack can not perform at line speed for smaller packets. This is less an issue for larger packets, where the costs of creating the headers is compensated by the transmission of the actual frame due to the stacks pipelined design. The stack can receive packets close to line speed. Sending and receiving packets simultaneously is again hindered by the transmission of frames.

5.2 Main contributions

The UDP/IP stack was created with several requirements in mind. These requirements were a small area footprint on an FPGA, the ability to run at 100Mbit/s and the ability to send and receive UDP packets. The main contributions of this thesis are;

- A functional UDP/IP stack with a minimal set of requirements has been designed and created in reconfigurable hardware.

- The UDP/IP stack has been optimized by deducing the most time-consuming processes per send and received packet. Candidates for optimization are functions that operate on almost every byte of a frame and/or that need relatively large amounts, meaning more than 32 bytes, of memory.
- The design ran first at a global clock frequency of 50MHz, while the memory operated at 100MHz. Both clocks could be replaced with a single clock running at 80MHz by optimizing the finiste state machines and pipelining. This optimization lead to a 60% increase in throughput.
- The copy function, which are responsible for moving bytes from one buffer to another, the Address Resolution Protocol and the Cyclic Redundancy Check function are the most compute-intensive function and are implemented as separate modules in reconfigurable logic.
- The UDP/IP stack uses 1328 slice flip flops and 1767 four input LUTs, which is $< 20\%$ of the available resources in the FPGA used in this project.
- A demo application is provided that illustrates the interfacing with the UDP/IP stack. The demo application includes a functioning Dynamic Host Configuration Protocol client.
- Both the demo and stack are released under the GNU Lesser General Public License, promoting free software and future research.

5.3 Recommendations for future research

This section discusses future opportunities to improve the UDP/IP stack. As future FPGAs are likely to grow in size the design can also grow, while keeping the design relatively the same size. With the increase of size new functionalities can be added. Perhaps the most important and urgent functionality is the addition of the Transmission Control Protocol to the stack. With TCP support a whole new range of applications can start to make use of FTP, SMTP and HTTP. Besides TCP, the following directions are recommended for future research:

- Receive and transmit fragmented packets. The firmware running on the Picoblaze can be altered to utilize the increased capacity when more memory is readily available on FPGAs. Adding support for fragmented packets would make it more compliant to the standards.
- The addition of IPv6 support. IPv6 will become more and more popular now that the free IPv4 addresses become more scarce. IPv6 support would make the stack future proof.
- Standard support for popular busses. The usability and retargetability of the UDP/IP stack can be significantly increased by supporting standard bus interface standards, such as AMBA, CoreConnect and Wishbone.

- Improving the throughput towards the Gigabit Ethernet standard. A higher bandwidth could relatively easy be achieved by parallelizing, changing the embedded processor to a 16 bit version and implementing more logic in reconfigurable hardware.
- Add support for multiple Ethernet interfaces. Multiple interfaces would make routing and/or switching possible, which could be interesting if one would want to create applications such as routers and firewalls.
- Separate the design in a receive and transmit side, where each side has a dedicated Picoblaze, for an increase in performance.

Bibliography

[1] L. Anghel, R. Velazco, S. Saleh, S. Deswaertes, and A. El Moucary, *Preliminary validation of an approach dealing with processor obsolescence*, Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on (2003).

[2] Anon., *Opencores*, june 2009.

[3] R. Braden, *Requirements for internet hosts - communication layers*, RFC1122 Internet Engineering Task Force, October 1989.

[4] G. Combs, *Wireshark*, june 2009.

[5] Denic, *http://www.denic.de/en/domains/statistiken/hostentwicklung/hostcount.html*.

[6] A. Dollas, I. Ermis, I. Koidis, I. Zisis, and C. Kachris, *An open tcp/ip core for reconfigurable logic*, (2005).

[7] R. Droms, *Rfc2131 - dynamic host configuration protocol*, IETF, 1997.

[8] A. Dunkels, *Full TCP/IP for 8 Bit Architectures*, Proceedings of the First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys 2003) (San Francisco), USENIX, MAY 2003.

[9] F. Eady, *Networking and internetworking with microcontrollers*, Newnes, 2004.

[10] IEEE, *Ieee 802.3-2005 section 1*, IEEE, 2005.

[11] M. Kuster, *Firmware for reconfigurable hardware os platform*, Master's thesis, Swiss Federal Institute of Technology Zurich, 2004.

[12] Mediatronics, *pblazide*, Mediatronics, 2008.

[13] Microsoft, http://www.microsoft.com/whdc/device/network/TCP_tool.mspx, *How to use ntttcp to test network performance*, 4 2008.

[14] S. Narayanaswamy, *High performance tcp/ip on xilinx fpga devices using the treck embedded tcp/ip stack*, 12 2004.

[15] RIPE NCC, *Ripe ncc service region hostcount*, Internet, March 2007.

[16] S. Nobs, *Prototype board for reconfigurable os*, Master's thesis, Swiss Federal Institute of Technology Zurich, 2003.

[17] D.C. Plummer, *Rfc826 - ethernet address resolution protocol*, IETF, 1982.

[18] J. Postel, *Rfc768 - user datagram protocol*, IETF, 1980.

[19] ———, *Rfc792 - internet control message protocol*, IETF, 1982.

[20] SMSC, *Lan83c185 high performance single chip low power 10/100 ethernet physical layer transceiver (phy)*, SMSC, 2008.

[21] W. Richard Stevens, B. Fenner, and A.M. Rudoff, *Unix network programming volume 1, third edition: The sockets networking*, Addison-Wesley Pub Co., 2003.

[22] P. Sutton, J. Brennan, A. Partis, and J. Peddersen, *Vhdl ip stack*, February 2001.

[23] S. Thammanur and C. Borrelli, *Tcp/ip on virtex-ii pro devices using lwip*, 8 2004.

[24] Xilinx, *Picoblaze, jtag loader*, Xilinx, 2004.

[25] ———, *Xapp464 - using look-up tables as distributed ram in spartan-3 generation fpgas*, Xilinx, 2005.

[26] ———, *Picoblaze 8-bit embedded microcontroller user guide*, Xilinx, 2008.

[27] ———, *Spartan-3e fpga starter kit board user guide*, Xilinx, 2008.

[28] ———, *Xilinx : Chipscope pro*, Xilinx, 2008.

[29] ———, *Xilinx ise 10.1 design suite software manuals and help*, Xilinx, 2008.

6 Appendix A: Source code

The following listings are not the complete code. The listings are a part of the code written for the implementation by the author of this thesis.

