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

October 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

This undergraduate thesis presents the design and implementation of both a hardware Ethernet Media Access Control (MAC) and packet filter on a Xilinx Artix 7 100T FPGA, specifically with the Digilent Artix 7 FPGA development board which includes a Reduced Media-Independant Interface (RMII) physical (PHY) interface chip. The primary objective of this work was to implement a firewall to improve security in the embedded systems space and to then host a web server on an onboard RISC-V softcore for configuration. More specifically, a NEORV32 RISC-V System on Chip (SoC) was used to interface the hardware over a Wishbone bus with the software hosting the webserver with FreeRTOS utilisitng both the Freertos-Plus-TCP and FreeRTPS-Plus-FAT libraries.

The wirespeed hardware five-tuple packet filter, analysing the destination IP, source IP, destination port, source port and protocol, showcased an added delay of just $4\mu s$ irrespective of packet lengths while potentially enhancing security over software based implementations. Many performance benchmarks were also conducted and concluded in a relative power draw of 0.51W including the microprocessor. In comparison other platforms such as the Nucleo-F767ZI, Raspberry pi Pico with WIZ5500 and MilkV-Duo were evaluated for their performance and efficiency.

In addition, the web server hosted a static single page application style website using Vue.js and Tailwindcss which was all stored on a microSD card and accessed over the SPI interface and using the FAT32 filesystem. UDP round trip times were also measured for all platforms resulting in an average delay of 1.45ms for the FPGA board which included an added 1ms delay.

Although effective, the packet classifier lacks support for IPv6 and only is applied to incoming traffic, while the firmware forgoes support for HTTPS. Given the FPGA's resource consumption of 11,738 slice LUTs and 12,505 slice registers, potential optimisations are discussed to overcome these shortcomings. A recommendation for future designs includes incorporating the efficiency and performance of the MilkV Duo RISC-V (CVITEK CV1800B based) board with an integrated hardware packet filter for a fast and secure embedded system platform.

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
RTT	Round Trip Time

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. Consider the infamous Mirai botnet which strictly targeted IoT devices to become a part of their potent Distributed-Denial-of-Service (DDoS) network to cripple services such as Dyn (a DNS provider), Sony, Facebook, CNN among others like they did in 2016 [1]. In more recent years, the Australian Cyber Security Center (ACSC) received in excess of 76,000 cybercrime reports and growing in the 2021-22 financial year [2]. 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 [3]. 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 [4] 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 [4]. Consequently, smaller and more efficient computers can be deployed at the edge/perimeter of these networks [5].

1.2 Aim and Objectives

To help alleviate the growing number of cyber attacks, this thesis aims to increase the security of IoT devices against cyber threads by designing and implementing a hardware firewall with Ethernet controller on an FPGA for use with a RISC-V processor. The work conducted in this thesis hopes to inspire future microcontroller/SoC designs to include hardware firewalls to help protect against cyber attacks.

The key objectives of this thesis are:

- Design and implement a hardware firewall capable of wire-speed filtering on an FPGA for IoT devices,
- Reduce the latency of hardware firewalls in embedded systems, ensuring packet classification adds the minimum possible delay after the relevant headers have been parsed, and to
- Establish a simple HTTP webserver on the RISC-V processor to facilitate user configuration of the firewall.

1.3 Scope

This thesis focuses on the core development of a hardware firewall and Ethernet controller with a RISC-V processor system. The scope of this thesis is limited to the following:

- Development of a 5-tuple binding packet filter for IPv4 networks to block unauthorised packets from reaching the microcontroller,
- Hardware design of Ethernet controller with integration to a RISC-V processor, and
- Configurability of the packet filter via a Web application.

Since cyber security is a broad topic and is constantly changing, this thesis will not cover all aspects of it. As such, the following topics are out of scope for this thesis:

- Protecting against all attacks,
- IPv6 packet handling and filtering,
- Deep packet inspection, and
- IEEE802.1Q VLANs.

