EVALUATION OF PICOBLAZE AND IMPLEMENTATION OF A NETWORK INTERFACE ON A FPGA

Thesis project at Elektronics system
Linköping Institute of Technology
by

Robert Mattsson

Reg nr: LiTH-ISY-EX-ET-0288-2004
Linköping 2004-06-04
EVALUATION OF PICOBLAZE AND IMPLEMENTATION OF A NETWORK INTERFACE ON A FPGA

Thesis project at Elektronics system
Linköping Institute of Technology

by

Robert Mattsson

Reg nr: LiTH-ISY-EX-ET-0288-2004

Supervisor: Peter Johansson
Examiner: Johnny Lindgren

Linköping, 4 June 2004.
Språk
Language
Svenska/Swedish
X Engelska/English

Report category
Licentiatavhandling
X Examensarbete
C-uppsats
D-uppsats
Övrig rapport

ISBN

ISRN LITH-ISY-EX-ET-0288-2004

Serietitel och serienummer
Title of series, numbering

ISSN

URL för elektronisk version

Titel
Title
Utvärdering av PicoBlaze och implementering av ett nätverksinterface på en FPGA
Evaluation of PicoBlaze and implementation of a network interface on a FPGA

Författare
Author
Robert Mattsson

Sammanfattning
Abstract
The use of microcontrollers and FPGAs is getting more and more widespread in electronic designs. A recent development has been to implement microcontrollers onboard the FPGA, there are a lot of benefits but also disadvantages with this. Often the microcontroller requires a lot of resources in the expensive FPGA. This is where PicoBlaze, a microcontroller provided by Xilinx, fits in. It is designed with one main object, keep it as small and powerful as possible.

In this report PicoBlaze is evaluated and documented. Two implementations have been done. One smaller to show how to use PicoBlaze and one larger implementation of an Ethernet network interface. The function of the implementations have been verified on an experiment board utilizing a Virtex-II FPGA.

The conclusion is that PicoBlaze is a very powerful microcontroller in comparison to the resources it uses on the FPGA. It uses only a little more than 80 slices on a Virtex II FPGA. This is its main advantage, the disadvantages of PicoBlaze is its limited program memory and the limited address space.

Nyckelord
Keyword
PicoBlaze, microcontroller, VHDL, FPGA, networkinterface, Internet, ARP, ICMP, IP
Abstract

The use of microcontrollers and FPGAs is getting more and more widespread in electronic designs. A recent development has been to implement microcontrollers onboard the FPGA, there are a lot of benefits but also disadvantages with this. Often the microcontroller requires a lot of resources in the expensive FPGA. This is where PicoBlaze, a microcontroller provided by Xilinx, fits in. It is designed with one main object, keep it as small and powerful as possible.

In this report PicoBlaze is evaluated and documented. Two implementations have been done. One smaller to show how to use PicoBlaze and one larger implementation of an Ethernet network interface. The function of the implementations have been verified on a experiment board utilizing a Virtex-II FPGA.

The conclusion is that PicoBlaze is a very powerful microcontroller in comparison to the resources it uses on the FPGA. It uses only a little more than 80 slices on a Virtex II FPGA. This is its main advantage, the disadvantages of PicoBlaze is its limited program memory and the limited address space.
Acknowledgements

I would like to take the opportunity to thank my supervisor Peter Johansson, without him I would never have been able come as far as I have with this work, and my examiner Johnny Lindgren. I should also pay my tributes to all personnel on Electronics Systems, I have probably received help from most of them in some way or another. My parents, who always have encouraged and supported me, should also be mentioned, specially my mother who has spent a lot of time reading this report and helped me correct spelling and grammatical errors.

Robert Mattsson, Linköping 2004
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>Programming language for computers</td>
</tr>
<tr>
<td>ARP</td>
<td>Address Resolutions Protocol</td>
</tr>
<tr>
<td>CLB</td>
<td>Configurable Logic Block</td>
</tr>
<tr>
<td>CPLD</td>
<td>Complex Programmable Logic Device</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>ICMP</td>
<td>Internet Control Message Protocol</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>JTAG</td>
<td>Joint Test Action Group, Protocol for accessing and controlling electronic devices</td>
</tr>
<tr>
<td>MAC</td>
<td>Media Access</td>
</tr>
<tr>
<td>MII</td>
<td>Media Independent Interface</td>
</tr>
<tr>
<td>MIPS</td>
<td>Million Instructions Per Second</td>
</tr>
<tr>
<td>Nibble</td>
<td>A 4-bit data word</td>
</tr>
<tr>
<td>PHY</td>
<td>Physical, used instead of physical layer</td>
</tr>
<tr>
<td>PIC</td>
<td>Peripheral Interface Controller</td>
</tr>
<tr>
<td>PLD</td>
<td>Programmable Logic Device</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very High Speed Integrated Circuit</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver-Transmitter</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

1 Introduction .................................................. 1
   1.1 Background ........................................... 1
   1.2 Purpose ............................................... 2
   1.3 Method ............................................... 2
   1.4 Reading instructions ..................................... 3
       1.4.1 Typographical conventions ......................... 3

2 Facts and background ........................................ 5
   2.1 FPGA .................................................. 5
       2.1.1 Virtex-II FPGA architecture ...................... 5
   2.2 VHDL .................................................. 6
       2.2.1 Unisim ........................................... 6
   2.3 Microcontroller ......................................... 7
       2.3.1 Implement a microcontroller on a FPGA .......... 7
   2.4 Internet ................................................ 7
       2.4.1 Ethernet ........................................ 8
       2.4.2 IP ................................................. 10
       2.4.3 ICMP ........................................... 12
       2.4.4 ARP .............................................. 13

3 PicoBlaze ..................................................... 15
   3.1 Presentation ............................................ 15
       3.1.1 PicoBlaze versions ................................ 15
       3.1.2 Architecture .................................... 16
       3.1.3 Instruction set ................................... 17
   3.2 I/O ports and signals .................................... 18
       3.2.1 out_port[7:0] and in_port[7:0] .................. 18
       3.2.2 port_id[7:0] ..................................... 18
       3.2.3 read_strobe and write_strobe .................... 18
       3.2.4 interrupt ....................................... 19
       3.2.5 reset ............................................ 19
   3.3 Assembler ................................................ 19
       3.3.1 Program syntax and assembler directives ........ 19
       3.3.2 Assembler files .................................. 20
3.3.3 pBlazeIDE .............................. 21
3.4 Update program memory .............. 21

4 Using PicoBlaze .......................... 23
4.1 Counter ................................. 23
4.1.1 Design ............................... 23
4.1.2 Problems encountered ............. 24
4.1.3 Writing a program for PicoBlaze .. 24
4.2 Update program memory .............. 25
4.2.1 Design ............................... 25
4.2.2 Downloading new program memory . 26
4.3 Results ................................. 27

5 Implementation of a network interface 29
5.1 Design .................................. 29
5.1.1 PHY interface ....................... 30
5.1.2 MAC layer ........................... 31
5.1.3 IP Layer ............................. 31
5.2 Hardware Solutions ................... 32
5.2.1 RX ................................. 33
5.2.2 TX ................................. 37
5.2.3 IP ................................ 38
5.3 Programs ............................... 40
5.3.1 Program for PicoBlaze in MAC layer (RX) 40
5.3.2 Program for PicoBlaze in IP layer .... 41
5.4 Functional verification ................. 42
5.5 Physical verification .................... 43
5.6 Result ................................ 43

6 Conclusion ............................... 45
References ................................ 47
Appendix A ................................. 49
INTRODUCTION

The use of microcontrollers and FPGAs is getting more and more widespread in electronic designs. A recent development has been to implement microcontrollers onboard the FPGA, there are a lot of benefits but also disadvantages with this. Often the microcontroller requires a lot of resources in the expensive FPGA. This is where PicoBlaze, a microcontroller provided by Xilinx fits in. It is designed with one main object, keep it as small and powerful as possible.

This thesis work has been done at Electronic Systems, Department of Electrical Engineering, Linköping Institute of Technology with the purpose to document PicoBlaze, its advantages and disadvantages and how to use it. It has been done by research, mostly on Internet, and by implementing a network interface using PicoBlaze on a Virtex II FPGA.

1.1 BACKGROUND

The use of microcontrollers are widely spread, they are used in embedded systems and can be found in almost all electronic products on the market, for example in dishwashers, cars and TV’s. Programable logic is also spread and while the microcontrollers don’t change very much, the programable logic devices (PLD) are getting both cheaper and larger in means of capacity and speed.

This far most of the programable devices have been used to realize smaller parts of a larger design. But it’s getting reasonable to implement larger parts
of a design including the microcontroller inside of the more advanced PLDs. This makes the design process faster and cheaper.