Listing 6.1: stack.psm

```
                        ; Mac ports
2
                        ; host inteface ports
4                       CONSTANT command_port, 90
                        CONSTANT command_id,90
6                       CONSTANT command_arg0, 91
                        CONSTANT command_arg1, 92
8                       CONSTANT command_arg2, 93
                        CONSTANT command_arg3, 94
10                      CONSTANT command_arg4, 95
                        CONSTANT command_arg5, 96
12                      CONSTANT command_arg6, 97
                        CONSTANT command_arg7, 98
14                      CONSTANT command_arg8, 99
                        CONSTANT command_arg9, 9A
16                      CONSTANT command_argA, 9B
                        CONSTANT command_argB, 9C
18                      CONSTANT command_argC, 9D
                        CONSTANT command_argD, 9E
20                      CONSTANT command_argE, 9F
                        CONSTANT result_port,  90
22                      CONSTANT result_id,    90
                        CONSTANT result_arg0,  91
24                      CONSTANT result_arg1,  92
                        CONSTANT result_arg2,  93
26                      CONSTANT result_arg3,  94
                        CONSTANT result_arg4,  95
28                      CONSTANT result_arg5,  96
                        CONSTANT result_arg6,  97
30                      CONSTANT result_arg7,  98
                        CONSTANT result_arg8,  99
32                      CONSTANT result_arg9,  9A
                        CONSTANT result_argA,  9B
34                      CONSTANT result_argB,  9C
                        CONSTANT result_argC,  9D
36                      CONSTANT result_argD,  9E
                        CONSTANT result_argE,  9F
38
                        ; mac_status_port
40                      CONSTANT mac_status_port, C8
                        CONSTANT tx_mac_packet_ready,80
42                      CONSTANT tx_mac_eop_bit,40
                        CONSTANT rx_mac_packet_ready,01
44
                        ; rx_mac_packet_status / rx_mac_get_packet
46                      CONSTANT  rx_mac_get_packet,D0
                        ;CONSTANT rx_mac_packet_status, D0
48                      ;CONSTANT rx_mac_eop, 80
                        ;CONSTANT rx_mac_sop, 40
50                      ;CONSTANT rx_mac_get_packet,20

52                      ; module_sfr
                        CONSTANT module_sfr, B8
54                      CONSTANT module_mac, 00
                        CONSTANT module_copy, 01
56                      CONSTANT module_crc, 02
                        CONSTANT module_copy_rx, 03
58
                        ; crc module
60                      CONSTANT crc_start, B0
                        CONSTANT crc_address_l, B1
62                      CONSTANT crc_address_h, B2
                        CONSTANT crc_length_l, B3
64                      CONSTANT crc_length_h, B4
                        CONSTANT crc_l, B1
66                      CONSTANT crc_h, B2
```

```
68                      ; copy module
                        CONSTANT copy_start , C0
70                      CONSTANT copy_from_address_l , C1
                        CONSTANT copy_from_address_h , C2
72                      CONSTANT copy_length_l , C3
                        CONSTANT copy_length_h , C4
74                      CONSTANT copy_to_address_l , C5
                        CONSTANT copy_to_address_h , C6
76
                        ; copy module rx
78                      CONSTANT copy_rx_start , A0
                        CONSTANT copy_rx_from_address_l , A1
80                      CONSTANT copy_rx_from_address_h , A2
                        CONSTANT copy_rx_length_l , A3
82                      CONSTANT copy_rx_length_h , A4
                        CONSTANT copy_rx_to_address_l , A5
84                      CONSTANT copy_rx_to_address_h , A6

86
                        ; mac_sfr
88                      CONSTANT mac_sfr , D8
                        CONSTANT mac_reset , 80
90                      CONSTANT mac_rx_buffer , 00
                        CONSTANT mac_tx_buffer , 40
92                      CONSTANT mac_send_packet , 20
                        CONSTANT mac_eop , 10
94                      CONSTANT mac_done , 00

96                      ; mac_sfr2
                        CONSTANT mac_sfr2 , F8
98                      CONSTANT mac_up , 01
                        CONSTANT mac_down , FE
100                     CONSTANT new_packet , 02
                        CONSTANT no_new_packet , FD
102
                        ; Ethernet frame constants
104                     CONSTANT eth_type_h , 0C
                        CONSTANT eth_type_l , 0D
106
                        ; Arp packet constant
108                     CONSTANT arp_hdr_opcode , 15
                        CONSTANT arp_request , 01
110                     CONSTANT arp_reply , 02

112                     ; IP packet constants
                        CONSTANT ip_version_ihl , 0E
114                     CONSTANT ip_tos , 0F
                        CONSTANT ip_length_h , 10
116                     CONSTANT ip_length_l , 11
                        CONSTANT ip_identification_h , 12
118                     CONSTANT ip_identification_l , 13
                        CONSTANT ip_flags , 14
120                     CONSTANT ip_offset , 15
                        CONSTANT ip_ttl , 16
122                     CONSTANT ip_protocol , 17
                        CONSTANT ip_checksum_h , 18
124                     CONSTANT ip_checksum_l , 19
                        CONSTANT ip_source , 1A
126                     CONSTANT ip_source3 , 1A
                        CONSTANT ip_source2 , 1B
128                     CONSTANT ip_source1 , 1C
                        CONSTANT ip_source0 , 1D
130                     CONSTANT ip_destination , 1E
                        CONSTANT ip_destination3 , 1E
132                     CONSTANT ip_destination2 , 1F
                        CONSTANT ip_destination1 , 20
134                     CONSTANT ip_destination0 , 21

136                     ; UDP packet constants
                        CONSTANT udp_source_port_h , 22
138                     CONSTANT udp_source_port_l , 23
                        CONSTANT udp_destination_port_h , 24
140                     CONSTANT udp_destination_port_l , 25
                        CONSTANT udp_length_h , 26
142                     CONSTANT udp_length_l , 27
                        CONSTANT udp_checksum_h , 28
144                     CONSTANT udp_checksum_l , 29
                        CONSTANT udp_data , 2A
146
                        ; UDP pseudoe header constants
148                     CONSTANT udp_pseudo_source_ip , 16
                        CONSTANT udp_pseudo_source_ip3 ,16
150                     CONSTANT udp_pseudo_source_ip2 ,17
                        CONSTANT udp_pseudo_source_ip1 ,18
152                     CONSTANT udp_pseudo_source_ip0 ,19
                        CONSTANT udp_pseudo_destination_ip ,1A
154                     CONSTANT udp_pseudo_destination_ip3 , 1A
```

```
                        CONSTANT udp_pseudo_destination_ip2, 1B
                        CONSTANT udp_pseudo_destination_ip1, 1C
                        CONSTANT udp_pseudo_destination_ip0, 1D
                        CONSTANT udp_pseudo_zero,1E
                        CONSTANT udp_pseudo_protocol,1F
                        CONSTANT udp_pseudo_length_h,20
                        CONSTANT udp_pseudo_length_l,21

                        ; ICMP packet constant
                        CONSTANT icmp_type, 22
                        CONSTANT icmp_echo_reply, 00
                        CONSTANT icmp_echo_request, 08
                        CONSTANT icmp_checksum_h, 24
                        CONSTANT icmp_checksum_l, 25

                        ; Arp table constants
                        CONSTANT arp_table_status, E0
                        CONSTANT arp_table_write, 01
                        CONSTANT arp_table_read, 02
                        CONSTANT arp_table_found, 04
                        CONSTANT arp_table_notfound, 08

                        CONSTANT arp_table_eth0, E1
                        CONSTANT arp_table_ip0, E7
                        CONSTANT arp_table_ip1, E8
                        CONSTANT arp_table_ip2, E9
                        CONSTANT arp_table_ip3, EA

                        ;Special Register usage
                        ;
                        NAMEREG sE, store_pointer
;used to pass location of data in scratch pad memory
                        ;
                        ;Scratch Pad Memory Locations
                        ;
                        CONSTANT loc_gateway_ip0, 0E
                        CONSTANT loc_gateway_ip1, 0F
                        CONSTANT loc_gateway_ip2, 10
                        CONSTANT loc_gateway_ip3, 11

                        CONSTANT loc_ip_identification_h, 12
                        CONSTANT loc_ip_identification_l, 13

                        CONSTANT loc_readstatus, 14
                        CONSTANT loc_writestatus, 15
                        CONSTANT loc_prepstatus, 31
                        CONSTANT last_command_id, 16

                        ;Sockets are stored as two bytes, sourceport_h = socketnumber * 2 + socket_start,
                        ;sourceport_l = socketnumber * 2 + socket_start
                        CONSTANT socket_random_port_h, 1D
                        CONSTANT socket_random_port_l, 1E
                        CONSTANT socket_alloc, 1F
                        CONSTANT socket_start, 20
                        CONSTANT string_start, 30
                        ;Initialise the system
                        ;
                        ;

           cold_start:  LOAD    sA, mac_reset                    ; Reset the mac
                        OUTPUT  sA, mac_sfr
                        LOAD    sA, mac_done
                        OUTPUT  sA, mac_sfr
       set_get_packet:  OUTPUT  sA, rx_mac_get_packet
; Write to register rx_mac_get_packet to get new packet into buffer
            main_loop:  FETCH   sA, loc_readstatus
; check if the packetbuffer is blocked
                        COMPARE sA, 00
                        JUMP    NZ, handle_commands
                        INPUT   sA, mac_status_port              ; Read status register
                        TEST    sA, rx_mac_packet_ready
; Check if there is a packet in the buffer
                        JUMP    NZ, received_packet
     handle_commands:   FETCH   sB, last_command_id
                        INPUT   sA, command_port
                        COMPARE sA, sB
                        JUMP    NZ, command_menu
                        JUMP    main_loop

     received_packet:   INPUT   sA, eth_type_h
; Read the ethernet type of the packet
                        INPUT   sB, eth_type_l
                        CALL    check_for_arp
; Check if the packet is an arp packet
                        CALL    Z, process_arp
; If it is an arp packet process it
                        CALL    check_for_ip
```

```
    ; Check if the packet is an ip packet
                            JUMP    NZ, set_get_packet              ; No IP packet so lets drop it
236                         INPUT   sA, ip_protocol
                            COMPARE sA, 01
    ; Check if the packet is an icmp packet
238                         JUMP    NZ, packet_not_icmp
              handle_icmp:  INPUT   sA, icmp_type
240                         COMPARE sA, icmp_echo_request
    ; Check if the packet is an icmp echo request
                            CALL    Z, process_icmp_echo_req
242                         JUMP    set_get_packet
          packet_not_icmp:  COMPARE sA, 11
    ; check if packet is an udp packet
244                         CALL    Z, process_udp
                            FETCH   sA, loc_readstatus
    ; if blocked do not request the new packet yet
246                         COMPARE sA, 01
                            JUMP    Z, main_loop
248                         JUMP    set_get_packet                  ; Restart main loop

250


252
            check_for_arp:  COMPARE sA, 08
    ; Subroutine to check if the packet is a arp packet, if arp packet Z=1
254                         JUMP    NZ, check_for_arp_n
                            COMPARE sB, 06
256         check_for_arp_n: RETURN

258
             check_for_ip:  COMPARE sA, 08
    ; Subroutine to check if the packet is an ip packet, if ip packet Z=1
260                         JUMP    NZ, check_for_ip_n
                            COMPARE sB, 00
262          check_for_ip_n: RETURN

264            command_menu: STORE   sA, last_command_id
                            INPUT   sA, command_arg0
266                         COMPARE sA, 0A
                            JUMP    Z, command_sendto
268                         COMPARE sA, 0B
                            JUMP    Z, command_recvfrom
270                         COMPARE sA, 01
                            JUMP    Z, command_get_ip
272                         COMPARE sA, 02
                            JUMP    Z, command_set_ip
274                         COMPARE sA, 03
                            JUMP    Z, command_get_gw
276                         COMPARE sA, 04
                            JUMP    Z, command_set_gw
278                         COMPARE sA, 05
                            JUMP    Z, command_get_nm
280                         COMPARE sA, 06
                            JUMP    Z, command_set_nm
282                         COMPARE sA, 07
                            JUMP    Z, command_socket
284                         COMPARE sA, 08
                            JUMP    Z, command_bind
286                         COMPARE sA, 09
                            JUMP    Z, command_close
288                         COMPARE sA, 0C
                            JUMP    Z, command_get_hw
290                         COMPARE sA, E0
                            JUMP    Z, command_read_port
292                         COMPARE sA, E1
                            JUMP    Z, command_write_port
294                         COMPARE sA, E2
                            JUMP    Z, command_read_mem
296                         COMPARE sA, E3
                            JUMP    Z, command_write_mem
298                         JUMP    main_loop

300       command_write_port: INPUT   sA, command_arg1
                            INPUT   sB, command_arg2
302                         OUTPUT  sB, (sA)
                            FETCH   sA, last_command_id
304                         OUTPUT  sA, result_id
                            JUMP    main_loop
306
         command_read_port: INPUT   sA, command_arg1
308                         INPUT   sB, (sA)
                            OUTPUT  sB, result_arg0
310                         FETCH   sA, last_command_id
                            OUTPUT  sA, result_id
312                         JUMP    main_loop

314        command_write_mem: INPUT   sA, command_arg1
```

```
                        INPUT   sB, command_arg2
316                     STORE   sB, (sA)
                        FETCH   sA, last_command_id
318                     OUTPUT  sA, result_id
                        JUMP    main_loop
320
      command_read_mem: INPUT   sA, command_arg1
322                     FETCH   sB, (sA)
                        OUTPUT  sB, result_arg0
324                     FETCH   sA, last_command_id
                        OUTPUT  sA, result_id
326                     JUMP    main_loop

328       command_get_hw: FETCH   sA, 04
                        OUTPUT  sA, result_arg0
330                     FETCH   sA, 05
                        OUTPUT  sA, result_arg1
332                     FETCH   sA, 06
                        OUTPUT  sA, result_arg2
334                     FETCH   sA, 07
                        OUTPUT  sA, result_arg3
336                     FETCH   sA, 08
                        OUTPUT  sA, result_arg4
338                     FETCH   sA, 09
                        OUTPUT  sA, result_arg5
340                     FETCH   sA, last_command_id
                        OUTPUT  sA, result_id
342                     JUMP    main_loop

344
      command_recvfrom: FETCH   sA, loc_readstatus      ; Check if there is a blocked buffer
346                     COMPARE sA, 00
                        JUMP    Z, cmd_rec_nopacket
348                     INPUT   s9, command_arg1
                        RL      s9
350                     ADD     s9, socket_start
                        FETCH   sB, (s9)
352                     INPUT   sC, udp_destination_port_h
                        COMPARE sB, sC
354                     JUMP    NZ, cmd_rec_nosocket
                        ADD     s9, 01
356                     FETCH   sB, (s9)
                        INPUT   sC, udp_destination_port_l
358                     COMPARE sB, sC
                        JUMP    NZ, cmd_rec_nosocket
360                                               ; there is a packet and it is for this socket, lets co

362                     INPUT   sA, command_arg2
                        OUTPUT  sA, copy_rx_to_address_h
364                     INPUT   sA, command_arg3
                        OUTPUT  sA, copy_rx_to_address_l
366
                        LOAD    sA, udp_data
368                     OUTPUT  sA, copy_rx_from_address_l
                        LOAD    sA, 00
370                     OUTPUT  sA, copy_rx_from_address_h

372                     INPUT   s8, udp_length_h
                        INPUT   s9, udp_length_l
374
                        SUB     s9, 08
376                     SUBCY   s8, 00

378                     OUTPUT  s9, copy_rx_length_l
                        OUTPUT  s9, result_arg8
380                     OUTPUT  s8, copy_rx_length_h
                        OUTPUT  s8, result_arg7
382
                        LOAD    sA, module_copy_rx
384                     OUTPUT  sA, module_sfr
                        OUTPUT  sA, copy_rx_start
386
                        TEST    s9,01                     ;nop
388                     TEST    s9,01
                        TEST    s9,01
390     cmd_recv_wait:  INPUT   sA, copy_rx_start
                        COMPARE sA, 00
392                     JUMP    Z,  cmd_recv_wait

394
                        LOAD    sA, 00                    ; reset the module_sfr
396                     OUTPUT  sA, module_sfr
                                                          ; clear new frame signal
398                     INPUT   sA, mac_sfr2
                        AND     sA, no_new_packet
400                     OUTPUT  sA, mac_sfr2
                        LOAD    sA, 00                    ; set the buffer for a new packet
```

```
402                         STORE   sA, loc_readstatus

404                         OUTPUT  sA, rx_mac_get_packet   ; Write to register rx_mac_get_packet to get new packet into buff

406                         LOAD    sA, 00                  ; return 00 for succesful
                            OUTPUT  sA, result_arg0
408                         INPUT   sA, udp_source_port_h   ; return source port
                            OUTPUT  sA, result_arg1
410                         INPUT   sA, udp_source_port_l
                            OUTPUT  sA, result_arg2
412                         INPUT   sA, ip_source3          ; return source ip
                            OUTPUT  sA, result_arg3
414                         INPUT   sA, ip_source2
                            OUTPUT  sA, result_arg4
416                         INPUT   sA, ip_source1
                            OUTPUT  sA, result_arg5
418                         INPUT   sA, ip_source0
                            OUTPUT  sA, result_arg6
420                         FETCH   sA, last_command_id
                            OUTPUT  sA, result_id
422                         JUMP    main_loop

424

426       cmd_rec_nopacket:
          cmd_rec_nosocket:
428                         LOAD    sA, FF                  ; return FF for failure
                            OUTPUT  sA, result_arg0
430                         FETCH   sA, last_command_id
                            OUTPUT  sA, result_id
432                         JUMP    main_loop

434        command_socket:  FETCH   sA, socket_alloc        ; Find a free socket, allocate it a source port and return socket
                            LOAD    sB, 01                  ; return FF if there is no available socket
436                         LOAD    s9, 00
          c_socket_try:     TEST    sA, sB                  ; check if the first bit is zero
438                         JUMP    Z,  c_socket_free       ; if so we found us a free socket
                            RL      sB
440                         ADD     s9, 01
                            COMPARE s9, 08
442                         JUMP    NZ, c_socket_try
                            LOAD    s9, FF                  ; no free socket
444                         OUTPUT  s9, result_arg0
                            FETCH   sA, last_command_id
446                         OUTPUT  sA, result_id
                            JUMP    main_loop
448
          c_socket_free:    OR      sA, sB                  ; allocate buffer
450                         STORE   sA, socket_alloc
               get_port:    FETCH   sA, socket_random_port_h
452                         FETCH   sB, socket_random_port_l
                            ADD     sB, 01
454                         ADDCY   sA, 00
                            STORE   sA, socket_random_port_h
456                         STORE   sB, socket_random_port_l
                            CALL    check_duplicate_port
458                         COMPARE sC, 01
                            JUMP    Z,  get_port
460                         OUTPUT  s9, result_arg0         ; return port number
                            LOAD    s8, socket_start
462                         RL      s9
                            ADD     s8, s9
464                         STORE   sA, (s8)
                            ADD     s8, 01
466                         STORE   sB, (s8)
                            FETCH   sA, last_command_id
468                         OUTPUT  sA, result_id
                            JUMP    main_loop
470

472   check_duplicate_port: LOAD    sC, socket_start        ; expects port sA sB
                            SUB     sC, 02
474       check_dup_loop_a: ADD     sC, 02
            check_dup_loop: COMPARE sC, string_start
476                         JUMP    Z,  check_dup_finish
                            FETCH   sD, (sC)
478                         COMPARE sA, sD
                            JUMP    NZ, check_dup_loop_a
480                         ADD     sC, 01
                            FETCH   sD, (sC)
482                         ADD     sC, 01
                            COMPARE sB, sD
484                         JUMP    NZ, check_dup_loop
                            LOAD    sC, 01                  ; oops port in use set Sc = 1
486                         RETURN
       check_dup_finish:    LOAD    sC, 00                  ; not duplicate
488                         RETURN
```

```
490
        command_bind:     INPUT    s9, command_arg1
492                       AND      s9, 07                  ; make sure socket is between 0 and 7
                          INPUT    sA, command_arg2
494                       INPUT    sB, command_arg3

496                       CALL     check_duplicate_port
                          COMPARE  sC, 01
498                       JUMP     Z, command_bind_fail
                          RL       s9
500                       ADD      s9, socket_start
                          STORE    sA, (s9)
502                       ADD      s9, 01
                          STORE    sB, (s9)
504     command_bind_fail: OUTPUT  sC, result_arg0
                          FETCH    sA, last_command_id
506                       OUTPUT   sA, result_id
                          JUMP     main_loop
508
        command_close:    INPUT    sA, command_arg1        ; get the socket to close
510                       AND      sA,07                   ; make sure socket is between 0 and 7
                          LOAD     sB,01                   ; rotate the bit to find the matching bit position
512                       LOAD     sC,00
      command_close_l:    COMPARE  sC,sA
514                       JUMP     Z, command_close_f
                          RL       sB
516                       ADD      sC,01
                          JUMP     command_close_l
518     command_close_f:  FETCH    sC, socket_alloc
                          XOR      sC, sB
520                       STORE    sC, socket_alloc
                          RL       sA
522                       ADD      sA, socket_start
                          LOAD     sB, 00
524                       STORE    sB, (sA)
                          ADD      sA, 01
526                       STORE    sB, (sA)
                          FETCH    sA, last_command_id
528                       OUTPUT   sA, result_id
                          JUMP     main_loop
530
        command_set_nm:   INPUT    sA, command_arg1
532                       STORE    sA, 0A
                          INPUT    sA, command_arg2
534                       STORE    sA, 0B
                          INPUT    sA, command_arg3
536                       STORE    sA, 0C
                          INPUT    sA, command_arg4
538                       STORE    sA, 0D
                          FETCH    sA, last_command_id
540                       OUTPUT   sA, result_id
                          JUMP     main_loop
542
        command_get_nm:   FETCH    sA, 0A
544                       OUTPUT   sA, result_arg0
                          FETCH    sA, 0B
546                       OUTPUT   sA, result_arg1
                          FETCH    sA, 0C
548                       OUTPUT   sA, result_arg2
                          FETCH    sA, 0D
550                       OUTPUT   sA, result_arg3
                          FETCH    sA, last_command_id
552                       OUTPUT   sA, result_id
                          JUMP     main_loop
554
        command_set_gw:   INPUT    sA, command_arg1
556                       STORE    sA, 0E
                          INPUT    sA, command_arg2
558                       STORE    sA, 0F
                          INPUT    sA, command_arg3
560                       STORE    sA, 10
                          INPUT    sA, command_arg4
562                       STORE    sA, 11
                          FETCH    sA, last_command_id
564                       OUTPUT   sA, result_id
                          JUMP     main_loop
566
        command_get_gw:   FETCH    sA, 0E
568                       OUTPUT   sA, result_arg0
                          FETCH    sA, 0F
570                       OUTPUT   sA, result_arg1
                          FETCH    sA, 10
572                       OUTPUT   sA, result_arg2
                          FETCH    sA, 11
574                       OUTPUT   sA, result_arg3
                          FETCH    sA, last_command_id
```

```
576                         OUTPUT  sA, result_id
                            JUMP    main_loop
578
        command_set_ip:     INPUT   sA, command_arg1
580                         STORE   sA, 00
                            INPUT   sA, command_arg2
582                         STORE   sA, 01
                            INPUT   sA, command_arg3
584                         STORE   sA, 02
                            INPUT   sA, command_arg4
586                         STORE   sA, 03
                            FETCH   sA, last_command_id
588                         OUTPUT  sA, result_id
                            JUMP    main_loop
590
        command_get_ip:     FETCH   sA, 00
592                         OUTPUT  sA, result_arg0
                            FETCH   sA, 01
594                         OUTPUT  sA, result_arg1
                            FETCH   sA, 02
596                         OUTPUT  sA, result_arg2
                            FETCH   sA, 03
598                         OUTPUT  sA, result_arg3
                            FETCH   sA, last_command_id
600                         OUTPUT  sA, result_id
                            JUMP    main_loop
602
        command_sendto:
             no_change:     FETCH   sA, loc_writestatus
                            COMPARE sA, 02
606                         JUMP    Z, command_sendto_b_empty
    ; if bottom empty check size and try
                            COMPARE sA, 01
608                         JUMP    Z, command_sendto_t_empty
    ; if top empty check size and try
                            COMPARE sA, 00
610                         JUMP    Z, command_sendto_empty
    ; if nothing in the buffer just prep
                            JUMP    check_change
    ; if bottom and top in use wait for one to be empty
612
 command_sendto_empty:      INPUT   sA, command_arg2                ; get length of packet
614                         COMPARE sA, 04
                            JUMP    C,  empty_small_packet
616                         LOAD    sA, 04                          ; prep bottom and top
                            STORE   sA, loc_prepstatus
618                         JUMP    sendto_prep

620    empty_small_packet:  LOAD    sA, 01                          ; prep only bottom
                            STORE   sA, loc_prepstatus
622                         JUMP    sendto_prep

624 command_sendto_b_empty: INPUT   sA, command_arg2
                            COMPARE sA, 04
626                         JUMP    NC, check_change
    ; packet is to big we need to wait for the entire buffer to be empty
                            LOAD    sA, 01
628                         STORE   sA, loc_prepstatus              ; prep only bottom
                            JUMP    sendto_prep
630

632 command_sendto_t_empty: INPUT   sA, command_arg2
                            COMPARE sA, 04
634                         JUMP    NC, check_change
    ; packet is to big we need to wait for the entire buffer to be empty
                            LOAD    sA, 02
636                         STORE   sA, loc_prepstatus              ; prep only top
                            JUMP    sendto_prep
638
          check_change:     INPUT   sA, mac_status_port
    ; lets check if we can change the write status
640                         TEST    sA, tx_mac_packet_ready
                            JUMP    Z,  no_change
642                         LOAD    sA, 00
                            STORE   sA, loc_writestatus
644                         JUMP    command_sendto
                                                                    ; now we are here we can have the following value
646                                                                 ; prepstatus = 001 lets prepare the bottom
                                                                    ; prepstatus = 010 lets prepare the top
648                                                                 ; prepstatus = 100 lets prepare the bottom
            sendto_prep:    INPUT   s7, command_arg1                ; store the socket id
650                         AND     s7, 07
                            INPUT   s8, command_arg2
652                         OUTPUT  s8, copy_length_h
                            INPUT   s9, command_arg3
654                         OUTPUT  s9, copy_length_l
                            INPUT   sA, command_arg4
```

```
                        OUTPUT  sA, copy_from_address_h          ; store the addr_h
                        INPUT   sA, command_arg5
                        OUTPUT  sA, copy_from_address_l          ; store the addr_l
                        FETCH   sB, loc_prepstatus
; set copy_to_address_h to 04 for top half and to 00 for bottom
                        AND     sB, 02
                        RL      sB
                        OUTPUT  sB, copy_to_address_h
                        LOAD    sA, 2A
; start of udp data assuming ip header with no options
                        OUTPUT  sA, copy_to_address_l
                        LOAD    sA, module_copy
                        OUTPUT  sA, module_sfr
                        OUTPUT  sA, copy_start
                        INPUT   sA, copy_start
                        INPUT   sA, copy_start
                        INPUT   sA, copy_start

    build_udp_wait:     INPUT   sA, copy_start
                        COMPARE sA, 00
                        JUMP    Z, build_udp_wait
                        ; 1 = gelijk , 00 = bezig

                        LOAD    sA, module_mac
                        OUTPUT  sA, module_sfr

                        FETCH   sB, loc_prepstatus
                        AND     sB, 02
                        RL      sB
                        RL      sB
                        INPUT   sA, mac_sfr                      ; set mac offset
                        OR      sA, sB
                        OUTPUT  sA, mac_sfr


                        ADD     s9, 08
; udp length = data length + header length
                        ADDCY   s8, 00

                        LOAD    sA, 00
; create the udp header - source port, destination port, udp length , checksum
                        OUTPUT  sA, udp_checksum_h
                        OUTPUT  sA, udp_checksum_l

                        RL      s7
; get the source port from the socket
                        ADD     s7, socket_start
                        FETCH   sA, (s7)
                        OUTPUT  sA, udp_source_port_h
                        ADD     s7, 01
                        FETCH   sA, (s7)
                        OUTPUT  sA, udp_source_port_l

                        INPUT   sA, command_arg6
                        OUTPUT  sA, udp_destination_port_h

                        INPUT   sA, command_arg7
                        OUTPUT  sA, udp_destination_port_l
                        OUTPUT  s8, udp_length_h
                        OUTPUT  s9, udp_length_l

                        FETCH   sB, 00
                        OUTPUT  sB, udp_pseudo_source_ip3
                        FETCH   sB, 01
                        OUTPUT  sB, udp_pseudo_source_ip2
                        FETCH   sB, 02
                        OUTPUT  sB, udp_pseudo_source_ip1
                        FETCH   sB, 03
                        OUTPUT  sB, udp_pseudo_source_ip0

                        INPUT   s0, command_arg8
                        OUTPUT  s0, udp_pseudo_destination_ip3

                        INPUT   s1, command_arg9
                        OUTPUT  s1, udp_pseudo_destination_ip2

                        INPUT   s2, command_argA
                        OUTPUT  s2, udp_pseudo_destination_ip1

                        INPUT   s3, command_argB
                        OUTPUT  s3, udp_pseudo_destination_ip0

                        LOAD    sA, 00                           ; zero
                        OUTPUT  sA, udp_pseudo_zero
                        LOAD    sA, 11                           ; protocol
                        OUTPUT  sA, udp_pseudo_protocol
```

```
738                         OUTPUT  s8, udp_pseudo_length_h
                            OUTPUT  s9, udp_pseudo_length_l
740
                            LOAD    sA, udp_pseudo_source_ip        ; calculate udp checksum
742                         OUTPUT  sA, crc_address_l
                            FETCH   sB, loc_prepstatus
744                         AND     sB, 02
                            RL      sB
746                         OUTPUT  sB, crc_address_h

748                         ADD     s9, 0C                          ; length for crc = pseudo + header + data
                            ADDCY   s8, 00
750                         OUTPUT  s9, crc_length_l
                            OUTPUT  s8, crc_length_h
752                         SUB     s9, 0C
                            SUBCY   s9, 00
754                         LOAD    sA, module_crc
                            OUTPUT  sA, module_sfr
756                         OUTPUT  sA, crc_start

758                         INPUT   sA, crc_start
                            INPUT   sA, crc_start
760                         INPUT   sA, crc_start

762       build_udp_crc_w:  INPUT   sA, crc_start
                            COMPARE sA, 00
764                         JUMP    Z, build_udp_crc_w

766                         LOAD    sA, module_mac
                            OUTPUT  sA, module_sfr
768
                            INPUT   sA, crc_l
770                         OUTPUT  sA, udp_checksum_l
                            INPUT   sA, crc_h
772                         OUTPUT  sA, udp_checksum_h

774                         LOAD    s7, 11  ; protocol udp

776                         CALL    build_ip

778
                            OUTPUT  s8, result_arg0
780                         OUTPUT  s9, result_arg1
                            FETCH   sA, last_command_id
782                         OUTPUT  sA, result_id
                            JUMP    main_loop
784
786               build_ip:
    ; create the ip header, length in s8 s9, protocol in s7, s0-s3 ip address
                            LOAD    sA, 45
788                         OUTPUT  sA, ip_version_ihl
                            LOAD    sA, 00
790                         OUTPUT  sA, ip_tos
                            OUTPUT  sA, ip_flags
792                         OUTPUT  sA, ip_offset
                            OUTPUT  sA, ip_checksum_h
794                         OUTPUT  sA, ip_checksum_l
                            OUTPUT  s7, ip_protocol
796
                            ADD     s9, 14
798                         ADDCY   s8, 00
                            OUTPUT  s9, ip_length_l
800                         OUTPUT  s8, ip_length_h

802
                            FETCH   sA, loc_ip_identification_h     ; fetch the identifcation counter, use it, update it, and
804                         FETCH   sB, loc_ip_identification_l
                            OUTPUT  sA, ip_identification_h
806                         OUTPUT  sB, ip_identification_l
                            ADD     sB, 01
808                         ADDCY   sA, 00
                            STORE   sA, loc_ip_identification_h
810                         STORE   sB, loc_ip_identification_l

812                         LOAD    sA, 80
                            OUTPUT  sA, ip_ttl
814
                            FETCH   sB, 00
816                         OUTPUT  sB, ip_source3
                            FETCH   sB, 01
818                         OUTPUT  sB, ip_source2
                            FETCH   sB, 02
820                         OUTPUT  sB, ip_source1
                            FETCH   sB, 03
822                         OUTPUT  sB, ip_source0
```

```
824                             OUTPUT  s0, ip_destination3
                                OUTPUT  s1, ip_destination2
826                             OUTPUT  s2, ip_destination1
                                OUTPUT  s3, ip_destination0
828
                                                                        ; calculate ip header checksum
830                             LOAD    sA, ip_version_ihl
                                OUTPUT  sA, crc_address_l
832                             FETCH   sA, loc_prepstatus
                                AND     sA, 02
834                             RL      sA
                                OUTPUT  sA, crc_address_h
836
                                LOAD    sA, 14
838                             OUTPUT  sA, crc_length_l
                                LOAD    sA, 00
840                             OUTPUT  sA, crc_length_h
                                LOAD    sA, module_crc
842                             OUTPUT  sA, module_sfr
                                OUTPUT  sA, crc_start
844
                                INPUT   sA, crc_start
846                             INPUT   sA, crc_start
                                INPUT   sA, crc_start
848
850             build_ip_crc_w: INPUT   sA, crc_start
                                COMPARE sA, 00
852                             JUMP    Z, build_ip_crc_w

854                             LOAD    sA, module_mac
                                OUTPUT  sA, module_sfr
856
                                INPUT   sA, crc_l
858                             OUTPUT  sA, ip_checksum_l
                                INPUT   sA, crc_h
860                             OUTPUT  sA, ip_checksum_h

862                             LOAD    sA, 08
                                LOAD    sB, 00
864                             CALL    build_ethernet

866                             RETURN

868              build_ethernet: OUTPUT  sB, 0D                         ; build the ethernet frame, ethertype in sA, SH
                                OUTPUT  sA, 0C
870                             FETCH   sB, 04
                                OUTPUT  sB, 06
872                             FETCH   sB, 05
                                OUTPUT  sB, 07
874                             FETCH   sB, 06
                                OUTPUT  sB, 08
876                             FETCH   sB, 07
                                OUTPUT  sB, 09
878                             FETCH   sB, 08
                                OUTPUT  sB, 0A
880                             FETCH   sB, 09
                                OUTPUT  sB, 0B
882                             CALL    set_destination_mac
                                CALL    send_packet
884                             RETURN

886               send_packet:  INPUT   sA, mac_status_port
; before sending we need to wait the packet to be send completely
                                TEST    sA, tx_mac_packet_ready
888                             JUMP    Z,  send_packet

890                             INPUT   sA, mac_sfr
                                AND     sA, DF
892                             OUTPUT  sA, mac_sfr

894                             FETCH   sA, loc_prepstatus
                                STORE   sA, loc_writestatus
896
                                COMPARE sA, 02
; if we send the top half we need to reset the send counter to top
898                             JUMP    NZ, send_now
                                INPUT   sA, mac_sfr2
900                             OR      sA, mac_up
                                OUTPUT  sA, mac_sfr2
902
                                LOAD    sA, 00                          ; reset prepstatus
904                             STORE   sA, loc_prepstatus

906
                    send_now:   INPUT   sA, mac_sfr
; set the tx bit high so that the mac starts sending
```

```
908                         OR      sA, mac_send_packet
                            OUTPUT  sA, mac_sfr
910
                            INPUT   sA, mac_sfr2
912                         AND     sA, mac_down                    ; reset mac_sfr2 to down
                            OUTPUT  sA, mac_sfr2
914                         RETURN

916

918               make_arp: LOAD    s9, 10
; Sends an arp request for the ip address in sA,sB,sC,sD
                            OUTPUT  s9, mac_sfr
920                         OUTPUT  sD, 29
                            LOAD    s9, 00
922                         OUTPUT  s9, mac_sfr
                            OUTPUT  sA, 26
924                         OUTPUT  sB, 27
                            OUTPUT  sC, 28
926                         LOAD    sB, 00                          ; write our ip
                            LOAD    sC, 1C
928                         LOAD    sA, 04
                            CALL    copy_scratch2buffer
930                         LOAD    sB, 04                          ; write our mac
                            LOAD    sC, 16
932                         LOAD    sA, 06
                            CALL    copy_scratch2buffer
934                         LOAD    sB, 04
                            LOAD    sC, 06
936                         LOAD    sA, 06
                            CALL    copy_scratch2buffer
938                                                                 ; from now sA = 00 sB = 01, sC = 08, sD=06
                            LOAD    sA, 00
940                         LOAD    sB, 01
                            LOAD    sC, 08
942                         LOAD    sD, 06
                            OUTPUT  sA, 20
944                         OUTPUT  sA, 21
                            OUTPUT  sA, 22
946                         OUTPUT  sA, 23
                            OUTPUT  sA, 24
948                         OUTPUT  sA, 25
                            OUTPUT  sA, 14
950                         OUTPUT  sB, 15
                            LOAD    s9, 04                          ; set protocol size
952                         OUTPUT  s9, 13
                            OUTPUT  sD, 12
954                         OUTPUT  sA, 11
                            OUTPUT  sC, 10
956                         OUTPUT  sB, 0F
                            OUTPUT  sA, 0E
958                         OUTPUT  sD, 0D
                            OUTPUT  sC, 0C
960                         LOAD    sA, FF
                            OUTPUT  sA, 00
962                         OUTPUT  sA, 01
                            OUTPUT  sA, 02
964                         OUTPUT  sA, 03
                            OUTPUT  sA, 04
966                         OUTPUT  sA, 05
                            ;CALL   send_packet       KAAS
968                         RETURN

970           process_udp:  INPUT   sA, ip_destination3             ; check if it is a broadcast
                            COMPARE sA, FF
972                         JUMP    NZ, process_udp_no_bc
                            INPUT   sA, ip_destination2
974                         COMPARE sA, FF
                            JUMP    NZ, process_udp_no_bc
976                         INPUT   sA, ip_destination1
                            COMPARE sA, FF
978                         JUMP    NZ, process_udp_no_bc
                            INPUT   sA, ip_destination0
980                         COMPARE sA, FF
                            JUMP    NZ, process_udp_no_bc
982                         JUMP    process_udp_check_p

984
     process_udp_no_bc:     INPUT   sA, ip_destination3
; check if packet is really for our ip
986                         FETCH   sB, 00
                            COMPARE sA, sB
988                         RETURN  NZ
                            INPUT   sA, ip_destination2
990                         FETCH   sB, 01
                            COMPARE sA, sB
992                         RETURN  NZ
```

```
                        INPUT   sA, ip_destination1
994                     FETCH   sB, 02
                        COMPARE sA, sB
996                     RETURN  NZ
                        INPUT   sA, ip_destination0
998                     FETCH   sB, 03
                        COMPARE sA, sB
1000                    RETURN  NZ

1002  process_udp_check_p:  INPUT   sA, udp_destination_port_h
    ; check if there is a socket open with this port
                        INPUT   sB, udp_destination_port_l
1004                    CALL    check_duplicate_port
                        COMPARE sC, 00
1006                    RETURN  Z
                                                          ; it is a valid packet so now we need to block
1008                                                      ; all other incoming traffic into this buffer
                                                          ; until a process requests this data
1010                    LOAD    sA, 01
                        STORE   sA, loc_readstatus
1012
                                                          ; lets create a signal for the applications
1014                    INPUT   sA, mac_sfr2
                        OR      sA, new_packet
1016                    OUTPUT  sA, mac_sfr2

1018                    RETURN

1020 process_icmp_echo_req:  INPUT   sA, ip_length_h
    ; get length of original echo request
                        INPUT   sB, ip_length_l
1022                    ADD     sB, 0E
    ; add length of ethernet to get total packet length in sA, sB
                        ADDCY   sA, 00
1024                    LOAD    sC, 00
    ; create address sC, sD , where sD < 128
                        LOAD    sD, 00
1026      copy_icmp_packet:  LOAD    s7, sC
                        LOAD    s8, sD
1028                    SL0     s8
                        SLA     s7
1030                    SR0     s8
                        AND     s7, 0F
1032                    OUTPUT  s7, mac_sfr
                        INPUT   s9, (s8)                  ; read byte from rx buffer
1034                    OUTPUT  s9, (s8)                  ; write it back to tx buffer
                        ADD     sD, 01                    ; increase pointer
1036                    ADDCY   sC, 00
                        COMPARE sB, sD
1038                    JUMP    NZ, copy_icmp_packet
                        COMPARE sA, sC
1040                    JUMP    NZ, copy_icmp_packet      ; copy remaining bytes
          make_icmp_req:  OR      s7, 10                    ; last byte lets set eop
1042                    OUTPUT  s7, mac_sfr
                        OUTPUT  s9, (s8)
1044                    LOAD    sA, 00                    ; reset mac_sfr
                        OUTPUT  sA, mac_sfr
1046                    LOAD    sA, icmp_echo_reply       ; set type to reply
                        OUTPUT  sA, icmp_type
1048                    INPUT   sA, icmp_checksum_h
    ; make checksum with cheating, just add 8 since only type field changes
                        INPUT   sB, icmp_checksum_l
1050                    ADD     sA, 08
                        ADDCY   sB, 00
1052                    OUTPUT  sA, icmp_checksum_h
                        OUTPUT  sB, icmp_checksum_l
1054                    LOAD    sB, 1A
    ; swap source ip and destination ip in ip
                        LOAD    sC, 1E
1056                    LOAD    sA, 04
                        CALL    copy_buffer
1058                    LOAD    sB, 1E
                        LOAD    sC, 1A
1060                    LOAD    sA, 04
                        CALL    copy_buffer
1062                    LOAD    sB, 00
    ; swap source arp and destination arp in ethernet
                        LOAD    sC, 06
1064                    LOAD    sA, 06
                        CALL    copy_buffer
1066                    LOAD    sB, 06
                        LOAD    sC, 00
1068                    LOAD    sA, 06
                        CALL    copy_buffer
1070                    CALL    send_packet
                        RETURN
1072
```

```
1074            process_arp:
                        LOAD    sA, 26
    ; Check if the incoming arp packet has our ip in it
1076                    LOAD    sB, 00
                        LOAD    sC, 04
1078  process_arp_check_ip: INPUT   s9, (sA)
                        FETCH   s8, (sB)
1080                    COMPARE s9, s8                          ; if not return
                        RETURN  NZ
1082                    ADD     sA, 01
                        ADD     sB, 01
1084                    SUB     sC, 01
                        JUMP    NZ, process_arp_check_ip
1086                                                            ; ok the arp packet is for us
                        INPUT   sA, arp_hdr_opcode
    ; two options now, either it is a request or a reply
1088                    COMPARE sA, arp_request
                        JUMP    Z, process_arp_request
1090      process_arp_reply: JUMP    save_in_arp
    ; if it is a reply we only need to store the result in the table
    process_arp_request: LOAD    sB, 00
    ; copy 64 bytes of the received packet buffer to the tx buffer
1092                    LOAD    sC, 00
                        LOAD    sA, 40
1094                    CALL    copy_buffer
                        LOAD    sA, arp_reply
    ; change packet from request to reply
1096                    OUTPUT  sA, arp_hdr_opcode
                        LOAD    sB, 06
    ; copy the source mac address to the destination mac address
1098                    LOAD    sC, 00
                        LOAD    sA, 06
1100                    CALL    copy_buffer
                        LOAD    sB, 16
    ; copy the source mac address to the destination mac address in arp
1102                    LOAD    sC, 20
                        LOAD    sA, 0A
1104                    CALL    copy_buffer
                        LOAD    sB, 04
    ; copy the mac address to the source mac address
1106                    LOAD    sC, 06
                        LOAD    sA, 06
1108                    CALL    copy_scratch2buffer
                        LOAD    sB, 04
    ; copy the mac address to the sender mac address in arp
1110                    LOAD    sC, 16
                        LOAD    sA, 06
1112                    CALL    copy_scratch2buffer
                        LOAD    sB, 00
    ; copy the ip address to the sender ip address in arp
1114                    LOAD    sC, 1C
                        LOAD    sA, 04
1116                    CALL    copy_scratch2buffer
                        LOAD    sA, mac_eop                     ; set the end of the packet
1118                    OUTPUT  sA, mac_sfr
                        LOAD    sA, 00
1120                    OUTPUT  sA, 30
                        OUTPUT  sA, mac_sfr
1122                    CALL    send_packet
            save_in_arp: LOAD    sB, 16
    ; save in arp table, copy arp + ip to arp registers
1124                    LOAD    sC, arp_table_eth0
                        LOAD    sA, 0A
1126                    CALL    copy_buffer
                        LOAD    sA, arp_table_write             ; store in arp table
1128                    OUTPUT  sA, arp_table_status
                        LOAD    sA, 00
1130                    OUTPUT  sA, arp_table_status
                        RETURN
1132
    set_destination_mac: LOAD    sA, 1E
    ; assumes that the packet is already in tx buffer on 1e - 21
1134                    LOAD    sB, 00
    ; sA pointer to tx buffer destination ip, sB point to own ip, sC to mask
                        LOAD    sC, 0A
    ; s6 points to the arp table, lets fill in the fields already assuming it is a local transfer
1136                    LOAD    s6, arp_table_ip0
                        LOAD    sD, 04
1138                    INPUT   s9, mac_sfr
                        OR      s9, mac_tx_buffer
1140                    OUTPUT  s9, mac_sfr
    ; set mac_sfr so we can read from tx buffer
    set_destination_mac_l: INPUT   s7, (sA)                     ; current ip byte in tx buffer
1142                    OUTPUT  s7, (s6)
    ; put current ip in lookup for arp table
                        FETCH   s8, (sB)
```

```
; current ip byte of own address
                        FETCH   s9, (sC)                        ; current mask byte
                        AND     s7, s9
                        AND     s8, s9
                        COMPARE s7, s8
; after anding the bytes they should be similar for local address
                        JUMP    NZ, set_destination_not_loc
; if they do not match it is a non local address
                        ADD     sA, 01
                        ADD     sB, 01
                        ADD     sC, 01
                        ADD     s6, 01
                        SUB     sD, 01
                        JUMP    NZ, set_destination_mac_l
                        JUMP    set_destination_loc             ; address is local

set_destination_not_loc:

                        INPUT   sA, 1E                          ; check if it is a broadcast
                        COMPARE sA, FF
                        JUMP    NZ, set_dest_no_bc
                        INPUT   sA, 1F
                        COMPARE sA, FF
                        JUMP    NZ, set_dest_no_bc
                        INPUT   sA, 20
                        COMPARE sA, FF
                        JUMP    NZ, set_dest_no_bc
                        INPUT   sA, 21
                        JUMP    NZ, set_dest_no_bc
                        LOAD    sA, FF                        ; it is a broadcast
                        OUTPUT  sA, 00
                        OUTPUT  sA, 01
                        OUTPUT  sA, 02
                        OUTPUT  sA, 03
                        OUTPUT  sA, 04
                        OUTPUT  sA, 05
                        JUMP    set_dest_finish

       set_dest_no_bc: FETCH   sA, loc_gateway_ip0
; get the ip of the gateway and fill it in arp table lookup
                        OUTPUT  sA, arp_table_ip0
                        FETCH   sB, loc_gateway_ip1
                        OUTPUT  sB, arp_table_ip1
                        FETCH   sC, loc_gateway_ip2
                        OUTPUT  sC, arp_table_ip2
                        FETCH   sD, loc_gateway_ip3
                        OUTPUT  sD, arp_table_ip3
                        CALL    arp_lookup
                        JUMP    Z, set_dest_not_loc_arp
                        LOAD    sB, arp_table_eth0
                        LOAD    sC, 00
                        LOAD    sA, 06
                        CALL    copy_buffer
      set_dest_finish: LOAD     s9, mac_rx_buffer
                        OUTPUT  s9, mac_sfr
                        RETURN

set_dest_not_loc_arp:   CALL    make_arp
                        RETURN

  set_destination_loc:  CALL    arp_lookup
                        JUMP    Z, set_dest_loc_arp
; if not lets go to set_destination_nf
                        LOAD    sB, arp_table_eth0
; we found the address so lets copy the mac address to the packet
                        LOAD    sC, 00
                        LOAD    sA, 06
                        CALL    copy_buffer
                        LOAD    s9, mac_rx_buffer
                        OUTPUT  s9, mac_sfr
                        RETURN

    set_dest_loc_arp:   INPUT   sA, 1E
; We did not find the address so we need to request it with arp and drop the current packet
                        INPUT   sB, 1F
                        INPUT   sC, 20
                        INPUT   sD, 21
                        CALL    make_arp
                        RETURN

          arp_lookup:   LOAD    s9, arp_table_read
                        OUTPUT  s9, arp_table_status
                        LOAD    s9, 00
                        OUTPUT  s9, arp_table_status
```

```
1224        arp_lookup_wait:  INPUT    s9, arp_table_status
                              AND      s9, 0C
1226                          JUMP     Z, arp_lookup_wait
                              INPUT    s9, arp_table_status
1228                          TEST     s9, arp_table_found
                              RETURN
1230

1232
              copy_buffer:    COMPARE sA, 00
; copies sA bytes from sB -> sC
1234                          RETURN   Z
                              INPUT    sD,(sB)
1236                          OUTPUT   sD,(sC)
                              ADD      sB, 01
1238                          ADD      sC, 01
                              SUB      sA, 01
1240                          JUMP     copy_buffer

1242 ;   copy_buffer2scratch: COMPARE sA, 00                          ; copies sA bytes from sB -> scratch(sC)
                              ;RETURN  Z
1244                          ;INPUT     sD, (sB)
                              ;STORE     sD, (sC)
1246                          ;ADD       sB, 01
                              ;ADD       sC, 01
1248                          ;SUB       sA, 01
                              ;JUMP    copy_buffer2scratch
1250

1252    copy_scratch2buffer:  COMPARE sA, 00
; copies sA bytes from Scratch(sB) -> sC
                              RETURN   Z
1254                          FETCH    sD,(sB)
                              OUTPUT   sD,(sC)
1256                          ADD      sB, 01
                              ADD      sC, 01
1258                          SUB      sA, 01
                              JUMP     copy_scratch2buffer
```

