THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

FPGA packet filter with Ethernet MAC and web server using a RISC-V softcore processor

Thesis

Matthew Gilpin

45801600

2023

The University of Queensland

School of Electrical Engineering and Computer Science

Matthew John Gilpin
m.gilpin@uq.net.au

September 28, 2023

Prof Michael Bruenig
Head of School
School of Electrical Engineering and Computer Science
The University of Queensland
St Lucia, QLD 4072

Dear Professor Bruenig,

In accordance with the requirements of the degree of Bachelor of Engineering (Honours) in the division of Electrical Engineering, I present the following thesis entitled

"FPGA packet filter with Etherent MAC and web server using a RISC-V softcore processor".

This work was performed in under the supervision of Dr. Matthew D'Souza. I declare that the work submitted in the thesis is my own, except as acknowledged in the text and footnotes, and that it has not previously been submitted for a degree at the University of Queensland or any other institution.

Yours sincerely,

_____-

Matthew John Gilpin

Abstract

Start this section on a new page [this template will automatically handle this].

The abstract should outline the main approach and findings of the thesis and normally must be between 300 and 800 words.

List of Abbreviations

Abbreviations	
IoT	Internet of Things
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
PF	Packet Filter
MAC	Medium Access Control
ISA	Instruction Set Architecture
ASIC	Application Specific Integrated Circuit
SoC	System on Chip
TRL	Technology Readiness Level
IP	Intellectual Property
PHY	Physical layer
RMII	Reduced Media Independent Interface
CRC	Cyclic Redundancy Check
FIFO	First-In First-Out
LSB	Least Significant Bit
FSM	Finite State Machine
CLI	Command Line Interface
GUI	Graphical User Interface
RTOS	Real Time Operating System

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In a technology era of increasing numbers of cyber attacks and record number of connected devices, ensuring these devices operate safely and securely is paramount. The Australian Cyber Security Center (ACSC) received in excess of 76,000 cybercrime reports and growing in the 2021-22 financial year [1]. The growing trend of Internet of Things (IoT) will provide more opportunity for black hats (malicious attackers). IHS Markit estimates 125 billion IoT devices will be connected by 2030 [2]. This proliferation of IoT devices necessitates robust and adaptable security measures to counter the evolving threats posed by malicious actors.

To manage the surge of IoT devices, a shift to edge computing has emerged in favour of the traditionally more centralised cloud computing architectures. Edge computing as [3] puts it, is the paradigm which involves the computation and analysis of data at the *edge* of the network to be as close as possible to the source of the data. This has many advantages including: lower latency, lower bandwidth requirements, enhanced availability, energy efficiency, improved security and privacy [3]. Consequently, smaller and more efficient computers can be deployed at the edge/perimeter of these networks [4].

Just like any other computer connected to the broader network, edge networks must also be safeguarded from malicious bad actors. Field programmable gate arrays (FPGAs) offer the flexibility of custom hardware that can be designed to incentivise low latency, high throughput yet efficient wire-speed firewalls. While current hardware firewalls exist in todays markets, they often come at a cost and high power usage rendering them unsuitable in edge networks. To address this, this thesis proposal attempts to design a FPGA firewall that fulfils these criteria.

Chapter 2

Literature review

Research into existing literature has been conducted on multiple topics in relation to the project. These topics include, field programmable gate arrays, packet filter firewalls, RISC-V processors, Ethernet MAC, webs servers and network stacks.

2.1 Field Programmable Gate Arrays

First introduced by Xilinx in 1984, field programmable gate arrays (FPGAs) allowed for large custom logic designs to be recognised without the need for expensive application specific integrated circuits (ASICs). More importantly, FPGAs did not suffer from the same scalability issues that programmable array logic (PAL) encountered and has allowed for larger and more complex designs [5].

A big advantage to custom logic is the ability to create highly parallelised designs with lower latencies than software based serialised algorithms. This comes down to having a great degree of freedom when it comes to designing the architecture and ability to optimise for specific tasks. As such, FPGAs have became ubiquitous in both digital signal processing and for accelerating an assortment of heterogeneous computing architectures and processes [6]. System on chip (SoC) design with custom hardware acceleration modules is an active area research. As [6] points out, there is a focus towards using both hardware and software in *edge* devices due to growing numbers of IoT devices.

Several papers, [7] [8] [9], have proposed a range of other related FPGA based firewalls that have different properties and focus on different optimisations. The key benefit to these firewalls is their high performance - namely, low latency, and high throughput. Article [7] proposed an Ethernet firewall using LwIP (A TCP/IP stack) with five-tuple binding (the five filtered parameters in packet filters) to achieve a throughput of 950Mbps with a latency of 61.266us. A conference proceeding in 2000 [8] used a comparator unit to check the fields of the IP headers obtained a filtering rate of 500,000 packets per second.

The enabling concept behind the above FPGA based firewalls is SoC design which involves integrating multiple components into a single package, or in this case a single FPGA. Often these will include small softcore microprocessors and some custom hardware such as the Ethernet or packet

filtering like the proposed packet filters in [7]. Having a microprocessor in the FPGA design can significantly reduce the complexity of the design and allows for quick and easy development in software instead of hardware [10]. In FPGA design, softcore processors are configurable and can be modelled in a hardware description language (HDL) which can then be synthesised onto ASICs or FPGAs hardware [10]. There are several softcore processors available for FPGA designs including ARM Cortex, Nios II, MicroBlaze, and RISC-V.

While recently the royalty free RISC-V based cores have been popular amongst many SoC designs, other older processors are still common in the literature. The two big FPGA vendors, Xilinx (now AMD) and Altera (now Intel) have their own RISC based softcores. As an example, Janik et al. [11] used Xilinx's MicroBlaze processor as a media converter between optical (SFP interface) and copper (Ethernet) networks. Likewise, Altera's Nios II can be found in a variety of research papers including an embedded web server which significantly simplified the design [12].

2.2 Packet Filter Firewall

Usually, the first line of defence against bad actors, firewalls play a vital component in computer networks and as such can become vastly complex. In essence, the job of a firewall is to isolate and restrict access to an internal network from an external one to increase security [13].

There are several types of firewalls such as packet filters (PF), stateful packet firewalls and application firewalls [14]. Traditional PFs are considered as stateless and filter exclusively on the fields in the network (layer 2) and transport (layer 3) layer headers [14]. Such fields include IP addresses, port numbers and protocol type.

Due to this, PFs are inherently simple and efficient. Consequently, they are widely available and can be either implemented in software or in hardware [13]. The book, [13], also highlights some inherent flaws with PFs which include not being able to suppress sophisticated attacks and in some cases, can be challenging to properly configure. More advanced firewalls can perform deep packet inspection which explore the contents of the higher layers to better evaluate a packets true intention [14].

While firewalls such as *iptables* in Linux are software based, hardware acceleration can vastly improve the performance of a packet filter. As stated in section 2.1, hardware acceleration allows for parallelised algorithms to be executed independently of a central processing unit (CPU). Wicaksana and Sasongko, [15], proposed a packet classification engine as shown in figure 2.1. To obtain a fast and reconfigurable packet classifier, the authors of [15] used a hierarchical tree-based algorithm that inspects the multidimensional fields of the IP header through the use of parallel decision trees.

Essentially, the architecture in figure 2.1 employs memory to store the ruleset and uses a multiplexer and a comparator to evaluate each of the fields in the header. As an safegaurd, the authors opted for a *default-deny* ruleset to prevent any unwanted traffic.

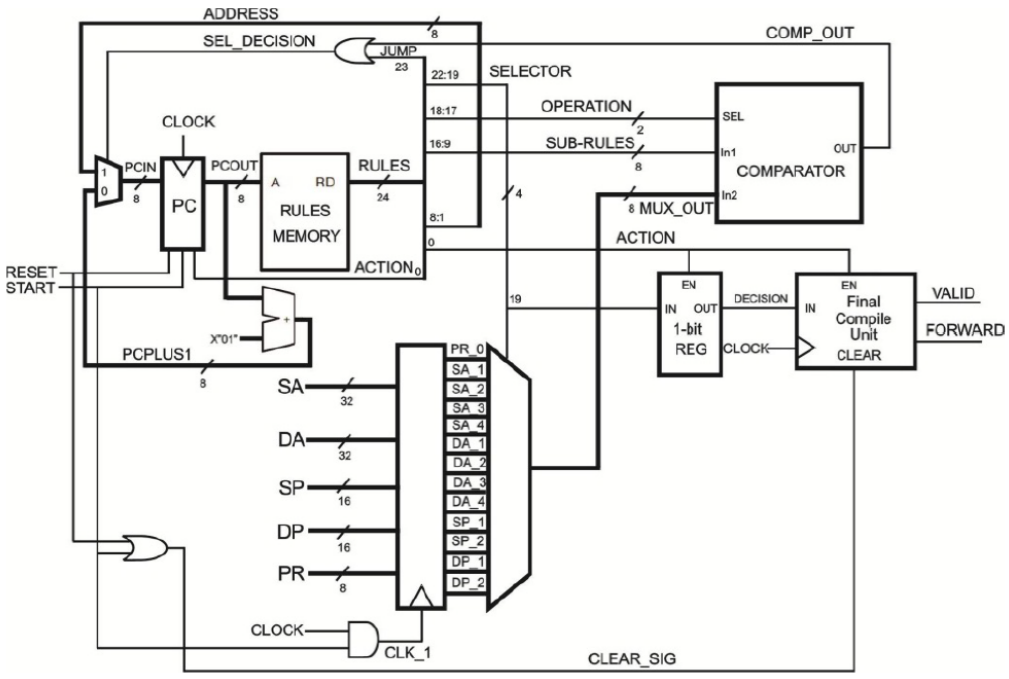


Figure 2.1: Packet classifier [15]

Wasti [16] presents several other classification algorithms for both hardware and software packet filters. *'Sequential matching'* provides the most trivial solution as it matches each rule to the incoming packet. While simple, this design has scalability issues as more rules get added. Another method proposed in [16] is by using a *'Grid of tries'* which uses tries (a type of tree datastructure) to help pattern match the packets, but fails to extend to multiple fields. Hardware algorithms using *Ternary CAMs* (stores words with 3-valued-digits - namely '0', '1' and '*') and *Bit-parallelism* were also discussed. Both of these exploited the parallelised nature of hardware design. One limiting factor with the classification methods cited in [16] is their configurability and expandability.

2.3 RISC-V processor

In the world of processor architectures, there are four major families, namely AMD64, x86, ARM and RISC-V. The two former instruction set architectures (ISA) are apart of the complex instructions sets (CISC) and are found in the majority of computers. ARM and RISC-V have a reduced instruction set compared to the CISC family and subsequently fall under the RISC family and are ideal for low power microprocessors [17].

RISC-V is an open and royalty free ISA and as a result, a plethora of softcore based custom implementations have been designed [18]. Consequently, there is an abundance of articles delving into RISC-V from evaluating the ISA [19] to creating multicore architectures [20]. A 2019 paper, [18] evaluated a variety of different RISC-V softcore processors. RISC-V International have also published a list[1] of different RISC-V implementations that have a unique architecture ID. The majority of these are either written in a HDL for either application specific integrated circuits (ASICs) or FPGAs. The

[1] See: https://github.com/riscv/riscv-isa-manual/blob/master/marchid.md

NEORV32 RISC-V softcore processor is written purely in vendor-agnostic VHDL and importantly has a considerable amount of documentation.