PicoBlaze is a microcontroller that is developed by Xilinx to be used on their hardware. Since it is only supposed to run on Xilinx devices it can be optimized to utilize the hardware on a device and save resources that are needed for other applications.

A popular use for microcontrollers is to add the necessary hardware and then connect them to the Internet. As an example the microcontroller could supervise measuring devices and provide the status via a webpage or send an email at a certain interval or when a specific value is reached. This has inspired the implementation of the network interface that will be described in this report.

1.2 PURPOSE

The purpose of this report is to document and evaluate PicoBlaze, a 8-bit microcontroller to be implemented on a FPGA from Xilinx. Facts about and advantages and disadvantages of PicoBlaze will be presented so that a potential user of a microcontroller for a FPGA can decide if PicoBlaze fulfills the requirements for the microcontroller. How to use and add functionality to PicoBlaze will also be presented so that a potential user can get a quick start with the processor.

1.3 METHOD

The purpose is reached by literature studies and by creating designs and implementing them on hardware. Most of the literature concerning PicoBlaze is found on Xilinx’s webpage. Facts used when implementing the network interface have been found in Understanding Data Communications [1] and on the Internet.

Tools for design and implementation have been Mentor Graphics FPGA Advantage for HDL design, release 5.4. FPGA Advantage 5.4 is a program suite consisting of HDL Designer version 2002.1b, ModelSim SE Version 5.6f and LeonardoSpectrum Version 2002e. Xilinx ISE 6.1 is used to generate the bit file and downloading it to the FPGA.
The hardware used to run the designs on is an experiment board from Insight MEMEC called Virtex-II MicroBlaze Development Board. The board utilizes a Xilinx Virtex-II XC2V1000-4FG456C FPGA. Other hardware on the board that will be used in the implementations are a 100 MHz oscillator that will be used to clock the device, two 7-segment displays, a PHY interface from Broadcom and a RJ45 connector.

1.4 READING INSTRUCTIONS

This report is written for a reader with basic knowledge in electronics and digital design. In chapter 2 theory about FPGAs and microcontrollers is presented. Internet and Ethernet is also handled since a network interface will be designed. Chapter 3 describes the PicoBlaze, the purpose of the chapter is to describe how PicoBlaze works and what it is capable of. How to implement and how to use PicoBlaze is presented in chapter 4 where a simple implementation is shown. In chapter 5 the implementation of a network interface with PicoBlaze is described.

1.4.1 TYPOGRAPHICAL CONVENTIONS

When signals are described their status is described as high or low or with a ‘0’ or ‘1’. The value of a bus or register are shown with either an bit array e.g. “0001” or in hexadecimal. To clarify that a hexadecimal value is being used, hexadecimal values are followed by (hex) e.g. 3FF (hex). Signal names are written in Italic e.g. signal, buses are also written in Italic. Assembler instructions and logical operations are written with capital letters in courier e.g. JUMP.
FACTS AND BACKGROUND

2.1 FPGA

A FPGA (Field Programable Gate Array) is a sort of a programable logic device (PLD). FPGAs have generally been expensive and their use has been limited to products produced in small numbers used for verification of functionality before a design has been implemented on a chip. The development of FPGAs has been fast the recent years and their capacity and clock speeds have increased and at the same time the prices have decreased making them more price worthy. Their use is getting more widespread and can be found in more and more applications.

2.1.1 VIRTEX-II FPGA ARCHITECTURE

Virtex-II FPGAs are built of CLBs (Configurable Logic Block). Each CLB consists of 4 slices. One slice is the smallest programable unit and can realize a small logical expression. Since slices or CLBs are the smallest programable units the capacity of a FPGA is often given in the number of slices it has. Also the utilization or the size of a design, is often given in slices. In addition to the CLBs the FPGAs contain blocks designed for a special function, examples are blocks for multiplication or blocks containing RAMs (blockRAMs).

Virtex-II is Xilinx most advanced FPGA family and can supply 46,594 CLBs and 168 blockRAMs. However the FPGA used in this work, Virtex II XC2V100-4FG456C, is a smaller model supplying 5,120 slices and 40 block-RAMs.
2.2 VHDL

VHDL (VHSIC Hardware Description Language) is a description language that is used to simulate and implement digital designs, the syntax is similar to the programming language Ada. It was first developed by USA's Department of Defence in the beginning of the 1980s, who needed a standardised way to describe electronic systems. In 1987 VHDL was standardised by IEEE, the standard has later been updated in 1993 and 2001. In the end of the eighties simulators was developed for computers so that the function of the design could be verified. The standard was set to design digital models, it is in the last ten years VHDL has been used for designing a working system.[2]

Designing a system with VHDL follows the same flow most of the time but this section will be more specific about the design flow for FPGA design. The work starts with the designer describing how the system should work. When the function is verified in a simulator it is possible to make changes to specify exactly how a task should be carried out. When the design is to satisfaction the design is synthesised. A synthesis tool translates the VHDL, a high level description, down to how the design will be realized on the FPGA. This is done in a couple of steps, first the VHDL description is translated into logical functions such as boolean expressions. Then these functions are mapped to CLBs or special function blocks that exists on the FPGA. Finally it is decided on which physical block on the FPGA each function will reside. Now the synthesis step is done and the result is converted into a bit file that is downloaded to the FPGA.

2.2.1 UNISIM

Unisim is a library that is used when creating VHDL designs for Xilinx devices. Unisim contains functional descriptions of special function blocks on Xilinx devices. This makes it possible to map a function directly to a logical building block and simulate it. The library also contains descriptions for functions that are used a lot but don’t have any special function blocks. These functions are implemented in CLBs, they are optimized to take advantage of the components that are used in a CLB.
2.3 MICROCONTROLLER

A microcontroller is a small computer that usually is totally embedded on one chip, including program memory and a small RAM. Unlike personal computers that can run a wide range of programs microcontrollers runs one small program stored in their program memory. Microcontrollers are usually used in embedded systems, often in consumer products where they control features or actions of products such as dishwashers, TV sets or cars.

In the more advanced microcontrollers the processor designs used are often based on processors that were used in desktop computers in the eighties such as Motorolas 6800 series or Intel 80386. In less advanced microcontrollers special processor designs are used to keep the microcontroller small and to keep down the power dissipation and cost. These microcontrollers are often referred to as PIC microcontrollers and was created by a company called Microchip, other companies have followed Microchip and have developed extremely small and cheap microcontrollers, an example is Atmel, who has a series of small microcontrollers called AVR.[ 10]

2.3.1 IMPLEMENT A MICROCONTROLLER ON A FPGA

It is possible to find two cases where it is an advantage to implement microcontrollers on a FPGA instead of using a microcontroller implemented on its own chip. In a design where both a FPGA and a microcontroller is used it could be an advantage to implement the microcontroller directly on the FPGA, this reduces the number of components in the design thus reducing production costs and development time. This requires that the microcontroller will fit in the FPGA without using a FPGA that is larger and more expensive than the first solution. The second case is when a large FPGA is used. In a large FPGA the design will probably use some control logic, here a microcontroller might be the best choice in means of resources used for controlling the design.

2.4 INTERNET

Since the main implementation will be a network interface a shorter presentation of the different layers and protocols that Internet is built on will follow. In this work data will be sent by the Internet Protocol (IP) and Address Reso-
Evaluation of PicoBlaze and implementation of a network interface on a FPGA

Communication via Internet are separated in different levels. This is to make the communication flexible. The communication can be divided into three layers. First there is a physical link, the media which the communication goes through called physical layer (PHY). Next there is a layer controlling the communication on the media, this is called the MAC layer (Media Access Control). And at last the software layer comes, this is a stack of protocols such as the Internet Protocol (IP). Data sent via the Internet is transmitted in packets or frames. Each protocol that is used adds to a part of that frame.

**2.4.1 ETHERNET**

Ethernet is a standard that defines the physical layer and the MAC layer. Originally Ethernet was developed for a local network with communication on one cable connecting all units or nodes on the network. The standard has grown with new technologies as networking has developed, but the basics are still the same. [20]

When the medium is used by a node no one else may try to transmit on the medium, the units waits until the medium is free. If two devices should try to start a transmission at the same time the units stop and wait for a random period of time before trying to transmit again.

Each node connected to the network has an unique MAC or Ethernet address that it listens to. There is also an broadcast address that all nodes listen to. All data transmitted on an Ethernet network is framed in a Ethernet frame that tells the destination and source of the frame.