Listing 6.2: demo.psm

```
                              ; uart
2                             CONSTANT UART_read_port, E0               ;UART Rx data input
                              CONSTANT UART_write_port, E0              ; UART Tx data output
4                             CONSTANT UART_status_port, C0             ;UART status input
                              CONSTANT tx_half_full, 01                 ;  Transmitter
    half full - bit0
6                             CONSTANT tx_full, 02                      ;    FIFO
    full - bit1
                              CONSTANT rx_half_full, 04                 ;  Receiver
    half full - bit2
8                             CONSTANT rx_full, 08                      ;    FIFO
    full - bit3
                              CONSTANT rx_data_present, 10              ;
    data present - bit4
10
                              CONSTANT status_port, C0
12                            CONSTANT new_packet, 20


14
                              ; counter
16                            CONSTANT counter_write_port, F0
                              CONSTANT counter_read_0, F0
18                            CONSTANT counter_read_1, F1
                              CONSTANT counter_read_2, F2
20                            CONSTANT counter_read_3, F3
                              CONSTANT counter_reset, 01
22                            CONSTANT counter_start, 02
                              CONSTANT counter_stop,  00
24
                              ;LCD interface ports
26                            ;
                              ;The master enable signal is not used by the LCD display itself
28                            ;but may be required to confirm that LCD communication is active.
                              ;This is required on the Spartan-3E Starter Kit if the StrataFLASH
30                            ;is used because it shares the same data pins and conflicts must be avoided.
                              ;
32                            CONSTANT LCD_output_port, A0              ;LCD character module output data and control
                              CONSTANT LCD_E, 01                        ;    active High Enable
    E - bit0
34                            CONSTANT LCD_RW, 02                       ;    Read=1 Write=0
    RW - bit1
                              CONSTANT LCD_RS, 04                       ;    Instruction=0 Data=1
    RS - bit2
36                            CONSTANT LCD_drive, 08                    ;
    Master enable (active High) - bit3
```

```
                        CONSTANT LCD_DB4, 10                ;   4-bit
Data DB4 - bit4
                        CONSTANT LCD_DB5, 20                ;   interface
Data DB5 - bit5
                        CONSTANT LCD_DB6, 40                ;
Data DB6 - bit6
                        CONSTANT LCD_DB7, 80                ;
Data DB7 - bit7
                        ;
                        ;
                        CONSTANT LCD_input_port, A0         ;LCD character module input data
                        CONSTANT LCD_read_spare0, 01        ;    Spare bits
- bit0
                        CONSTANT LCD_read_spare1, 02        ;    are zero
- bit1
                        CONSTANT LCD_read_spare2, 04        ;
- bit2
                        CONSTANT LCD_read_spare3, 08        ;
- bit3
                        CONSTANT LCD_read_DB4, 10           ;    4-bit
Data DB4 - bit4
                        CONSTANT LCD_read_DB5, 20           ;    interface
Data DB5 - bit5
                        CONSTANT LCD_read_DB6, 40           ;
Data DB6 - bit6
                        CONSTANT LCD_read_DB7, 80           ;
Data DB7 - bit7


                        CONSTANT command_id, 80
                        CONSTANT command_arg0, 81
                        CONSTANT command_arg1, 82
                        CONSTANT command_arg2, 83
                        CONSTANT command_arg3, 84
                        CONSTANT command_arg4, 85
                        CONSTANT command_arg5, 86
                        CONSTANT command_arg6, 87
                        CONSTANT command_arg7, 88
                        CONSTANT command_arg8, 89
                        CONSTANT command_arg9, 8A
                        CONSTANT command_argA, 8B
                        CONSTANT command_argB, 8C
                        CONSTANT command_argC, 8D
                        CONSTANT command_argD, 8E
                        CONSTANT command_argE, 8F
                        CONSTANT result_id, 80
                        CONSTANT result_arg0, 81
                        CONSTANT result_arg1, 82
                        CONSTANT result_arg2, 83
                        CONSTANT result_arg3, 84
                        CONSTANT result_arg4, 85
                        CONSTANT result_arg5, 86
                        CONSTANT result_arg6, 87
                        CONSTANT result_arg7, 88
                        CONSTANT result_arg8, 89
                        CONSTANT result_arg9, 8A
                        CONSTANT result_argA, 8B
                        CONSTANT result_argB, 8C
                        CONSTANT result_argC, 8D
                        CONSTANT result_argD, 8E
                        CONSTANT result_argE, 8F


                        ;
                        ;The main operation of the program uses 1ms delays to set the shift rate
                        ;of the LCD display. A 16-bit value determines how many milliseconds
                        ;there are between shifts
                        ;
                        ;Tests indicate that the fastest shift rate that the LCD display supports is
                        ;500ms. Faster than this and the display becomes less clear to read.
                        ;
                        CONSTANT shift_delay_msb, 01        ;delay is 500ms (01F4 hex)
                        CONSTANT shift_delay_lsb, F4
                        ;
                        ;
                        ;
                        ;
                        ;Constant to define a software delay of 1us. This must be adjusted to reflect the
                        ;clock applied to KCPSM3. Every instruction executes in 2 clock cycles making the
                        ;calculation highly predictable. The '6' in the following equation even allows for
                        ;'CALL delay_1us' instruction in the initiating code.
                        ;
                        ; delay_1us_constant =  (clock_rate - 6)/4
Where 'clock_rate' is in MHz
                        ;
                        ;Example: For a 50MHz clock the constant value is (10-6)/4 = 11
(0B Hex).
```

```
110                         ;For clock rates below 10MHz the value of 1 must be used and the operation will
                            ;become lower than intended.
112                         ;
                            CONSTANT delay_1us_constant, 12
114                         ;
                            ;
116                         ;

118                         ;Special Register usage
                            ;
120                         NAMEREG sF, UART_data
;used to pass data to and from the UART
                            ;
122                         NAMEREG sE, store_pointer
;used to pass location of data in scratch pad memory
                            NAMEREG sD, socket
124                         ;
                            ;Scratch Pad Memory Locations
126                         ;
                            ;UART character strings will be stored in scratch pad memory ending in carriage return.
128                         ;A string can be up to 16 characters with the start location defined by this constant.
                            ;
130                         CONSTANT xid3, 10
                            CONSTANT xid2, 11
132                         CONSTANT xid1, 12
                            CONSTANT xid0, 13
134
                            CONSTANT loc_server_id3, 14
136                         CONSTANT loc_server_id2, 15
                            CONSTANT loc_server_id1, 16
138                         CONSTANT loc_server_id0, 17
                            CONSTANT loc_mask3, 18
140                         CONSTANT loc_mask2, 19
                            CONSTANT loc_mask1, 1A
142                         CONSTANT loc_mask0, 1B
                            CONSTANT loc_gw3, 1C
144                         CONSTANT loc_gw2, 1D
                            CONSTANT loc_gw1, 1E
146                         CONSTANT loc_gw0, 1F

148
                            CONSTANT ip_address3, 20
150                         CONSTANT ip_address2, 21
                            CONSTANT ip_address1, 22
152                         CONSTANT ip_address0, 23

154                         CONSTANT last_command_id, 25

156                         CONSTANT socket_echo, 26

158                         CONSTANT string_start, 30
                            ;
160                         ;
                            ;Initialise the system
162                         ;
                            ;
164                         LOAD    sA, 01
                            STORE   sA, last_command_id
166                         CALL    LCD_reset
                            LOAD    s5, 10                          ;Line 1 position 0
168                         CALL    LCD_cursor
                            CALL    disp_uipdemo
170                         CALL    dhcp_init
                            CALL    echo_init
172
            main_loop:      INPUT   sA, UART_status_port            ; Check uart for bytes
174                         TEST    sA, rx_data_present
                            JUMP    NZ, prompt_input                ; Goto debugging
176                         CALL    start_echo
                            JUMP    main_loop                       ; Restart main loop
178

180         prompt_input:   CALL    send_prompt                     ;Prompt 'Demo>'
                            CALL    receive_string
;obtain input string and maintain the time
182                         LOAD    s1, string_start
                            CALL    fetch_char_from_memory
184
                            COMPARE s0, character_R
186                         JUMP Z, read_command
                            COMPARE s0, character_W
188                         JUMP Z, write_command
                            COMPARE s0, character_D
190                         JUMP z, dumppacket_command
                            COMPARE s0, character_X
192                         JUMP z, main_loop
                            COMPARE s0, character_F
```

```
194                         JUMP z, command_fill
                            COMPARE s0, character_S
196                         JUMP z, start_dhcp
                            COMPARE s0, character_M
198                         JUMP z, dumpmem
                            COMPARE s0, character_L
200                         CALL  Z, parse_options
                            COMPARE s0, character_G
202                         CALL z, clearscratch
                            COMPARE s0, character_T
204                         JUMP z, send_test
                            COMPARE s0, character_U
206                         JUMP z, dump_pip_mem
                            COMPARE s0, character_I
208                         JUMP z, dump_pip_port_tx
                            COMPARE s0, character_O
210                         JUMP z, write_pip_port
                            COMPARE s0, character_P
212                         JUMP z, dump_pip_port_rx
                            COMPARE s0, character_Y
214                         JUMP z, do_receive_test
                            JUMP prompt_input
216
    do_receive_test:        LOAD      s2, 00                          ;packet counter in s2
218                         LOAD      s1, counter_reset
                            OUTPUT    s1, counter_write_port
220
                            LOAD      sA, 07                          ; get socket
222                         OUTPUT    sA, command_arg0
                            CALL      set_command_id
224                         CALL      get_result
                            INPUT     s3, result_arg0                 ; keep socket in socket
226
                            LOAD      sA, 08                          ; bind socket to port 2500
228                         OUTPUT    sA, command_arg0
                            OUTPUT    s3, command_arg1
230                         LOAD      sA, 09
                            OUTPUT    sA, command_arg2
232                         LOAD      sA, C4
                            OUTPUT    sA, command_arg3
234                         CALL      set_command_id
                            CALL      get_result
236
    wait_for_first_packet:  INPUT     s1, status_port
238                         TEST      s1, new_packet
                            JUMP      Z, wait_for_first_packet
240                         LOAD      s1, counter_start               ;start timer
                            OUTPUT    s1, counter_write_port
242                         JUMP      receive_process
    receive_next_packet:    INPUT     s1, status_port
244                         TEST      s1, new_packet
                            JUMP      Z, receive_next_packet
246 receive_process:        LOAD      sA, 0B
                            OUTPUT    sA, command_arg0
248                         OUTPUT    s3, command_arg1
                            LOAD      sA, 00
250                         OUTPUT    sA, command_arg2
                            OUTPUT    sA, command_arg3
252                         CALL      set_command_id
                            CALL      get_result
254                         ADD       s2, 01
                            JUMP      NZ, receive_next_packet
256                         LOAD      s1, counter_stop                ; stop timer
                            OUTPUT    s1, counter_write_port
258
                            LOAD      sA, 09                          ; close socket
260                         OUTPUT    sA, command_arg0
                            OUTPUT    s3, command_arg1
262                         CALL      set_command_id
                            CALL      get_result
264
                            LOAD      s1, counter_stop                ; get packet
266                         OUTPUT    s1, counter_write_port
                            LOAD      s1, counter_stop
268                         LOAD      s1, counter_stop
                            INPUT     s1, counter_read_3
270                         CALL      value2ser
                            INPUT     s1, counter_read_2
272                         CALL      value2ser
                            INPUT     s1, counter_read_1
274                         CALL      value2ser
                            INPUT     s1, counter_read_0
276                         CALL      value2ser
                            JUMP      prompt_input
278
                start_echo: INPUT     s1, status_port           ; wait for signal
280                         TEST      s1, new_packet
```

```
                        RETURN    Z
282                     LOAD      sA, 0B                    ; get next packet
                        OUTPUT    sA, command_arg0
284                     FETCH     sC, socket_echo
                        OUTPUT    sC, command_arg1
286                     LOAD      sA, 00
                        OUTPUT    sA, command_arg2
288                     OUTPUT    sA, command_arg3
                        CALL      set_command_id
290                     CALL      get_result

292                     LOAD      sA, 0A                    ; return packet
                        OUTPUT    sA, command_arg0
294                     OUTPUT    sC, command_arg1
                        INPUT     sA, result_arg7
296                     OUTPUT    sA, command_arg2
                        INPUT     sA, result_arg8
298                     OUTPUT    sA, command_arg3
                        LOAD      sA, 00
300                     OUTPUT    sA, command_arg4
                        OUTPUT    sA, command_arg5
302                     INPUT     sA, result_arg1
                        OUTPUT    sA, command_arg6
304                     INPUT     sA, result_arg2
                        OUTPUT    sA, command_arg7
306                     INPUT     sA, result_arg3
                        OUTPUT    sA, command_arg8
308                     INPUT     sA, result_arg4
                        OUTPUT    sA, command_arg9
310                     INPUT     sA, result_arg5
                        OUTPUT    sA, command_argA
312                     INPUT     sA, result_arg6
                        OUTPUT    sA, command_argB
314                     CALL      set_command_id
                        CALL      get_result
316                     RETURN

318
        write_pip_port: LOAD      sA, E1
320                     OUTPUT    sA, command_arg0
                        LOAD      sA, D8
322                     OUTPUT    sA, command_arg1
                        CALL      hex2value   ; result in s3
324                     OUTPUT    s3, command_arg2
                        CALL      set_command_id
326                     CALL      get_result
                        CALL      hex2value
328                     OUTPUT    s3, command_arg1
                        CALL      hex2value
330                     OUTPUT    s3, command_arg2
                        CALL      set_command_id
332                     CALL      get_result
                        JUMP      prompt_input
334

336
         set_command_id: FETCH  sA, last_command_id
338                     ADD    sA, 01
                        STORE  sA, last_command_id
340                     OUTPUT sA, command_id
                        RETURN
342

344      dump_pip_port_tx: LOAD      s8, 00              ; sAsB
                        LOAD      s9, 00
346       dump_pip_outer:                               ; 1111 01111111
                        LOAD      sC, E1
348                     OUTPUT    sC, command_arg0
                        LOAD      sC, D8
350                     OUTPUT    sC, command_arg1
                        OR        s8, 40
352                     OUTPUT    s8, command_arg2
                        CALL      set_command_id
354                     CALL      get_result
       dump_pip_inner: LOAD      sC, E0
356                     OUTPUT    sC, command_arg0
                        OUTPUT    s9, command_arg1
358                     CALL      set_command_id
                        CALL      get_result
360                     INPUT     s1, result_arg0
                        CALL      value2ser
362                     ADD       s9, 01
                        COMPARE   s9, 80
364                     JUMP      NZ, dump_pip_inner
                        AND       s8, 0F
366                     ADD       s8, 01
                        COMPARE   s8, 10
```

```
368                         JUMP    Z, dump_pip_port_tx_f
                            LOAD    s9, 00
370                         JUMP    dump_pip_outer
      dump_pip_port_tx_f:   LOAD    sC, E1
372                         OUTPUT  sC, command_arg0
                            LOAD    sC, D8
374                         OUTPUT  sC, command_arg1
                            LOAD    sC, 00
376                         OUTPUT  sC, command_arg2
                            CALL    set_command_id
378                         CALL    get_result
                            JUMP    prompt_input
380
        dump_pip_port_rx:   LOAD    s8, 00              ; sAsB
382                         LOAD    s9, 00
       dump_pip_outer_rx:                               ; 1111 01111111
384                         LOAD    sC, E1
                            OUTPUT  sC, command_arg0
386                         LOAD    sC, D8
                            OUTPUT  sC, command_arg1
388                         OUTPUT  s8, command_arg2
                            CALL    set_command_id
390                         CALL    get_result
       dump_pip_inner_rx:   LOAD    sC, E0
392                         OUTPUT  sC, command_arg0
                            OUTPUT  s9, command_arg1
394                         CALL    set_command_id
                            CALL    get_result
396                         INPUT   s1, result_arg0
                            CALL    value2ser
398                         ADD     s9, 01
                            COMPARE s9, 80
400                         JUMP    NZ, dump_pip_inner_rx
                            AND     s8, 0F
402                         ADD     s8, 01
                            COMPARE s8, 10
404                         JUMP    Z, dump_pip_port_rx_f
                            LOAD    s9, 00
406                         JUMP    dump_pip_outer_rx
      dump_pip_port_rx_f:   LOAD    sC, E1
408                         OUTPUT  sC, command_arg0
                            LOAD    sC, D8
410                         OUTPUT  sC, command_arg1
                            LOAD    sC, 00
412                         OUTPUT  sC, command_arg2
                            CALL    set_command_id
414                         CALL    get_result
                            JUMP    prompt_input
416
418
          dump_pip_mem:     LOAD    s8, 00
420                         LOAD    sA, E2
                            OUTPUT  sA, command_arg0
422       dump_pip_mem_l:   COMPARE s8, 40
                            JUMP    Z, prompt_input
424                         OUTPUT  s8, command_arg1
                            CALL    set_command_id
426                         CALL    get_result
                            INPUT   s1, result_arg0
428                         CALL    value2ser
                            ADD     s8, 01
430                         JUMP    dump_pip_mem_l

432             send_test:  CALL    hex2value   ; result in s3
                            LOAD    s6, s3
434                         CALL    hex2value
                            LOAD    s8, s3
436                         LOAD    s7, 00              ; send 255 packets
                            LOAD    sA, 00
438                         OUTPUT  sA, C0
                            LOAD    sA, 0A                  ; send packet
440                         OUTPUT  sA, command_arg0
                            LOAD    sA, 00
442                         OUTPUT  sA, command_arg1        ; send socket
                            OUTPUT  s6, command_arg2        ; length
444                         OUTPUT  s8, command_arg3
                            LOAD    sB, 00
446                         OUTPUT  sB, command_arg4        ; address
                            OUTPUT  sB, command_arg5
448                         LOAD    sA, 00                  ; port
                            OUTPUT  sA, command_arg6
450                         LOAD    sA, 43
                            OUTPUT  sA, command_arg7
452                         LOAD    sA, C0                  ; broadcast
                            OUTPUT  sA, command_arg8
454                         LOAD    sA, A8
```

```
                        OUTPUT    sA, command_arg9
456                     LOAD      sA, 0A
                        OUTPUT    sA, command_argA
458                     LOAD      sA, 01
                        OUTPUT    sA, command_argB
460
462           loop_test:
                        OUTPUT    s7, 07
464                     CALL      set_command_id
                        CALL      get_result            ; wait for packet to be send and consume length of packet
466                     ADD       s7, 01
                        COMPARE   s7, 00
468                     JUMP      Z, main_loop
                        JUMP      loop_test
470
          clearscratch:LOAD s8, 00
472                     LOAD sA, 00
             clearsc:   STORE sA, (s8)
474                     ADD   s8, 01
                        COMPARE s8, 40
476                     JUMP NZ, clearsc
                        RETURN
478
              dumpmem: LOAD     s8,00
480           dumpmem_l: FETCH    s1, (s8)
                        CALL     value2ser
482                     ADD      s8, 01
                        COMPARE s8, 40
484                     JUMP     NZ, dumpmem_l
                        JUMP     prompt_input
486
488
           start_dhcp:
490          try_again: CALL      send_discover
                        CALL      delay_1s
492                     LOAD      sA, 0B                    ; check every second for packet
                        OUTPUT    sA, command_arg0
494                     OUTPUT    socket, command_arg1
                        LOAD      sA, 00
496                     OUTPUT    sA, command_arg2
                        OUTPUT    sA, command_arg3
498                     CALL      set_command_id
                        CALL      get_result                ; get result
500                     INPUT     sA, result_arg0
                        COMPARE   sA, 00
502                     JUMP      NZ, try_again
; if no dhcp offer is received try again
                        CALL      parse_options
504                     LOAD      sA, 00
                        OUTPUT    sA, C0                  ; save yiaddress
506                     INPUT     sA, 10
                        STORE     sA, ip_address3
508                     INPUT     sA, 11
                        STORE     sA, ip_address2
510                     INPUT     sA, 12
                        STORE     sA, ip_address1
512                     INPUT     sA, 13
                        STORE     sA, ip_address0
514                     CALL      send_request
                        LOAD      s9, 00
516         wait_for_ack: COMPARE   s9, 0A
                        JUMP      Z, try_again              ; if after ten seconds no reply then retry
518                     ADD       s9, 01
                        CALL      delay_1s
520                     LOAD      sA, 0B                    ; check every second for packet
                        OUTPUT    sA, command_arg0
522                     OUTPUT    socket, command_arg1
                        LOAD      sA, 00
524                     OUTPUT    sA, command_arg2
                        OUTPUT    sA, command_arg3
526                     CALL      set_command_id
                        CALL      get_result                ; get result
528                     INPUT     sA, result_arg0
                        COMPARE   sA, 00
530                     JUMP      NZ, wait_for_ack
; if no dhcp offer is received try again
                        CALL      parse_options
532                     LOAD      sA, 04                ; set gw
                        OUTPUT    sA, command_arg0
534                     FETCH     sA, loc_gw3
                        OUTPUT    sA, command_arg1
536                     FETCH     sA, loc_gw2
                        OUTPUT    sA, command_arg2
538                     FETCH     sA, loc_gw1
                        OUTPUT    sA, command_arg3
```

```
                        FETCH     sA, loc_gw0
                        OUTPUT    sA, command_arg4
                        CALL      set_command_id
                        CALL      get_result
                        LOAD      sA, 06              ; set mask
                        OUTPUT    sA, command_arg0
                        FETCH     sA, loc_mask3
                        OUTPUT    sA, command_arg1
                        FETCH     sA, loc_mask2
                        OUTPUT    sA, command_arg2
                        FETCH     sA, loc_mask1
                        OUTPUT    sA, command_arg3
                        FETCH     sA, loc_mask2
                        OUTPUT    sA, command_arg4
                        CALL      set_command_id
                        CALL      get_result
                        LOAD      sA, 02                          ; set ip
                        OUTPUT    sA, command_arg0
                        FETCH     sA, ip_address3
                        OUTPUT    sA, command_arg1
                        FETCH     sA, ip_address2
                        OUTPUT    sA, command_arg2
                        FETCH     sA, ip_address1
                        OUTPUT    sA, command_arg3
                        FETCH     sA, ip_address0
                        OUTPUT    sA, command_arg4
                        CALL      set_command_id
                        CALL      get_result
                        LOAD      s5, 20                          ;Line 2 position 0
                        CALL      LCD_cursor
                        CALL      disp_ip
                        JUMP      prompt_input

        send_request:   CALL      dhcp_create_msg
                        LOAD      sA, 01
                        OUTPUT    sA, C0      ; add options
                        LOAD      sA, 35      ; dhcp request
                        OUTPUT    sA, 70
                        LOAD      sA, 01
                        OUTPUT    sA, 71
                        LOAD      sA, 03
                        OUTPUT    sA, 72
                        LOAD      sA, 32      ; add option 50
                        OUTPUT    sA, 73
                        LOAD      sA, 04
                        OUTPUT    sA, 74
                        FETCH     sA, ip_address3
                        OUTPUT    sA, 75
                        FETCH     sA, ip_address2
                        OUTPUT    sA, 76
                        FETCH     sA, ip_address1
                        OUTPUT    sA, 77
                        FETCH     sA, ip_address0
                        OUTPUT    sA, 78
                        LOAD      sA, 36      ; add option 54
                        OUTPUT    sA, 79
                        LOAD      sA, 04
                        OUTPUT    sA, 7A
                        FETCH     sA, loc_server_id3
                        OUTPUT    sA, 7B
                        FETCH     sA, loc_server_id2
                        OUTPUT    sA, 7C
                        FETCH     sA, loc_server_id1
                        OUTPUT    sA, 7D
                        FETCH     sA, loc_server_id0
                        OUTPUT    sA, 7E
                        LOAD      sA, FF
                        OUTPUT    sA, 7F
                        LOAD      sA, 0A          ; send packet
                        OUTPUT    sA, command_arg0
                        OUTPUT    socket, command_arg1     ; send socket
                        LOAD      sA, 01          ; length
                        OUTPUT    sA, command_arg2
                        LOAD      sA, 00
                        OUTPUT    sA, command_arg3
                        LOAD      sB, 00
                        OUTPUT    sB, command_arg4          ; address
                        OUTPUT    sB, command_arg5
                        LOAD      sA, 00          ; port
                        OUTPUT    sA, command_arg6
                        LOAD      sA, 43
                        OUTPUT    sA, command_arg7
                        LOAD      sA, FF          ; broadcast
                        OUTPUT    sA, command_arg8
                        OUTPUT    sA, command_arg9
                        OUTPUT    sA, command_argA
```

```
                        OUTPUT    sA, command_argB
628                     CALL      set_command_id
                        CALL      get_result     ; consume length of packet and wait for the packet to be send
630                     LOAD      sA, 00
                        OUTPUT    sA, C0
632                     RETURN

634

636
          parse_options: LOAD     sC, 01
638                     OUTPUT    sC, C0
                        LOAD      sB, 70
640          parse_op_l: COMPARE  sC, 03
                        RETURN    Z
642                     INPUT     sA, (sB)         ; get the option at pointer
                        COMPARE   sA, 35           ; check if option 53
644                     JUMP      NZ, not_op53
                        LOAD      s9, 02
646                     CALL      inc_address
                        INPUT     s8, (sB)         ; return type in s8
648                     LOAD      s9, 01
                        CALL      inc_address
650                     JUMP      parse_op_l
              not_op53: COMPARE   sA, 36
652                     JUMP      NZ, not_op54
                        LOAD      s9, 02
654                     CALL      inc_address
                        INPUT     sA, (sB)
656                     STORE     sA, loc_server_id3
                        LOAD      s9, 01
658                     CALL      inc_address
                        INPUT     sA, (sB)
660                     STORE     sA, loc_server_id2
                        CALL      inc_address
662                     INPUT     sA, (sB)
                        STORE     sA, loc_server_id1
664                     CALL      inc_address
                        INPUT     sA, (sB)
666                     STORE     sA, loc_server_id0
                        CALL      inc_address
668                     JUMP      parse_op_l
              not_op54: COMPARE   sA, FF
670                     JUMP      NZ, not_op256
                        RETURN
672           not_op256: COMPARE  sA, 01
                        JUMP      NZ, not_op01
674                     LOAD      s9, 02
                        CALL      inc_address
676                     INPUT     sA, (sB)
                        STORE     sA, loc_mask3
678                     LOAD      s9, 01
                        CALL      inc_address
680                     INPUT     sA, (sB)
                        STORE     sA, loc_mask2
682                     CALL      inc_address
                        INPUT     sA, (sB)
684                     STORE     sA, loc_mask1
                        CALL      inc_address
686                     INPUT     sA, (sB)
                        STORE     sA, loc_mask0
688                     CALL      inc_address
                        JUMP      parse_op_l
690
              not_op01: COMPARE   sA, 03
692                     JUMP      NZ, not_op03
                        LOAD      s9, 02
694                     CALL      inc_address
                        INPUT     sA, (sB)
696                     STORE     sA, loc_gw3
                        LOAD      s9, 01
698                     CALL      inc_address
                        INPUT     sA, (sB)
700                     STORE     sA, loc_gw2
                        CALL      inc_address
702                     INPUT     sA, (sB)
                        STORE     sA, loc_gw1
704                     CALL      inc_address
                        INPUT     sA, (sB)
706                     STORE     sA, loc_gw0
                        CALL      inc_address
708                     JUMP      parse_op_l

710           not_op03: CALL      inc_address
                        INPUT     s9, (sB)
712                     CALL      inc_address
                        LOAD      s9, 01
```

```
714                         CALL        inc_address
                            JUMP        parse_op_l
716
718             inc_address: ADD        sB, s9          ; sD is step size to increase
                            COMPARE     sB, 80
720                         RETURN      C
                            ADD         sC, 01
722                         OUTPUT      sC, C0
                            SUB         sB, 80
724                         RETURN

726
                dhcp_init: LOAD         sA, 07            ; get socket
728                         OUTPUT      sA, command_arg0
                            CALL        set_command_id
730                         CALL        get_result
                            INPUT       socket, result_arg0       ; keep socket in socket
732
                            LOAD        sA, 08           ; bind socket to port 68
734                         OUTPUT      sA, command_arg0
                            OUTPUT      socket, command_arg1
736                         LOAD        sA, 00
                            OUTPUT      sA, command_arg2
738                         LOAD        sA, 44
                            OUTPUT      sA, command_arg3
740                         CALL        set_command_id
                            CALL        get_result
742                         RETURN

744             echo_init: LOAD         sA, 07                ; get socket
                            OUTPUT      sA, command_arg0
746                         CALL        set_command_id
                            CALL        get_result
748                         INPUT       sB, result_arg0       ; keep socket in socket
                            STORE       sB, socket_echo
750                         LOAD        sA, 08                ; bind socket to port 07
                            OUTPUT      sA, command_arg0
752                         OUTPUT      sB, command_arg1
                            LOAD        sA, 00
754                         OUTPUT      sA, command_arg2
                            LOAD        sA, 07
756                         OUTPUT      sA, command_arg3
                            CALL        set_command_id
758                         CALL        get_result
                            RETURN
760
              get_result: FETCH         sB, last_command_id
762           get_result_l: INPUT       sA, result_id
                            COMPARE     sA, sB
764                         JUMP        NZ, get_result_l
                            RETURN
766
          dhcp_create_msg: LOAD         sA, 00
768                         OUTPUT      sA, C0        ; set to first page
                            LOAD        sA, 01
770                         OUTPUT      sA, 00        ; DHCP_REQUEST
                            OUTPUT      sA, 01        ; HType = Ethernet
772                         LOAD        sA, 06
                            OUTPUT      sA, 02        ; HLEN = 6
774                         LOAD        sA, 00
                            OUTPUT      sA, 03        ; HOPS = 0
776                         LOAD        sA, 0D        ; set static xid to 0D0D0D0D
                            OUTPUT      sA, 04
778                         OUTPUT      sA, 05
                            OUTPUT      sA, 06
780                         OUTPUT      sA, 07
                            LOAD        sA, 00        ; set secs = 0
782                         OUTPUT      sA, 08
                            OUTPUT      sA, 09
784                         LOAD        sA, 80        ; set broadcast flag
                            OUTPUT      sA, 0A
786                         LOAD        sA, 00
                            OUTPUT      sA, 0B
788                         LOAD        sA, 00        ; set ciaddr to zero
                            OUTPUT      sA, 0C
790                         OUTPUT      sA, 0D
                            OUTPUT      sA, 0E
792                         OUTPUT      sA, 0F
                            OUTPUT      sA, 10        ; set yiaddr to zero
794                         OUTPUT      sA, 11
                            OUTPUT      sA, 12
796                         OUTPUT      sA, 13
                            OUTPUT      sA, 14        ; set siaddr to zero
798                         OUTPUT      sA, 15
                            OUTPUT      sA, 16
800                         OUTPUT      sA, 17
```

```
                        OUTPUT    sA, 18       ; set giaddr to zero
802                     OUTPUT    sA, 19
                        OUTPUT    sA, 1A
804                     OUTPUT    sA, 1B

806                     LOAD      sA, 0C       ; get and set the chaddr
                        OUTPUT    sA, command_arg0
808                     CALL      set_command_id
                        CALL      get_result
810
                        INPUT     sA, result_arg0
812                     OUTPUT    sA, 1C
                        INPUT     sA, result_arg1
814                     OUTPUT    sA, 1D
                        INPUT     sA, result_arg2
816                     OUTPUT    sA, 1E
                        INPUT     sA, result_arg3
818                     OUTPUT    sA, 1F
                        INPUT     sA, result_arg4
820                     OUTPUT    sA, 20
                        INPUT     sA, result_arg5
822                     OUTPUT    sA, 21

824                     LOAD      sA, 00       ; set sname to blank
                        LOAD      sB, 22
826           sname_loop: OUTPUT  sA, (sB)
                        ADD       sB, 01
828                     COMPARE   sB, 62
                        JUMP      NZ, sname_loop
830
                        LOAD      sA, 00       ; set file to blank, first part till 80
832                     LOAD      sB, 62
           sfile_loop: OUTPUT     sA, (sB)
834                     ADD       sB, 01
                        COMPARE   sB, 80
836                     JUMP      NZ, sfile_loop

838                     LOAD      sA, 01
                        OUTPUT    sA, C0       ; set address to 80->
840
                        LOAD      sA, 00
842                     LOAD      sB, 00
         sfile_loop2:   OUTPUT    sA, (sB)
844                     ADD       sB, 01
                        COMPARE   sB, 6F
846                     JUMP      NZ, sfile_loop2

848                     LOAD      sA, 63       ; set magic cookie
                        OUTPUT    sA, 6C
850                     LOAD      sA, 82
                        OUTPUT    sA, 6D
852                     LOAD      sA, 53
                        OUTPUT    sA, 6E
854                     LOAD      sA, 63
                        OUTPUT    sA, 6F
856
                        LOAD      sA, 00
858                     OUTPUT    sA, C0     ; reset address preset
                        RETURN
860
       send_discover:   CALL      dhcp_create_msg
862
                        LOAD      sA, 01
864                     OUTPUT    sA, C0      ; add options
                        LOAD      sA, 35      ; dhcp discover
866                     OUTPUT    sA, 70
                        LOAD      sA, 01
868                     OUTPUT    sA, 71
                        OUTPUT    sA, 72
870
                        LOAD      sA, 37
872                     OUTPUT    sA, 73
                        LOAD      sA, 02
874                     OUTPUT    sA, 74
                        LOAD      sA, 01
876                     OUTPUT    sA, 75
                        LOAD      sA, 03
878                     OUTPUT    sA, 76

880                     LOAD      sA, FF
                        OUTPUT    sA, 77
882                     LOAD      sA, 00
                        OUTPUT    sA, 78
884

886                     LOAD      sA, 0A     ; send packet
                        OUTPUT    sA, command_arg0
```

```
888                         OUTPUT    socket, command_arg1    ; send socket
                            LOAD      sB, 00
890                         OUTPUT    sB, command_arg2    ; length
                            LOAD      sA, F9
892                         OUTPUT    sA, command_arg3

894
                            OUTPUT    sB, command_arg4    ; address
896                         OUTPUT    sB, command_arg5
                            OUTPUT    sB, command_arg6    ; port
898                         LOAD      sA, 43
                            OUTPUT    sA, command_arg7
900                         LOAD      sA, FF    ; broadcast
                            OUTPUT    sA, command_arg8
902                         OUTPUT    sA, command_arg9
                            OUTPUT    sA, command_argA
904                         OUTPUT    sA, command_argB
                            CALL      set_command_id
906                         CALL      get_result ; consume length of packet

908                         LOAD      sA, 00
                            OUTPUT    sA, C0
910                         RETURN

912
          command_fill: LOAD    sA, FF
914                     LOAD    sB, 00
             fill_loop: OUTPUT  sA, (sB)
916                     ADD     sB, 01
                        COMPARE sB, 80
918                     JUMP    NZ, fill_loop
                        JUMP    prompt_input
920
    dumppacket_command: LOAD    s7,00
922                     LOAD    s8,00

924           dp_loop: OUTPUT  s7,C0
                        INPUT   s1,(s8)
926                     CALL    value2ser
                        ADD     s8,01
928                     COMPARE s8,80
                        JUMP    NZ, dp_loop
930                     ADD     s7,01
                        AND     s8,7F
932                     COMPARE s7, 03
                        JUMP    NZ, dp_loop
934                     LOAD    s7, 00
                        OUTPUT  s7, C0
936                     JUMP prompt_input

938       read_command: CALL hex2value    ; result in s3
                        INPUT s2,(s3)
940                     LOAD s1,s2
                        CALL value2ser    ; value2ser input in s1
942                     JUMP prompt_input

944      write_command: CALL hex2value    ; result in s3
                        LOAD s4,s3
946                     CALL hex2value
                        OUTPUT s3,(s4)
948                     JUMP prompt_input

950                     ;Send 'Demo>' prompt to the UART
                        ;
952        send_prompt: CALL send_CR                              ;start new line
                        LOAD UART_data, character_D
954                     CALL send_to_UART
                        LOAD UART_data, character_e
956                     CALL send_to_UART
                        LOAD UART_data, character_m
958                     CALL send_to_UART
                        LOAD UART_data, character_o
960                     CALL send_to_UART

962                     ;
                        ;Send '>' character to the UART
964                     ;
 send_greater_than: LOAD UART_data, character_greater_than
966                     CALL send_to_UART
                        RETURN
968                     ;
                        ;
970                     ;Receive ASCII string from UART
                        ;
972                     ;An ASCII string will be read from the UART and stored in scratch pad memory
                        ;commencing at the location specified by a constant named 'string_start'.
974                     ;The string will will have a maximum length of 16 characters including a
```

```
                        ;carriage return (0D) denoting the end of the string.
976                     ;
                        ;As each character is read, it is echoed to the UART transmitter.
978                     ;Some minor editing is supported using backspace (BS=08) which is used
                        ;to adjust what is stored in scratch pad memory and adjust the display
980                     ;on the terminal screen using characters sent to the UART transmitter.
                        ;
982                     ;A test is made for the receiver FIFO becoming full. A full status is treated as
                        ;a potential error situation and will result in a 'Overflow Error' message being
984                     ;transmitted to the UART, the receiver FIFO being purged of all data and an
                        ;empty string being stored (carriage return at first location).
986                     ;
                        ;Registers used s0, s1, s2 and 'UART_data'.
988                     ;
        receive_string: LOAD s1, string_start                   ;locate start of string
990                     LOAD s2, s1                             ;compute 16 character address
                        ADD s2, 10
992     receive_full_test: INPUT s0, UART_status_port           ;test Rx_FIFO buffer for full
                        TEST s0, rx_full
994                     JUMP NZ, read_error
                        CALL read_from_UART                     ;obtain and echo character
996                     STORE UART_data, (s1)                   ;write to memory
                        COMPARE UART_data, character_CR         ;test for end of string
998                     RETURN Z
                        COMPARE UART_data, character_BS         ;test for back space
1000                    JUMP Z, BS_edit
                        ADD s1, 01                              ;increment memory pointer
1002                    COMPARE s1, s2
;test for pointer exceeding 16 characters
                        JUMP NZ, receive_full_test              ;next character
1004                    CALL send_backspace
;hold end of string position on terminal display
              BS_edit: SUB s1, 01                               ;memory pointer back one
1006                    COMPARE s1, string_start                ;test for under flow
                        JUMP C, string_start_again
1008                    CALL send_space
;clear character at current position
                        CALL send_backspace                     ;position cursor
1010                    JUMP receive_full_test                  ;next character
    string_start_again: CALL send_greater_than                  ;restore '>' at prompt
1012                    JUMP receive_string                     ;begin again
                        ;Receiver buffer overflow condition
1014          read_error: CALL send_CR                          ;Transmit error message
                        STORE UART_data, string_start
;empty string in memory (start with CR)
1016                    CALL send_CR
    clear_UART_Rx_loop: INPUT s0, UART_status_port              ;test Rx_FIFO buffer for data
1018                    TEST s0, rx_data_present
                        RETURN Z                                ;finish when buffer is empty
1020                    INPUT UART_data, UART_read_port         ;read from FIFO and ignore
                        JUMP clear_UART_Rx_loop
1022                    ;
                        ;
1024                    ;
                        ;Read one character from the UART
1026                    ;
                        ;Character read will be returned in a register called 'UART_data' and will be
1028                    ;echoed to the UART transmitter.
                        ;
1030                    ;The routine first tests the receiver FIFO buffer to see if data is present.
                        ;If the FIFO is empty, the routine waits until there is a character to read.
1032                    ;As this could take any amount of time the wait loop includes a call to the
                        ;subroutine which updates the real time clock.
1034                    ;
                        ;Registers used s0 and UART_data
1036                    ;
        read_from_UART: INPUT s0, UART_status_port              ;test Rx_FIFO buffer
1038                    TEST s0, rx_data_present
                        JUMP NZ, read_character
1040                    JUMP read_from_UART
        read_character: INPUT UART_data, UART_read_port         ;read from FIFO
1042                    CALL send_to_UART                       ;echo received character
                        RETURN
1044                    ;
                        ;
1046                    ;
                        ;Transmit one character to the UART
1048                    ;
                        ;Character supplied in register called 'UART_data'.
1050                    ;
                        ;The routine first tests the transmit FIFO buffer to see if it is full.
1052                    ;If the FIFO is full, the routine waits until there is space which could
                        ;be as long as it takes to transmit one complete character.
1054                    ;
                        ;     Baud Rate     Time per Character (10 bits)
1056                    ;       9600            1,024us
                        ;      19200              521us
```

```
1058                            ;      38400              260us
                                ;      57600              174us
1060                            ;     115200               87us
                                ;
1062                            ;Since this is a relatively long duration, the wait loop includes a
                                ;call to the subroutine which updates the real time clock.
1064                            ;
                                ;Registers used s0
1066                            ;
              send_to_UART: INPUT s0, UART_status_port              ;test Tx_FIFO buffer
1068                          TEST s0, tx_full
                              JUMP Z, UART_write
1070                          JUMP send_to_UART
                UART_write: OUTPUT UART_data, UART_write_port
1072                          RETURN
                              ;
1074                          ;
                              ;Fetch character from memory, convert to upper case
1076                          ;and increment memory pointer.
                              ;
1078                          ;The memory pointer is provided in register s1.
                              ;The character obtained is returned in register s0.
1080                          ;
                              ;Registers used s0 and s1.
1082                          ;
  fetch_char_from_memory: FETCH s0, (s1)                            ;read character
1084                          CALL upper_case                       ;convert to upper case
                              ADD s1, 01                            ;increment memory pointer
1086                          RETURN
                              ;
1088                          ;
                              ;
1090                          ;Send Carriage Return to the UART
                              ;
1092                  send_CR: LOAD UART_data, character_CR
                              CALL send_to_UART
1094                          RETURN
                              ;
1096                          ;
                              ;
1098                          ;Send a space to the UART
                              ;
1100              send_space: LOAD UART_data, character_space
                              CALL send_to_UART
1102                          RETURN
                              ;
1104                          ;
                              ;Send a back space to the UART
1106                          ;
           send_backspace: LOAD UART_data, character_BS
1108                          CALL send_to_UART
                              RETURN
1110
1112                          ; input in s0

1114               hex2value: CALL serhex2value
                              LOAD s2, s0
1116                          SL0  s2
                              SL0  s2
1118                          SL0  s2
                              SL0  s2
1120                          CALL serhex2value
                              OR   s2,s0
1122                          LOAD s3,s2
                              RETURN
1124
              serhex2value: CALL fetch_char_from_memory
1126                          ADD s0, C6
  ;reject character codes above '9' (39 hex)
                              JUMP C,serhex2valuegt9
  ;carry flag is set
1128                          SUB s0, F6
  ;reject character codes below '0' (30 hex)
                              RETURN
  ;carry is set if value not in range
1130         serhex2valuegt9: ADD s0 ,03
                              RETURN
1132
1134                          ;Convert byte to ascii hex
                              ;IN : S1 byte waarde
1136                          ;OUT: Serial hex ascii
                              ;
1138               value2ser: LOAD s2, s1
                              SR0  s2
1140                          SR0  s2
```

```
                        SR0   s2
1142                    SR0   s2
                        CALL nibble2hex
1144                    LOAD UART_data, s2
                        CALL send_to_UART
1146                    LOAD s2,s1
                        AND  s2,0F
1148                    CALL nibble2hex
                        LOAD UART_data, s2
1150                    CALL send_to_UART
                        RETURN
1152                    ;
                        ; Convert a nibble in register s2 into an ASCII character, 0-9 A-F
1154                    ; The value provided must be in the range of 0 to 15 and will be converted into
                        ; one ASCII character
1156                    ;
                        ;
1158          nibble2hex: COMPARE s2,0A
                        JUMP NC, niblebt9
1160                    ADD s2,30
                        RETURN
1162            niblebt9: ADD s2,37
                        RETURN
1164                    ;
                        ;Convert character to upper case
1166                    ;
                        ;The character supplied in register s0.
1168                    ;If the character is in the range 'a' to 'z', it is converted
                        ;to the equivalent upper case character in the range 'A' to 'Z'.
1170                    ;All other characters remain unchanged.
                        ;
1172                    ;Registers used s0.
                        ;
1174           upper_case: COMPARE s0, 61
;eliminate character codes below 'a' (61 hex)
                        RETURN C
1176                    COMPARE s0, 7B
;eliminate character codes above 'z' (7A hex)
                        RETURN NC
1178                    AND s0, DF
;mask bit5 to convert to upper case
                        RETURN
1180                    ;
                        ;Display a space on LCD at current cursor position
1182                    ;
                        ;
1184           disp_space: LOAD s5, character_space
                        CALL LCD_write_data
1186                    RETURN

1188
               disp_ip: LOAD    s5, 10                          ;Line 1 position 0
1190                    CALL    LCD_cursor
                        LOAD    sA, 01
1192                    OUTPUT  sA, command_arg0
                        CALL    set_command_id
1194                    CALL    get_result

1196                    INPUT   sA, result_arg0
                        CALL    display_number
1198                    LOAD    s5, character_dot
                        CALL    LCD_write_data
1200                    INPUT   sA, result_arg1
                        CALL    display_number
1202                    LOAD    s5, character_dot
                        CALL    LCD_write_data
1204                    INPUT   sA, result_arg2
                        CALL    display_number
1206                    LOAD    s5, character_dot
                        CALL    LCD_write_data
1208                    INPUT   sA, result_arg3
                        CALL    display_number
1210                    RETURN

1212
       display_number: LOAD    sB, 64                        ; displays number in sA, changes sB and s5
1214                    CALL    times
                        JUMP    Z, hundred_is_zero
1216                    ADD     s5, 30
                        CALL    LCD_write_data
1218
  hundred_is_not_zero: LOAD    sB, 0A
1220                    CALL    times
                        ADD     s5, 30
1222                    CALL    LCD_write_data
                        JUMP    do_one
1224
```

```
        hundred_is_zero: LOAD     sB, 0A
1226                     CALL     times
                         JUMP     Z, do_one
1228                     ADD      s5, 30
                         CALL     LCD_write_data
1230
                do_one: LOAD     sB, 01
1232                     CALL     times
                         ADD      s5, 30
1234                     CALL     LCD_write_data
                         RETURN
1236
                 times: LOAD     s5, 00              ; input sA, input sB, output s5
1238            times_l: ADD      s5, 01
                         SUB      sA, sB
1240                     JUMP     NC, times_l
                         ADD      sA, sB
1242                     SUB      s5, 01
                         RETURN
1244
                         ;
1246                     ;
                         ;
1248                     ;
                         ;*****************************************************************************
1250                     ;Software delay routines
                         ;*****************************************************************************
1252                     ;
                         ;
1254                     ;
                         ;Delay of 1us.
1256                     ;
                         ;Constant value defines reflects the clock applied to KCPSM3. Every instruction
1258                     ;executes in 2 clock cycles making the calculation highly predictable. The '6' in
                         ;the following equation even allows for 'CALL delay_1us' instruction in the initiating
1260                     ;
                         ; delay_1us_constant =  (clock_rate - 6)/4
    Where 'clock_rate' is in MHz
1262                     ;
                         ;Registers used s0
1264                     ;
             delay_1us: LOAD s0, delay_1us_constant
1266              wait_1us: SUB s0, 01
                         JUMP NZ, wait_1us
1268                     SUB s0, 00 ; needed to compensate clock
                         RETURN
1270                     ;
                         ;Delay of 40us.
1272                     ;
                         ;Registers used s0, s1
1274                     ;
            delay_40us: LOAD s1, 28                          ;40 x 1us = 40us
1276             wait_40us: CALL delay_1us
                         SUB s1, 01
1278                     JUMP NZ, wait_40us
                         RETURN
1280                     ;
                         ;
1282                     ;Delay of 1ms.
                         ;
1284                     ;Registers used s0, s1, s2
                         ;
1286             delay_1ms: LOAD s2, 19                          ;25 x 40us = 1ms
              wait_1ms: CALL delay_40us
1288                     SUB s2, 01
                         JUMP NZ, wait_1ms
1290                     RETURN
                         ;
1292                     ;Delay of 20ms.
                         ;
1294                     ;Delay of 20ms used during initialisation.
                         ;
1296                     ;Registers used s0, s1, s2, s3
                         ;
1298            delay_20ms: LOAD s3, 14                          ;20 x 1ms = 20ms
             wait_20ms: CALL delay_1ms
1300                     SUB s3, 01
                         JUMP NZ, wait_20ms
1302                     RETURN
                         ;
1304                     ;Delay of approximately 1 second.
                         ;
1306                     ;Registers used s0, s1, s2, s3, s4
                         ;
1308               delay_1s: LOAD s4, 32                          ;50 x 20ms = 1000ms
                wait_1s: CALL delay_20ms
1310                     SUB s4, 01
```

```
                        JUMP NZ, wait_1s
1312                    RETURN
                        ;
1314                    ;
                        ;
1316                    ;*****************************************************************************
                        ;LCD Character Module Routines
1318                    ;*****************************************************************************
                        ;
1320                    ;LCD module is a 16 character by 2 line display but all displays are very similar
                        ;The 4-wire data interface will be used (DB4 to DB7).
1322                    ;
                        ;The LCD modules are relatively slow and software delay loops are used to slow down
1324                    ;KCPSM3 adequately for the LCD to communicate. The delay routines are provided in
                        ;a different section (see above in this case).
1326                    ;
                        ;
1328                    ;Pulse LCD enable signal 'E' high for greater than 230ns (1us is used).
                        ;
1330                    ;Register s4 should define the current state of the LCD output port.
                        ;
1332                    ;Registers used s0, s4
                        ;
1334        LCD_pulse_E: XOR s4, LCD_E                      ;E=1
                        OUTPUT s4, LCD_output_port
1336                    CALL delay_1us
                        XOR s4, LCD_E                     ;E=0
1338                    OUTPUT s4, LCD_output_port
                        RETURN
1340                    ;
                        ;Write 4-bit instruction to LCD display.
1342                    ;
                        ;The 4-bit instruction should be provided in the upper 4-bits of register s4.
1344                    ;Note that this routine does not release the master enable but as it is only
                        ;used during initialisation and as part of the 8-bit instruction write it
1346                    ;should be acceptable.
                        ;
1348                    ;Registers used s4
                        ;
1350    LCD_write_inst4: AND s4, F8                         ;Enable=1 RS=0 Instruction, RW=0 Write, E=0
                        OUTPUT s4, LCD_output_port        ;set up RS and RW >40ns before enable pulse
1352                    CALL LCD_pulse_E
                        RETURN
1354                    ;
                        ;
1356                    ;Write 8-bit instruction to LCD display.
                        ;
1358                    ;The 8-bit instruction should be provided in register s5.
                        ;Instructions are written using the following sequence
1360                    ; Upper nibble
                        ; wait >1us
1362                    ; Lower nibble
                        ; wait >40us
1364                    ;
                        ;Registers used s0, s1, s4, s5
1366                    ;
    LCD_write_inst8: LOAD s4, s5
1368                    AND s4, F0                          ;Enable=0 RS=0 Instruction, RW=0 Write, E=0
                        OR s4, LCD_drive                    ;Enable=1
1370                    CALL LCD_write_inst4                ;write upper nibble
                        CALL delay_1us                      ;wait >1us
1372                    LOAD s4, s5                         ;select lower nibble with
                        SL1 s4                              ;Enable=1
1374                    SL0 s4                              ;RS=0 Instruction
                        SL0 s4                              ;RW=0 Write
1376                    SL0 s4                              ;E=0
                        CALL LCD_write_inst4                ;write lower nibble
1378                    CALL delay_40us                     ;wait >40us
                        LOAD s4, F0                         ;Enable=0 RS=0 Instruction, RW=0 Write, E=0
1380                    OUTPUT s4, LCD_output_port          ;Release master enable
                        RETURN
1382                    ;
                        ;
1384                    ;
                        ;Write 8-bit data to LCD display.
1386                    ;
                        ;The 8-bit data should be provided in register s5.
1388                    ;Data bytes are written using the following sequence
                        ; Upper nibble
1390                    ; wait >1us
                        ; Lower nibble
1392                    ; wait >40us
                        ;
1394                    ;Registers used s0, s1, s4, s5
                        ;
1396     LCD_write_data: LOAD s4, s5
                        AND s4, F0                          ;Enable=0 RS=0 Instruction, RW=0 Write, E=0
```

```
1398                    OR s4, 0C                       ;Enable=1 RS=1 Data, RW=0 Write, E=0
                        OUTPUT s4, LCD_output_port      ;set up RS and RW >40ns before enable pulse
1400                    CALL LCD_pulse_E                ;write upper nibble
                        CALL delay_1us                  ;wait >1us
1402                    LOAD s4, s5                     ;select lower nibble with
                        SL1 s4                          ;Enable=1
1404                    SL1 s4                          ;RS=1 Data
                        SL0 s4                          ;RW=0 Write
1406                    SL0 s4                          ;E=0
                        OUTPUT s4, LCD_output_port      ;set up RS and RW >40ns before enable pulse
1408                    CALL LCD_pulse_E                ;write lower nibble
                        CALL delay_40us                 ;wait >40us
1410                    LOAD s4, F0                     ;Enable=0 RS=0 Instruction, RW=0 Write, E=0
                        OUTPUT s4, LCD_output_port      ;Release master enable
1412                    RETURN
                        ;
1414                    ;
                        ;
1416                    ;
                        ;Read 8-bit data from LCD display.
1418                    ;
                        ;The 8-bit data will be read from the current LCD memory address
1420                    ;and will be returned in register s5.
                        ;It is advisable to set the LCD address (cursor position) before
1422                    ;using the data read for the first time otherwise the display may
                        ;generate invalid data on the first read.
1424                    ;
                        ;Data bytes are read using the following sequence
1426                    ; Upper nibble
                        ; wait >1us
1428                    ; Lower nibble
                        ; wait >40us
1430                    ;
                        ;Registers used s0, s1, s4, s5
1432                    ;
LCD_read_data8:         LOAD s4, 0E                     ;Enable=1 RS=1 Data, RW=1 Read, E=0
1434                    OUTPUT s4, LCD_output_port      ;set up RS and RW >40ns before enable pulse
                        XOR s4, LCD_E                   ;E=1
1436                    OUTPUT s4, LCD_output_port
                        CALL delay_1us                  ;wait >260ns to access data
1438                    INPUT s5, LCD_input_port        ;read upper nibble
                        XOR s4, LCD_E                   ;E=0
1440                    OUTPUT s4, LCD_output_port
                        CALL delay_1us                  ;wait >1us
1442                    XOR s4, LCD_E                   ;E=1
                        OUTPUT s4, LCD_output_port
1444                    CALL delay_1us                  ;wait >260ns to access data
                        INPUT s0, LCD_input_port        ;read lower nibble
1446                    XOR s4, LCD_E                   ;E=0
                        OUTPUT s4, LCD_output_port
1448                    AND s5, F0                      ;merge upper and lower nibbles
                        SR0 s0
1450                    SR0 s0
                        SR0 s0
1452                    SR0 s0
                        OR s5, s0
1454                    LOAD s4, 04                     ;Enable=0 RS=1 Data, RW=0 Write, E=0
                        OUTPUT s4, LCD_output_port      ;Stop reading 5V device and release master enable
1456                    CALL delay_40us                 ;wait >40us
                        RETURN
1458                    ;
                        ;
1460                    ;Reset and initialise display to communicate using 4-bit data mode
                        ;Includes routine to clear the display.
1462                    ;
                        ;Requires the 4-bit instructions 3,3,3,2 to be sent with suitable delays
1464                    ;following by the 8-bit instructions to set up the display.
                        ;
1466                    ;  28 = '001' Function set, '0' 4-bit mode, '1' 2-line, '0' 5x7 dot matrix, 'xx'
                        ;  06 = '000001' Entry mode, '1' increment, '0' no display shift
1468                    ;  0C = '00001' Display control, '1' display on, '0' cursor off, '0' cursor blink off
                        ;  01 = '00000001' Display clear
1470                    ;
                        ;Registers used s0, s1, s2, s3, s4
1472                    ;
LCD_reset:              CALL delay_20ms                 ;wait more that 15ms for display to be ready
1474                    LOAD s4, 30
                        CALL LCD_write_inst4            ;send '3'
1476                    CALL delay_20ms                 ;wait >4.1ms
                        CALL LCD_write_inst4            ;send '3'
1478                    CALL delay_1ms                  ;wait >100us
                        CALL LCD_write_inst4            ;send '3'
1480                    CALL delay_40us                 ;wait >40us
                        LOAD s4, 20
1482                    CALL LCD_write_inst4            ;send '2'
                        CALL delay_40us                 ;wait >40us
1484                    LOAD s5, 28                     ;Function set
```

```
                        CALL LCD_write_inst8
1486                    LOAD s5, 06                         ;Entry mode
                        CALL LCD_write_inst8
1488                    LOAD s5, 0C                         ;Display control
                        CALL LCD_write_inst8
1490          LCD_clear: LOAD s5, 01                        ;Display clear
                        CALL LCD_write_inst8
1492                    CALL delay_1ms                      ;wait >1.64ms for display to clear
                        CALL delay_1ms
1494                    RETURN
                        ;
1496                    ;Position the cursor ready for characters to be written.
                        ;The display is formed of 2 lines of 16 characters and each
1498                    ;position has a corresponding address as indicated below.
                        ;
1500                    ;                    Character position
                        ;           0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
1502                    ;
                        ; Line 1 - 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
1504                    ; Line 2 - C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF
                        ;
1506                    ;This routine will set the cursor position using the value provided
                        ;in register s5. The upper nibble will define the line and the lower
1508                    ;nibble the character position on the line.
                        ; Example s5 = 2B will position the cursor on line 2 position 11
1510                    ;
                        ;Registers used s0, s1, s2, s3, s4
1512                    ;
           LCD_cursor: TEST s5, 10                          ;test for line 1
1514                    JUMP Z, set_line2
                        AND s5, 0F                          ;make address in range 80 to 8F for line 1
1516                    OR s5, 80
                        CALL LCD_write_inst8                ;instruction write to set cursor
1518                    RETURN
            set_line2: AND s5, 0F                           ;make address in range C0 to CF for line 2
1520                    OR s5, C0
                        CALL LCD_write_inst8                ;instruction write to set cursor
1522                    RETURN
                        ;
1524                    ;This routine will shift the complete display one position to the left.
                        ;The cursor position and LCD memory contents will not change.
1526                    ;
                        ;
1528                    ;Registers used s0, s1, s2, s3, s4, s5
                        ;
1530        LCD_shift_left: LOAD s5, 18                     ;shift display left
                        CALL LCD_write_inst8
1532                    RETURN
                        ;
1534          disp_uipdemo: LOAD s5, character_u
                        CALL LCD_write_data
1536                    LOAD s5, character_I
                        CALL LCD_write_data
1538                    LOAD s5, character_P
                        CALL LCD_write_data
1540                    CALL disp_space
                        LOAD s5, character_D
1542                    CALL LCD_write_data
                        LOAD s5, character_E
1544                    CALL LCD_write_data
                        LOAD s5, character_M
1546                    CALL LCD_write_data
                        LOAD s5, character_O
1548                    CALL LCD_write_data
                        RETURN
1550
                        ;
1552                    ;Useful constants
                        ;
1554                    ;
                        ;ASCII table
1556                    ;
                        CONSTANT character_a, 61
1558                    CONSTANT character_b, 62
                        CONSTANT character_c, 63
1560                    CONSTANT character_d, 64
                        CONSTANT character_e, 65
1562                    CONSTANT character_f, 66
                        CONSTANT character_g, 67
1564                    CONSTANT character_h, 68
                        CONSTANT character_i, 69
1566                    CONSTANT character_j, 6A
                        CONSTANT character_k, 6B
1568                    CONSTANT character_l, 6C
                        CONSTANT character_m, 6D
1570                    CONSTANT character_n, 6E
                        CONSTANT character_o, 6F
```

```
1572                        CONSTANT character_p , 70
                            CONSTANT character_q , 71
1574                        CONSTANT character_r , 72
                            CONSTANT character_s , 73
1576                        CONSTANT character_t , 74
                            CONSTANT character_u , 75
1578                        CONSTANT character_v , 76
                            CONSTANT character_w , 77
1580                        CONSTANT character_x , 78
                            CONSTANT character_y , 79
1582                        CONSTANT character_z , 7A
                            CONSTANT character_A , 41
1584                        CONSTANT character_B , 42
                            CONSTANT character_C , 43
1586                        CONSTANT character_D , 44
                            CONSTANT character_E , 45
1588                        CONSTANT character_F , 46
                            CONSTANT character_G , 47
1590                        CONSTANT character_H , 48
                            CONSTANT character_I , 49
1592                        CONSTANT character_J , 4A
                            CONSTANT character_K , 4B
1594                        CONSTANT character_L , 4C
                            CONSTANT character_M , 4D
1596                        CONSTANT character_N , 4E
                            CONSTANT character_O , 4F
1598                        CONSTANT character_P , 50
                            CONSTANT character_Q , 51
1600                        CONSTANT character_R , 52
                            CONSTANT character_S , 53
1602                        CONSTANT character_T , 54
                            CONSTANT character_U , 55
1604                        CONSTANT character_V , 56
                            CONSTANT character_W , 57
1606                        CONSTANT character_X , 58
                            CONSTANT character_Y , 59
1608                        CONSTANT character_Z , 5A
                            CONSTANT character_0 , 30
1610                        CONSTANT character_1 , 31
                            CONSTANT character_2 , 32
1612                        CONSTANT character_3 , 33
                            CONSTANT character_4 , 34
1614                        CONSTANT character_5 , 35
                            CONSTANT character_6 , 36
1616                        CONSTANT character_7 , 37
                            CONSTANT character_8 , 38
1618                        CONSTANT character_9 , 39
                            CONSTANT character_colon , 3A
1620                        CONSTANT character_semi_colon , 3B
                            CONSTANT character_less_than , 3C
1622                        CONSTANT character_greater_than , 3E
                            CONSTANT character_equals , 3D
1624                        CONSTANT character_space , 20
                            CONSTANT character_CR , 0D              ;carriage return
1626                        CONSTANT character_question , 3F        ;'?'
                            CONSTANT character_dollar , 24
1628                        CONSTANT character_BS , 08              ;Back Space command character
                            CONSTANT character_dot , 2E
1630                        ;
```