Being a softcore processor, control is given over which modules are implemented. Some basic features of the *NEORV32 RISC-V* include UART, SPI, and GPIO interfaces [21]. The datasheet, [21], also mentions that it supports a *'Wishbone b4 classic'* external bus interface. A Wishbone B4 (or just 'wishbone') interconnection is designed specifically to connect modular pieces of hardware together on a SoC into the memory mapped 32bit address space in the processor [22]. This approach has the benefit of not needing to create custom instructions for the microprocessor.

2.4 Ethernet MAC

First introduced in 1983, the IEEE 802.3 standard [23], more commonly known by the name of 'Ethernet', defines the *'Medium Access Control'* (MAC) protocol amongst other things for two or more devices to communicate over a network. This standard is just one part in the layered network models such as the OSI model or TCP/IP model, namely the network layer - layer 2.

A core function of the Ethernet MAC is to attach the required MAC layer headers to the head and tail of the layer 3 payload to create an Ethernet packet. The fields in an Ethernet packet can be seen in figure 2.2.

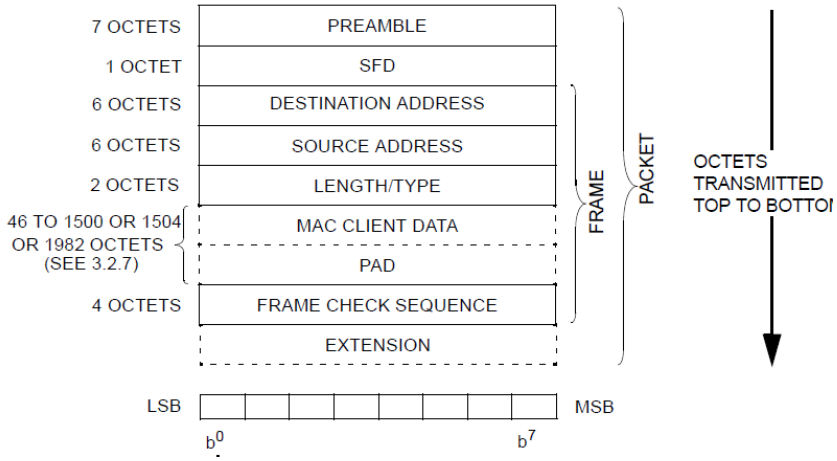


Figure 2.2: MAC layer headers [23]

After the packet has been constructed, the data is forwarded out to the physical (PHY) layer least significant bit (LSB) first [23]. Typically, a PHY management chip is used to handle the physical layer channel encoding amongst other things. These PHY chips often can be interfaced with the media independent interfaces such as MII, RMII, GMII and RGMII [24]. The reduced media independent interface (RMII) is one of these standards defined in [23] and consists of a reference clock, 2 bit wide transmit (TX), 2 bit wide receive (RX) lines and a few other supplementary signals as defined in the LAN8720A datasheet [25].

The MAC layer itself is usually implemented in hardware as it has several advantages over a software implementation. The core reasons behind this are due to parallelised nature of FPGAs

and that parts of the MAC can operate independently [26]. One key example is the calculation of the frame check sequence (FCS in figure 2.2). The FCS for Ethernet is a 32bit cyclic redundancy check (CRC) [23] and in addition to Etherent, the CRC32 can be found in an extensive amount of applications. As such, research has been conducted into parallelising the calculation. Noteably, Mitra and Nayak [27] proposed a low latency parallelised architecture for FPGA design on CRC32. As a result, packets can be assembled faster and offload additional processing burden from the CPU.

Numerous articles [24] [28] [29] can be found about Ethernet MACs implemented on FPGAs each with a slightly different approach. Fundamentally though, as best highlighted in [24], a simple way of implementing a MAC is by employing a finite state machine (FSM) to set the required fields. Another technique found in these articles is the use first-in first-out (FIFO) buffers to cross clock domains. This is a common technique used in FPGA design as it allows you to have the packet assembly logic at a much higher clock rate than the output RMII reference clock speed [28].

In addition to the papers, there are a plethora of intellectual property (IP) blocks for xMII interfaces in HDL which have their own benefits and drawbacks. Some freely available HDL modules for Ethernet MACs can be found in both a complete [1] [2] [3] and incomplete state [4].

2.5 Web servers and network stacks

Almost all firewalls need to be configured with a ruleset which can be configured in two common ways, using a command line interface (CLI) or by a web-based graphical user interface (GUI). Before a web server can be realised, the network stack (Layers 3, and 4) need to be established since a web server operates at the application layer (layer 4). As embedded platforms are resource limited, special precautions need to be taken into consideration when it comes to memory and resource usage [30].

Article [7] investigated using the open source lightweight IP (LwIP) network stack as a mechanism for interfacing with the firewall. The LwIP library is a popular lightweight TCP/IP stack which has been investigated in a plethora of research papers and projects [31] [30]. Often these papers run LwIP on real time operating systems (RTOS) such as FreeRTOS or Zephyr.

FreeRTOS is a leading RTOS for microprocessors and is distributed freely under the MIT license. As an RTOS, it provides an abstraction to the hardware that allows for multitasking and brings other OS-Like features to embedded systems. Several ports are available including one for RISC-V.

FreeRTOS also provide their own TCP/IP network stack called *FreeRTOS-Plus-TCP* which includes a HTTP web server example and is much newer than LwIP. Consequently, less research can be found apart from existing documentation. The library aims to provide a threadsafe Berkley sockets API and network stack supporting multiple protocols such as DHCP, DNS, TCP, and UDP [32]. LwIP is not threadsafe and typically suffers from memory issues as found in [30].

[1] See: https://github.com/yol/ethernet_mac
[2] See: https://github.com/alexforencich/verilog-ethernet/
[3] See: https://opencores.org/projects/ethernet_tri_mode
[4] See: https://github.com/pabennett/ethernet_mac

Chapter 3

Design overview

This chapter details the design decisions and steps taken to complete the project. The project itself can be broken down into three main areas: hardware, firmware and software.

3.1 Hardware

3.1.1 FPGA

Digilent, parented by National Instruments, make a wide range of Xilinx based FPGA development boards and test equipment. In this project, the Digilent Nexys A7-100T FPGA development board (figure 3.1) was used due to it's availability and features including: a Xilinx Artix 7 100T FPGA (part number XC7A100T-1CSG324C), LAN8720A 100MBit/s RMII PHY, micro SD card slot and PMOD (auxiliary outputs) among other IO.

Xilinx has multiple FPGAs in their 7-series lineup with different target audiences. The Artix-7 family is optimised for low power designs with high logic throughput. The XC7A100T has 101,440 logic cells, 4,860Kbits of Block RAM (BRAM) and 240 DSP blocks [33]. There is a varient of the Nexys A7 FPGA board that consists of a XC7A50T FPGA (fewer resources), but ultimately the XC7A100T variant was used due to its larger amount of resources.

Importantly, there are four ways this FPGA can be configured (essentially *'programmed'*), at each power on cycle using JTAG, nonvolatile SPI flash, microSD card or using a USB stick through the HID interface. These modes are switchable using jumpers, JP1 and JP2, on the board. The JTAG interface is ideal for testing and as such it was used throughout development process, while storing these configurations on a microSD card was used once the design was solidified.

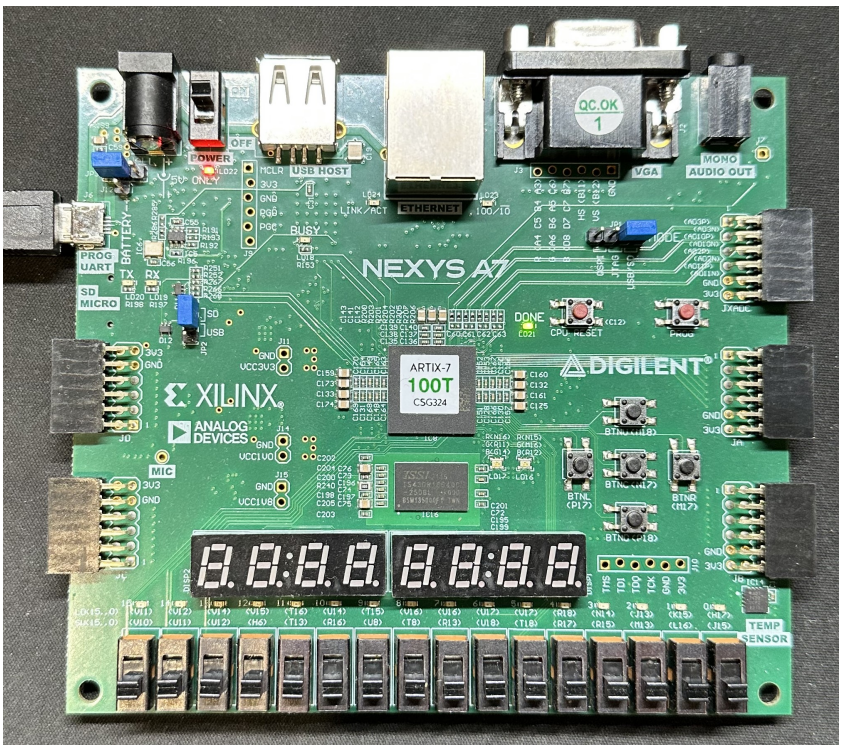


Figure 3.1: Digilent Nexys A7 FPGA development board.

3.1.2 MicroSD card

After the FPGA has been configured using the microSD card, the onboard microcontroller on the Nexys A7 board power cycles the microSD card and relinquishes control of the bus. On power up of the RISC-V softcore processor, it has full control of the card.

The selected MicroSD card for use in this project is the Patriot LX Series 32GB card, seen in figure 3.2. SD cards, like the patriot card have 2 modes of operation: native SDIO mode and SPI mode. While the native SDIO mode allows for higher speeds, it adds complexity to the design. As the files stored on the SD card are minimal ($< 100KB$), to keep things simple, the microSD card was connected in SPI mode.

Figure 3.2: MicroSD card used in project.

The files that were stored on the microSD card include the bitstream file for the FPGA itself and web assets for the webserver. While the bitstream file needed to be at the root directory of the filesystem, the web assets were stored in their own folder structure to help segregate the files.

3.1.3 System on Chip

A benefit to using an FPGA is that full control is given to the overall system design. At the heart of the SoC, a NEORV32 softcore processor[1] controls the hardware and runs the higher layers of the network and webserver tasks.

The NEORV32 processor is RISC-V compatible and designed by GitHub user *stnolting* and is highly configurable. In this design, seen in figure 3.3, the Wishbone, SPI, UART and external interrupts interfaces were enabled and configured. In addition to these, the M extension (Multiplier) was configured to use the DSP blocks to reduce the number of LUTs needed to handle multiplication in the core.

The Wishbone B4 classic bus is an open source interface that allows for multiple bits of hardware to connect and communicate together. In this project, the bus is 32bits wide and clocked at 50MHz, giving a bandwidth of $32 \times 50 \times 10^6 = 1.6 \times 10^9 bit/s = 1.6 Gbit/s$. Due to its relatively high bandwidth, it was used to connect the MAC with the NEORV32 as packets of 1500 bytes would need to be transferred quickly to not bottleneck the 100Mbit Ethernet interface. In addition to this, the MAC had an interrupt line to the NEORV32 processor to notify it when a packet has been recieved and ready for processing in the higher layers. This connects into the XIRQ lines which creates a fast interrupt request by firing a mcause trap event (RISC-V terminology).

Serial Perhipheral Interface (SPI) was used to connect to both the MicroSD card and Packet classifier. These are comparatively low speed and low priority perhipherals and so do not require a high speed interface. UART was connected to the onboard serial to USB converter chip for CLI commands and debugging.