*Figure 1: Different layers in a network interface*
Ethernet is standardized by IEEE and is then called IEEE 802. IEEE differs slightly from Ethernet, but the name Ethernet is still commonly used since the changes are minor. A presentation of a IEEE 802.3 frame will follow, 802.3 is the standard for twisted pair networks, read *Understanding Data Communications* [1] for more detailed information of Ethernet and the Ethernet frame.

<table>
<thead>
<tr>
<th>Field size in bytes</th>
<th>7</th>
<th>1</th>
<th>6</th>
<th>6</th>
<th>2</th>
<th>46-1500</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preamble</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SFD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Destination Address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source Address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type / Length</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 2: Ethernet (IEEE 802.3) frame format*

**Preamble**

The preamble field consists of 7 bytes containing “0101 0101” or 55 (hex).

**SFD - Start of Frame Delimiter**

The end of preamble is indicated with SFD. SFD is one byte and the value is “1101 0101” or D5 (hex).

Preamble and SFD result in a series of ‘1’s and ‘0’s ended with ‘11’ since the least significant bit is sent first. e.g. “1010 1010 ... 1010 1011” this sequence is used to synchronize the sender and receiver.

**Destination Address**

This field contains the MAC address to the destination of the frame. If the frame is intended to be received by all clients on the network all bits in this field is set to ‘1’ this address is called the broadcast address.

**Source Address**

Contains the senders MAC address.

**Type / Length**

This field tells the length in bytes of the following data field. If the value is higher than 1500 the field tells what kind of protocol the data field contains e.g. 0806 (hex) for ARP packets and 0800 (hex) for IP packets.

**Data**

The Data field contains the Data to be sent such as a IP or ARP packet. It may
not be smaller than 46 bytes to ensure that the frame is 64 at least bytes, not including preamble and SFD. If the field is not large enough padding will have to be done. Generally the padding is done with zeroes. The maximum length of the data field is 1500 bytes.

**FCS - Frame Check sequence**

The value in this field is used to control that the frame is received without errors. The value is calculated over the whole frame not including the preamble, SFD and FCS fields. The algorithm to calculate the value is a 32 bit cyclic redundancy check (CRC). The sender calculates the value and sends it in the FCS field. The receiver calculates its own value and compares it to the FCS field, if they do not match the frame is discarded.

### 2.4.2 IP

Internet protocol (IP) is the fundamental protocol when transmitting data on the Internet. IP is not used alone, there are always protocols used above IP. IP contains information about where the packet is going where it comes from and other information about the transmission. If a larger amount of data, a datagram, is sent it will not fit in one packet, in this case the datagram will be fragmented and put into a number of packets. Information in the IP header informs if it is a fragmented datagram and where in the datagram the packet fits.

![Figure 3: Structure of an IP packet](image)

Figure 3 is a diagram of the structure of a IP packet according to RFC 791 -
Internet Protocol [17], the four last fields, except options, are not explained since they should explain themselves. When a value of a field is presented it is given in hexadecimal.

**Version**
Indicates what IP version is used, 4 for IPv4 and 6 for IPv6.

**Header Length**
Length of the IP header in 32 bit words.

**Type of Service**
Used to set priority of the packet in a network, all bits could be set to ‘0’. For more details see RFC 791 [17].

**Total Length**
Size of entire packet in bytes including IP header.

**Identification**
Identification number to tell the receiver of which datagram the packet belongs to.

**DF, MF**
Three bits used as flags, first bit is reserved and always set to zero. Second bit is set if fragmentation of the packet is allowed, the third bit is set if there are more fragments after this packet.

**Fragment Offset**
This field indicates where in the datagram this packet belongs.

**Time To Live**
This value indicates the maximum time the packet may live. The value is decreased each time the packet is processed on its way to the destination. When the value is 0 the packet must be destroyed.

**Protocol**
Indicates the next level protocol e.g. 01 for ICMP protocol.

**Header Checksum**
Checksum of the IP header to indicate errors in IP head. The value is calculated by the receiver when the packet is received and compared to the value in
Header checksum. The value is computed at each point the packet is processed. When the checksum is calculated Header Checksum field is set to 0000. For algorithm see RFC 791 [17] or Calculating IP Checksums [11] for a good example.

Options

Options are an optional field, see RFC 791 [17] for more information. In some cases there are no option field at all, this applies to ICMP packets.

2.4.3 ICMP

Internet Control Message Protocol (ICMP) is a protocol that is used for finding problems and diagnosing the network. There are a number of different ICMP packets, but in this work only echo request and echo reply will be used. When an echo request packet is received at its destination it is returned as an echo reply message. ‘Ping’ that is included in most operating systems is a well known user of ICMP echo messages. ‘Ping’ can be used to see if an IP address is in use and if its associated host is connected to the network. A detailed presentation of ‘Ping’ can be found at Freesoft.org [19].

Figure 4 is a diagram of the ICMP frame according to RFC 792 - Internet Control Message Protocol [18].

![Figure 4: Structure of a ICMP packet](image)

Type

Tells what type of ICMP packet it is, 8 for an echo request message and 0 for an echo reply message

Code
Set to zero

Checksum

The same usage and algorithm as the IP Header checksum. Calculated over Type, Code, Checksum, Identifier, Sequence number and Data. Checksum field is set to 0 when checksum is calculated.

Identifier, Sequence number

Used to match echo requests and replies, may be zero. Used by the echo sender, the echoer returns the values received.

Data

Data must be returned in the echo reply message. Size and content of data is defined by the echo sender.

2.4.4 ARP

Address Resolution Protocol (ARP) is used as a link between the Internet Protocol and Ethernet. If an Ethernet frame is to be sent the MAC destination address must be known, but the sender usually only knows the IP address. The ARP message works as a question, “Who has X.X.X.X tell Y.Y.Y.Y” where X.X.X.X and Y.Y.Y.Y are IP addresses. The message uses the broadcast address so that all nodes on the network receive the frame. Then the affected node forms an answer with its MAC address and its IP address. Resolved addresses are normally saved in a cache to reduce the number of ARP requests.[14]

When a request is sent and the MAC address is unknown all bits in the destination address in the Ethernet frame are set to ‘1’ which is the broadcast address. The MAC destination address in the ARP package is usually set 0.0.0.0.0.0 in an ARP request. At a reply the source address is simply moved to the destination address field and in the source field the senders address is added. A diagram of the structure of an ARP package is shown in figure [5] followed by an explanation of the fields, all values are given in hexadecimal representation.
MAC type
Identifies the network type, 0001 for Ethernet.

Protocol type
Identifies the network protocol, 0800 IP.

MAC length
Length of the hardware address, 06 for Ethernet.

Network length
Length of the network protocol address, 4 for IP.

Operation
Identifies request or reply message, 0001 for request and 0000 for reply.
3

PICOBLAZE

3.1 PRESENTATION

PicoBlaze is a 8-bit microcontroller developed and maintained by Xilinx and Ken Chapman. The microcontroller is described in VHDL and is to be implemented on Xilinx’s different FPGAs and CPLDs. It is free to use as long as it is implemented in a FPGA or CPLD that comes from Xilinx [9].

PicoBlaze is well documented, the application notes are detailed and well written. On Xilinx’s webpage for PicoBlaze [16] there is a forum where PicoBlaze users can ask questions and help each other, this forum is often visited by Ken Chapman who answers questions. There is also a range of free tools that have been developed to use with PicoBlaze. All this makes PicoBlaze easy to work with.

PicoBlaze is downloaded from the PicoBlaze Softprocessor homepage [16]. The package contains a number of files including the VHDL definition of the PicoBlaze, an assembler and the files that go with it. Included with PicoBlaze is also a VHDL definition of a UART transmitter and receiver, documentation for the UART and a display decoder for a 4-bit word to a 7-segment display. All of these VHDL definitions use the Unisim library.

3.1.1 PICOBLAZE VERSIONS

Right now there are three different versions of PicoBlaze due to limitations in different devices that effect the design of the microcontroller. The versions
differ in fields such as size of the program memory, number of internal registers and the stack depth. There is one version for CoolRunner-II CPLD with eight general purpose 8-bit registers and a 4-entry program counter stack. Another version is for Virtex, Virtex-E, Spartan-II and Spartan-IIE FPGAs. This version has 16 general purpose 8-bit registers and a 15-entry program counter stack. Common for these two versions is that both have a program memory that can store 256 instructions [6] [7].

The last version is the one for Virtex-II FPGAs, it is often referred to as PicoBlaze2. It has 32 general purpose 8-bit registers, and a 31-entry program counter stack. The program memory can store 1024 instructions. The performance of this version is in the range of 40-70 MIPS, depending on device speed grade [8]. This is the version that will be used in the implementations and if nothing else is indicated this will be the referred version.