Listing 6.3: app_interface.v

```
`timescale 1ns / 1ps
2 //////////////////////////////////////////////////////////////////////////////////
  // Company:
4 // Engineer:
  //
6 // Create Date:    20:44:38 03/30/2009
  // Design Name:
8 // Module Name:    app_interface
  // Project Name:
10 // Target Devices:
  // Tool versions:
12 // Description:
  //
14 // Dependencies:
  //
16 // Revision:
  // Revision 0.01 - File Created
18 // Additional Comments:
  //
20 //////////////////////////////////////////////////////////////////////////////////
  module app_interface(
22     input            clk,
       input      [7:0] input_data,
```

```
24        input        [7:0] port_id,
          input              write_enable,
26        input              read_enable,

28        output       [7:0] output_data,
          output             empty
30      );

32    reg [7:0] fifo [15:0];
      reg [7:0] output_data_int = 8'b00000000;
34    reg [3:0] write_address = 4'b0000;
      reg [3:0] read_address  = 4'b0000;
36
      assign empty       = (read_address == write_address) ? 1'b1 : 1'b0;
38    assign output_data = (~port_id[0]) ? output_data_int : {7'b0000000, empty};

40    always @(posedge clk)
         if (write_enable)
42          write_address <= write_address + 1;

44    always @(posedge clk)
         if (read_enable & ~empty)
46          read_address <= read_address + 1;

48    always @(posedge clk)
         if (write_enable)
50          fifo[write_address] <= input_data;

52    always @(posedge clk)
         output_data_int <= fifo[read_address];
54
endmodule
```