[1] See: https://github.com/stnolting/neorv32

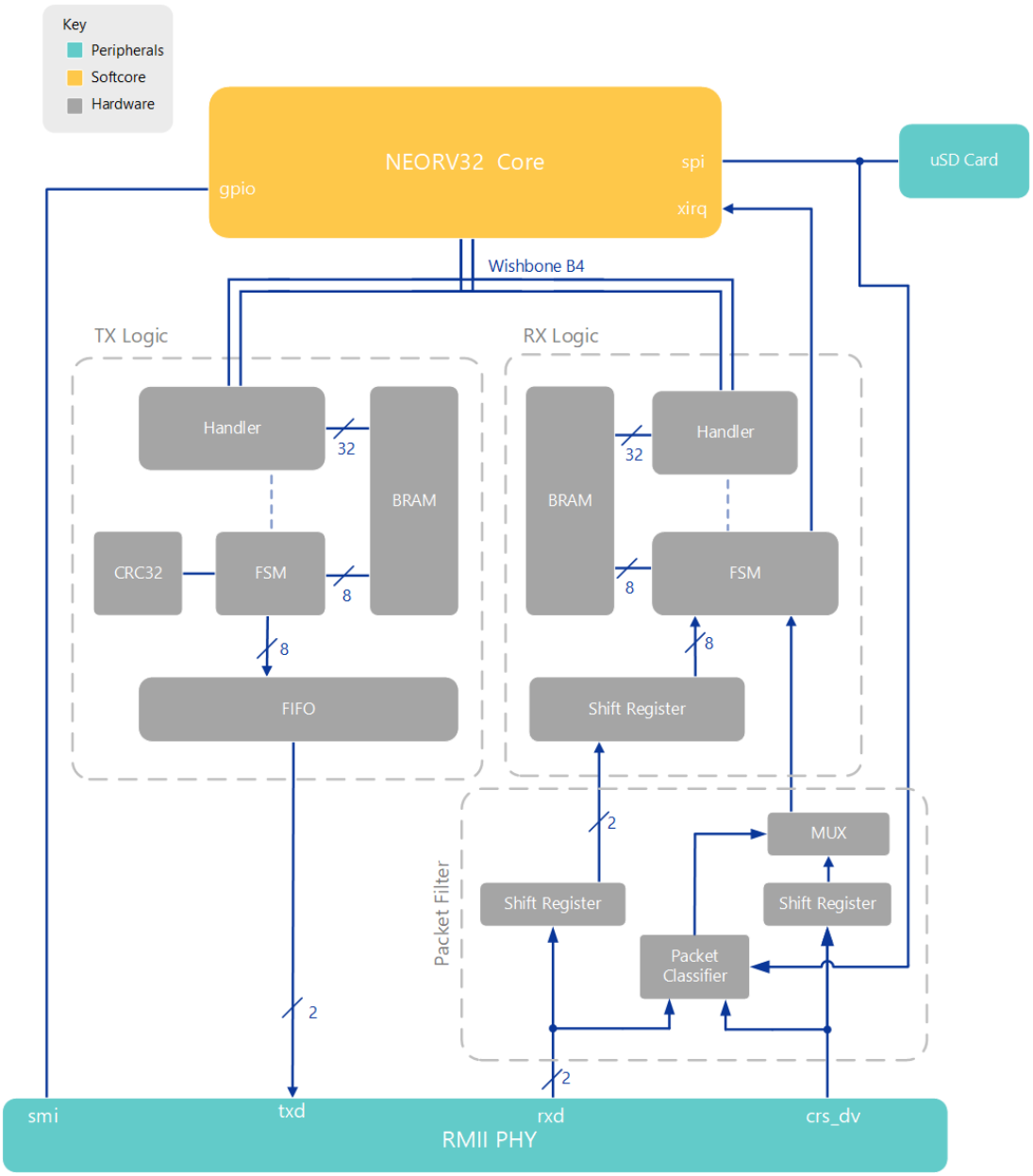


Figure 3.3: System on Chip high level architecture.

Since the design is only concerned about incoming filtering, the packet filter was only placed between the RMII PHY and the MAC. By filtering at the RMII interface level, the Ethernet MAC is indifferent to the filter and ultimately doesn't care about it. This allows for a simpler modular design compared to integrating the filter in the MAC hardware. This filter consists of classifier (discussed in section 3.1.5) which would determine whether to forward or block an incoming packet.

To do the filtering itself, a shift register can be used to essentially delay the inputs from the RMII PHY until the packet classifier has determined whether to forward or drop the packet. A simple MUX can then be used to either allow the packet to enter the wishbone MAC or to not.

As the hardware in the MAC only processes the input if crs_dv is high, we only need to gate the crs_dv and can always have the rxd lines always attached. However, these should also go through a shift register.

By doing this, it means that we can operate the filter at wirespeed with the only downside is the

extra latency that the shift registers bring. The delay that these registers add to the latency can be found to be $T_{latency} = N \times T_{clk}$ where N is the size of the registers. In the design a size of 224 ticks was used. This is because at a minimum, the packet classifier needed to input a maximum of $22 + 24 + 4 = 50$ bytes (22 for MAC headers including preamble, maximum 24 bytes for IP header and 4 bytes for the TCP/UDP headers (only need to check the source and destination port)) need to be processed. While this is enough, a margin of 6 bytes was arbitrarily chosen to allow propagation of other parts of the design to have taken effect. This gave 56 bytes, where each byte takes 4 clock cycles to input into the MAC meaning that we need a register size of $56 \times 4 = 224$ for the data to propagate to the end of the registers after the packet classifier has determined whether to drop or allow the packet. Importantly this does not effect the speed/bandwidth of the connection.

It is assumed that any traffic leaving from the device is safe and trusted. In a larger network where there are several devices behind the firewall, it may be desirable to also have a packet filter on the output.

3.1.4 Ethernet Media Access Controller

The advantage of using an FPGA is that custom hardware can be designed for specific tasks. In this design the MAC layer was done in hardware to free up the microprocessor by handling the lower level logic.

This MAC was implemented as a memory-mapped perhipheral which used the MCU's Wishbone B4 classic interface. This then made it easily accessible over the memory address space of the MCU.

The hardware can be broken down into two main sections: the transmit logic and receive logic.

In the receive logic, there are two main functions, one that stores the incoming frame into BRAM and then another to interface the BRAM with the Wishbone interface. On the input side the data is shifted into a 8bit wide shift register - shown in figure 3.3. While crs_dv is asserted, after every four clock cycles (modulo four since 2 bits is received at a time) the contents of the shift register is stored into BRAM. After each byte has been added to BRAM, a counter is incremented to store the next byte in the next index. The end of the packet is signified when crs_dv is deasserted, at which point the payload length is stored for use when the processor receives the frame over the wishbone interface. After the first two bits equals ”01” (first 2 bits of the preamble) have been received, a trigger output is asserted so that it can fire a CPU interrupt.

On the wishbone side of the receive logic, only read requests are accepted and processed. A register access returns the payload size of the received frame to help the driver (section 3.2.1) identify how much data it needs to extract from the hardware buffer. When accessing the BRAM memory locations over the Wishbone interface two things are considered. The first is that an offset of eight is needed since the BRAM stores the preamble and SFD which is not wanted by the processor. The second is that the payload is stored in 8bit values whereas the wishbone interface can send 32bits at once. Therefore some conversion between the memory addresses and the memory accesses to BRAM take place.

```
1 if wb_i_stb = '1' and wb_i_addr(31 downto 16) = x"1338" then
2     wb_o_ack <= '1';
```

```
3       if wb_i_we = '0' then -- Ensure write enable is reset to read.
4           if wb_i_addr(15 downto 0) = MAC_DAT_SIZE then -- Payload size
5               wb_o_dat <= std_logic_vector(to_unsigned(payloadLen, 32));
6           elsif wb_i_addr(15 downto 0) >= x"0008" and wb_i_addr(15 downto 0)
                <= x"05F8" then -- BRAM access
7               virtAddr := to_integer((unsigned(wb_i_addr(15 downto 0)) - 8));
8               wb_o_dat <= FRAME_BUFFER(virtAddr) &  FRAME_BUFFER(1 + virtAddr)
                    & FRAME_BUFFER(2 + virtAddr) & FRAME_BUFFER(3 + virtAddr);
9           end if;
10      end if;
11  else
12      wb_o_ack <= '0';
13  end if;
```

Listing 3.1: Wishbone access logic for Ethernet receive

By splitting the address over 2 if conditions, the amount of resources can be greatly reduced as the nested if conditions only need to compare 16bits instead of 32bits each time. Another important design decision is that this version of the Ethernet MAC does not validate the FCS after receiving a packet, instead it assumes it's a valid packet.

The transmit logic is broken down into three parts: Wishbone handler, main FSM and RMII conversion. The process of creating and sending an Ethernet frame starts with the processor sending data over the Wishbone interface. Like the receive logic, there are two types of commands that can be sent over the Wishbone interface, one that controls the logic, the other that stores payload data into BRAM. There are three configuration commands, one to initialise the hardware and FSM, another to start the transmission of data (used after all data has been transferred into BRAM) and one to set the payload length of the packet. The other register addresses allow the processor to store the payload in the frame buffer (BRAM). See figure 3.4 for the format of data in BRAM. The preamble, SFD and FCS are left out as these are appended in hardware. To be accurate, this is a quasi-MAC as the MAC addresses and type fields in the header (expanded view of payload in figure 3.4) are populated in software.

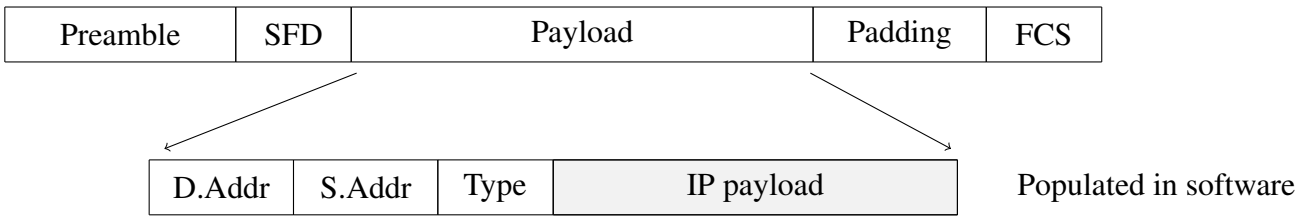


Figure 3.4: Format of frame in BRAM

Once a payload has been stored in the frame buffer, the main FSM takes control and at this point the CPU is free to do anything else. Since the FCS is calculated in hardware, the FSM resets the FCS hardware and begins to send the bytes to both the FCS hardware (to calculate the CRC32) and to a FIFO buffer. Once the payload has been transmitted to the FIFO, the resulting CRC32 FCS is sent out to the FIFO without missing a clock cycle.

A FIFO buffer is used to cross the domains since the RMII interface is 50MHz at 2bits wide, whereas the bytes are stored as 8bit vectors in the frame buffer. This also allows the FSM to have a higher clock speed as well. The current implementation of the FSM uses a 50MHz clock signal, consequently the equivalent bit rate is four times larger than the output needed to the RMII interface. The FIFO used in this design, figure 3.5, is slightly modified so that the read clock is one quarter the 50MHz output frequency since the FIFO itself returns 8 bits at a time. An FSM operating at 50MHz then sends each 2bit nibble out to the RMII PHY at a time in a circular fashion. The tx_en line is asserted while the FIFO is non-empty.