PicoBlaze was first named KCPSM (Constant (k) Coded Programable State Machine), but was renamed to PicoBlaze to follow the naming of other Xilinx products. Despite this PicoBlaze is often referred to as KCPSM. File names are still named KCPSM. To keep track of the files for the different versions, the one for Virtex-II is often referred to as KCPSM2.

There is a fourth version of PicoBlaze that works on Virtex 2 and Spartan 3 FPGAs. It is still under development and is not available on Xilinx’s homepage, but by request Ken Chapman can send a copy by e-mail. This version is often referred to as PicoBlaze3 or KCPSM3. It has some new features that were missing in the earlier versions, such as test and compare instructions, and a 64 byte scratch pad memory that works like an internal RAM. [4]

### 3.1.2 ARCHITECTURE

PicoBlaze is totally implemented into an FPGA or a CPLD and requires no external circuits to work. A diagram of the PicoBlaze architecture can be found in figure 6. A single block RAM is used to form a ROM to store the program in, for PicoBlaze2 it holds 1024 18 bit instructions. It has been designed to be small and exploit the hardware it is running on. Ken Chapman discusses how he reasoned while designing and optimizing PicoBlaze in an interesting paper called Creating Embedded Microcontrollers [21].

When PicoBlaze is synthesized with the program memory LeonardoSpectrum reports that 82 of the 5,120 available slices on the Virtex-II XC2V1000 are used.
Chapter 3 – PicoBlaze

3.1.3 INSTRUCTION SET

PicoBlaze has 49 different instructions, a list of them can be found in appendix A. There is no specific accumulator register, all operations can be done on any of the 32 different registers. Most of the instructions use a register as a operand and returns the result to the same register, if an instruction uses a second operand it can be either another register or a constant. All instructions execute over two clock periods.

The instructions can be categorized in 6 groups; program control, logical, arithmetic, shift and rotate, input and output and interrupt. In means of program control the processor features jumps in program, call and return for subroutines, all can be conditional or unconditional. Logical instructions
include an instruction to load a value to a register and boolean operations such as \texttt{AND}, \texttt{OR} and \texttt{XOR}. Arithmetic instructions consist of addition and subtraction with or without a carry flag.

Shift and rotate groups include instructions for both left and right shifts or rotations on a single register. Input and output instructions use a first operand to tell in which register to put the input value or from which register the output value should come from. The address, \texttt{port\_id}, are defined by a second operand. Worth to mention is that it is not possible to write a constant directly to the outport, it must first be loaded into a register. The instructions in the interrupt group make it possible to enable or disable the use of interrupt. There is also a special return instruction to use after an interrupt.

### 3.2 I/O PORTS AND SIGNALS

#### 3.2.1 \texttt{OUT\_PORT}[7:0] AND \texttt{IN\_PORT}[7:0]

\texttt{in\_port} and \texttt{out\_port} are used for data in- and output from the controller. The data on these ports come from the first register declared in the in- or output instruction. Data is stable during the two clock cycles that the instruction executes, but during execution of other instructions the value of the ports change.

#### 3.2.2 \texttt{PORT\_ID}[7:0]

\texttt{port\_id} is used to direct where from or to the data will be read or written. The 8 bits may be a constraint since it only allows for 256 different ports. The value on \texttt{port\_id} is provided by the second operand in the in/out-put instruction. Data on \texttt{port\_id} is, just as \texttt{out\_port} and \texttt{in\_port}, stable during the two clock cycles that the instruction executes, but during execution of other instructions the value of the port changes.

#### 3.2.3 \texttt{READ\_STROBE} AND \texttt{WRITE\_STROBE}

\texttt{read\_strobe} and \texttt{write\_strobe} are used to indicate that data is read or written. They are only high during the last clock cycle that a read or write instruction uses. For more information and a diagram of the signals timing of input and output operations see \texttt{XAPP627} [ 8].
3.2.4 INTERRUPT

The interrupt signal is active high, at an interrupt the zero and carry flags are preserved and further interrupts are disabled. The interrupt forces the program counter to go to the last instruction in the program memory. This instruction is typically a jump to the interrupt sub routine. To return from a interrupt there is a special return instruction for interrupts that restore the flags.

3.2.5 RESET

The reset is active high. Reset forces the processor to start from its initial state, address 000 (hex). Interrupts are disabled and flags and the CALL/RETURN stack is reset, the registers are not affected.

3.3 ASSEMBLER

The assembler is a DOS program called KCPSM2.EXE. Program code for the assembler may be written in any standard text editor such as Notepad in Windows or Emacs in a UNIX environment. The file containing the program code should be saved with a ‘.psm’ extension. The assembler uses the ‘.psm’ file and a template called ‘ROM_form.vhd’ to create a VHDL description of the program memory. The template is provided in the package with PicoBlaze. The assembler also uses another template provided in the package called ‘ROM_form.coe’, to generate a coefficient file to be used by the core generator if desired.

3.3.1 PROGRAM SYNTAX AND ASSEMBLER DIRECTIVES

In excess of the program instructions the assembler uses a few directives to define labels and force the assembler to a specific address. This section will contain a summary of the program syntax and the directives for the assembler. For a more detailed description see XAPP627 [8].

Constants and addresses are specified with two hexadecimal values in the range 00 to FF (hex). The 32 internal registers are defined as sXX where XX is two hexadecimal digits in the range 00 to 1F (hex). Anything written after a semicolon ‘;’ is ignored, making it possible to add comments. Blank lines will be ignored and removed from the formatted file, to keep a blank line a
A semicolon can be used.

It is possible to define constants and registers or a special line in the program to a label. Labels are case sensitive, valid characters are A-Z, a-z and 0-9. Registers defined by a label are only possible to access with the label name, thus reducing the chance that the register is used by accident somewhere else in the program. Defining a constant makes it easier to change values that are used repeatedly in the code. Defining a line label makes it possible to declare program jumps to a label instead of a specific address. Common for all the labels is that, used properly, the program code is be easier to understand and some errors will therefore be avoided.

### 3.3.2 ASSEMBLER FILES

As earlier mentioned the assembler uses the program file ‘<filename>.psm’ and ‘ROM_form.vhd’ and ‘ROM_form.coe’ as inputs. ‘ROM_form.vhd’ is basically a initiation file that contains the values of the program memory. The assembler turns out totally 13 different files, a description of them follows:

- **<filename>.vhd** Contains the description of the BlockRAM memory that is to be used as program memory and the program it holds.
- **<filename>.coe** As above but a description for the core generator
- **<filename>.fmt** Contains the original program, but formatted by the assembler.
- **<filename>.log** Presents details of the assembly process, it shows the addresses and the opcodes associated with each line of the program.
- **<filename>.hex** Contains the opcodes in hexadecimal format
- **<filename>.dec** Contains the opcodes in decimal format
- **constant.txt** Presents a list of constants and their values defined by the CONSTANT instruction in the program
- **labels.txt** Presents a list of line labels and their associated addresses defined in the program
- **pass[1-5].dat** Files created in the assembly process, may be useful for debugging.
3.3.3 PBLAZEIDE

pBlazeIDE is a tool to help develop programs for PicoBlaze microcontrollers. It is developed and provided for free by Mediatronix[15]. It can be used as a debugger, running the program step by step or as a developing environment capable of generating all the files that the PicoBlaze assembler does. pBlaze IDE can compile code and indicate where errors may occur, when the code is compiled pBlaze can simulate a PicoBlaze microcontroller and execute the code just as the processor does. It is also possible to run the program step by step or to use breakpoints to make the simulator stop running the program at a specific line in the code. It is possible to add directives specific to the pBlaze. Among other things these directives can define in- and output devices connected to the processor while running the program on the simulator. Most useful are probably the possibilities to simulate RAMs and ROMs.

The assembler used in pBlazeIDE uses another syntax than the one used by the PicoBlaze assembler. This syntax is supposed to be more similar to regular assembler code and easier to use by users that are familiar with assembler programing. It is also more advanced, for example constants can be defined with both decimal and hexadecimal values. For use of pBlazeIDE as a debugging tool for code written for the PicoBlaze assembler pBlazeIDE has an import function. This function can convert programs written with the PicoBlaze syntax to the syntax used by pBlazeIDE PicoBlaze assembler. It is recommended to use the formatted file that KCPSM generates when importing code since pBlazeIDE might have problems converting unformatted code.

3.4 UPDATE PROGRAM MEMORY

One of the advantages of microcontrollers is that the program code is easy to write, and is often easy to update the program memory and the result of changes in the program code can be viewed directly. This advantage is lost with a microcontroller that runs on a FPGA. A small change in the program code results in the time consuming procedure of first running an assembler, including the generated VHDL file in the design and then synthesising the design to generate a bit file to download to the FPGA. It would be an advantage if the program code could be updated without going through the whole implementation procedure.