Listing 6.4: app_interface_tf.v

```
1  `timescale 1ns / 1ps

3  ////////////////////////////////////////////////////////////////////////////////
   // Company:
5  // Engineer:
   //
7  // Create Date:   21:29:24 03/30/2009
   // Design Name:   app_interface
9  // Module Name:   C:/NoSpace/final_from_scratch/thesis/app_interface_tf.v
   // Project Name:  thesis
11 // Target Device:
   // Tool versions:
13 // Description:
   //
15 // Verilog Test Fixture created by ISE for module: app_interface
   //
17 // Dependencies:
   //
19 // Revision:
   // Revision 0.01 - File Created
21 // Additional Comments:
   //
23 ////////////////////////////////////////////////////////////////////////////////

25 module app_interface_tf;

27    // Inputs
      reg clk;
29    reg [7:0] input_data;

31    reg write_enable;
      reg read_enable;
33
      // Outputs
35    wire [7:0] output_data;
      wire empty;
37
      // Instantiate the Unit Under Test (UUT)
39    app_interface uut (
         .clk(clk),
41       .input_data(input_data),
         .write_enable(write_enable),
43       .read_enable(read_enable),
         .output_data(output_data),
45       .empty(empty)
      );
47
      initial begin
49       clk = 1'b0;
         forever #10 clk = ~clk;
```

```
51      end

53
        initial begin
55          // Initialize Inputs
            input_data = 0;
57          write_enable = 0;
            read_enable = 0;
59
            // Wait 100 ns for global reset to finish
61
            #110;
63          input_data = 1;
            write_enable = 1;
65          read_enable = 0;
            #20
67          input_data = 2;
            write_enable = 1;
69          read_enable = 0;
            #20
71          write_enable = 0;
            read_enable = 1;
73          #20
            write_enable = 0;
75          read_enable = 1;
            #20
77          input_data = 3;
            read_enable = 0;
79          write_enable = 1;
            #20
81          input_data = 4;
            write_enable = 1;
83          read_enable = 1;
            #20;
85          write_enable = 0;
            read_enable = 1;
87          #100;
            input_data = 5;
89          write_enable = 0;
            #20;
91          write_enable = 0;

93

95

        end
97
    endmodule
```

Listing 6.5: copy.v

```
1   //////////////////////////////////////////////////////////////////////////////////
    // Copyright 2009 Michel Bieleveld
3   //
    // This file is part of the UDP/IP stack.
5   //
    // The UDP/IP stack is free software: you can redistribute it and/or modify
7   // it under the terms of the GNU Lesser General Public License as published by
    // the Free Software Foundation, either version 3 of the License, or
9   // (at your option) any later version.
    //
11  // The UDP/IP stack is distributed in the hope that it will be useful,
    // but WITHOUT ANY WARRANTY; without even the implied warranty of
13  // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    // GNU Lesser General Public License for more details.
15  //
    // You should have received a copy of the GNU Lesser General Public License
17  // along with the UDP/IP stack.  If not, see <http://www.gnu.org/licenses/>.
    //
19  //////////////////////////////////////////////////////////////////////////////////

21  `timescale 1ns / 1ps
    module copy(
23
        input               clk,
25
        // from
27      output reg  [15:0]  copy_from_address = 16'b0000000000000000,
        input       [7:0]   copy_from_data,
29
        // to
31      output reg  [15:0]  copy_to_address = 16'b0000000000000000,
        output reg          copy_to_we = 1'b0,
33      output reg  [7:0]   copy_to_data = 8'b00000000,
        output reg          last_byte = 1'b0,
```

```
35
        // registers
37      input     [7:0]     reg_address,
        input               reg_we,
39      input     [7:0]     reg_data_in,
        output    [7:0]     reg_data_out
41    );

43      reg last_byte_d = 1'b0;
        // address      write               read
45      // 0            start_copy          status
        // 1            from_address_l      status
47      // 2            from_address_h      status
        // 3            length_l            status
49      // 4            length_h            status
        // 5            to_address_l        status
51      // 6            to_address_h        status

53      reg           ready           = 1'b1;
        reg           start_copy      = 1'b0;
55      reg [15:0]    length          = 16'b0000000000000000;

57      reg           last_one        = 1'b0;
        reg           length_ce       = 1'b0;
59      reg           to_address_ce   = 1'b0;
        reg           from_address_ce= 1'b0;
61
        assign        reg_data_out = {7'b0000000, ready};
63
        always @(posedge clk)
65         copy_to_data <= copy_from_data;

67      always @(posedge clk)
           start_copy <= (reg_address[2:0] == 3'b000 & reg_we);
69
        always @(posedge clk)
71         if (from_address_ce)
              copy_from_address <= copy_from_address + 1;
73         else if (reg_address[2:0] == 3'b010 & reg_we)
              copy_from_address[15:8] <= reg_data_in;
75         else if (reg_address[2:0] == 3'b001 & reg_we)
              copy_from_address[7:0] <= reg_data_in;
77
        always @(posedge clk)
79         if (to_address_ce)
              copy_to_address <= copy_to_address + 1;
81         else if (reg_address[2:0] == 3'b101 & reg_we)
              copy_to_address[7:0] <= reg_data_in;
83         else if (reg_address[2:0] == 3'b110 & reg_we)
              copy_to_address[15:8] <= reg_data_in;
85
        always @(posedge clk)
87         if (length_ce)
              length <= length - 1;
89         else if (reg_address[2:0] == 3'b011 & reg_we)
              length[7:0] <= reg_data_in;
91         else if (reg_address[2:0] == 3'b100 & reg_we)
              length[15:8] <= reg_data_in;
93


95      always @(posedge clk)
           if (length == 16'b0000000000000010)
97            last_one <= 1'b1;
           else
99            last_one <= 1'b0;

101     always @(posedge clk)
           last_byte_d <= last_one;
103
        always @(posedge clk)
105        last_byte <= last_byte_d;

107

        parameter st_idle    = 6'b000001;
109     parameter st_offset  = 6'b000010;
        parameter st_offset2 = 6'b000100;
111     parameter st_copy    = 6'b001000;
        parameter st_finish  = 6'b010000;
113     parameter st_finish2 = 6'b100000;

115      (* FSM_ENCODING="ONE-HOT", SAFE_IMPLEMENTATION="NO" *) reg [5:0] state = st_idle;

117     always@(posedge clk)
           (* FULL_CASE, PARALLEL_CASE *) case (state)
119           st_idle : begin
                 if (start_copy)
121                 state <= st_offset;
```

```
                else
                    state <= st_idle;
                length_ce <= 1'b0;
                to_address_ce <= 1'b0;
                from_address_ce <= 1'b0;
                copy_to_we <= 1'b0;
                ready <= 1'b1;
            end
            st_offset : begin
                state <= st_offset2;
                length_ce <= 1'b1;
                to_address_ce <= 1'b0;
                from_address_ce <= 1'b1;
                copy_to_we <= 1'b0;
                ready <= 1'b0;
            end
            st_offset2 : begin
                state <= st_copy;
                length_ce <= 1'b1;
                to_address_ce <= 1'b0;
                from_address_ce <= 1'b1;
                copy_to_we <= 1'b0;
                ready <= 1'b0;
            end
            st_copy : begin
                if (last_one)
                    state <= st_finish;
                else
                    state <= st_copy;
                length_ce <= 1'b1;
                to_address_ce <= 1'b1;
                from_address_ce <= 1'b1;
                copy_to_we <= 1'b1;
                ready <= 1'b0;
            end
            st_finish : begin
                state <= st_idle;
                length_ce <= 1'b0;
                to_address_ce <= 1'b1;
                from_address_ce <= 1'b0;
                copy_to_we <= 1'b1;
                ready <= 1'b0;
            end
            st_finish2 : begin
                state <= st_idle;
                length_ce <= 1'b0;
                to_address_ce <= 1'b1;
                from_address_ce <= 1'b0;
                copy_to_we <= 1'b1;
                ready <= 1'b0;
            end

        endcase

endmodule
```