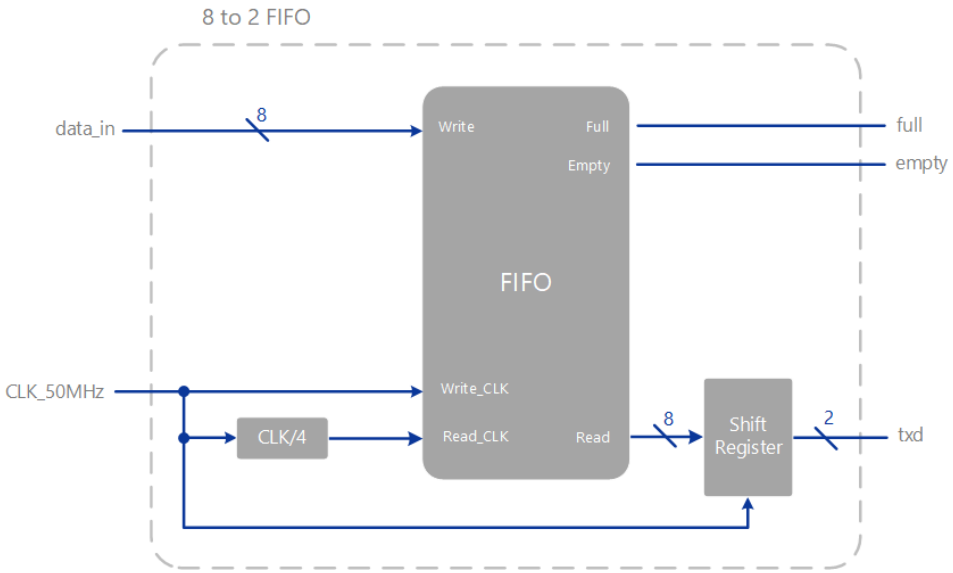


Figure 3.5: 8bit to 2bit FIFO used for RMII output.

The documentation for the NEORV32 states that all memory accesses that do not target specific processor-internal address regions (see appendix A.1) get forwarded to the external interfaces, such as the Wishbone interface. As such, there is a large block of unused memory between the IMEM and DMEM regions. The memory mapping, figure 3.6, used in this design ranges from 0x13370000 to 0x133805F8.

This mapping is required for developing the driver (section 3.2.1) to access the correct registers.

3.1.5 Packet Classifier

To further save MCU resources, the packet classification was done in hardware. Not only did this reduce the load on the MCU itself - giving it more time to do other things - it allowed the interface to run at *'wirespeed'*. That is, at the full speed of the interface - 100Mbit/s.

This was possible by having the rulset been evaluated in parallel as the data is coming into the firewall. This method however is not suitable for large rulesets as the fan-in and fan-out limit the maximum number of parallel comparisons. For every new rule, the number of gates grows exponentially. Hence a design decision of a maximum ruleset of size 8 was chosen.

The way this classifier was designed was to be a *'default-block'* where all connections were blocked except for the ones specifically whitelisted in the ruleset. The specific rules had a few options, namely

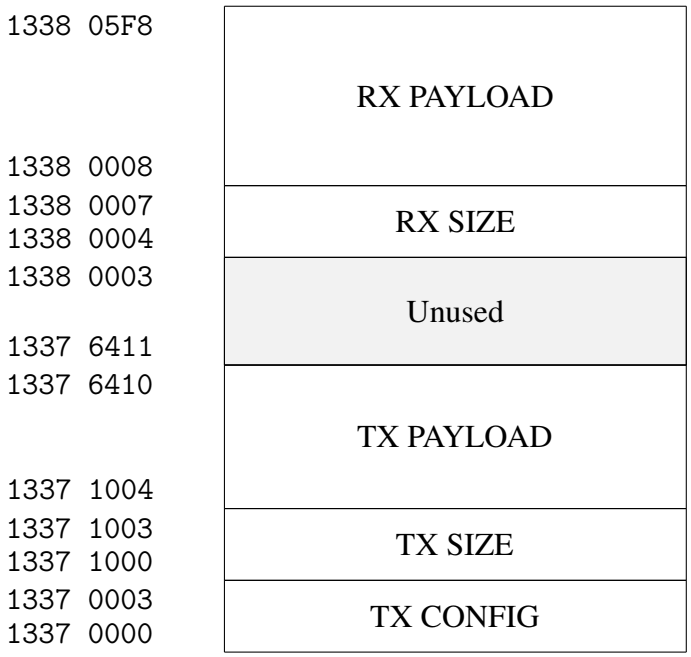


Figure 3.6: Memory Address Layout

the source IP address, destination IP address, source port, destination port and protocol could be configured. In addition to these, each field had a wildcard operator which allowed all values for that specific option to be classified.

The design of the packet classifier hardware, figure 4.1 stores the firewall rules in block memory. The rules are stored in BRAM as an array of 112 bits with the format shown in figure 3.7.

$Wildcard$	IP_{Dest}	IP_{Src}	$Port_{Dest}$	$Port_{Src}$	$Protocol$

Figure 3.7: Format of rules stored in BRAM

The wildcard attribute signifies whether to allow all possible combinations (in other words, disregard) for the positional attribute where the most significant bit refers to the IP_{Dest} and the least significant bit refers to the $Protocol$.

A FSM then records the position of the incoming and configures the multiplexers on the BRAM to output the current property to the comparators where they compare with the shift register which contains the current field being classified. On a successful match, a bit is left set in the result register, otherwise clear if no match. Importantly, the bits only get set on the first iteration of the classification.

After passing through all the fields, if there is any bit set in the results register, it indicates that a rule matched and that a packet should be forwarded.

This reduces the required resources as only 1 set of comparators are needed, which is important as for each rule that exists, another comparator is needed. In this design, there are a total of 8 comparators, 8 multiplexers and the results register is 8bits wide.

A more resource efficient design is possible at the cost of latency. This is one of the critiques of the design mentioned in [15] as multiple clock cycles would be needed to classify the headers. In theory

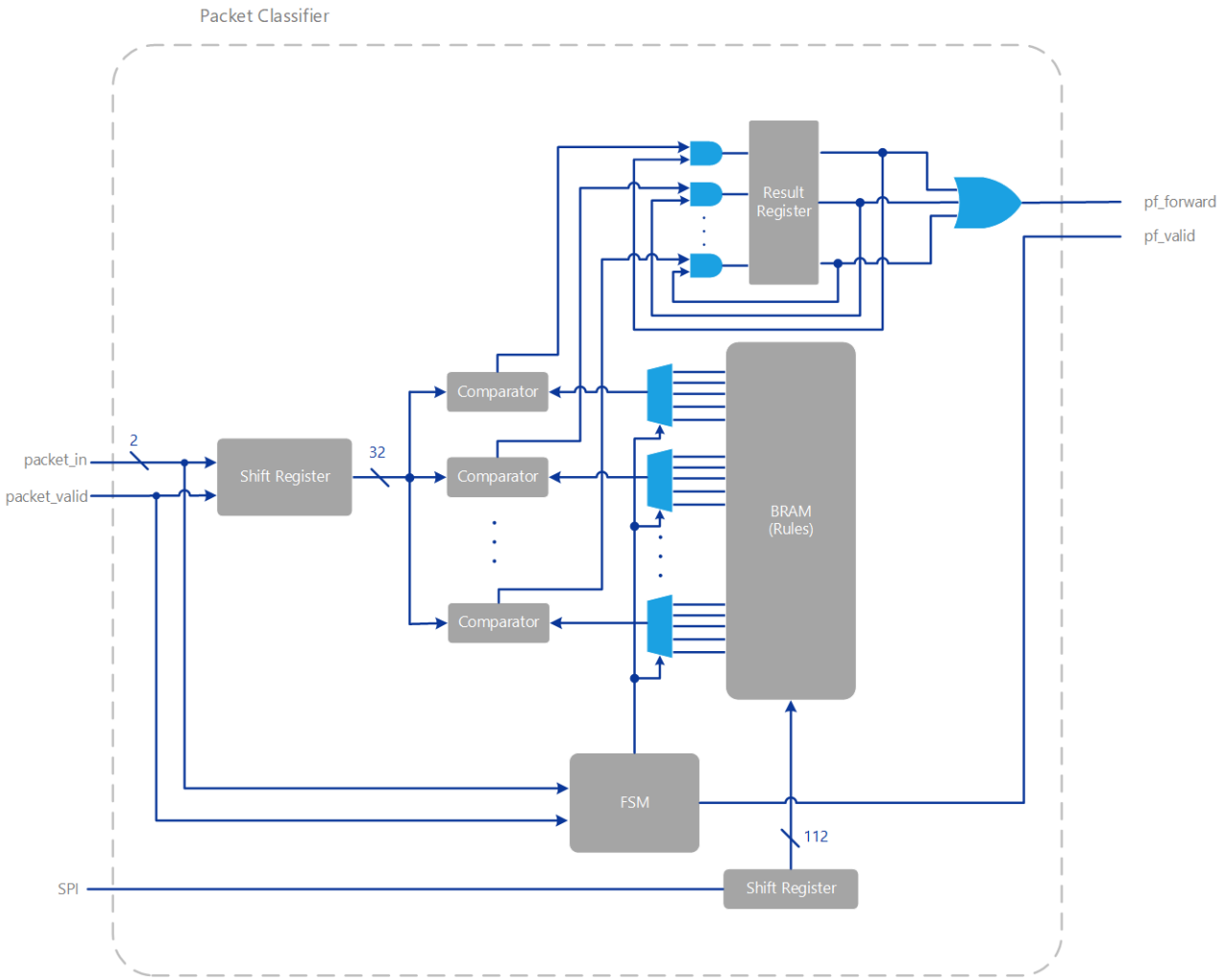


Figure 3.8: Packet classifier architecture. Clock signals have been omitted.

the clock speed could be faster than Ethernet frequency and as only two bits from the RMII interface are processed at a time, it could be a viable alternative, however this may likely fall apart when higher speeds are required.

Several options exist for configuring the ruleset in the packet classifier such as: Wishbone, I2C, UART, SPI or a completely custom solution. For simplicity, SPI was used in configuring the packet filter. Importantly, data would only flow in one direction, from the microcontroller to the classifier and not the other way round. This means that the microcontroller needs to keep the state of the rules inside the packet filter and needs to resend the rules to be sure of the configuration. This is not an issue as the rules need to be stored in flash on the microSD card to keep settings between power cycles. The format, figure 3.9 that the SPI hardware expects is similar to how it's stored in BRAM to make it quick and easy to transfer.

Index	*Wildcard*	IP_{Dest}	IP_{Src}	$Port_{Dest}$	$Port_{Src}$	*Protocol*

Figure 3.9: Format of SPI messages

Notably, the data is received into a shift register and after all bits have been received the BRAM is updated at the corresponding index in a single clock cycle.

3.2 Firmware

3.2.1 Ethernet drivers

Since the Ethernet hardware was custom, drivers were needed to interface with the hardware in software. There were two types of commands that were needed, first the RMII serial managment interface (SMI) and secondly the MAC drivers - the drivers that would handle the data. The SMI interface is used to control the mode of operation of the PHY chip including the speed, Auto-MDIX, duplex settings. The LAN8720A datasheet, ([25]) provided some details (seen in figure 3.10) into how the protocol operated. The datasheet also outlined that a maximum frequency of 2.5MHz, but no lower bound. As such the interface was *'bitbanged'* with GPIO pins to reduce complexity. The maximum switching frequency of the NEORV32's GPIO was measured to be $\approx$ 1MHz, thus no additional delays were needed in the code and that the GPIO pin could be toggled at the fasted speed possible.