One method is to edit the bit file, this can be done with a tool called DATA2BRAM, but this tool does not work on Virtex II devices and the bit file
must still be downloaded to the FPGA. Instead the fact that the program memory is not a ROM but actually a dual port ram can be used. This opens up for a lot of different solutions for updating the program memory. The processor could actually be made to update its own program code. However these solutions would use resources on the FPGA that might be needed to something else.

There is one solution that comes almost to no cost at all, at least as long as the design is in the development stage. It uses the JTAG port that is used to download the bit file to the Virtex II FPGA. Since the FPGA is already connected to the computer via the JTAG port no extra hardware is needed and no pins on the FPGA are occupied. All newer FPGAs from Xilinx have a special hardware component onboard that allows custom logic to be connected to the JTAG port by a special instruction, on Virtex II devices this hardware is accessed as a component called BSCAN_VIRTEX2 in the Unisim library. Data out from this part is serial, so it requires a serial to parallel register, this register can be fitted to a single CLB.

With the serial to parallel register, the BSCAN-block and the cable to connect the computer to the FPGAs JTAG port, all hardware is supplied to update the microcontrollers blockRAM. This procedure is more thoroughly explained in Reconfiguring Block RAMs by Kristian Chaplin at Xilinx [21]. This is also done and described in chapter 4.2.
4

USING PICOBLAZE

To show how PicoBlaze is used and to show how the program memory can be updated via the JTAG port a small design realizing a counter has been implemented and is presented in this chapter.

4.1 COUNTER

The main purpose of this implementation is to get an understanding of how to use the PicoBlaze and the tools involved to implement it on the Virtex-II FPGA. Including writing a simple program and using the assembler. The task is to create a counter that counts from 00 to FF (hex) and is displayed on two 7-segment displays on the experiment board, the counter should be possible to restore with a reset button and halted for a few seconds with an interrupt signal. As far as possible the components in the PicoBlaze package will be used. By this specification a basic range of functions and tools are used without making the implementation too complex.

4.1.1 DESIGN

The counter is made out of three main building blocks (figure [ 7]) that come with the PicoBlaze, the PicoBlaze microcontroller, the description of a 4-bit to 7-segment decoder and the program memory generated by the assembler. The two signals reset and interrupt, generated by two switch buttons on the experiment board, is active low, which results in a block that inverts the reset
and interrupt signals before they reach the controller. The block two_segment consists of a 8-bit register and two 4-bit to 7-segment decoders that are provided with PicoBlaze. The register is used to store the latest values that were a valid output on out_port. The register is updated on the rising edge of write_strobe.

![Diagram of counter design](image)

**Figure 7: Block diagram of counter design**

### 4.1.2 PROBLEMS ENCOUNTERED

There were some problems encountered during the implementation of the counter. The most time consuming problem was that LeonardoSpectrum had problems with instances of design units. The source of the problem was that all top views of each design was called `top` and that caused an error while synthesising the design. The solution was to give each top design a specific name. A lot of time was spent on this problem until Peter Johansson discovered the source of the problem.

### 4.1.3 WRITING A PROGRAM FOR PICOBLAZE

In the application note [8], the syntax and the instructions are well described, the effects of each instruction and what flags it affects is presented. There are also some hints and tips about how to make test and compare operations in an efficient way and other ways to write efficient code.

No big problems were encountered while writing the program, but some problems occurred while simulating it on pBlazeIDE. The conversion of counter.psm sometimes did not work properly, however by using the file formatted by the assembler this problem was solved. Properly imported the program did not run, this because pBlazeIDE was set to simulate PicoBlaze for a
Virtex or Spartan chip, thus it did not accept using registers 16 through 31. After setting pBlazeIDE to simulate a PicoBlaze implemented on a Virtex-II it worked well.

4.2 UPDATE PROGRAM MEMORY

The object of this implementation is to show how the BlockRAM used for program memory can be used as a dualport RAM with one port connected to the processor and the other port controlled by the JTAG port. This will be done by using tools created by Kris Chaplin at Xilinx, the package of tools is called PicoBlaze JTAG loader.

In this implementation both the counter design and microcontroller program that was created in the earlier implementation will be used. The program memory will be changed so that it can be updated via the JTAG port and necessary logic will be added to support this. The new program that will be downloaded to the program memory will be a copy of the earlier program only with the difference that this program starts at FF and decreases its value instead.

4.2.1 DESIGN

With the PicoBlaze loader (figure 8) from Kristian Chaplin a new template for the program memory is provided. This template uses a component description of a dual port memory, instead of a single port as in the original template. In addition to this and the memory’s initial values it also contains the logic for receiving data from the JTAG port and logic to write the data to...
Evaluation of PicoBlaze and implementation of a network interface on a FPGA

The JTAG port is accessed by a component called BSCAN_VIRTEX2 in the UNISIM library. The logic is a serial to parallel register made out of 20 flip flops, by optimizing the design for Virtex II Kris has managed to use only one CLB for the shift register, read more about this in Reconfiguring Block RAMs [21].

Using the new template makes the change in the design (figure [9]) minor since all logic is hidden in the memory block, the only visible change in the design is a new output signal from the program memory. It is a reset signal called proc_reset that is active high and resets the processor while updating the program memory. This signal is combined with reset with a logical OR in the inverter block.

![Block diagram of counter with updateable program memory](image)

**Figure 9: Block diagram of counter with updateable program memory**

### 4.2.2 DOWNLOADING NEW PROGRAM MEMORY

The new program code is downloaded to the FPGA with a program called PlayXSVF. To obtain a file that PlayXSVF can download a few steps must be taken where the hex file obtained in the assembly process is used to generate the ‘.xsvf’ file that PlayXSVF can use. This is a straightforward procedure well described in the documentation for the PicoBlaze loader.

To generate the ‘.xsvf’ file, information of where the FPGA is in the JTAG chain and the instruction length of the devices in the chain is entered in a setup program. On the experiment board used there where is JTAG devices, the Virtex-II FPGA is placed in the middle, the instruction length of the first device is 8 and the instruction length of the last device is 4.
4.3 RESULTS

The impression of PicoBlaze after these two implementations is that it is easy to use. No problems was encountered when simulating or synthesizing the controller and memory, the problems encountered have been caused by either the user or by bugs in FPGA Advantage. The application note for PicoBlaze is very well written and covers all information needed by the first time user.

Table 4.1: Resources used by Counter on Virtex-II XC2V100 reported by NGD Build

<table>
<thead>
<tr>
<th>Block</th>
<th>Counter</th>
<th>Counter w. PicoBlaze loader</th>
<th>Available on FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAMB16(BlocRAM)</td>
<td>1</td>
<td>1</td>
<td>40</td>
</tr>
<tr>
<td>SLICE</td>
<td>97</td>
<td>103</td>
<td>5120</td>
</tr>
<tr>
<td>BSCAN</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

As seen in table 4.1 the designs are small in comparison to the resources available on the FPGA. Of the 97 slices used in the counter design PicoBlaze uses 82 of them. Specially interesting is the difference when the PicoBlaze loader is used, with the PicoBlaze loader the design uses seven more slices. According to Kris Chaplin [21] the only resources used by PicoBlaze loader on the FPGA would be the BSCAN block and one CLB or four slices. The reason that seven slices are used could be found in the synthesising tool and the level of optimization. However the seven slices used are cheap in comparison to the added functionality.
5 IMPLEMENTATION OF A NETWORK INTERFACE

The purpose of implementing a network interface is to show how PicoBlaze could be used in a larger design and to some extent show what it is capable of. The goal is to realize the hardware that a network interface requires and to implement some of the basic networking protocols and run them on a PicoBlaze to show that it is capable to connect to the Internet.

5.1 DESIGN

The network interface will be designed to connect to a 10 Mbit twisted pair Ethernet network. The relative low speed of 10 Mbit might seem a bit out of date but is motivated by the fact that implementations run by a 8-bit microcontroller will not require high data rates. Keeping the data speed rates low reduces the requirements of the design and will give more time to implement the software protocols.

There is already some hardware available to connect the experiment board to a 10/100 mbit ethernet network. It consists of a RJ45 connector which is used to connect the board to a physical network and a 10/100 Ethernet PHY interface.
5.1.1 PHY INTERFACE

The PHY (physical layer) interface constitutes a link between the physical signals transmitted on the network and the digital environment on the FPGA. The PHY interface is implemented on a single chip called BCM5221 and comes from a company called Broadcom. The chip handles data and clock recovery and data encoding and decoding. It communicates with the MAC layer via Media Independent Interface (MII) which is an industry standard interface for PHY devices. This means that the system designer does not need to consider whether the physical network consists of a 10 Mbit or 100 Mbit twisted pair network or even a fiber optic network. As long as the PHY device uses MII the communication between the PHY device and the MAC layer is the same, the only difference is the speed of the data transmission.