Listing 6.6: copy_rx.v

```
////////////////////////////////////////////////////////////////////////////////
// Copyright 2009 Michel Bieleveld
//
// This file is part of the UDP/IP stack.
//
// The UDP/IP stack is free software: you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// The UDP/IP stack is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with the UDP/IP stack.  If not, see <http://www.gnu.org/licenses/>.
//
////////////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ps
module copy_rx(

        input               clk,

        // from
```

```
27        output reg  [15:0]   copy_from_address = 16'b0000000000000000,
          input       [7:0]    copy_from_data,
29
          // to
31        output reg  [15:0]   copy_to_address = 16'b0000000000000000,
          output reg           copy_to_we = 1'b0,
33        output reg  [7:0]    copy_to_data = 8'b00000000,
          output reg           last_byte = 1'b0,
35
          // registers
37        input     [7:0]    reg_address,
          input              reg_we,
39        input     [7:0]    reg_data_in,
          output    [7:0]    reg_data_out
41      );

43        reg last_byte_d = 1'b0;
          // address      write              read
45        // 0            start_copy         status
          // 1            from_address_l     status
47        // 2            from_address_h     status
          // 3            length_l           status
49        // 4            length_h           status
          // 5            to_address_l       status
51        // 6            to_address_h       status

53        reg          ready          = 1'b1;
          reg          start_copy     = 1'b0;
55        reg [15:0]   length         = 16'b0000000000000000;

57        reg          last_one       = 1'b0;
          reg          length_ce      = 1'b0;
59        reg          to_address_ce  = 1'b0;
          reg          from_address_ce= 1'b0;
61
          assign       reg_data_out = {7'b0000000, ready};
63


65
          always @(posedge clk)
67           copy_to_data <= copy_from_data;

69
          always @(posedge clk)
71           start_copy <= (reg_address[2:0] == 3'b000 & reg_we);

73        always @(posedge clk)
             if (from_address_ce)
75              copy_from_address <= copy_from_address + 1;
             else if (reg_address[2:0] == 3'b010 & reg_we)
77              copy_from_address[15:8] <= reg_data_in;
             else if (reg_address[2:0] == 3'b001 & reg_we)
79              copy_from_address[7:0] <= reg_data_in;

81        always @(posedge clk)
             if (to_address_ce)
83              copy_to_address <= copy_to_address + 1;
             else if (reg_address[2:0] == 3'b101 & reg_we)
85              copy_to_address[7:0] <= reg_data_in;
             else if (reg_address[2:0] == 3'b110 & reg_we)
87              copy_to_address[15:8] <= reg_data_in;

89        always @(posedge clk)
             if (length_ce)
91              length <= length - 1;
             else if (reg_address[2:0] == 3'b011 & reg_we)
93              length[7:0] <= reg_data_in;
             else if (reg_address[2:0] == 3'b100 & reg_we)
95              length[15:8] <= reg_data_in;

97
          always @(posedge clk)
99           if (length == 16'b0000000000000010)
                last_one <= 1'b1;
101          else
                last_one <= 1'b0;
103
          always @(posedge clk)
105          last_byte_d <= last_one;

107       always @(posedge clk)
             last_byte <= last_byte_d;
109

111       parameter st_idle    = 8'b00000001;
          parameter st_offset  = 8'b00000010;
113       parameter st_offset2 = 8'b00000100;
```

```
    parameter st_copy    = 8'b00001000;
    parameter st_finish  = 8'b00010000;
    parameter st_finish2 = 8'b00100000;
    parameter st_offset3 = 8'b01000000;
    parameter st_offset4 = 8'b10000000;

     (* FSM_ENCODING="ONE-HOT", SAFE_IMPLEMENTATION="NO" *) reg [7:0] state = st_idle;

    always@(posedge clk)
       (* FULL_CASE, PARALLEL_CASE *) case (state)
           st_idle : begin
              if (start_copy)
                 state <= st_offset;
              else
                 state <= st_idle;
              length_ce       <= 1'b0;
              to_address_ce   <= 1'b0;
              from_address_ce <= 1'b0;
              copy_to_we      <= 1'b0;
              ready           <= 1'b1;
           end
           st_offset : begin
              state           <= st_offset2;
              length_ce       <= 1'b1;
              to_address_ce   <= 1'b0;
              from_address_ce <= 1'b1;
              copy_to_we      <= 1'b0;
              ready           <= 1'b0;
           end
           st_offset2 : begin
              //state <= st_copy;
              state           <= st_offset3;
              length_ce       <= 1'b1;
              to_address_ce   <= 1'b0;
              from_address_ce <= 1'b1;
              copy_to_we      <= 1'b0;
              ready           <= 1'b0;
           end
           st_offset3 : begin
              //state <= st_copy;
              state           <= st_copy;
              length_ce       <= 1'b1;
              to_address_ce   <= 1'b1;
              from_address_ce <= 1'b1;
              copy_to_we      <= 1'b1;
              ready           <= 1'b0;
           end

           st_copy : begin
              if (last_one)
                 state <= st_finish;
              else
                 state <= st_copy;
              length_ce       <= 1'b1;
              to_address_ce   <= 1'b1;
              from_address_ce <= 1'b1;
              copy_to_we      <= 1'b1;
              ready           <= 1'b0;
           end
           st_finish : begin
              state           <= st_finish2;
              length_ce       <= 1'b0;
              to_address_ce   <= 1'b1;
              from_address_ce <= 1'b0;
              copy_to_we      <= 1'b1;
              ready           <= 1'b0;
           end
           st_finish2 : begin
              state           <= st_idle;
              length_ce       <= 1'b0;
              to_address_ce   <= 1'b1;
              from_address_ce <= 1'b0;
              copy_to_we      <= 1'b1;
              ready           <= 1'b0;
           end


        endcase

endmodule
```

Listing 6.7: crc_8bit.v

```
//////////////////////////////////////////////////////////////////////////////
```

```
2  // Copyright 2009 Michel Bieleveld
   //
4  // This file is part of the UDP/IP stack.
   //
6  // The UDP/IP stack is free software: you can redistribute it and/or modify
   // it under the terms of the GNU Lesser General Public License as published by
8  // the Free Software Foundation, either version 3 of the License, or
   // (at your option) any later version.
10 //
   // The UDP/IP stack is distributed in the hope that it will be useful,
12 // but WITHOUT ANY WARRANTY; without even the implied warranty of
   // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
14 // GNU Lesser General Public License for more details.
   //
16 // You should have received a copy of the GNU Lesser General Public License
   // along with the UDP/IP stack.  If not, see <http://www.gnu.org/licenses/>.
18 //
   ////////////////////////////////////////////////////////////////////////////////
20
   `timescale 1ns / 1ps
22 module crc_8bit(
      input                clk,
24    output reg  [15:0]   address = 16'b0000000000000000,
      input       [7:0]    data_in,
26    input       [7:0]    reg_address,
      input       [7:0]    reg_data_in,
28    input                reg_we,
      output reg  [7:0]    reg_data_out
30    );

32    reg         [15:0]   length = 16'b0000000000000000;
      reg         [15:0]   CRC = {16{1'b0}};
34    reg                  start_crc = 1'b0;
      reg                  start_crc_d;
36    reg                  start_crc_d_d;

38    reg                  last_one_d;
      reg                  last_one_d_d;
40    reg                  last_one_d_d_d;
      reg                  last_one_d_d_d_d;
42
      reg                  last_one = 1'b0;
44    reg                  odd = 1'b0;
      reg                  crc_ready = 1'b1;
46
      reg   [7:0]  data_h;
48    reg   [7:0]  data_l;
      reg          CRC_Carry;
50    reg          crc_reset;

52    // address     write             read
      // 0           start_crc         status
54    // 1           address_l         crc_0
      // 2           address_h         crc_1
56    // 3           length_l          0
      // 4           length_h          0
58
      always @(posedge clk)
60       if (reg_address[2:0] == 3'b000 & reg_we)
            crc_ready <= 1'b0;
62       else if (last_one_d_d_d)
            crc_ready <= 1'b1;
64
      always @(posedge clk)
66       if (last_one) begin
            start_crc <= 1'b0;
68          crc_reset <= 1'b0;
         end else if (reg_address[2:0] == 3'b000 & reg_we) begin
70          start_crc <= 1'b1;
            odd       <= length[0];
72          crc_reset <= 1'b1;
         end else begin
74          crc_reset <= 1'b0;
         end
76
      always @(reg_address, crc_ready, CRC)
78       case (reg_address[1:0])
            2'b00: reg_data_out = {7'b0000000, crc_ready};
80          2'b01: reg_data_out = ~CRC[7:0];
            2'b10: reg_data_out = ~CRC[15:8];
82          2'b11: reg_data_out = 8'b10101010;
         endcase
84
      always @(posedge clk)
86       last_one <= (length == 16'b0000000000000010);

88    always @(posedge clk)
```

```
        if (start_crc)
90          length <= length -1;
            //length <= length;
92      else if (reg_address[2:0] == 3'b011 & reg_we)
            length[7:0] <= reg_data_in;
94      else if (reg_address[2:0] == 3'b100 & reg_we)
            length[15:8] <= reg_data_in;
96
    always @(posedge clk)
98      if (start_crc)
            address <= address +1;
100     else if (reg_address[2:0] == 3'b001 & reg_we)
            address[7:0] <= reg_data_in;
102     else if (reg_address[2:0] == 3'b010 & reg_we)
            address[15:8] <= reg_data_in;
104


106
    always @(posedge clk) begin
108     start_crc_d <= start_crc;
        start_crc_d_d <= start_crc_d;
110     last_one_d <= last_one;
        last_one_d_d <= last_one_d;
112     last_one_d_d_d <= last_one_d_d;
        last_one_d_d_d_d <= last_one_d_d_d;
114 end

116 always @(posedge clk)
        if (~start_crc_d | odd & last_one_d)
118         data_l <= 8'b00000000;
        else if (~address[0])
120         data_l <= data_in;

122 always @(posedge clk)
        if (~start_crc_d)
124         data_h <= 8'b00000000;
        else if (address[0])
126         data_h <= data_in;

128 always @(posedge clk)
        if (crc_reset) begin
130        CRC <= {16{1'b0}};
           CRC_Carry <= 1'b0;
132     end else if (address[0] & start_crc_d_d | last_one_d & odd | last_one_d_d
| last_one_d_d_d | last_one_d_d_d_d)
            {CRC_Carry, CRC } <= CRC + {data_h, data_l} + CRC_Carry;
134


136
endmodule
```

Listing 6.8: dcm.v

```
////////////////////////////////////////////////////////////////////////////////
2 // Copyright 2009 Michel Bieleveld
  //
4 // This file is part of the UDP/IP stack.
  //
6 // The UDP/IP stack is free software: you can redistribute it and/or modify
  // it under the terms of the GNU Lesser General Public License as published by
8 // the Free Software Foundation, either version 3 of the License, or
  // (at your option) any later version.
10 //
  // The UDP/IP stack is distributed in the hope that it will be useful,
12 // but WITHOUT ANY WARRANTY; without even the implied warranty of
  // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
14 // GNU Lesser General Public License for more details.
  //
16 // You should have received a copy of the GNU Lesser General Public License
  // along with the UDP/IP stack.  If not, see <http://www.gnu.org/licenses/>.
18 //
  ////////////////////////////////////////////////////////////////////////////////
20
  `timescale 1ns / 1ps
22
  module dcma(CLKIN_IN,
24            RST_IN,
              CLKFX_OUT,
26            CLKIN_IBUFG_OUT,
              CLK0_OUT,
28            CLK2X_OUT,
              LOCKED_OUT);
30
    input CLKIN_IN;
32  input RST_IN;
```

```
    output CLKFX_OUT;
    output CLKIN_IBUFG_OUT;
    output CLK0_OUT;
    output CLK2X_OUT;
    output LOCKED_OUT;

    wire CLKFB_IN;
    wire CLKFX_BUF;
    wire CLKIN_IBUFG;
    wire CLK0_BUF;
    wire CLK2X_BUF;
    wire GND_BIT;

    assign GND_BIT = 0;
    assign CLKIN_IBUFG_OUT = CLKIN_IBUFG;
    assign CLK0_OUT = CLKFB_IN;
    BUFG CLKFX_BUFG_INST (.I(CLKFX_BUF),
                          .O(CLKFX_OUT));
    IBUFG CLKIN_IBUFG_INST (.I(CLKIN_IN),
                            .O(CLKIN_IBUFG));
    BUFG CLK0_BUFG_INST (.I(CLK0_BUF),
                         .O(CLKFB_IN));
    BUFG CLK2X_BUFG_INST (.I(CLK2X_BUF),
                          .O(CLK2X_OUT));
    DCM_SP DCM_SP_INST (.CLKFB(CLKFB_IN),
                        .CLKIN(CLKIN_IBUFG),
                        .DSSEN(GND_BIT),
                        .PSCLK(GND_BIT),
                        .PSEN(GND_BIT),
                        .PSINCDEC(GND_BIT),
                        .RST(RST_IN),
                        .CLKDV(),
                        .CLKFX(CLKFX_BUF),
                        .CLKFX180(),
                        .CLK0(CLK0_BUF),
                        .CLK2X(CLK2X_BUF),
                        .CLK2X180(),
                        .CLK90(),
                        .CLK180(),
                        .CLK270(),
                        .LOCKED(LOCKED_OUT),
                        .PSDONE(),
                        .STATUS());
    defparam DCM_SP_INST.CLK_FEEDBACK = "1X";
    defparam DCM_SP_INST.CLKDV_DIVIDE = 2.0;
    defparam DCM_SP_INST.CLKFX_DIVIDE = 5;
    defparam DCM_SP_INST.CLKFX_MULTIPLY = 8;
    defparam DCM_SP_INST.CLKIN_DIVIDE_BY_2 = "FALSE";
    defparam DCM_SP_INST.CLKIN_PERIOD = 20.000;
    defparam DCM_SP_INST.CLKOUT_PHASE_SHIFT = "NONE";
    defparam DCM_SP_INST.DESKEW_ADJUST = "SYSTEM_SYNCHRONOUS";
    defparam DCM_SP_INST.DFS_FREQUENCY_MODE = "LOW";
    defparam DCM_SP_INST.DLL_FREQUENCY_MODE = "LOW";
    defparam DCM_SP_INST.DUTY_CYCLE_CORRECTION = "TRUE";
    defparam DCM_SP_INST.FACTORY_JF = 16'hC080;
    defparam DCM_SP_INST.PHASE_SHIFT = 0;
    defparam DCM_SP_INST.STARTUP_WAIT = "FALSE";
endmodule
```

Listing 6.9: module_arp.v

```
//////////////////////////////////////////////////////////////////////////////
// Copyright 2009 Michel Bieleveld
//
// This file is part of the UDP/IP stack.
//
// The UDP/IP stack is free software: you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// The UDP/IP stack is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with the UDP/IP stack.  If not, see <http://www.gnu.org/licenses/>.
//
//////////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ps
module module_arp(
      input         clk,
      input  [7:0] port_id,
```

```
      input  [7:0] in_port,
      input        write_strobe,
      output reg [7:0] out_port
    );

////////////////////////////////////////////////////////////////////////////////
// Instantiate registers
////////////////////////////////////////////////////////////////////////////////

reg [7:0] ip_0 = 8'b00000001;
reg [7:0] ip_1 = 8'b00000000;
reg [7:0] ip_2 = 8'b00000000;
reg [7:0] ip_3 = 8'b00000000;

reg [7:0] eth_0 = 8'b00000001;
reg [7:0] eth_1 = 8'b00000000;
reg [7:0] eth_2 = 8'b00000000;
reg [7:0] eth_3 = 8'b00000000;
reg [7:0] eth_4 = 8'b00000000;
reg [7:0] eth_5 = 8'b00000000;


reg [1:0] status_r = 4'b00;
reg [1:0] status_r_int = 4'b00;
reg [1:0] status_w = 4'b00;
reg       status_w_reset = 1'b0;

wire status_write;
wire status_read;

assign status_write = status_w[0];
assign status_read  = status_w[1];

always @(posedge clk)
      if (status_write | status_read ) begin
         status_r <= 4'b00;
      end else begin
         status_r <= status_r_int;
      end

always @(posedge clk)
      if (status_w_reset) begin
         status_w <= 2'b00;
      end else if (port_id[3:0] == 4'b0000 & write_strobe) begin
         status_w <= in_port[1:0];
      end

always @(posedge clk)
      if (port_id[3:0] == 4'b0001 & write_strobe)
         eth_0 <= in_port;

always @(posedge clk)
      if (port_id[3:0] == 4'b0010 & write_strobe)
         eth_1 <= in_port;

always @(posedge clk)
      if (port_id[3:0] == 4'b0011 & write_strobe)
         eth_2 <= in_port;

always @(posedge clk)
      if (port_id[3:0] == 4'b0100 & write_strobe)
         eth_3 <= in_port;

always @(posedge clk)
      if (port_id[3:0] == 4'b0101 & write_strobe)
         eth_4 <= in_port;

always @(posedge clk)
      if (port_id[3:0] == 4'b0110 & write_strobe)
         eth_5 <= in_port;

always @(posedge clk)
      if (port_id[3:0] == 4'b0111 & write_strobe)
         ip_0 <= in_port;

always @(posedge clk)
      if (port_id[3:0] == 4'b1000 & write_strobe)
         ip_1 <= in_port;

always @(posedge clk)
      if (port_id[3:0] == 4'b1001 & write_strobe)
         ip_2 <= in_port;

always @(posedge clk)
      if (port_id[3:0] == 4'b1010 & write_strobe)
         ip_3 <= in_port;
```

```
112
////////////////////////////////////////////////////////////////////////////////
114 // Instantiate address counters
////////////////////////////////////////////////////////////////////////////////
116
    reg [3:0] read_address_counter = 4'b0000;
118
    reg read_address_counter_ce = 1'b0;
120 reg write_address_counter_ce = 1'b0;
    reg read_address_counter_loop = 1'b0;
122 reg read_address_counter_reset = 1'b0;
    reg read_address_counter_up = 1'b1;
124
    reg [3:0] compensate = 4'b0000;
126
    always @(posedge clk)
128    if (read_address_counter_reset | (read_address_counter_ce & ~read_address_counter_up))
          compensate <= 4'b0000;
130    else if (read_address_counter_ce & read_address_counter_up & ~(compensate[0] & compensate[1]))
          compensate <= compensate + 1;
132
    always @(posedge clk)
134    if (read_address_counter_reset) begin
          read_address_counter <= 0;
136       read_address_counter_loop <= 0;
       end else if (read_address_counter_ce & read_address_counter_up)
138       {read_address_counter_loop,read_address_counter} <= read_address_counter + 1;
       else if (read_address_counter_ce)
140       read_address_counter <= read_address_counter - compensate;

142 reg [3:0] write_address_counter = 4'b0000;

144 always @(posedge clk)
       if (write_address_counter_ce)
146       write_address_counter <= write_address_counter + 1;

148 ////////////////////////////////////////////////////////////////////////////////
    // Instantiate distributed ram
150 ////////////////////////////////////////////////////////////////////////////////
    parameter RAM_WIDTH = 80;
152 parameter RAM_ADDR_BITS = 4;

154 reg [RAM_WIDTH-1:0] arp_table [(2**RAM_ADDR_BITS)-1:0];

156 wire [RAM_WIDTH-1:0] arp_table_output;
    wire [RAM_WIDTH-1:0] arp_table_input;
158 reg                  arp_table_we;

160 assign arp_table_input[RAM_WIDTH-1:0] = {eth_5,eth_4,eth_3,eth_2,eth_1,eth_0,ip_3,ip_2,ip_1,ip_0};

162 always @(posedge clk)
       if (arp_table_we)
164       arp_table[write_address_counter] <= arp_table_input;

166 assign arp_table_output = arp_table[read_address_counter];

168 ////////////////////////////////////////////////////////////////////////////////
    // out_port
170 ////////////////////////////////////////////////////////////////////////////////

172 //    always @(posedge clk)
    //       case (port_id[2:0])
174 //          3'b000: out_port = {4'b0000,status_r,status_w};
    //          3'b001: out_port = arp_table_output[39:32];
176 //          3'b010: out_port = arp_table_output[47:40];
    //          3'b011: out_port = arp_table_output[55:48];
178 //          3'b100: out_port = arp_table_output[63:56];
    //          3'b101: out_port = arp_table_output[71:64];
180 //          3'b110: out_port = arp_table_output[79:72];
    //          3'b111: out_port = 8'b10101010;
182 //       endcase

184 //  always @(port_id[2:0], status_r[1:0], status_w[1:0],arp_table_output[39:32], arp_table_output[47:40],
    //           arp_table_output[55:48], arp_table_output[63:56], arp_table_output[71:64], arp_table_output[79:72])
186 //       case (port_id[2:0])
    //          3'b000: out_port = {4'b0000,status_r[1:0],status_w[1:0]};
188 //          3'b001: out_port = arp_table_output[39:32];
    //          3'b010: out_port = arp_table_output[47:40];
190 //          3'b011: out_port = arp_table_output[55:48];
    //          3'b100: out_port = arp_table_output[63:56];
192 //          3'b101: out_port = arp_table_output[71:64];
    //          3'b110: out_port = arp_table_output[79:72];
194 //          3'b111: out_port = 8'b01010101;
    //       endcase
196
       always @(posedge clk)
198    case (port_id[2:0])
```

```
            3'b000: out_port = {4'b0000,status_r[1:0],status_w[1:0]};
            3'b001: out_port = arp_table_output[39:32];
            3'b010: out_port = arp_table_output[47:40];
            3'b011: out_port = arp_table_output[55:48];
            3'b100: out_port = arp_table_output[63:56];
            3'b101: out_port = arp_table_output[71:64];
            3'b110: out_port = arp_table_output[79:72];
            3'b111: out_port = 8'b01010101;
        endcase

////////////////////////////////////////////////////////////////////////////////
// Instantiate signals
////////////////////////////////////////////////////////////////////////////////

    reg matching_ip = 1'b0;

    always @(posedge clk)
        if (arp_table_output[31:0] == {ip_3,ip_2,ip_1,ip_0})
            matching_ip <= 1'b1;
        else
            matching_ip <= 1'b0;

////////////////////////////////////////////////////////////////////////////////
// Instantiate FSM
////////////////////////////////////////////////////////////////////////////////
    parameter st_idle   = 9'b000000001;
    parameter st_write  = 9'b000000010;
    parameter st_read   = 9'b000000100;
    parameter st_lookup = 9'b000001000;
    parameter st_found  = 9'b000010000;
    parameter st_nfound = 9'b000100000;
    parameter st_nop    = 9'b001000000;
    parameter st_nop2   = 9'b010000000;
    parameter st_nop3   = 9'b100000000;

    (* FSM_ENCODING="ONE-HOT", SAFE_IMPLEMENTATION="NO" *) reg [8:0] state = st_idle;

    always@(posedge clk)
          (* FULL_CASE, PARALLEL_CASE *) case (state)
             st_idle : begin
                if (status_write)
                   state <= st_write;
                else if (status_read)
                   state <= st_read;
                else
                   state <= st_idle;
                status_w_reset <= 1'b0;
                arp_table_we <= 1'b0;
                write_address_counter_ce <= 1'b0;
                read_address_counter_ce <= 1'b0;
                read_address_counter_reset <= 1'b0;
                read_address_counter_up <= 1'b1;
             end
             st_write : begin
                status_w_reset <= 1'b1;
                arp_table_we <= 1'b1;
                write_address_counter_ce <= 1'b1;
                read_address_counter_ce <= 1'b0;
                read_address_counter_reset <= 1'b0;
                //read_address_counter_up <= 1'b1;
                read_address_counter_up <= 1'b0;
                status_r_int <= 2'b00;
                state <= st_nop;
             end
             st_read : begin
                if (read_address_counter_loop)
                   state <= st_read;
                else
                   //state <= st_lookup;
                   state <= st_nop2;
                status_w_reset <= 1'b1;
                arp_table_we <= 1'b0;
                write_address_counter_ce <= 1'b0;
                read_address_counter_ce <= 1'b0;
                read_address_counter_reset <= 1'b1;
                read_address_counter_up <= 1'b1;
                status_r_int <= 2'b00;
             end
             st_lookup : begin
                if (matching_ip)
                   state <= st_found;
                else if (read_address_counter_loop)
                   state <= st_nfound;
                else
                   state <= st_lookup;
                status_w_reset <= 1'b1;
                arp_table_we <= 1'b0;
```

```
286                 write_address_counter_ce <= 1'b0;
                    read_address_counter_ce <= 1'b1;
288                 read_address_counter_reset <= 1'b0;
                    read_address_counter_up <= 1'b1;
290                 status_r_int <= 2'b00;
                end
292             st_found : begin
                    status_w_reset <= 1'b0;
294                 arp_table_we <= 1'b0;
                    write_address_counter_ce <= 1'b0;
296                 read_address_counter_ce <= 1'b1;
                    read_address_counter_reset <= 1'b0;
298                 read_address_counter_up <= 1'b0;
                    status_r_int <= 2'b01;
300                 state <= st_idle;
                end
302             st_nfound : begin
                    status_w_reset <= 1'b0;
304                 arp_table_we <= 1'b0;
                    write_address_counter_ce <= 1'b0;
306                 read_address_counter_ce <= 1'b0;
                    read_address_counter_reset <= 1'b0;
308                 read_address_counter_up <= 1'b1;
                    status_r_int <= 2'b10;
310                 state <= st_idle;
                end
312             st_nop : begin
                    status_w_reset <= 1'b0;
314                 arp_table_we <= 1'b0;
                    write_address_counter_ce <= 1'b0;
316                 read_address_counter_ce <= 1'b0;
                    read_address_counter_reset <= 1'b0;
318                 //read_address_counter_up <= 1'b1;
                    read_address_counter_up <= 1'b0;
320                 state <= st_idle;
                end
322             st_nop2 : begin
                    status_w_reset <= 1'b0;
324                 arp_table_we <= 1'b0;
                    write_address_counter_ce <= 1'b0;
326                 read_address_counter_ce <= 1'b0;
                    read_address_counter_reset <= 1'b0;
328                 read_address_counter_up <= 1'b0;
                    status_r_int <= 2'b00;
330                 state <= st_nop3;
                end
332             st_nop3 : begin
                    status_w_reset <= 1'b0;
334                 arp_table_we <= 1'b0;
                    write_address_counter_ce <= 1'b0;
336                 read_address_counter_ce <= 1'b0;
                    read_address_counter_reset <= 1'b0;
338                 read_address_counter_up <= 1'b0;
                    status_r_int <= 2'b00;
340                 state <= st_lookup;
                end
342
            endcase
344
endmodule
```