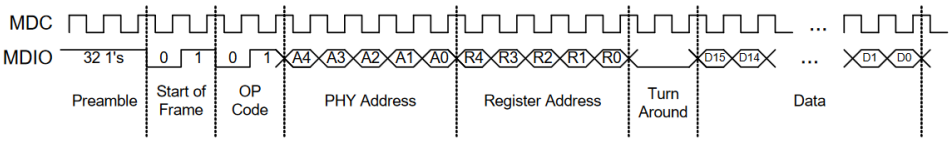


Figure 3.10: SMI Write message structure. [25]

On power up the RMII interface would be set to full duplex, 100Mbps and auto-negotiate in accordance to the LAN8720A datasheet. 10Mbit and half duplex modes were excluded from the design as further hardware design would be required.

As the Ethernet hardware used the Wishbone interface, the register locations were mapped into the processors address space. Accessing these registers is analogous to accessing any other variable in memory, using pointers to the memory address. As an example, to access memory location `0x12345678` the following C code can be used `*(volatile uint32_t *)0x12345678`

To simplify the design and improve readability, a range of macros was setup, such example of macros can be seen below.

```
#define ETH_MAC_TX_BASE 0x13371000
#define ETH_MAC_CMD_BASE 0x13370000

#define ETH_MAC_CMD  (*(volatile uint32_t *)ETH_MAC_CMD_BASE)
#define ETH_MAC_TX ((EthMacTx *) ETH_MAC_TX_BASE)

typedef struct __attribute__((__packed__))  {
    volatile uint32_t SIZE;
    volatile uint32_t DATA[375]; // 1500 / 4 = 375.
} EthMacTx;
```

Listing 3.2: Python example

This allowed for the connected wishbone Ethernet to be accessed like any other register in the embedded system.

There are three fundamental functions that the driver itself must fulfill, initialisation, send data and receive data. The intialisation resets the Ethernet MAC and resets the interrupt registers.

The send method is also trivial, but importantly it takes in two parameters, the first is an array of data to send and the second is the amount or length of the data. Like with the SMI interface, these instructions and data transfers were actioned without any delays as the hardware was capable of handling the native speed of the processor. After the data had been transferred, the hardware is instructed to send out the packet. There is no need to provide the FCS as this is calculated on the fly in hardware.

Similarly, a receive method was created that took in a buffer to store the bytes into as it transfers the data into memory from the registers. Both the transmit and receive functions handled the data translation from 32bit values over the wishbone interface to 8bit bytes in software. In addition to this, as the Ethernet hardware used external interrupts to signal to the processor that there is an Ethernet packet ready for processing. This would use direct task notifications to signal to other functions in the code to call the receive method.

3.2.2 SD card drivers

To use a micro SD card, drivers must be created so that the card can be initialised, written to and read from. More preciesly, in accordance to the documentation for the FreeRTOS-Plus-FAT file system, the driver had to implement a function that reads sectors from the media and one that rights sectors to the media.

Unlike the Ethernet MAC, initialising a MicroSD card isn't as trivial and requires multiple steps. A guide online for AVRs[1] was followed and the code was ported to the NEORV32 system. In a nutshell, the SPI interface was initalised with a clock divisor of 1 and a prescalar of 3, several commands were sent and recieved from the SD card and then it was put into IDLE mode after increasing the speed of the SPI interface to have no clock divisor and a prescalar of 1.

Similarly, the same guide was followed to implement the read and write sector functions. Importantly, these functions would read and write a whole block at a time. On a microSD card, a block is considered to be 512 bytes.

In addition to these, a simple function to determine if a SD card is present in the slot was also implemented to warn the user if data was attempted to be written to or read from without a physical card in the slot.

3.2.3 Packet classifier drivers

An initialisation function was created to enable and configure the SPI interface on the microprocessor for mode 0, and with no clock diviser and a prescalar of 1. The SPI hardware for the classifier can handle speeds in excess of 50Mhz (seen in section **??**), hence there was no need to slow down the

[1] See: http://www.rjhcoding.com/avrc-sd-interface-1.php

clock. Moreover, calling this initialise function is only required if the microSD card intialise function has not been called.

A single function was created that takes in all the attributes needed in the packet filter and that sends these out in the correct order and format to the hardware.

3.3 Software

3.3.1 Real Time Operating System

In addition to simplifying the software, an RTOS was used to allow the use of network TCP stacks and handle multiple concurrent connections at a time. FreeRTOS version V10.4.4 was used due to its familirity and compatability with the NEORV32 MCU. FreeRTOS also had integration with their own filesystem (section 3.3.2) and TCP/IP stack (section 3.3.3).

3.3.2 Filesystem

The FreeRTOS-Plus-FAT filesystem library was used to allow the system to read and write files (such as web assets) to a microSD card. The library is managed by FreeRTOS and is DOS compatible which allows FAT32 formatted drives to work. Other popular filesystem modules such as FatFS[1] and LittleFS [2] were also considered, but did not have the same level of integration and active support as the FreeRTOS option.

The library then uses the microSD card drivers (section 3.2.2) to access the disk.

3.3.3 Network Stack

Two main options for the network stack were available, LwIP and FreeRTOS-Plus-TCP. The main concern with LwIP was that it was not threadsafe and had memory issues. In addition to this, as FreeRTOS was chosen as the RTOS, their own TCP stack had tighter integration. The stack provides a Berkeley sockets API which is the same used in full-blown operating systems such as Linux. It also includes support for ARP, DHCP, DNS and ICMP protocols, which were used throughout this project.

Like with the filesystem library, the stack needed to be ported to the NEORV32 processor and importantly the custom Etherent hardware. In addition to the transmit, receive and initalisation methods, functions to return random numbers were needed. These are for the TCP sequence numbers and are required to truely random for security. As such, the hardware based true random number generator in the NEORV32 core was used.

[1] http://elm-chan.org/fsw/ff/00index_e.html
[2] https://github.com/littlefs-project/littlefs

3.3.4 Webserver

A simple HTTP webserver running on top of a TCP server was used to serve the webpages for the project. FreeRTOS give an example of how to setup a TCP and HTTP server that uses the FAT filesystem to get the content such as the html, css and javascript files.

... TALK ABOUT WHAT TYPE OF FILES ARE BEEN SERVED, VUE, PLAIN HTML ETC...

3.3.4.1 API server

A subset of the webserver is the API server itself. To make the design simpler, an API was created so that the interface to set and get the firewall rules was independant of the web content. The way the software distinguishes between requests to load a webpage and the API server itself is by inspecting both the route/URL (pcUrlData) and the method type (GET, POST, PUT).

For setting the firewall rules, a POST request to the *'/api/firewall'* endpoint can be made. The body of the request would contain the rule in the following format

$$payload = Index|Wildcard|IP_{Dest}|IP_{Src}|Port_{Dest}|Port_{Src}|Protocol$$

Where | is the concatination operator and all fields are in hexadecimal. As an example, to insert a rule at index 0, and with a wildcard operator for all items with a destination IP of 10.20.1.120, source IP of 10.0.0.159, source and destination port of 80 and a protocol of TCP, the following body would need to be sent to the API: `payload=003F0A1401780A00009F0050005006`. The API server then takes the necessary action and applies the rule to the packet filter by calling the methods in the packet filter driver (section 3.2.3).

3.3.5 Command line interface

To aid with debugging the FreeRTOS-Plus-CLI framework was used. This would easily allow certain actions to be executed on demand without the need to reflash or reset the device each time. Such examples include configuring the PHY, ethernet MAC, sending ICMP packets and filesystem related commands. In additon, the firewall rules can be set from the CLI in the event that the firewall blocks HTTP connections.

Chapter 4

Results

To ensure that the firewall was performing properly, a few tests were conducted. In this chapter, the testing results for the performance and the resource utilisation among other features are discussed.

4.1 Modifications

The design was changed from the 2 ethernet interfaces to a design with just a single ethernet interface.

4.2 Performance

Talk about wishbone bus speed. Initial tests were also conducted at 100MHz, but due to timing issues, it was reverted to 50MHz.

To test the speed of the packet classifier, a Agilent MSO6054A MSO was used (4GSa/s). A pin was set to the output of the crs_dv of the PHY and the crs_dv after the packet filter. The time between the rising and falling edge of the output will be the delay added by the packet filter.

As the shift register in the hardware has a length of 224 bits, at a clock frequency of 50Mhz, the added latency is $224 \times \frac{1}{50 \times 10^6} = 4.48 \times 10^{-6} = 4.48uS$.

There is two pulses in this graph. This is due to when the crs_dv line gets disasserted at the end of the packet. From this the time taken to receive the packet can be found to be 8.94uS. This is just an observation and is proportional to the packet size.

The same setup was used to test the preexisting solution, except this time GPIO pins were set high and low. It is assumed that the latency of setting the pin high cancels out with the latency of setting the pin low.

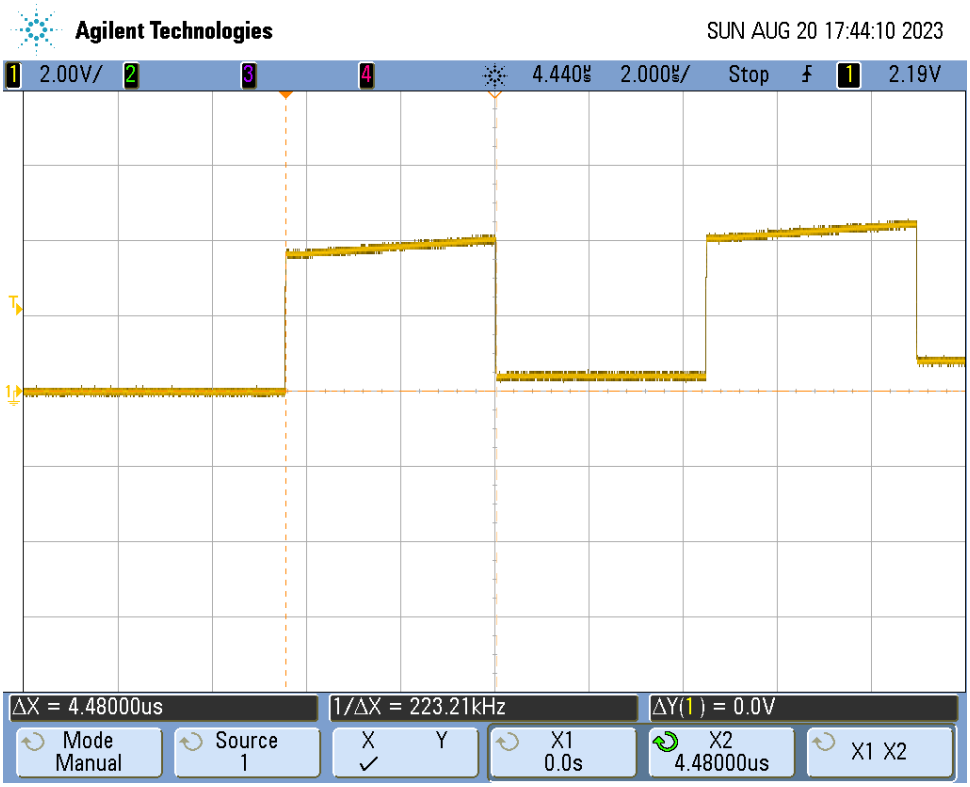


Figure 4.1: Added latency by packet filter waveform.