It is possible to control the PHY device and get information about the network via the MII interface, but to keep the design small these options will not be used. Only the necessary functions for receiving and transmitting data will be used. For transmitting and receiving data MII works like a basic parallel data bus (figure [10]). It uses two 4-bit data buses, one for receiving and one for transmitting data. With each bus, MII provides a clock signal to synchronize the data on the bus, when data is transferred in 10 mbit, the clock rate is 2.5 MHz. It also provides signals to tell when data on the bus is valid, when the physical link is available and collisions have occurred. In the datasheet for BCM5221 [3] more details about MII and the signal timing can be found.

![Figure 10: Interface between PHY and FPGA](image)

Basically three signals are used while receiving data, in addition to the data bus that is called RXD. The signals are RXC which is the receive clock provided by the PHY, RXER indicates if an error has occurred in the transmission and RXDV informs the MAC layer that the data on the TXD bus is valid.
When a frame is received it starts with the PHY synchronising with the sender, this is during the transmission of the preamble bytes, this means that the number of received preamble bytes may vary. When the PHY is in sync \textit{RXDV} is set high to indicate that the data on the \textit{RXD} bus is valid. \textit{RXDV} is asserted high until the whole frame is received. New data on \textit{TXD} is valid at the rising edge of \textit{RXC}. If an error occurs during the transmission \textit{RXER} is asserted high.

Before transmitting a frame the MAC layer must first wait until the link is free to use, this is indicated by the \textit{CRS} signal that is asserted high while the link is active. If two devices are trying to transmit at the same time a collision will occur, this is indicated by the \textit{COL} signal being asserted high. If \textit{COL} is asserted high the result is that the entire frame will have to be transmitted again when the link is inactive. The MAC layer indicates that data provided on \textit{TXD} is valid with \textit{TXEN}. The PHY provides its own 2.5 MHz clock on \textit{TXC}, note that the clock is not in sync with \textit{RXC}. When the clock signal is high and the \textit{TXEN} is asserted high the PHY will transmit the data on \textit{TXD}.

\section*{5.1.2 MAC Layer}

The MAC layer will take care of receiving the data from the PHY and put it together into a complete ethernet frame and check that the checksum in the CRC field is right, thus controlling that the frame is transmitted without errors. When the whole frame is received the MAC layer must communicate this to the IP layer. In the same way the MAC layer will also handle the transmission of data, provided from the IP layer on TX ram, through the PHY, but instead of checking the checksum the MAC layer will have to add a checksum at the end of the frame.

\section*{5.1.3 IP Layer}

The IP layer is where the IP and ARP protocols is implemented. It is responsible for identifying that the ethernet frame has the right MAC and IP addresses. Then it decides what to do with the data and how to reply to the frame received.
5.2 HARDWARE SOLUTIONS

Ethernet and Internet communication is built in layers where each layer has a specific task. It is appropriate to build the hardware in the same way. At the top there is the IP layer communicating with the MAC layer. The MAC layer consists of two independent blocks, one block receiving frames and one transmitting frames. A block diagram of the interface can be found in figure [11].

An ethernet frame can be as big as 1500 bytes, so storing the frames in some type of memory is necessary. The internal dual-port BlockRAMs that is available on the Virtex-II is ideal for this. One BlockRAM can store 2048 8-bit words, and the dual ports make accessing and writing data with two units easy. Assuming that the rate the frames are sent with is low only two RAMs are used, one called RX RAM to store received frames in, and the other, called TX ram, to build new frames to transmit. This means that when one frame is received a new frame can not be received before the first frame is computed in a proper way by the IP layer. Since the address space of the PicoBlaze is constrained to 256 ports, a solution for addressing the RAMs 2048 addresses must be found. The fact that the controller will access both RAMs for reading and writing will also needs to be considered.

Figure 11: Top view of network interface
5.2.1 RX

Receiving a frame could be solved with a statemachine specific for this task but since PicoBlaze, which is a programable statemachine, is available, a PicoBlaze microcontroller is used to control the receiving part. Since it is a 8-bit microcontroller and the data in is 4-bit, some logic is required to receive two nibbles (a nibble is a 4 bit data word) and forge them into a byte. It is also important to put the nibbles together in the right order, the first nibble of a byte that is sent contains the least significant bits and the last nibble contains the most significant bits. To know when a frame is received the microcontroller must be able to read the RXDV signal. Since this design will run in a controlled environment and to keep the design small no CRC check is made, neither will the RXER signal be used.

The processor will store the received frame in the RX RAM and must be able to write to 2048 addresses and also read from address 000 (hex). Address 000 (hex) in RX RAM is dedicated for communication between RX and the IP part. If the least significant bit on address 000 (hex) is set the memory is occupied by a frame. If the bit is set, RX may not store a new frame in the memory. The bit is set by RX and reset by the IP side.

A register is used to forge the nibbles together into a byte. A small statemachine (figure [13])is used to control the register with an enable signal. The statemachine has three states. It leaves its start state if RXDV is high and the

![Figure 12: Schematic view of RX](image-url)
received nibble is “0101” or 5 (hex). It stays in the first state as long as RXDV is high and the received nibble is 5 (hex). The statemchine enters the third state if RXDV still is high and the received nibble is “1101” or D (hex). At the third state the enable signal is set high and the register starts saving nibbles. The reason that the first nibbles are discarded is that they are part of the preamble and SFD sequence that is made out of fourteen 5 (hex) nibbles and ended with a D (hex) nibble. By staying in the first state and discarding the preamble and SFD it does not matter how large part of the preamble sequence is received. When the ending D (hex) is sent the next nibble that is sent contains the least significant bits for the first byte.

When the enable signal is set high the register stores the nibbles received on each positive edge of RXC. The register is divided into two parts, a lower part storing the nibble with the least significcant bits and the upper part storing the nibble with the most significant bits. To store the nibbles in the right part a toggle bit is used that works as a second enable signal. It is used to enable the lower or the upper part of the register, its initial value is ‘0’ which enables the lower part of the register. When a nibble is stored in the lower part of the register the toggle bit is set to ‘1’ and vice versa when a nibble is stored in the upper part of the register.

The processor reads data from three different sources. It will read from the register, dataoutA on RX RAM to see if the memory is free to use, and then it will read RXDV to see if the entire frame is received and at last it will read

![Figure 13: Statemachine controlling input register and interrupts](image-url)
data from the register. This is solved with a MUX (input control in figure 12) that is controlled by \textit{port\_id}. When \textit{port\_id} is 01 (hex) the processor will read from \textit{RXDV}, at 02 (hex) it will read from RX RAM and in all other cases it will read from the register.

Using two different clocks in one design is always a problem. The register is updated with a frequency of 2.5 MHz and the processor runs at a speed of 100 MHz. It is critical that the processor reads a new value from the register at the right time so that no bytes are lost or stored twice. To guarantee that the read operations are made at the right time an interrupt (figure [14]) is generated by the statemachine and the register. The input to the interrupt register is a logical AND of the enable signal and the toggle bit. When the processor receives an interrupt it runs a routine to read the new value in the register and store it in RX RAM.

The \textit{port\_id} bus is 8 bits wide which makes it possible to access 256 different output ports, but to be able to access all 2048 addresses on the RX ram a bus that is 11 bits wide is required. This problem is solved by using two output instructions for each byte that is written to the memory. The first write operation only provides the most significant part of the address and the data on \textit{out\_port} is discarded, the last address provides the least significant 8 bits of the address and the data to be written to the memory.
This is done by using a statemachine (figure [15]) and a register. The register stores the three most significant bits and the statemachine generates control signals for both the register and RAM. The statemachine is clocked by the global 100 MHz clock.

The register (figure [16]) storing the three bits from the first operation is updated on the positive edge of read_addr, and the data is written to the ram when write_enable is high. By default the address sent to RX ram is 000 (hex), the address is changed to the assembled address when write_op is set high. This solution makes it possible to read from the first address of the memory with only one instruction and without adding more logic.
5.2.2 TX

The TX part of the MAC layer consists of two parts, a statemachine controlling the transmission and a CRC generator, both are clocked by TXC. The first address in TX ram is, as in RX ram, dedicated for communication. The least significant bit is set high if the ram contains a frame that is not yet transmitted. The bit is set by the IP layer and is reset by the statemachine when the whole frame is sent.