Listing 6.10: pip.v

```
1  ////////////////////////////////////////////////////////////////////////////////
   // Copyright 2009 Michel Bieleveld
3  //
   // This file is part of the UDP/IP stack.
5  //
   // The UDP/IP stack is free software: you can redistribute it and/or modify
7  // it under the terms of the GNU Lesser General Public License as published by
   // the Free Software Foundation, either version 3 of the License, or
9  // (at your option) any later version.
   //
11 // The UDP/IP stack is distributed in the hope that it will be useful,
   // but WITHOUT ANY WARRANTY; without even the implied warranty of
13 // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   // GNU Lesser General Public License for more details.
15 //
   // You should have received a copy of the GNU Lesser General Public License
17 // along with the UDP/IP stack.  If not, see <http://www.gnu.org/licenses/>.
   //
19 ////////////////////////////////////////////////////////////////////////////////

21 `timescale 1ns / 1ps
   module pip(
```

```
23        input             clk_50mhz,
          input             LOCKED_OUT,
25        input             E_RX_CLK,
          input             E_TX_CLK,
27        output            E_TX_ERR,
          output            E_TX_EN,
29        output  [7:0]     E_TXD,
          input             E_RX_ERR,
31        input             E_RX_DV,
          input   [7:0]     E_RXD,
33        input             E_CRS,
          input             E_COL,
35        inout             E_MDIO,
          output            E_MDC,
37
          // DCM
39        output            reset,
          input             reset_app,
41
          // Application Interface tx
43        output [15:0]     pip_tx_address,
          input  [7:0]      pip_tx_data,
45
          output [15:0]     pip_rx_address,
47        output [7:0]      pip_rx_data,
          output            pip_rx_we,
49
          output            pip_new_frame,
51
          input  [7:0]      pip_command_data,
53        input  [7:0]      pip_command_address,
          input             pip_command_wr_en,
55
          input             pip_result_rd_en,
57        input  [7:0]      pip_result_address,
          output reg [7:0]     pip_result_data
59
      );
61
    // wires for picoblaze
63  reg  [7:0]      out_port;
    reg  [7:0]      port_id;
65
    wire [17:0]     instruction ;
67  wire [9:0]      address;
    wire [7:0]      port_id_int;
69  wire [7:0]      out_port_int;
    reg  [7:0]      in_port = 8'b00000000;
71  wire            proc_reset ;
    wire            read_strobe;
73  wire            write_strobe;

75  reg [7:0]       input_mac;
    reg [7:0]       input_registers;
77
    // copy
79  wire [7:0]      copy_data_out;
    wire [15:0]     copy_to_address;
81  wire            copy_to_we;
    wire            copy_last_byte;
83
    wire [7:0]      copy_data_out_rx;
85  wire [15:0]     copy_from_address_rx;

87  // crc
    wire [7:0]      crc_data_out;
89  wire [15:0]     crc_address;
    wire            wr_crc;
91  reg             cs_crc;

93  // arp
    reg             cs_copy;
95  wire            wr_copy;
    reg             cs_copy_rx;
97  wire            wr_copy_rx;

99
    // command interface
101 reg  [7:0]      pip_command_data_out;
    wire            pip_command_rd_en;
103 reg             pip_command_cs;
    reg             pip_result_cs;
105
    // wires for arp module
107 reg             cs_arp;
    wire            wr_arp;
109 wire [7:0]      arp_out_port;
```

```
111     // reset for picoblaze when dcm is ready and/or jtag memory is programmed
        assign          proc_reset = reset | !LOCKED_OUT | reset_app;
113
        // mac signals
115
        // mac_sfr
117     // 7 Mac reset
        // 6 READBUFFER  (0 = Read RX buffer, 1 = Read TX buffer)
119     // 5 Send (sends packet)
        // 4 EOP (End of packet)
121     // 3 addr[10]
        // 2 addr[9]
123     // 1 addr[8]
        // 0 addr[7]
125
        reg [10:0]      mac_address;
127     wire [7:0]      mac_status_port;
        reg [7:0]       tx_mac_data;
129     wire [7:0]      copy_to_data;
        wire [7:0]      Tx_mac_data_rd;
131     wire [7:0]      Rx_mac_data;
        wire [7:0]      Rx_mac_packet_status;
133     wire            Rx_mac_packet_ready;
        wire            Tx_mac_packet_ready;
135     wire            Tx_mac_top_half;
        wire            Tx_mac_eop_bit;
137     wire            mac_reset;
        wire            Tx_mac_send;
139     reg             Tx_mac_write;
        wire            MAC_Host_CSB;
141

143     reg             Tx_mac_eop_bit_d;
        reg   [7:0]     mac_sfr = 8'b00000000;
145     reg   [7:0]     mac_sfr2 = 8'b00000000;
        reg   [7:0]     module_sfr  = 8'b00000000;
147     reg             cs_rx_mac;

149     assign          Tx_mac_top_half = mac_sfr2[0];
        assign          pip_new_frame = mac_sfr2[1];
151
        assign          mac_reset = mac_sfr[7];
153     assign          Tx_mac_send = mac_sfr[5];
        assign          Tx_mac_is_last_byte = mac_sfr[4] | copy_last_byte;
155
//  wire [35:0] CONTROL0;
157 //
//  scope instance_scope (
159 //      .CONTROL0(CONTROL0)
//    );
161 //
//  scope_ila instance_ila (
163 //      .CONTROL(CONTROL0),
//      .CLK(clk_50mhz),
165 //      .TRIG0(address)
//    );
167
    always @(posedge clk_50mhz)
169     if (read_strobe & !port_id_int[7])
            Tx_mac_eop_bit_d <= Tx_mac_eop_bit;
171
        assign          MAC_Host_CSB = 1'b1;
173
        assign          mac_status_port = {Tx_mac_packet_ready, Tx_mac_eop_bit_d,5'b00000,Rx_mac_packet_ready};
175
    always @(module_sfr, port_id_int[7], write_strobe, copy_to_we)
177     case (module_sfr[1:0])
            2'b00: Tx_mac_write = (!port_id_int[7] & write_strobe);
179         2'b01: Tx_mac_write = copy_to_we;
            2'b10: Tx_mac_write = 1'b0;
181         2'b11: Tx_mac_write = 1'b0;
        endcase
183
    always @(module_sfr, out_port_int, copy_to_data)
185     case (module_sfr[1:0])
            2'b00: tx_mac_data = out_port_int;
187         2'b01: tx_mac_data = copy_to_data;
            2'b10: tx_mac_data = out_port_int;
189         2'b11: tx_mac_data = out_port_int;
        endcase
191
    always @(module_sfr, mac_sfr[3:0], port_id_int[6:0], copy_to_address,crc_address,copy_from_address_rx)
193     case (module_sfr[1:0])
            2'b00: mac_address = {mac_sfr[3:0], port_id_int[6:0]};
195         2'b01: mac_address = copy_to_address[10:0];
            2'b10: mac_address = crc_address[10:0];
```

```
197           2'b11: mac_address = copy_from_address_rx[10:0];
        endcase
199
    MAC_top instance_mac (
201   .Reset(mac_reset), //
      .Clk_125M(),
203   .Clk_user(clk_50mhz), //
      .Clk_reg(clk_50mhz),  //
205   .Clk_mem(clk_50mhz), //
      .Speed(Speed),
207   .Rx_mac_packet_ready(Rx_mac_packet_ready), //
      .Rx_mac_get_packet(Rx_mac_get_packet),
209   .Rx_mac_addr(mac_address), //
      .Rx_mac_data(Rx_mac_data), //
211   .Rx_mac_packet_status(Rx_mac_packet_status), //
      .Tx_mac_addr(mac_address), //
213   .Tx_mac_data(tx_mac_data), //
      .Tx_mac_data_rd(Tx_mac_data_rd), //
215   .Tx_mac_write(Tx_mac_write), //
      .Tx_mac_send(Tx_mac_send), //
217   .Tx_mac_is_last_byte(Tx_mac_is_last_byte), //
      .Tx_mac_top_half(Tx_mac_top_half),
219   .Tx_mac_packet_ready(Tx_mac_packet_ready), //
      .TX_mac_eop_bit(Tx_mac_eop_bit), //
221   .Gtx_clk(Gtx_clk),
      .Rx_clk(E_RX_CLK), //
223   .Tx_clk(E_TX_CLK), //
      .Tx_er(E_TX_ERR), //
225   .Tx_en(E_TX_EN), //
      .Txd(E_TXD),  //
227   .Rx_er(E_RX_ERR), //
      .Rx_dv(E_RX_DV), //
229   .Rxd(E_RXD), //
      .Crs(E_CRS), //
231   .Col(E_COL), //
      .CSB(MAC_Host_CSB), //
233   .WRB(), //
      .CD_in(), //
235   .CD_out(), //
      .CA(), //
237   .Mdio(E_MDIO), //
      .Mdc(E_MDC) //
239   );

241   reg [7:0] command_arguments [15:0];

243   always @(posedge clk_50mhz)
        if (pip_command_wr_en)
245       command_arguments[pip_command_address[3:0]] <= pip_command_data;

247   //assign pip_command_data_out = command_arguments[port_id_int[3:0]];
      always @(posedge clk_50mhz)
249     pip_command_data_out <= command_arguments[port_id_int[3:0]];

251
      reg [7:0] result_arguments [15:0];
253
      always @(posedge clk_50mhz)
255     if (write_strobe & (port_id_int[7:4] == 4'b1001))
          result_arguments[port_id_int[3:0]] <= out_port_int;
257
      always @(posedge clk_50mhz)
259     pip_result_data <= result_arguments[pip_result_address[3:0]];

261   //assign pip_result_data = result_arguments[pip_result_address[3:0]];

263


265   app picoblaze_rom (
      .address(address),
267   .instruction(instruction),
      //.proc_reset(reset),
269   .clk(clk_50mhz)
      );
271


273
    module_arp instance_arp (
275   .clk(clk_50mhz),
      .port_id(port_id_int),
277   .in_port(out_port_int),
      .write_strobe(wr_arp),
279   .out_port(arp_out_port)
    );
281
    copy instance_copy_tx (
283   .clk(clk_50mhz),
```

```
    .copy_from_address(pip_tx_address),
    .copy_from_data(pip_tx_data),
    .copy_to_address(copy_to_address),
    .copy_to_we(copy_to_we),
    .copy_to_data(copy_to_data),
    .last_byte(copy_last_byte),
    .reg_address(port_id_int),
    .reg_we(wr_copy),
    .reg_data_in(out_port_int),
    .reg_data_out(copy_data_out)
  );

  copy_rx instance_copy_rx (
    .clk(clk_50mhz),
    .copy_from_address(copy_from_address_rx),
    .copy_from_data(Rx_mac_data),
    .copy_to_address(pip_rx_address),
    .copy_to_we(pip_rx_we),
    .copy_to_data(pip_rx_data),
    .last_byte(),
    .reg_address(port_id_int),
    .reg_we(wr_copy_rx),
    .reg_data_in(out_port_int),
    .reg_data_out(copy_data_out_rx)
  );


  crc_8bit inst_crc_tx (
    .clk(clk_50mhz),
    .address(crc_address),
    .data_in(Tx_mac_data_rd),
    .reg_address(port_id),  // port_id_int
    .reg_data_in(out_port_int),
    .reg_we(wr_crc),
    .reg_data_out(crc_data_out)
  );

  /*
  address             write               read
  0 XXXXXXX           rx_buffer           tx_buffer       mac_sfr[6] = 0
  0 XXXXXXX           tx_buffer           tx_buffer       mac_sfr[6] = 1

  1 0010 000  90      result_data         command_data
  1 0010 001  91      result_arg          command_arg
  1 0010 010  92      result_arg          command_arg
  .
  .
  1 0011 111  9F      result_arg          command_arg


  1 0100 000  A0      start_copy_rx       status
  1 0100 001  A1      from_address_l      status
  1 0100 010  A2      from_address_h      status
  1 0100 011  A3      length_l            status
  1 0100 100  A4      length_h            status
  1 0100 101  A5      to_address_l        status
  1 0100 110  A6      to_address_h        status

  1 0110 000  B0      start_crc           status
  1 0110 001  B1      address_l           crc_l
  1 0110 010  B2      address_h           crc_h
  1 0110 011  B3      length_l            0
  1 0110 100  B4      length_h            0

  1 0111 000  B8      module_sfr          module_sfr
     0 mac
     1 copy
     2 crc

  1 1000 000  C0      start_copy          status
  1 1000 001  C1      from_address_l      status
  1 1000 010  C2      from_address_h      status
  1 1000 011  C3      length_l            status
  1 1000 100  C4      length_h            status
  1 1000 101  C5      to_address_l        status
  1 1000 110  C6      to_address_h        status

  1 1001 000  C8      mac_status_port

  1 1010 000  D0      Rx_mac_packet_status  Rx_mac_get_packet
  // Rx_mac_packet_status = {Rx_mac_eop, Rx_mac_sop, Rx_mac_get_packet, Rx_mac_ra, state[3:0]};

  1 1011 000  D8      mac_sfr             mac_sfr

  1 110 0000  E0      arp_status          arp_status
  1 110 0001  E1      arp_eth0            arp_eth0
```

```
371     1 110 0010  E2     arp_eth1            arp_eth1
        1 110 0011  E3     arp_eth2            arp_eth2
373     1 110 0100  E4     arp_eth3            arp_eth3
        1 110 0101  E5     arp_eth4            arp_eth4
375     1 110 0110  E6     arp_eth5            arp_eth5
        1 110 0111  E7                         arp_ip0
377     1 110 1000  E8                         arp_ip1
        1 110 1001  E9                         arp_ip2
379     1 110 1010  EA                         arp_ip3

381     1 1110 000  F0
        1 1111 000  F8    mac_sfr2             mac_sfr2
383

385     */

387


389
        kcpsm3 picoblaze (
391         .address(address),
            .instruction(instruction),
393         .port_id(port_id_int),
            .write_strobe(write_strobe),
395         .out_port(out_port_int),
            .read_strobe(read_strobe),
397         .in_port(in_port),
            .interrupt(Tx_mac_packet_ready),
399         .interrupt_ack(interrupt_ack),
            .reset(proc_reset),
401         .clk(clk_50mhz)
        );
403
        // pipeline output decoding logic
405
        always @(posedge clk_50mhz) begin
407         out_port <= out_port_int;
            port_id  <= port_id_int;
409     end

411     // for copy
        always @(posedge clk_50mhz) begin
413         if (port_id_int[7:3] == 5'b11000) begin
                cs_copy <= 1'b1;
415         end else begin
                cs_copy <= 1'b0;
417         end
        end
419
        assign wr_copy = cs_copy & write_strobe;
421
        // for copy_rx
423     always @(posedge clk_50mhz) begin
            if (port_id_int[7:3] == 5'b10100) begin
425             cs_copy_rx <= 1'b1;
            end else begin
427             cs_copy_rx <= 1'b0;
            end
429     end

431     assign wr_copy_rx = cs_copy_rx & write_strobe;

433
        // for crc
435     always @(posedge clk_50mhz) begin
            if (port_id_int[7:3] == 5'b10110) begin
437             cs_crc <= 1'b1;
            end else begin
439             cs_crc <= 1'b0;
            end
441     end

443     assign wr_crc = cs_crc & write_strobe;

445     // for result interface
        always @(posedge clk_50mhz) begin
447       if (port_id_int[7:4] == 4'b1001) begin
              pip_result_cs <= 1'b1;
449       end else begin
              pip_result_cs <= 1'b0;
451       end
        end
453
        assign pip_result_wr_en = pip_result_cs & write_strobe;
455
        // for command interface
457     always @(posedge clk_50mhz) begin
```

```
        if (port_id_int[7:4] == 4'b1001) begin
459         pip_command_cs <= 1'b1;
        end else begin
461         pip_command_cs <= 1'b0;
        end
463 end

465 assign pip_command_rd_en = pip_command_cs & read_strobe;

467 // for arp
    always @(posedge clk_50mhz) begin
469     if (port_id_int[7:4] == 4'b1110) begin
            cs_arp <= 1'b1;
471     end else begin
            cs_arp <= 1'b0;
473     end
    end
475
    assign wr_arp = cs_arp & write_strobe;
477
    // for mac_sfr
479 always @(posedge clk_50mhz) begin
        if ((port_id_int[7:3] == 5'b11011) & write_strobe) begin
481         mac_sfr <= out_port;
        end
483 end

485 // for mac_sfr2
    always @(posedge clk_50mhz) begin
487     if ((port_id_int[7:3] == 5'b11111) & write_strobe) begin
            mac_sfr2 <= out_port;
489     end
    end
491

493 // for module_sfr
    always @(posedge clk_50mhz) begin
495     if ((port_id_int[7:3] == 5'b10111) & write_strobe) begin
            module_sfr <= out_port;
497     end
    end
499

501 // to get next packet
    always @(posedge clk_50mhz) begin
503     if (port_id_int[7:3] == 5'b11010) begin
            cs_rx_mac <= 1'b1;
505     end else begin
            cs_rx_mac <= 1'b0;
507     end
    end
509
    assign Rx_mac_get_packet     = cs_rx_mac & write_strobe;
511
// pipeline input decoding logic
513
    //always @(posedge clk_50mhz)
515 //always @(port_id_int[7], mac_sfr[6], Rx_mac_data, Tx_mac_data_rd, input_registers)
    //    case ({port_id_int[7], mac_sfr[6]})
517   always @(port_id[7], mac_sfr[6], Rx_mac_data, Tx_mac_data_rd, input_registers)
        case ({port_id[7], mac_sfr[6]})
519       2'b00: in_port = Rx_mac_data;
          2'b01: in_port = Tx_mac_data_rd;
521       2'b10: in_port = input_registers;
          2'b11: in_port = input_registers;
523     endcase

525
    //always @(port_id_int[6:3], mac_status_port, Rx_mac_packet_status, mac_sfr, arp_out_port, module_sfr, crc_data_out, copy_
527 //    case (port_id_int[6:3])
    always @(port_id[6:3], mac_status_port, Rx_mac_packet_status, mac_sfr, arp_out_port, module_sfr, crc_data_out, copy_data_o
529     case (port_id[6:3])
          4'b0000: input_registers = 8'b00000000;
531       4'b0001: input_registers = 8'b00000000;
          4'b0010: input_registers = pip_command_data_out;
533       4'b0011: input_registers = pip_command_data_out;
          4'b0100: input_registers = copy_data_out_rx;
535       4'b0101: input_registers = 8'b00000000;
          4'b0110: input_registers = crc_data_out;
537       4'b0111: input_registers = module_sfr;
          4'b1000: input_registers = copy_data_out;
539       4'b1001: input_registers = mac_status_port;
          4'b1010: input_registers = Rx_mac_packet_status;
541       4'b1011: input_registers = mac_sfr;
          4'b1100: input_registers = arp_out_port;
543       4'b1101: input_registers = arp_out_port;
          4'b1110: input_registers = 8'b00000000;
```

```
545             4'b1111: input_registers = mac_sfr2;
        endcase
547

549 endmodule
```