4.2.1 Limitations

4.2.1.1 PMOD Interface

There are 5 PMOD connectors on the development board. Initally, one of these would be used for a second Ethernet PHY, but due to bandwidth limitations of the interface, the design had to be altered. The recommended bandwidth of these ports are 25MHz while the Ethernet RMII PHY would have been using 50Mhz signals over the interface. As such, signal integrity issues arose (see figure 4.3) and restricted the use to just one interface - the onboard PHY. A new development board with two PHYs would be needed.

4.2.2 Testing setup

4.2.3 Results

4.3 Ultilisation

The design can be broken down into several parts including the NeoRV32 processor itself, Ethernet and packet filtering hardware.

The total number of resoruces on the FPGA are as follows: 63,400 LUTs, 126,800 flip flops, 135 BRAM tiles, 240 DSPs and 6 MMCMs.

LUT6 and LUT5 primatives were used the most in the design and by far the FDRE flip flop was the most used flip flop type. 2436 MUXF7s were also used. RAMD64E was also used in the design.

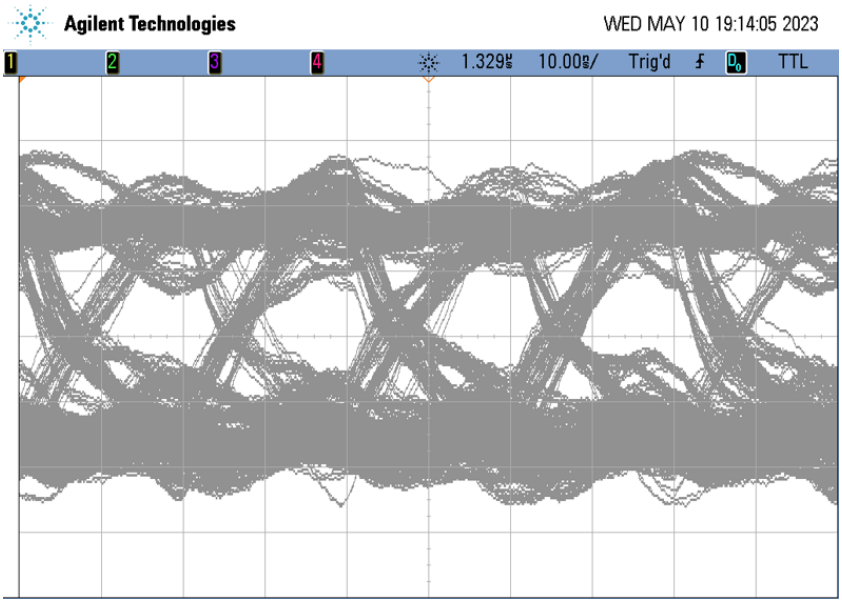


Figure 4.2: Eye diagram of TXD through PMOD interface.

Name	Slice LUTs	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Block RAM Tile (135)	DSPs (240)	Bonded IOB (210)	BUFGCTRL (32)	MMCME2_ADV (6)
hardware_top	40920	16457	2436	884	130	4	66	8	1
neorv32_top_inst (neorv32_top)	28455	2507	25	8	130	4	0	0	0
ethernet_mac (wb_ethernet)	11738	12505	2403	872	0	0	0	0	0
eth_tx (eth_tx_mac)	9175	12384	2296	848	0	0	0	0	0
FCS_CRC32 (CRC)	49	40	0	0	0	0	0	0	0
eth_rx (eth_rx_mac)	1398	73	51	0	0	0	0	0	0
rmii_int (rmii)	1165	48	56	24	0	0	0	0	0
pc (packet_classifier)	571	1145	0	0	0	0	0	0	0
pf_stats_spi (pf_stats)	127	104	8	4	0	0	0	0	0
clk_control (clk_master)	1	0	0	0	0	0	0	6	1

Figure 4.3: Summary of the resource utilisation on XC7A100T FPGA.

The report from vivado can be found in appendix A.2.

4.3.1 NeoRV32 processor

The NeoRV32 SoC was configured to use the SPI, UART, GPIO, external interupts (XIRQ), true random number generator and importantly the wishbone B4 classic interface. The DSP48 blocks were used by the SoC to handle the multiply operations as this would free up LUTs and compute the result faster.

The IMEM and DMEM sizes were chosen to consume as much of the remaining BRAM blocks left as possible. IMEM, the program storage was configured to 256KB while the DMEM, effectively the RAM, was configured to take the remaining amount left and set to 168KB in size.

After synthesis of the design at 50Mhz, it was deemed that there was some headroom in the critical path delays. Hence the design was updated to 80MHz. This however does not affect the resource utilisation except for taking up some extra clock buffers. A single MMCM was still used to add the additional 80MHz output.

The NeoRV32 itself took 28455 slice LUTs, 2507 Slice registers, 130 Block RAM tiles and 4 DSPs.

4.3.2 Ethernet hardware

Comparatively, the Ethernet hardware took 11738 Slice LUTs and 12505 Slice registers, most of which is consumed by the transmit logic. This is largely due to the required buffers when storing (to construct) and sending out the frame.

4.3.3 Packet filter

The packet classifier took a total of 571 Slice LUTs and 1145 Slice registers. This is lower than expected and indicates that the design could easily be increased to consolidate more rules. However, the fanout of the design will need to be considered due to the nature of the implementation.

4.4 Timing Summary

After running the post synthesis timing summary in Vivado, a worst negative slack of -2.634ns was reported for the setup timing with a total of 82 endpoints failing the constraints. For the hold summary, a worst hold slack of -0.021ns was found.

The critical path is from the wishbone interface to the BRAM block inside the ethernet hardware - seen in figure 4.4. The level is 3 and stems from the NeoRV32 processor's wishbone bus. This cannot be easily updated and so is the leading limitation in the speed of the design. Even though this, in practice, doesn't seem to cause any issues, a caution for reliabilty is raised.

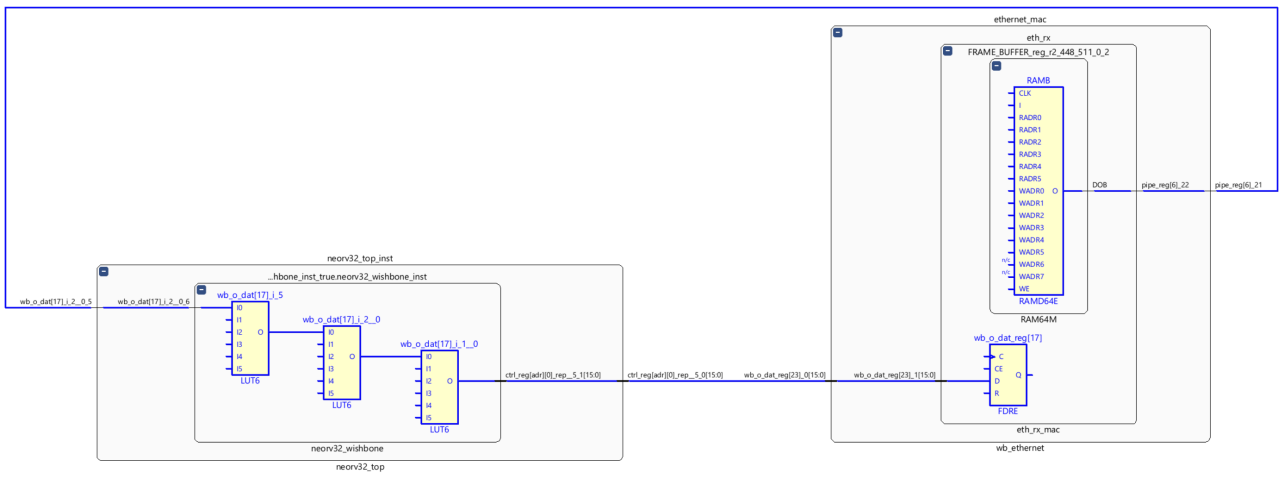


Figure 4.4: Critical path in SoC design.

4.5 Comparison to preexisting solutions

my solution is less suseptible to power glitch attacks since this is done at the hardware layer and not in software where instructions can be skipped. More secure and harder to bypass. faster

Testing with WIZ5500 Pico, avg udp rtt was 1.88ms from 1000 tests with a payload of 7 bytes. Testing with WIZ5500 Pico, avg udp rtt was 2.07ms from 1000 tests with a payload of 256 bytes.

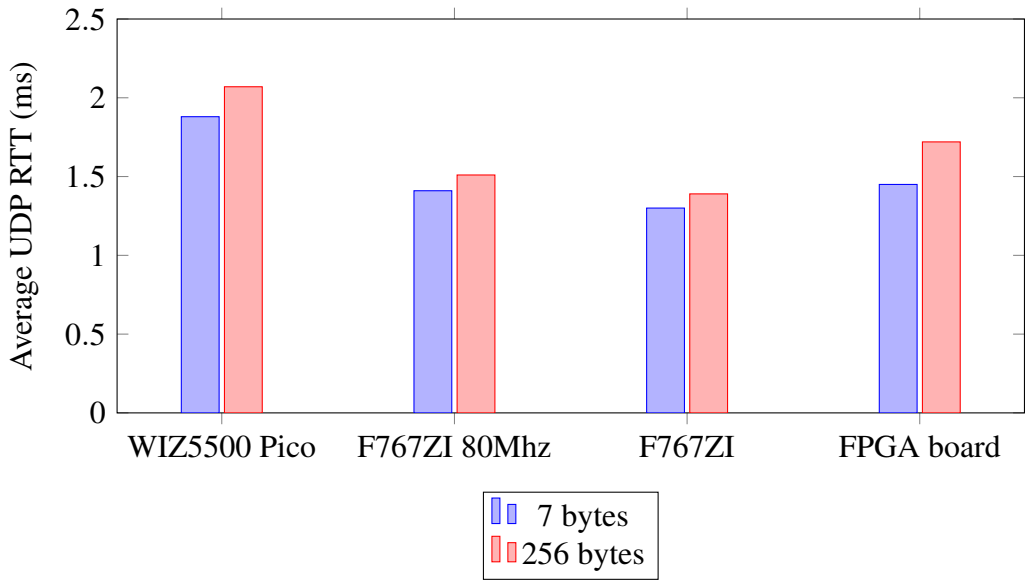


Figure 4.5: Average UDP RTT for different devices and payload sizes.

Testing with LwIP Nucleo-F767ZI at 80Mhz, avg udp rtt was 1.41ms from 1000 tests with a payload of 7 bytes. Testing with LwIP Nucleo-F767ZI at 80Mhz, avg udp rtt was 1.51ms from 1000 tests with a payload of 256 bytes.

Testing with LwIP Nucleo-F767ZI, avg udp rtt was 1.30ms from 1000 tests with a payload of 7 bytes. Testing with LwIP Nucleo-F767ZI, avg udp rtt was 1.39ms from 1000 tests with a payload of 256 bytes.

Testing with FPGA board, avg udp rtt was 1.45ms from 1000 tests with a payload of 7 bytes. Testing with FPGA board, avg udp rtt was 1.72ms from 1000 tests with a payload of 256 bytes.