Chapter 2

Literature review

This chapter introduces the necessary 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 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 [6].

There are several types of firewalls such as packet filters (PF), stateful packet firewalls and application firewalls [7]. Traditional PFs are considered as stateless and filter exclusively on the fields in the network and transport layer headers [7]. More information on the different layers in a network can be found in section 2.4. A five-tuple binding packet filter is an example of a stateless PF which filters on five key fields, source IP address, destination IP address, source port, destination port and protocol type.

Due to this, PFs are inherently simple, efficient and relatively effective in most situations. As a result, they are widely available and can be either implemented in software or in hardware [6]. The book, [6], 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 and factor in previous packets to better evaluate a packets true intention [7].

While firewalls such as *iptables* in Linux are software based, hardware acceleration can vastly improve the performance of a packet filter. As mentioned in section 2.2, hardware acceleration allows for parallelised algorithms to be executed independently of a central processing unit (CPU). Wicaksana and Sasongko, [8], proposed a packet classification engine as shown in figure 2.1. To obtain a fast yet reconfigurable and scalable packet classifier, the authors of [8] 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 a safeguard, the authors opted for a *default-deny* ruleset to prevent any unwanted traffic. One inherent downside to this design is that it requires additional clock cycles for each rule that gets added to the ruleset.

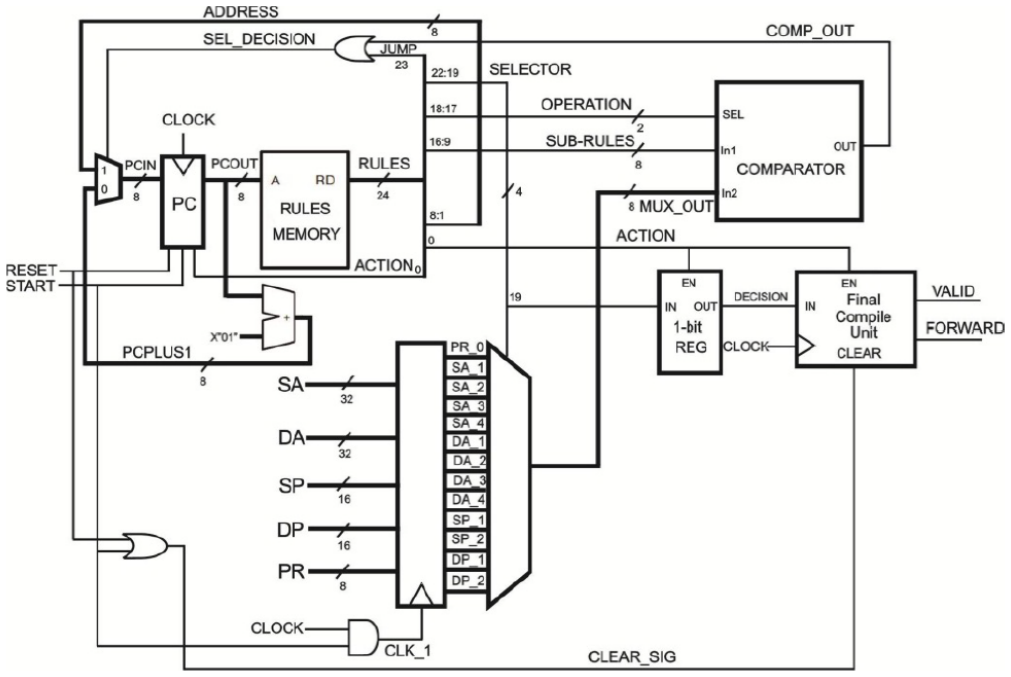


Figure 2.1: Packet classifier [8]

Wasti [9] 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 [9] is by using a *'Grid of tries'* which uses tries (a type of tree data-structure) 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 [9] is their configurability and expandability.