The whole frame including preamble and SFD is stored in the TX ram, each byte of the frame is stored in the order that it is transmitted. The statemachine uses addressA as a counter and increments addressA for each byte that is sent. On address 001 and 002 (hex) a value is stored by the IP layer. This is the address to the first byte in TX ram that will not be sent. When addressA is equal to this address the statemachine starts to send data from the CRC generator (see transition between state S8 and S9 figure 17). The least significant bits of the most significant byte of the CRC is sent first and so on. When the whole frame is sent the least significant bit on address 000 (hex) is set to ‘0’, and the statemachine returns to the start state.

![Figure 17: TX statemachine](image-url)
The statemachine is asynchronously reset by COL, this makes the statemachine automatically re-send a frame if a collision occurs. This does not follow the standard that includes an algorithm of how long time the device should wait until a new attempt to send the frame is done, but it works on a small network.

The CRC is not calculated on the preamble and SFD sequence, the statemachine keeps track of this and resets the CRC generator with CRC\_rst when the preamble and SFD sequence is sent with CRC\_rst (see state S6 and S7, figure 17). CRC\_rst is active low and resets the CRC generator to its initial state. It also uses an enable signal called CRC\_enable which is active high. This enable signal is used by the statemachine so that the CRC is not calculated on the same byte twice (see state S7 - S8 figure 17). The CRC generator uses 8 bit words to generate the CRC value, which can be accessed on CRC which is 32 bits wide (4 bytes).

The VHDL for the CRC comes from CRC Tool [12], which is a web based tool that generates VHDL code for CRC generators. Some changes were made to the VHDL code to fit the design.

5.2.3 IP

In the IP layer the microcontroller reads (input) from RX ram and stores (output) data in TX ram. The microcontroller will also read from address 000 (hex) in TX ram and write to address 000 (hex) in RX ram. This is solved with two instructions and a register storing the most significant part of the address, just as the out control in the RX part. Inputs will also be made by
using two input instructions to be able to access the entire RX ram.

A statemachine similar to the one used in the output control in the RX part is used (figure [ 18]. The statemachine will control both inputs and outputs, this is why it enters the first state when the read_strobe is high. The register is updated on positive edge op_one. To generate a write enable signal rw_strobe is used, but since rw_strobe is high on both input and output instructions some logic must be added, this is illustrated in figure 19.

Bit six and seven on port_id in the first input/output operation is used to indicate which of the rams that are targeted with an input or output instruction. If bit seven is high the operation targets the RX ram and if bit 6 is high the operation targets the TX ram.

To make the program easier to write, the design has been constructed so that only the special cases of an output operation to the RX RAM and an input operation from the TX RAM will have to use bit six and seven in the first in/out operation. Input operations from port 0000-07FF (hex) will read data from the according address on RX ram. Output operations on port 0000-07FF (hex) will write the data on out_port to the according address on TX ram. An input operation on port 4000 (hex) will read data from address 000 (hex) on TX ram, an output operation on port 8000 (hex) will have the result that the

![Figure 18: Statemachine for in and output control](image-url)

\[
\begin{array}{c}
S0 \\
\downarrow \\
\downarrow \\
S2 \\
\downarrow \\
\downarrow \\
S1 \\
\downarrow \\
\downarrow \\
S0 \\
\end{array}
\]

\begin{align*}
op_one &= '0' \\
rw_strobe &= '0'
\end{align*}

\begin{align*}
write_strobe &= '1' \text{ OR read_strobe } = '1' \\
op_one &= '0' \\
rw_strobe &= '1'
\end{align*}

\begin{align*}
write_strobe &= '1' \text{ OR read_strobe } = '1' \\
op_one &= '0' \\
rw_strobe &= '0'
\end{align*}
data on `out_port` will be written on address 000 (hex) on RX ram.

**Figure 19: In and output logic for microcontroller in IP layer**

### 5.3 PROGRAMS

#### 5.3.1 PROGRAM FOR PICOBLAZE IN MAC LAYER (RX)

With all the logic to support the microcontroller the program for the microcontroller is kept small. What it basically does is that it read data from address 000 (hex) on RX RAM, this is done by reading on port 01 (hex) when an interrupt occurs. If the least significant bit is ‘0’ it reads data from the receive logic and stores it in RX RAM. After an interrupt the controller waits until a new interrupt should have occurred and then reads the value of `RXDV` on port 01 (hex). If RXDV is ‘0’ the frame is received and the processor indicates this on address 000 (hex).
5.3.2 PROGRAM FOR PICOBLAZE IN IP LAYER

In the program (figure 20) for the IP layer ARP and IP protocols are implemented. The protocols are constrained to reply to ARP and ICMP echo requests. Replying successfully on a ARP and a ICMP shows that it is possible to reply and send messages higher up in the protocol hierarchy.

When a frame is received a test to establish if the MAC address is right or if a broadcast has been made. When this is done it is decided whether the data in the Ethernet frame is an ARP packet or an IP packet, deciding which subroutine to use. The subroutines check that the frame contains the right IP address and that it is a request, when it is established that all addresses are right and that an answer is demanded a new frame is built up in TX ram.

Large parts of the packets can be reused and copied from RX ram, but some things like source address and type must be changed and are not copied. For the ICMP packets the IP checksum does not have to be calculated since the only changes in the IP header is that some fields are swapped. However the
ICMP checksum must be calculated since the type field must be changed.

5.4 FUNCTIONAL VERIFICATION

A test bench was created to help simulate the design and verify the functionality. A file containing an entire Ethernet frame is used as an input. The output signals were recorded and stored in another file. Since it was difficult to see what data had been stored in the RX and TX rams the data on the memory buses was also recorded. This together with a diagram of the RX and TX ram and their contents for the different packets turned out to be a great help to find errors in the design and program code.

The function of the whole design was never verified. Finding out exactly what data and when it should be sent to the PHY and then compare it to the simulation result would have required too much time. Instead the design was considered to be ready for a test on the FPGA when the TX ram was loaded with the right data and the outputs to the PHY looked like they should.

When it came to simulating and debugging the programs pBlaze was not useful because of the special ways used to address the memories. It proved to be effective to use ModelSim and the microcontrollers internal signals for problem solving. An advantage was that this way the memories were loaded with the right data from the test bench.

As figure 21 shows it is possible to see what address in the program memory

```
11C  0C002           SUB s00, 02
11D  0E100           SUBCY s01, 00
11E  35D0D          JUMP NC, loopccs[10D]
11F  000FF          LOAD s00, FF
```

**Figure 21: Waveforms in ModelSim and according program code**

As figure 21 shows it is possible to see what address in the program memory
that is accessed and to see where in the program the controller is. It is also possible to see the status of the internal flags and registers and the result from an ALU operation.

5.5 PHYSICAL VERIFICATION

Physical verification of the design was made by connecting the experiment board to a 10 mbit Ethernet network consisting of a 10 Mbit hub and computer running Windows. The computer was used to send ARP and ICMP requests with the ping tool. On the computer a program called ‘Ethereal’ [13] was used to log the traffic on the network and view the data in the frames that was received.

An error occurred when synthesising the design during the NGDbuild process. A lot of time was spent finding the problem and getting around it but nothing worked until Peter Johansson did the synthesis on his computer with his program settings. It seems like some of the settings on the computer used were wrong, when the settings were copied from Peter’s computer the synthesis worked.

In the beginning there was no response on either ARP or ICMP requests. It turned out to be a program error. After changes in the program, replies were received from the experiment board. There were some errors in the packets so the replies was never accepted by ‘Ping’ but were detected with the help of ‘Ethereal’. After some adjustments in the program code the errors was corrected and the ping tool accepted the replies as valid. Still some errors in the program code remain, the generation of the ICMP checksum does not work properly when the data in the end of the ICMP packet is an odd number or when the size of the data field exceeds 202 bytes. Unfortunately there was no time to correct the errors.

5.6 RESULT

The implementation shows that it is possible to implement a network interface with a PicoBlaze microcontroller. The design is incomplete, but with minor changes in the design and a some programing of the microcontroller in the IP layer the functionality would be increased.

In addition to the errors in the program code more protocols need to be added
to get more functionality. By adding the TCP and HTTP protocols it would be possible to run a small web server on the experiment board. These changes are possible to do in the program code, the limiting factor is the 1024 instructions that the program memory can store, the program code implementing ARP and ICMP consist of 295 instructions. Among the changes needed in the design is a way to use the CRC generator to check the FCS field on received frames. Right now RXER is not used, it would be possible to check if any errors occurred at the same time that the status of RXDV is checked.