4.5.1 Firewall performance

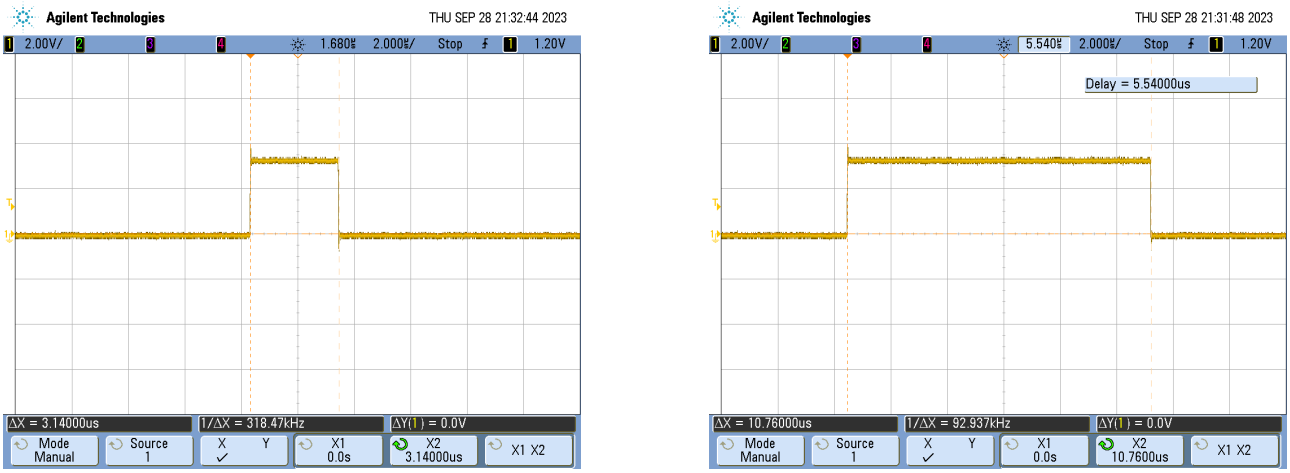
The hardware packet filter in this design has previously be found to induce a 4.48us delay and does not change the throughput of the device.

A software based implementation of the firewall was created on the Nucleo-F767ZI board. Also with 8 rules. After measuring with an with an oscilloscope for the time it takes to compute whether or not to forward the packet, the timings depended on first how many rules there are, where a valid rule is matched (start or end of the sequence), if there is like terms between invalid rules and so on.

As a best case, the time was found to be 3.14us (figure 4.6a) while an average-to-bad case was 10.76us (figure 4.6b).

4.5.2 Thermal analysis

A thermal camera was used to record the temperatures periodically. At an ambient room temperature of 24.8°C throughout the test, after 5mins the WIZ5500 ethernet chip heated to 58.0°C and RP2040 was at 46.6°C. While the FPGA was at 38.0°C. This is a bit of an unfair comparison as the physical size of the FPGA is much larger than the WIZ5500. After two hours of constant UDP ping requests



(a) Best case scenario

(b) Average case

Figure 4.6: Software packet classifier timings

to both devices, figure 4.7 shows the FPGA board and WIZ5500 board's temperature gradient. The FPGA plateaued to a maximum of 40.4°C while the WIZ5500 was 1.c°C cooler at, 56.8°C. This could be due to accuracy of the measurements, in addition to not getting aiming the thermal camera in the hottest part. The RP2040 chip however was measured to be 53.2°C. Some additional thermal images can be found in the appendix.

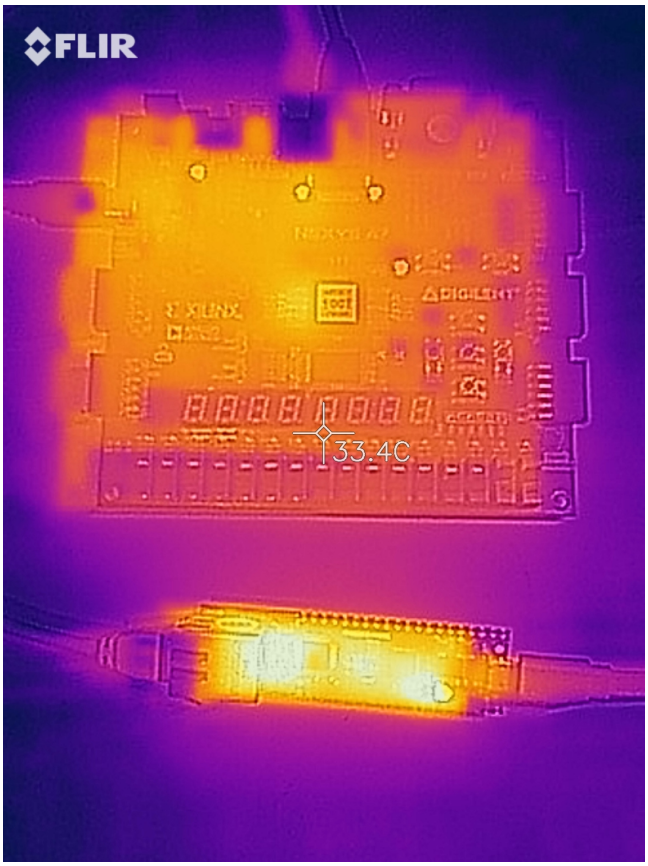


Figure 4.7: Thermal image of FPGA board and WIZ5500 after two hours.

4.6 Power analysis

As the voltage would remain constant between devices (all powered over USB), only the current was measured. These results however should be taken with caution as they do not account for regulator inefficiencies and do not give a true current reading of the device, rather just an indication.

The device for testing the current was the Nordic Semiconductor Power Profiler Kit 2 which can record at up to 100kSa/s. For the tests in this report, only a sampling rate of 10kSa/s was used as it produces less noisy results.

As a baseline, the Nexys A7 board draws 200mA with no design applied and is due to all the additional components on the board. With the design loaded up but without the processor flashed, the board took 284.4mA. After flashing the board, the idle current reached an average of 301.84mA.

A series of tests were then done and the currents were measured. By pinging the device every 50ms, the average current consumption was 300.72mA. Interestingly, the current consumption was cyclic similar to what is shown in figure 4.8. A UDP ping test was conducted and had an average current draw of 301.13mA. In both the ICMP and UDP pings, the current consumption was of similar style where the variance of the current was about 10mA.

If the packets are now blocked by the filter, more about the design in terms of power consumption can be learned. After adding a rule in the packet filter, very little could be observed in the current consumption over the unblocked case. The same cyclic pattern in figure **??** could be seen. An average of 300.92mA was been consumed by the device and is within the margin of error of the device. Figure 4.8 shows the that the period for the current waveform is about 83ms, which is much larger than the 50ms between pings to the device. After filming the status LED on the PHY output at 240 frames per second, the period of the LED was 20 frames equivalent to $\approx 84ms$ which aligns with the current measurements.

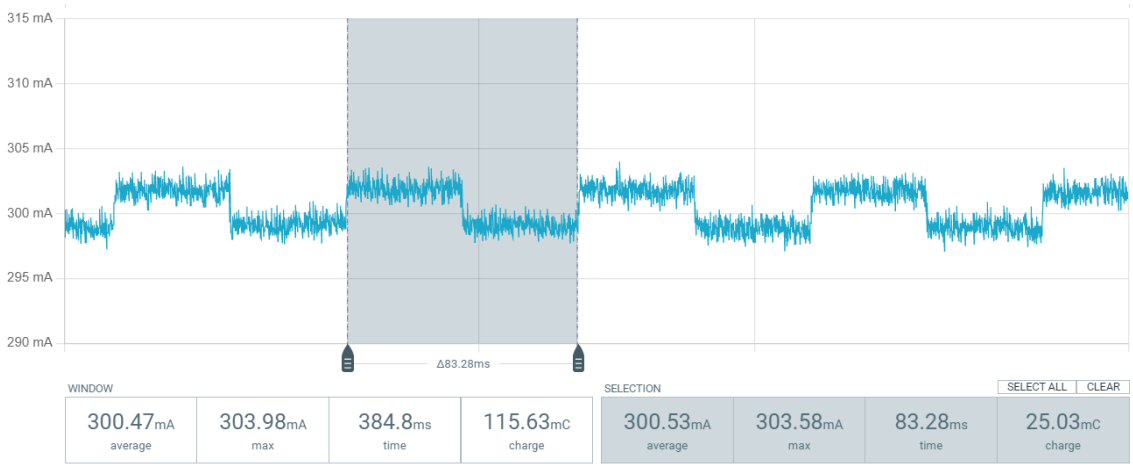


Figure 4.8: Zoomed in current consumption for ICMP pings.

The next test was done when accessing the webserver. Figure 4.9 shows 5 different regions where each region is a result of a different action. The left most tests is from the inital HTTP requests to get the index page. Notably, there is 2 separate sections here, this is because the client fetches the html,

css and favicon first and then requests the main (and much larger) javascript file after. The readings for each of these points is given as: average current, maximum current and time going from top to bottom.

The second test is what happens when you click to navigate to the about page. The third event is when navigating to the config page and the 4th event is what happens when you press the 'load rules' button. The final test case is a refresh on the main page for the statistics. Notably, as the javascript (thus client) is doing the routing and page handling, future requests to get the contents of the pages are not needed, but only small API requests to update the data. Coincidentally, these first 3 requests also trigger a SD card read and explains the higher current draw. The fourth request also creates a read request to the SD card, but only needs to read a single page. The fifth request does not involve a read or write to the SD card, but rather just a simple SPI transaction takes place and consequently doesn't draw much additional power.

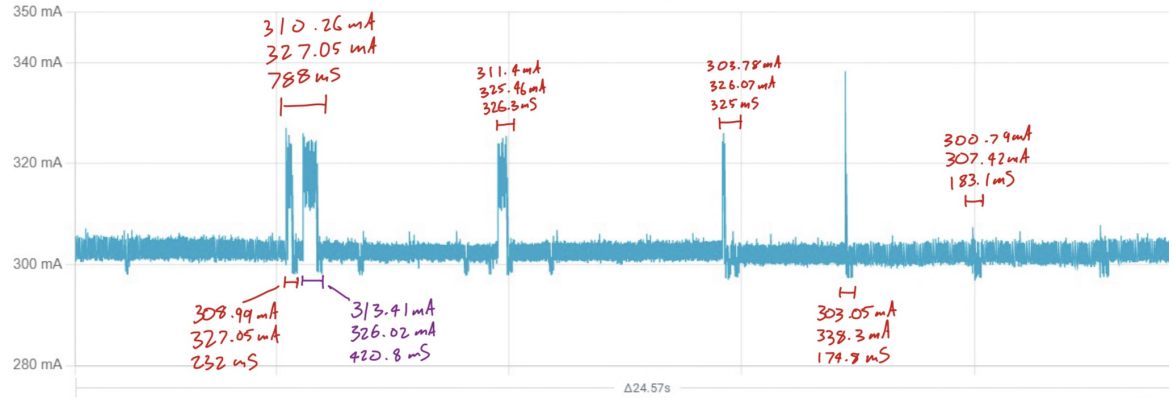


Figure 4.9: Current consumption of FPGA board with HTTP requests.

Chapter 5

Conclusion

Conclude your thesis.

5.0.1 Improvements

- Bidirectional filtering - currently only doing incomming filtering

Bibliography

[1] A. C. S. Center, "Acsc annual cyber threat report, july 2021 to june 2022." https://www.cyber.gov.au/acsc/view-all-content/reports-and-statistics/acsc-annual-cyber-threat-report-july-2021-june-2022, Nov 2022.

[2] IHS, "The internet of things: a movement, not a market." https://cdn.ihs.com/www/pdf/IoT_ebook.pdf, 2017.

[3] W. Shi, G. Pallis, and Z. Xu, "Edge computing [scanning the issue]," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1474–1481, 2019.