Listing 6.11: rx_packet.v

```
1  `timescale 1ns / 1ps
   //////////////////////////////////////////////////////////////////////////////////
3  // Company:
   // Engineer:
5  //
   // Create Date:     10:33:28 06/14/2007
7  // Design Name:
   // Module Name:     rx_packet
9  // Project Name:
   // Target Devices:
11 // Tool versions:
   // Description:
13 //
   // Dependencies:
15 //
   // Revision:
17 // Revision 0.01 - File Created
   // Additional Comments:
19 //
   //////////////////////////////////////////////////////////////////////////////////
21 module rx_packet(
   input            clk,
23 input            Clk_mem,
   input            reset,
25                 // Internal interface
   input            Rx_mac_ra               ,
27 output reg       Rx_mac_rd               ,
   input    [31:0]  Rx_mac_data_32          ,
29 input    [1:0]   Rx_mac_BE               ,
   input            Rx_mac_pa               ,
31 input            Rx_mac_sop              ,
   input            Rx_mac_eop              ,
33                 // User interface
   output reg        Rx_mac_packet_ready    ,
35 output [7:0] Rx_mac_packet_status        ,
   input             Rx_mac_get_packet      ,
37 input    [10:0]   Rx_mac_addr            ,
   output   [7:0]   Rx_mac_data
39 );

41 // Counter for addrb
   reg [8:0] counter_addrb = 0;
43 wire      counter_addrb_reset;
   wire      counter_addrb_ce;
45 wire [31:0] Rx_mac_data_32_c;
   assign Rx_mac_data_32_c = {Rx_mac_data_32[7:0],Rx_mac_data_32[15:8],Rx_mac_data_32[23:16],Rx_mac_data_32[31:24
47




51


53 // RAMB16_S9_S36: Virtex-II/II-Pro, Spartan-3/3E 2k/512 x 8/32 + 1/4 Parity bits Parity bits Dual-Port RAM
   // Xilinx HDL Language Template, version 9.1i
55
   RAMB16_S9_S36 #(
57    .INIT_A(9'h000),          // Value of output RAM registers on Port A at startup
      .INIT_B(36'h000000000),   // Value of output RAM registers on Port B at startup
59    .SRVAL_A(9'h000),         // Port A output value upon SSR assertion
      .SRVAL_B(36'h000000000), // Port B output value upon SSR assertion
61    .WRITE_MODE_A("WRITE_FIRST"), // WRITE_FIRST, READ_FIRST or NO_CHANGE
      .WRITE_MODE_B("WRITE_FIRST"), // WRITE_FIRST, READ_FIRST or NO_CHANGE
63    .SIM_COLLISION_CHECK("ALL"),  // "NONE", "WARNING_ONLY", "GENERATE_X_ONLY", "ALL"

65    // The following INIT_xx declarations specify the initial contents of the RAM
      // Port A Address 0 to 511, Port B Address 0 to 127
67    .INIT_00(256'h0000000000000000000000000000000000000000000000000000000000BBAA),
      .INIT_01(256'h0000000000000000000000000000000000000000000000000000000000000000),
69    .INIT_02(256'h0000000000000000000000000000000000000000000000000000000000000000),
      .INIT_03(256'h0000000000000000000000000000000000000000000000000000000000000000),
71    .INIT_04(256'h0000000000000000000000000000000000000000000000000000000000000000),
      .INIT_05(256'h0000000000000000000000000000000000000000000000000000000000000000),
73    .INIT_06(256'h0000000000000000000000000000000000000000000000000000000000000000),
      .INIT_07(256'h0000000000000000000000000000000000000000000000000000000000000000),
75    .INIT_08(256'h0000000000000000000000000000000000000000000000000000000000000000),
      .INIT_09(256'h0000000000000000000000000000000000000000000000000000000000000000),
77    .INIT_0A(256'h0000000000000000000000000000000000000000000000000000000000000000),
```

```
    .INIT_0B(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_0C(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_0D(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_0E(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_0F(256'h0000000000000000000000000000000000000000000000000000000000000000),
    // Port A Address 512 to 1023, Port B Address 128 to 255
    .INIT_10(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_11(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_12(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_13(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_14(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_15(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_16(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_17(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_18(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_19(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_1A(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_1B(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_1C(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_1D(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_1E(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_1F(256'h0000000000000000000000000000000000000000000000000000000000000000),
    // Port A Address 1024 to 1535, Port B Address 255 to 383
    .INIT_20(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_21(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_22(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_23(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_24(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_25(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_26(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_27(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_28(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_29(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_2A(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_2B(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_2C(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_2D(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_2E(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_2F(256'h0000000000000000000000000000000000000000000000000000000000000000),
    // Port A Address 1536 to 2047, Port B Address 384 to 511
    .INIT_30(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_31(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_32(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_33(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_34(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_35(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_36(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_37(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_38(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_39(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_3A(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_3B(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_3C(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_3D(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_3E(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INIT_3F(256'h0000000000000000000000000000000000000000000000000000000000000000),

    // The next set of INITP_xx are for the parity bits
    // Port A Address 0 to 511, Port B Address 0 to 127
    .INITP_00(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INITP_01(256'h0000000000000000000000000000000000000000000000000000000000000000),
    // Port A Address 512 to 1023, Port B Address 128 to 255
    .INITP_02(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INITP_03(256'h0000000000000000000000000000000000000000000000000000000000000000),
    // Port A Address 1024 to 1535, Port B Address 256 to 383
    .INITP_04(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INITP_05(256'h0000000000000000000000000000000000000000000000000000000000000000),
    // Port A Address 1536 to 2047, Port B Address 384 to 511
    .INITP_06(256'h0000000000000000000000000000000000000000000000000000000000000000),
    .INITP_07(256'h0000000000000000000000000000000000000000000000000000000000000000)
) packet_buffer_inst (
    .DOA(Rx_mac_data),      // Port A 8-bit Data Output
    .DOB(),  // Port B 32-bit Data Output
    .DOPA(),                // Port A 1-bit Parity Output
    .DOPB(),                // Port B 4-bit Parity Output
    .ADDRA(Rx_mac_addr),    // Port A 11-bit Address Input
    .ADDRB(counter_addrb), // Port B 9-bit Address Input
    .CLKA(clk),             // Port A Clock
    .CLKB(clk),             // Port B Clock
    .DIA(8'b00000000),      // Port A 8-bit Data Input
    .DIB(Rx_mac_data_32_c),  // Port B 32-bit Data Input
    .DIPA(1'b0),            // Port A 1-bit parity Input
    .DIPB(4'b0000),         // Port-B 4-bit parity Input
    .ENA(1'b1),             // Port A RAM Enable Input
    .ENB(1'b1),             // Port B RAM Enable Input
    .SSRA(1'b0),            // Port A Synchronous Set/Reset Input
    .SSRB(1'b0),            // Port B Synchronous Set/Reset Input
```

```
165     .WEA(1'b0),                 // Port A Write Enable Input
        .WEB(web)                   // Port B Write Enable Input
167 );

169 // End of RAMB16_S9_S36_inst instantiation

171
    always @(posedge clk)
173     if (counter_addrb_reset)
            counter_addrb <= 0;
175     else if (counter_addrb_ce)
            counter_addrb <= counter_addrb + 1;
177
    //
179 assign web                 = Rx_mac_pa;
    assign counter_addrb_ce    = Rx_mac_pa;
181 assign counter_addrb_reset = Rx_mac_eop;

183 parameter st_wait      = 6'b000001;
    parameter st_startup   = 6'b000010;
185 parameter st_read      = 6'b000100;
    parameter st_fetch     = 6'b001000;
187 parameter st_finished  = 6'b010000;
    parameter st_newpacket = 6'b100000;
189


191


193  (* FSM_ENCODING="ONE-HOT", SAFE_IMPLEMENTATION="NO" *) reg [5:0] state = st_wait;

195 assign Rx_mac_packet_status = {Rx_mac_eop,Rx_mac_sop,Rx_mac_get_packet,Rx_mac_ra,state[3:0]};

197 always@(posedge clk)
        (* FULL_CASE, PARALLEL_CASE *) case (state)
199         st_wait : begin
                if (Rx_mac_get_packet)
201                 state <= st_startup;
                else
203                 state <= st_wait;
                Rx_mac_packet_ready <= 1'b0;
205             Rx_mac_rd           <= 1'b0;
            end
207         st_startup : begin
                if (Rx_mac_ra)
209                 state <= st_read;
                else
211                 state <= st_startup;
                Rx_mac_packet_ready <= 1'b0;
213             Rx_mac_rd           <= 1'b0;
            end
215         st_read : begin
                if (Rx_mac_eop)
217                 state <= st_finished;
                else if (!Rx_mac_ra)
219                 state <= st_startup;
                else
221                 state <= st_read;
                Rx_mac_packet_ready <= 1'b0;
223             Rx_mac_rd           <= 1'b1;
            end
225         st_finished : begin
                if (Rx_mac_get_packet & Rx_mac_eop)
227                 state <= st_newpacket;
                else if (Rx_mac_get_packet)
229                 state <= st_startup;
                else
231                 state <= st_finished;
                Rx_mac_packet_ready <= 1'b1;
233             Rx_mac_rd           <= 1'b0;
            end
235         st_newpacket : begin
                if (!Rx_mac_ra)
237                 state <= st_startup;
                else
239                 state <= st_read;
                Rx_mac_packet_ready <= 1'b0;
241             Rx_mac_rd           <= 1'b1;
            end
243     endcase

245

247 endmodule
```

Listing 6.12: top.v

```
1  //////////////////////////////////////////////////////////////////////////////////
   // Copyright 2009 Michel Bieleveld (michel@zifnab.com)
3  //
   // This file is part of the UDP/IP stack.
5  //
   // The UDP/IP stack is free software: you can redistribute it and/or modify
7  // it under the terms of the GNU Lesser General Public License as published by
   // the Free Software Foundation, either version 3 of the License, or
9  // (at your option) any later version.
   //
11 // The UDP/IP stack is distributed in the hope that it will be useful,
   // but WITHOUT ANY WARRANTY; without even the implied warranty of
13 // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   // GNU Lesser General Public License for more details.
15 //
   // You should have received a copy of the GNU Lesser General Public License
17 // along with the UDP/IP stack.  If not, see <http://www.gnu.org/licenses/>.
   //
19 //////////////////////////////////////////////////////////////////////////////////

21 `timescale 1ns / 1ps
   module top(
23    input            clk_50mhz_outside,
      output           RS232_DTE_TXD,
25    input            RS232_DTE_RXD,
      input            E_RX_CLK,
27    input            E_TX_CLK,
      output           E_TX_ERR,
29    output           E_TX_EN,
      output  [7:0]    E_TXD,
31    input            E_RX_ERR,
      input            E_RX_DV,
33    input   [7:0]    E_RXD,
      input            E_CRS,
35    input            E_COL,
      inout            E_MDIO,
37    output           E_MDC,
      output           strataflash_oe,
39    output           strataflash_ce,
      output           strataflash_we,
41    output           lcd_rs,
      output           lcd_rw,
43    output           lcd_e,
      inout [7:4]      lcd_d
45

47    );

49    assign strataflash_oe = 1'b1;
      assign strataflash_ce = 1'b1;
51    assign strataflash_we = 1'b1;

53
      wire [3:0] lcd_output_data, lcd_input_data;
55    wire       lcd_rw_control, lcd_drive;

57    assign lcd_d[7:4] = (~lcd_rw_control & lcd_drive) ? lcd_output_data : 4'hz;
      assign lcd_rw     = (lcd_rw_control & lcd_drive);
59    assign lcd_input_data = lcd_d[7:4];

61
      // global clocks
63    wire            clk_50mhz;
      wire           LOCKED_OUT;
65    wire            reset;

67    // pip interface
      wire  [15:0]    pip_tx_address;
69    wire  [7:0]     pip_tx_data;
      wire  [15:0]    pip_rx_address;
71    wire  [7:0]     pip_rx_data;
      wire  [7:0]     pip_command_data;
73    wire  [7:0]     pip_command_address;
      wire  [7:0]     pip_result_address;
75    wire  [7:0]     pip_result_data;

77
      dcma instance_dcm (
79     .CLKIN_IN(clk_50mhz_outside),
       .RST_IN(reset),
81     .CLKFX_OUT(clk_50mhz),
       .CLKIN_IBUFG_OUT(CLKIN_IBUFG_OUT),
83     .CLK0_OUT(),
       .CLK2X_OUT(),
85     .LOCKED_OUT(LOCKED_OUT)
```

```
    );
87
   // Instantiate the module
89   pip instance_pip (
     .clk_50mhz(clk_50mhz),
91   .LOCKED_OUT(LOCKED_OUT),
     .E_RX_CLK(E_RX_CLK),
93   .E_TX_CLK(E_TX_CLK),
     .E_TX_ERR(E_TX_ERR),
95   .E_TX_EN(E_TX_EN),
     .E_TXD(E_TXD),
97   .E_RX_ERR(E_RX_ERR),
     .E_RX_DV(E_RX_DV),
99   .E_RXD(E_RXD),
     .E_CRS(E_CRS),
101  .E_COL(E_COL),
     .E_MDIO(E_MDIO),
103  .E_MDC(E_MDC),

105  // DCM
     .reset(reset),
107  .reset_app(reset_app),

109  // Application Interface
     .pip_new_frame(pip_new_frame),
111  .pip_tx_address(pip_tx_address),
     .pip_tx_data(pip_tx_data),
113  .pip_rx_address(pip_rx_address),
     .pip_rx_data(pip_rx_data),
115  .pip_rx_we(pip_rx_we),

117  .pip_command_data(pip_command_data),
     .pip_command_wr_en(pip_command_wr_en),
119  .pip_command_address(pip_command_address),

121  .pip_result_rd_en(pip_result_rd_en),
     .pip_result_address(pip_result_address),
123  .pip_result_data(pip_result_data)
     );
125
     application instance_app (
127  .pip_address(pip_tx_address[10:0]),
     .pip_data(pip_tx_data),
129  .pip_address_rx(pip_rx_address[10:0]),
     .pip_data_rx(pip_rx_data),
131  .pip_wr_rx(pip_rx_we),
     .pip_enable(pip_tx_address[15]),
133  .pip_new_frame(pip_new_frame),
     .pip_command_data(pip_command_data),
135  .pip_command_wr_en(pip_command_wr_en),
     .pip_command_address(pip_command_address),
137  .pip_result_rd_en(pip_result_rd_en),
     .pip_result_address(pip_result_address),
139  .pip_result_data(pip_result_data),
     .clk_50mhz(clk_50mhz),
141  .RS232_DTE_TXD(RS232_DTE_TXD),
     .RS232_DTE_RXD(RS232_DTE_RXD),
143  .reset(reset),
     .reset_app(reset_app),
145  .lcd_rw_control(lcd_rw_control),
     .lcd_drive(lcd_drive),
147  .lcd_output_data(lcd_output_data),
     .lcd_input_data(lcd_input_data),
149  .lcd_rs(lcd_rs),
     .lcd_e(lcd_e)
151  );

153
endmodule
```

Listing 6.13: top_tf.v

```
   `timescale 1ns / 1ps
2
   //////////////////////////////////////////////////////////////////////////////////
4  // Company:
   // Engineer:
6  //
   // Create Date:    15:58:34 02/15/2009
8  // Design Name:    top
   // Module Name:    C:/NoSpace/final_from_scratch/thesis/top_tf.v
10 // Project Name:   thesis
   // Target Device:
12 // Tool versions:
   // Description:
```

```
14 //
   // Verilog Test Fixture created by ISE for module: top
16 //
   // Dependencies:
18 //
   // Revision:
20 // Revision 0.01 - File Created
   // Additional Comments:
22 //
   ////////////////////////////////////////////////////////////////////////////////
24
   module top_tf;
26
      // Inputs
28    reg clk_50mhz_outside;
      reg RS232_DCE_RXD;
30    reg RS232_DTE_RXD;

32    // Outputs
      wire RS232_DCE_TXD;
34    wire RS232_DTE_TXD;

36    // Instantiate the Unit Under Test (UUT)
      top uut (
38       .clk_50mhz_outside(clk_50mhz_outside),
         .RS232_DCE_TXD(RS232_DCE_TXD),
40       .RS232_DCE_RXD(RS232_DCE_RXD),
         .RS232_DTE_TXD(RS232_DTE_TXD),
42       .RS232_DTE_RXD(RS232_DTE_RXD)
      );
44
      initial begin
46       clk_50mhz_outside = 0;
         forever #20 clk_50mhz_outside = ~clk_50mhz_outside;
48    end

50    initial begin
         // Initialize Inputs
52
         RS232_DCE_RXD = 0;
54       RS232_DTE_RXD = 0;

56       // Wait 100 ns for global reset to finish
         #100;
58
         // Add stimulus here
60
      end
62
   endmodule
```

Listing 6.14: tx_packet.v

```
   `timescale 1ns / 1ps
2  ////////////////////////////////////////////////////////////////////////////////
   // Company:
4  // Engineer:
   //
6  // Create Date:     10:45:06 07/16/2007
   // Design Name:
8  // Module Name:     tx_packet
   // Project Name:
10 // Target Devices:
   // Tool versions:
12 // Description:
   //
14 // Dependencies:
   //
16 // Revision:
   // Revision 0.01 - File Created
18 // Additional Comments:
   //
20 ////////////////////////////////////////////////////////////////////////////////
   module tx_packet(
22 input           clk,
   input           clk_mem,
24 input           reset,
                  //user interface
26 input  [10:0]  Tx_mac_addr,
   input  [7:0]   Tx_mac_data,
28 output [7:0]   Tx_mac_data_rd,
   output          TX_mac_eop_bit,
30 input           Tx_mac_write,
   input           Tx_mac_send,
32 input           Tx_mac_is_last_byte,
```

```
input           Tx_mac_top_half,
output reg      Tx_mac_packet_ready,
                //internal interface
input           Tx_mac_wa,
output reg      Tx_mac_wr,
output reg     [31:0] Tx_mac_data_int,
output reg     [1:0]  Tx_mac_BE,
output         Tx_mac_sop,
output reg     Tx_mac_eop
);

wire    [3:0]  eop_byte;
wire    [31:0] Dout;
reg            Tx_mac_sop_delay [2:0];
reg            Tx_mac_sop_delay_in = 0;
reg     [8:0]  mem_addr = 0;
reg            mem_addr_reset = 0;
reg            mem_addr_count = 0;
reg            Tx_mac_wr_ = 0;

reg            top_half_internal = 0;

always @(posedge clk)
   if (eop_byte[3] | eop_byte[2] | eop_byte[1] | eop_byte[0])
      top_half_internal <= 1'b0;
   else if (Tx_mac_top_half)
      top_half_internal <= 1'b1;

always @(posedge clk)
   if (mem_addr_reset)
      mem_addr <= {top_half_internal, 8'b00000000};
      //mem_addr <= 0;
   else if (mem_addr_count)
      mem_addr <= mem_addr + 1;

always @(posedge clk)
   Tx_mac_eop <= eop_byte[3] | eop_byte[2] | eop_byte[1] | eop_byte[0];

always @(posedge clk)
   case (eop_byte)
      4'b0001: Tx_mac_BE <= 2'b01;  // 01
      4'b0010: Tx_mac_BE <= 2'b10;  // 10
      4'b0100: Tx_mac_BE <= 2'b11;  // 11
      4'b1000: Tx_mac_BE <= 2'b00;  // 00
      default: Tx_mac_BE <= 2'b00;
   endcase

always @(posedge clk)
   Tx_mac_data_int <= {Dout[7:0],Dout[15:8],Dout[23:16],Dout[31:24]};


SRL16 #(
   .INIT(16'h0000)
) SRL16_inst (
   .Q(Tx_mac_sop),
   .A0(1'b0),
   .A1(1'b0),
   .A2(1'b1),
   .A3(1'b0),
   .CLK(clk),
   .D(Tx_mac_sop_delay_in)
);

parameter st_finish  = 8'b00000001;
parameter st_startup = 8'b00000010;
parameter st_initmem = 8'b00000100;
parameter st_send    = 8'b00001000;
parameter st_wait    = 8'b00010000;


(* FSM_ENCODING="ONE-HOT", SAFE_IMPLEMENTATION="NO" *) reg [7:0] state = st_finish;



always@(posedge clk)
      (* FULL_CASE, PARALLEL_CASE *) case (state)
         st_finish : begin
            if (Tx_mac_send)
               state <= st_startup;
            else
               state <= st_finish;
            Tx_mac_wr           <= 1'b0;
            Tx_mac_sop_delay_in <= 1'b1;
            Tx_mac_packet_ready <= 1'b1;
            mem_addr_reset      <= 1'b1;
            mem_addr_count      <= 1'b0;
```

```
120             end
                st_startup: begin
122                 state               <= st_initmem;
                    Tx_mac_wr           <= 1'b0;
124                 Tx_mac_sop_delay_in <= 1'b0;
                    Tx_mac_packet_ready <= 1'b0;
126                 mem_addr_reset      <= 1'b0;
                    mem_addr_count      <= 1'b0;
128             end
                st_initmem: begin
130                 state               <= st_send;
                    Tx_mac_wr           <= 1'b0;
132                 Tx_mac_sop_delay_in <= 1'b0;
                    Tx_mac_packet_ready <= 1'b0;
134                 mem_addr_reset      <= 1'b0;
                    mem_addr_count      <= 1'b0;
136             end
                st_send: begin
138                 if (eop_byte[0] | eop_byte[1] | eop_byte[2] | eop_byte[3])
                        state <= st_finish;
140                 else if (!Tx_mac_wa)
                        state <= st_wait;
142                 else
                        state <= st_startup;
144                 Tx_mac_wr           <= 1'b1;
                    Tx_mac_sop_delay_in <= 1'b0;
146                 Tx_mac_packet_ready <= 1'b0;
                    mem_addr_reset      <= 1'b0;
148                 mem_addr_count      <= 1'b1;
150             end
                st_wait: begin
                    if (Tx_mac_wa)
152                     state <= st_startup;
                    else
154                     state <= st_wait;
                    Tx_mac_wr           <= 1'b0;
156                 Tx_mac_sop_delay_in <= 1'b0;
                    Tx_mac_packet_ready <= 1'b0;
158                 mem_addr_reset      <= 1'b0;
                    mem_addr_count      <= 1'b0;
160             end
            endcase
162
    // RAMB16_S9_S36: Virtex-II/II-Pro, Spartan-3/3E 2k/512 x 8/32 + 1/4 Parity bits Parity bits Dual-Port RAM
164 // Xilinx HDL Language Template, version 9.1i

166 RAMB16_S9_S36 #(
        .INIT_A(9'h000),          // Value of output RAM registers on Port A at startup
168     .INIT_B(36'h000000000),   // Value of output RAM registers on Port B at startup
        .SRVAL_A(9'h000),         // Port A output value upon SSR assertion
170     .SRVAL_B(36'h000000000), // Port B output value upon SSR assertion
        .WRITE_MODE_A("WRITE_FIRST"), // WRITE_FIRST, READ_FIRST or NO_CHANGE
172     .WRITE_MODE_B("WRITE_FIRST"), // WRITE_FIRST, READ_FIRST or NO_CHANGE
        .SIM_COLLISION_CHECK("ALL"),  // "NONE", "WARNING_ONLY", "GENERATE_X_ONLY", "ALL"
174
        // The following INIT_xx declarations specify the initial contents of the RAM
176     // Port A Address 0 to 511, Port B Address 0 to 127
        .INIT_00(256'h0000000000000000000000000000000000000000000000000000000000000000),
178     .INIT_01(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_02(256'h0000000000000000000000000000000000000000000000000000000000000000),
180     .INIT_03(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_04(256'h0000000000000000000000000000000000000000000000000000000000000000),
182     .INIT_05(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_06(256'h0000000000000000000000000000000000000000000000000000000000000000),
184     .INIT_07(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_08(256'h0000000000000000000000000000000000000000000000000000000000000000),
186     .INIT_09(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_0A(256'h0000000000000000000000000000000000000000000000000000000000000000),
188     .INIT_0B(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_0C(256'h0000000000000000000000000000000000000000000000000000000000000000),
190     .INIT_0D(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_0E(256'h0000000000000000000000000000000000000000000000000000000000000000),
192     .INIT_0F(256'h0000000000000000000000000000000000000000000000000000000000000000),
        // Port A Address 512 to 1023, Port B Address 128 to 255
194     .INIT_10(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_11(256'h0000000000000000000000000000000000000000000000000000000000000000),
196     .INIT_12(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_13(256'h0000000000000000000000000000000000000000000000000000000000000000),
198     .INIT_14(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_15(256'h0000000000000000000000000000000000000000000000000000000000000000),
200     .INIT_16(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_17(256'h0000000000000000000000000000000000000000000000000000000000000000),
202     .INIT_18(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_19(256'h0000000000000000000000000000000000000000000000000000000000000000),
204     .INIT_1A(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_1B(256'h0000000000000000000000000000000000000000000000000000000000000000),
206     .INIT_1C(256'h0000000000000000000000000000000000000000000000000000000000000000),
```

```
        .INIT_1D(256'h0000000000000000000000000000000000000000000000000000000000000000),
208     .INIT_1E(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_1F(256'h0000000000000000000000000000000000000000000000000000000000000000),
210     // Port A Address 1024 to 1535, Port B Address 255 to 383
        .INIT_20(256'h0000000000000000000000000000000000000000000000000000000000000000),
212     .INIT_21(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_22(256'h0000000000000000000000000000000000000000000000000000000000000000),
214     .INIT_23(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_24(256'h0000000000000000000000000000000000000000000000000000000000000000),
216     .INIT_25(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_26(256'h0000000000000000000000000000000000000000000000000000000000000000),
218     .INIT_27(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_28(256'h0000000000000000000000000000000000000000000000000000000000000000),
220     .INIT_29(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_2A(256'h0000000000000000000000000000000000000000000000000000000000000000),
222     .INIT_2B(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_2C(256'h0000000000000000000000000000000000000000000000000000000000000000),
224     .INIT_2D(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_2E(256'h0000000000000000000000000000000000000000000000000000000000000000),
226     .INIT_2F(256'h0000000000000000000000000000000000000000000000000000000000000000),
        // Port A Address 1536 to 2047, Port B Address 384 to 511
228     .INIT_30(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_31(256'h0000000000000000000000000000000000000000000000000000000000000000),
230     .INIT_32(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_33(256'h0000000000000000000000000000000000000000000000000000000000000000),
232     .INIT_34(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_35(256'h0000000000000000000000000000000000000000000000000000000000000000),
234     .INIT_36(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_37(256'h0000000000000000000000000000000000000000000000000000000000000000),
236     .INIT_38(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_39(256'h0000000000000000000000000000000000000000000000000000000000000000),
238     .INIT_3A(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_3B(256'h0000000000000000000000000000000000000000000000000000000000000000),
240     .INIT_3C(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_3D(256'h0000000000000000000000000000000000000000000000000000000000000000),
242     .INIT_3E(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INIT_3F(256'h0000000000000000000000000000000000000000000000000000000000000000),
244
        // The next set of INITP_xx are for the parity bits
246     // Port A Address 0 to 511, Port B Address 0 to 127
        .INITP_00(256'h0000000000000000000000000000000000000000000000008000000000000000),
248     .INITP_01(256'h0000000000000000000000000000000000000000000000000000000000000000),
        // Port A Address 512 to 1023, Port B Address 128 to 255
250     .INITP_02(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INITP_03(256'h0000000000000000000000000000000000000000000000000000000000000000),
252     // Port A Address 1024 to 1535, Port B Address 256 to 383
        .INITP_04(256'h0000000000000000000000000000000000000000000000000000000000000000),
254     .INITP_05(256'h0000000000000000000000000000000000000000000000000000000000000000),
        // Port A Address 1536 to 2047, Port B Address 384 to 511
256     .INITP_06(256'h0000000000000000000000000000000000000000000000000000000000000000),
        .INITP_07(256'h0000000000000000000000000000000000000000000000000000000000000000)
258   ) RAMB16_S9_S36_inst (
        .DOA(Tx_mac_data_rd),        // Port A 8-bit Data Output
260     .DOB(Dout),                  // Port B 32-bit Data Output
        .DOPA(TX_mac_eop_bit),       // Port A 1-bit Parity Output
262     .DOPB(eop_byte),             // Port B 4-bit Parity Output
        .ADDRA(Tx_mac_addr),         // Port A 11-bit Address Input
264     .ADDRB(mem_addr),            // Port B 9-bit Address Input
        .CLKA(clk),                  // Port A Clock
266     .CLKB(clk),                  // Port B Clock
        .DIA(Tx_mac_data),           // Port A 8-bit Data Input
268     .DIB(32'h00000000),          // Port B 32-bit Data Input
        .DIPA(Tx_mac_is_last_byte), // Port A 1-bit parity Input
270     .DIPB(4'h0),                 // Port-B 4-bit parity Input
        .ENA(1'b1),                  // Port A RAM Enable Input
272     .ENB(1'b1),                  // Port B RAM Enable Input
        .SSRA(1'b0),                 // Port A Synchronous Set/Reset Input
274     .SSRB(1'b0),                 // Port B Synchronous Set/Reset Input
        .WEA(Tx_mac_write),          // Port A Write Enable Input
276     .WEB(1'b0)                   // Port B Write Enable Input
    );
278
endmodule
```