**Table 5.1: Resources used reported by NGDBuild**

<table>
<thead>
<tr>
<th>Part of design</th>
<th>Slices used</th>
<th>BlockRAMs used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whole design</td>
<td>307</td>
<td>4</td>
</tr>
<tr>
<td>RX</td>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>TX</td>
<td>97</td>
<td>-</td>
</tr>
<tr>
<td>PicoBlaze</td>
<td>85</td>
<td>1</td>
</tr>
<tr>
<td>TX statemach-</td>
<td>58</td>
<td>-</td>
</tr>
<tr>
<td>iene</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRC-generator</td>
<td>33</td>
<td>-</td>
</tr>
</tbody>
</table>

Using PicoBlaze in the RX part does not seem to be the optimal solution according to table 5.1. Even though the tasks are not exactly the same it is a good guess that a state machine in the RX part would use approximately the same resources as in the TX part. The PicoBlaze is larger than the TX statemachine and it needs some logic and registers to support it. This uses resources and takes time designing. However it shows that PicoBlaze could be a good replacement for statemachines when they grow in size. Such is the case in the IP layer.
CONCLUSION

PicoBlaze is a 8-bit microcontroller developed by Xilinx to be used on their devices. It is highly optimized for their products and requires only a small amount of resources. It is easy to use and the documentation is extensive and well written.

The largest advantage of PicoBlaze is that it is powerful in comparison to the resources that it uses on the FPGA, it uses 85 slices and a BlockRAM and is capable of 40-70 MIPS. The other advantage is that it is easy to use and has good documentation. The main disadvantage encountered in this work has been its limited address space, also compare and test instructions have been missed. Another limitation in the microcontroller design is the program memory being capable of storing 1024 instructions, even though it was not required in this work it might be a limitation for other applications.

To control a smaller flow a custom made statemachine might be preferred instead of using PicoBlaze. Even though relatively small state machines use almost as much resources as PicoBlaze, not including the BlockRAM used as program memory, PicoBlaze often needs some kind of logic to manage in and output of data. This adds to both time used for designing and resources used in the design. But as soon as the flow being controlled is increased and more conditions have to be tested PicoBlaze comes will show its advantages.

The implementation of the network interface shows that PicoBlaze could work as a small web server or something similar with small changes of the design and more protocols implemented in the microcontroller program.
REFERENCES

Books:

   John Wiley & Sons, Ltd, 2000

[2] Sjöholm, Lindh, VHDL för konstruktion, fjärde upplagan,
   Studentlitteratur, Lund, 2003
   ISBN 91-44-0247-1

Data sheets, application notes:

[3] Data sheet BCM5221, 05/29/01

   Available by request to Ken Chapman, ken.chapman@xilinx.com

   Insight MEMEC Dec. 2001

    Devices, ver 2.1

[7] XAPP387 PicoBlaze 8-bit Microcontroller for CPLD Devices, ver 1.1

[8] XAPP627 PicoBlaze 8-bit Microcontroller for Virtex-II Series Devices, ver 1.1

[9] Xilinx Reference Design License
Evaluation of PicoBlaze and implementation of a network interface on FPGA

URLs:

    http://electronics.howstuffworks.com/microcontroller.htm, (21/05/04)

    http://www.geocities.com/SiliconValley/2072/bit33.txt, (05/11/04)
    Description on how to calculate a IP checksum, uses a practical example

[12] CRC tool
    http://www.easics.be/webtools/crctool

[13] Ethereal, tool for analysing frames sent on an Ethernet network
    http://www.ethereal.com/

[14] Fairhurst Garry, 01/01/01, Address Resolution Protocol (arp)
    http://www.erg.abdn.ac.uk/users/gorry/course/inet-pages/arp.html, (05/03/04)
    Good overview of how ARP works and the structure of an ARP message.

    http://www.mediatronix.com

[16] PicoBlaze Soft Processor homepage

    http://www.faqs.org/rfcs/rfc791.html, (04/20/04)

    http://www.faqs.org/rfcs/rfc792.html, (04/20/04)

[19] Ping, 04/20/04
    http://www.freesoft.org/CIE/Topics/53.htm
    Overview of how ping programs work and uses ICMP.

[20] Pidgeon Nick, How Ethernet works
    http://computer.howstuffworks.com/ethernet.htm, (21/05/04)

[21] Xilinx support, TechXlusives
    Ken Chapman, 07/28/02, Creating embedded Microcontrollers, (03/18/04)
    Kris Chaplin, 05/08/03, Reconfiguring BlockRAMs, (03/22/04)
    http://www.xilinx.com/xlnx/xweb/xil_tx_home.jsp
    Reports on how to use Xilinx devices effectively written by Xilinx employees are published here.
Appendix A - Instruction set

sXX and sYY refer to the controllers internal registers. kk represents a constant value in range 00 to FF (hex). aa represent a line label or an address in range 000 to 3FF (hex) for the Virtex II version. In the end of the list the assembler directives are listed. See the application notes for a more consistently description of the instructions.

Program control:

JUMP aa          Unconditional jump
JUMP Z, aa       Conditional jump if ZERO
JUMP NZ, aa      Conditional jump if NOT ZERO
JUMP C, aa       Conditional jump if CARRY
JUMP NC, aa      Conditional jump if NOT CARRY

CALL is similar to JUMP but it also stores the program counter on the stack, used to call subroutines. CALL has no effect on the flags

CALL aa
CALL Z, aa
CALL NZ, aa
CALL C, aa
CALL NC, aa

RETURN complement CALL. RETRUN moves the latest saved value on the stack to the program counter. RETRUN does not effect the flags.

RETURN
RETURN Z
RETURN NZ
RETURN C
RETURN NC

Logic:

LOAD sXX, kk     Loads sXX with kk
AND sXX, kk      Bit-wise AND between sXX and kk
OR sXX, kk       Bit-wise OR between sXX and kk
XOR sXX, kk      Bit-wise XOR between sXX and kk
LOAD sXX, sYY  Loads sXX with sYY
AND sXX, sYY  Bit-wise AND between sXX and sYY
OR sXX, sYY  Bit-wise OR between sXX and sYY
XOR sXX, sYY  Bit-wise XOR between sXX and sYY

Arithmetic:

All arithmetic instructions effect ZERO and CARRY

ADD sXX, kk  Addition of sXX and kk
ADDCY sXX, kk  Addition of sXX, kk and CARRY
SUB sXX, kk  Subtraction of sXX and kk
SUBCY sXX, kk  Subtraction of sXX, kk and CARRY

ADD sXX, sYY  Addition of sXX and sYY
ADDCY sXX, sYY  Addition of sXX, sYY and CARRY
SUB sXX, sYY  Subtraction of sXX and sYY
SUBCY sXX, sYY  Subtraction of sXX, sYY and CARRY

Shift and rotate:

All shift and rotate instructions except SR1 and SL1 sets ZERO if all bits in result are 0 and reset ZERO in all other cases.

SR0 sXX  Shift in 0 in to the left, right-hand bit put in CARRY
SR1 sXX  Shift in 1 in to the left, right-hand bit put in CARRY
Sets ZERO to ‘0’
SRX sXX  Shift in left hand bit to the left, right-hand bit put in CARRY
SRA sXX  Shift in CARRY to the left, right-hand bit put in CARRY
RR sXX  Rotate righthand bit to the left, right-hand bit put in CARRY
SL0 sXX  Shift in 0 in to the right, left-hand bit put in CARRY
SL1 sXX  Shift in 1 in to the right, left-hand bit put in CARRY.
Sets ZERO to ‘0’
SLX sXX  Shift in right-hand bit to the right, left-hand bit put in CARRY
SLA sXX  Shift in CARRY to the right, left-hand bit put in CARRY
RL sXX  Rotate lefthand bit to the right, lefthand bit put in CARRY
Input/Output:

INPUT sXX, kk  Reads the value of port kk to sXX
INPUT sXX, (sYY)  Reads the value of port sYY to sXX

OUTPUT sXX, kk  Writes the value of sXX to port kk
OUTPUT sXX, (sYY)  Writes the value of sXX to port sYY

Interrupt:

RETURNI concludes an interrupt service routine. It loads the last address to the program counter and returns preserved CARRY and ZERO.

RETURNI ENABLE  Returns from subroutine and enables future interrupts
RETURNI DISABLE  Returns from subroutine and disables future interrupts

ENABLE INTERRUPT
DISABLE INTERRUPT

Assembler directives:

ADDRESS aa  Forces the assembly process to aa
CONSTANT label, kk  Assigns constant kk to label
NAMEREG sXX, label  Assigns sXX to label. After the NAMEREG instruction sXX can only be accessed by the label name
Evaluation of PicoBlaze and implementation of a network interface on FPGA
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlaget hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Robert Mattsson