[4] M. Caprolu, R. Di Pietro, F. Lombardi, and S. Raponi, "Edge computing perspectives: Architectures, technologies, and open security issues," in *2019 IEEE International Conference on Edge Computing (EDGE)*, pp. 116–123, IEEE, 2019.

[5] S. M. Trimberger, "Three ages of fpgas: A retrospective on the first thirty years of fpga technology," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 318–331, 2015.

[6] M. Gokhale and L. Shannon, "Fpga computing," *IEEE MICRO*, vol. 41, no. 4, pp. 6–7, 2021.

[7] S. Lin, D. Zhang, Y. Fu, and S. Wang, "A design of the ethernet firewall based on fpga," in *2017 10th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*, vol. 2018-, pp. 1–5, IEEE, 2017.

[8] A. Kayssi, L. Harik, R. Ferzli, and M. Fawaz, "Fpga-based internet protocol firewall chip," in *ICECS 2000. 7th IEEE International Conference on Electronics, Circuits and Systems (Cat. No.00EX445)*, vol. 1, pp. 316–319 vol.1, IEEE, 2000.

[9] S. M. Keni and S. Mande, "Packet filtering for ipv4 protocol using fpga," in *2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS)*, pp. 400–403, IEEE, 2018.

[10] S. Cuenca-Asensi, A. Martínez-Álvarez, F. Restrepo-Calle, F. R. Palomo, H. Guzmán-Miranda, and M. A. Aguirre, "Soft core based embedded systems in critical aerospace applications," *Journal of systems architecture*, vol. 57, no. 10, pp. 886–895, 2011.

[11] L. Janik, M. Novak, L. Hudcova, O. Wilfert, and A. Dobesch, "Lwip based network solution for microblaze," in *2016 International Conference on Broadband Communications for Next Generation Networks and Multimedia Applications (CoBCom)*, pp. 1–4, IEEE, 2016.

[12] N. Joshi, P. Dakhole, and P. Zode, "Embedded web server on nios ii embedded fpga platform," in *2009 Second International Conference on Emerging Trends in Engineering and Technology*, pp. 372–377, IEEE, 2009.

[13] E. D. Zwicky, S. Cooper, and D. B. Chapman, *Building Internet Firewalls (2nd Ed.)*. USA: O'Reilly and Associates, Inc., 2000.

[14] E. W. Fulp, "Chapter e74 - firewalls," in *Computer and Information Security Handbook*, pp. e219–e237, Elsevier Inc, third edition ed., 2017.

[15] A. Wicaksana and A. Sasongko, "Fast and reconfigurable packet classification engine in fpga-based firewall," in *Proceedings of the 2011 International Conference on Electrical Engineering and Informatics*, (Ithaca), pp. 1–6, IEEE, 2016.

[16] S. Wasti, "Hardware assisted packet filtering firewall." `https://madmuc.usask.ca/Pubs/shw320.pdf`, 2001.

[17] Y.-H. Cheng, L.-B. Huang, Y.-J. Cui, S. Ma, Y.-W. Wang, and B.-C. Sui, "Rv16: An ultra-low-cost embedded risc-v processor core," *Journal of computer science and technology*, vol. 37, no. 6, pp. 1307–1319, 2022.

[18] C. Heinz, Y. Lavan, J. Hofmann, and A. Koch, "A catalog and in-hardware evaluation of open-source drop-in compatible risc-v softcore processors," in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–8, IEEE, 2019.

[19] V. A. Frolov, V. A. Galaktionov, and V. V. Sanzharov, "Investigation of risc-v," *Programming and computer software*, vol. 47, no. 7, pp. 493–504, 2021.

[20] M. A. ISLAM and K. KISE, "An efficient resource shared risc-v multicore architecture," *IEICE transactions on information and systems*, vol. E105.D, no. 9, pp. 1506–1515, 2022.

[21] S. Nolting, "The neorv32 risc-v processor." `https://stnolting.github.io/neorv32/`, 2023. v1.8.1-r17-gd1b295de.

[22] "Wishbone B4 SoC Interconnection." `https://cdn.opencores.org/downloads/wbspec_b4.pdf`, Dec. 2010. Standard.

[23] "IEEE 802.3-2012 IEEE Standard for Ethernet," standard, The Institute of Electrical and Electronics Engineers, Inc., New York, USA, Dec. 2012.

[24] J. Hemanth, X. Fernando, P. Lafata, and Z. Baig, "An optimized packet transceiver design for ethernet-mac layer based on fpga," in *International Conference on Intelligent Data Communication Technologies and Internet of Things (ICICI) 2018*, vol. 26 of *Lecture Notes on Data Engineering and Communications Technologies*, pp. 725–732, Switzerland: Springer International Publishing AG, 2019.

[25] Microchip Technology Incorporated, "Small footprint rmii 10/100 ethernet transceiver with hp auto-mdix support." "https://ww1.microchip.com/downloads/en/DeviceDoc/8720a.pdf", 2012. Revision 1.4 (08-23-12).

[26] J. Sütő and S. Oniga, "Fpga implemented reduced ethernet mac," in *2013 IEEE 4th International Conference on Cognitive Infocommunications (CogInfoCom)*, pp. 29–32, 2013.

[27] J. Mitra and T. Nayak, "Reconfigurable very high throughput low latency vlsi (fpga) design architecture of crc 32," *Integration (Amsterdam)*, vol. 56, pp. 1–14, 2017.

[28] P. Guoteng, L. Li, O. Guodong, F. Qingchao, and B. Han, "Design and verification of a mac controller based on axi bus," in *2013 Third International Conference on Intelligent System Design and Engineering Applications*, pp. 558–562, IEEE, 2013.

[29] N. M. Khalilzad, F. Yekeh, L. Asplund, and M. Pordel, "Fpga implementation of real-time ethernet communication using rmii interface," in *2011 IEEE 3rd International Conference on Communication Software and Networks*, pp. 35–39, IEEE, 2011.

[30] T. Stuckenberg, M. Gottschlich, S. Nolting, and H. Blume, "Design and optimization of an arm cortex-m based soc for tcp/ip communication in high temperature applications," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Lecture Notes in Computer Science, (Cham), pp. 169–183, Springer International Publishing.

[31] L. Liu, N. Li, and L. Feng, "Improvement and optimization of lwip," in *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, pp. 1342–1345, IEEE, 2016.

[32] FreeRTOS, "Freertos plus tcp - a free thread aware tcp/ip stack for freertos." https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_TCP/index.html, Aug 2022.

[33] "7 Series FPGAs Data Sheet: Overview (DS180)." https://docs.xilinx.com/v/u/en-US/ds180_7Series_Overview, Sept. 2020. Standard.

Appendix A

Appendix

Write your appendix here. Following two are examples.

A.1 Neorv32 memory address space layout

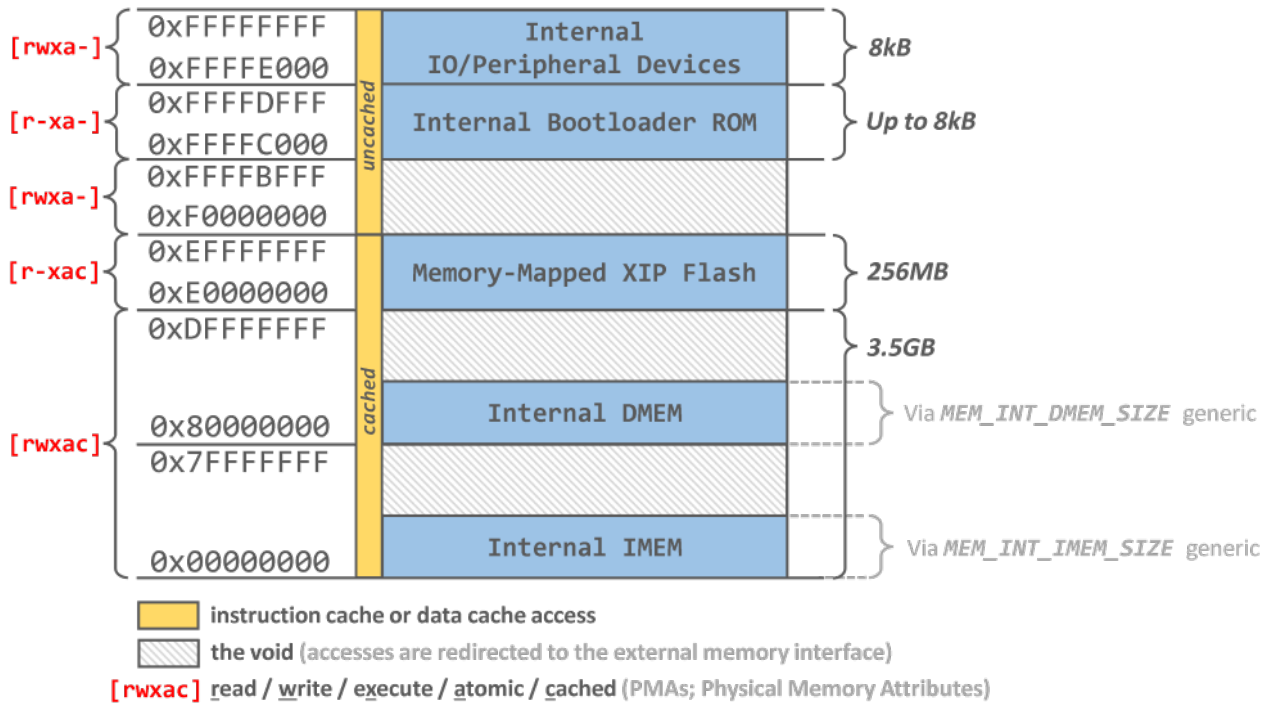


Figure A.1: Neorv32 Memory Address space.

A.2 FPGA primitives utilisation

Table A.1: FPGA primitives utilisation for XC7A100T

Ref Name	Used	Functional Category
LUT6	16262	LUT
LUT5	14820	LUT
FDRE	14500	Flop & Latch
LUT3	13222	LUT
MUXF7	2436	MuxFx
FDCE	1875	Flop & Latch
RAMD64E	1836	Distributed Memory
LUT4	1294	LUT
LUT2	1016	LUT
MUXF8	884	MuxFx
CARRY4	437	CarryLogic
LUT1	156	LUT
RAMB36E1	130	Block Memory
FDPE	41	Flop & Latch
OBUF	40	IO
LDCE	36	Flop & Latch
IBUF	24	IO
SRLC32E	21	Distributed Memory
OBUFT	11	IO
BUFG	8	Clock
FDSE	5	Flop & Latch
DSP48E1	4	Block Arithmetic
SRL16E	1	Distributed Memory
MMCME2_ADV	1	Clock

Table A.2: Memory Utilisation

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	130	0	0	135	96.30
RAMB36/FIFO*	130	0	0	135	96.30
RAMB36E1 only	130	-	-	-	-
RAMB18	0	0	0	270	0.00

Table A.3: Slice Logic Utilisation

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	40920	0	0	63400	64.54
LUT as Logic	39062	0	0	63400	61.61
LUT as Memory	1858	0	0	19000	9.78
LUT as Distributed RAM	1836	-	-	-	-
LUT as Shift Register	22	-	-	-	-
Slice Registers	16457	0	0	126800	12.98
Register as Flip Flop	16421	0	0	126800	12.95
Register as Latch	36	0	0	126800	0.03
F7 Muxes	2436	0	0	31700	7.68
F8 Muxes	884	0	0	15850	5.58