2.2 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 [10].

A large advantage to custom logic is the ability to create highly parallelised designs with lower latencies and higher throughput 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 including networking [11]. More specifically, system on chip (SoC) design with custom hardware acceleration modules is an active area research. As [11] points out, there is a focus towards using both hardware and software in *edge devices*[1] due to growing numbers of IoT devices.

Several papers, [12] [13] [14], have proposed a range of 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 [12] 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 [13] 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 Ethernet controllers or packet filtering hardware like the proposed designs in [12]. 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 [15]. In FPGA design, softcore processors are generally highly configurable and can be modelled in a hardware description language (HDL) which can then be synthesised onto ASICs or FPGAs hardware [15]. There are several softcore processor architectures 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. [16] 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 [17].

2.3 RISC-V processor

There are four major processor architecture families, namely AMD64, x86, ARM and RISC-V. The two former instruction set architectures (ISA) are a part of the complex instructions sets (CISC) and are found in the majority of computers such as laptops and servers. 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 [18].

RISC-V is an open and royalty free ISA and as a result, a plethora of softcore based custom implementations have been designed and are available for use in designs [19]. Consequently, there is an abundance of articles delving into RISC-V from evaluating the ISA [20] to creating multicore architectures [21]. A 2019 paper, [19] evaluated a variety of different RISC-V softcore processors.

[1] *Edge devices* are a result of the edge computing paradigm which moves computation closer to the source [4]

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 *NEORV32 RISC-V* softcore processor is written purely in vendor-agnostic VHDL and importantly has a considerable amount of documentation. The design is regularly updated by the maintainers with the original creator, Stephan Nolting, emphasising the importance of understandability.

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 [22]. The datasheet, [22], 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 [23]. This translates into accessing the hardware as a bunch of regular memory accesses like setting any other registers or values stored in main memory. This approach results in the benefit of not needing to create custom instructions for the microprocessor.

2.4 Computer Networks

To understand how one might go about creating an Ethernet interface and firewall, it is important to understand how networks operate. The TCP/IP model is a layered model that blueprints the different protocols and standards used in computer networks so that two devices can communicate in a prescriptive way. Figure 2.2 shows the updated five layers of the TCP/IP model mentioned in [24].

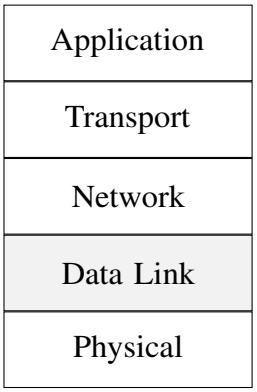


Figure 2.2: Updated Layers of the TCP/IP model

Each layer encapsulates the previous and provides a service to the layer above it. They are typically processed and handled by a separate process in either hardware or software. The physical layer is responsible for the transmission and reception of the above layers over a physical medium such as copper, radio frequency or fibre optics.

The data link layer (layer two, coloured grey) is responsible for the physical addressing of devices on the network and *links* devices together. Ethernet is the most common protocol used in the data link

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

layer and will be discussed in 2.5 and throughout the thesis. Other protocols such as Point-to-Point Protocol (PPP) also exist [24].

The network layer (layer three) is responsible for addressing and routing of packets between networks and devices. The services this layer provides is one that is analogous to a regular postal service. The one major protocol used in this layer is Internet Protocol (IP) and it works by comparing known addresses in a routing table to the destination address of the packet [24]. RFC791 [25], defines all the protocol headers and how it operates.

The transport layer (layer four) is responsible for the end-to-end communication between devices and is where the TCP and UDP protocols reside. The application layer (layer five) is what is used for application specific protocols such as HTTP, FTP, and DNS.

Putting this all together a UDP packet can be seen in figure 2.3. The dark grey headers are the ones that are inspected in the five-tuple binding firewall. The trailing layer two FCS field has been omitted for simplicity.

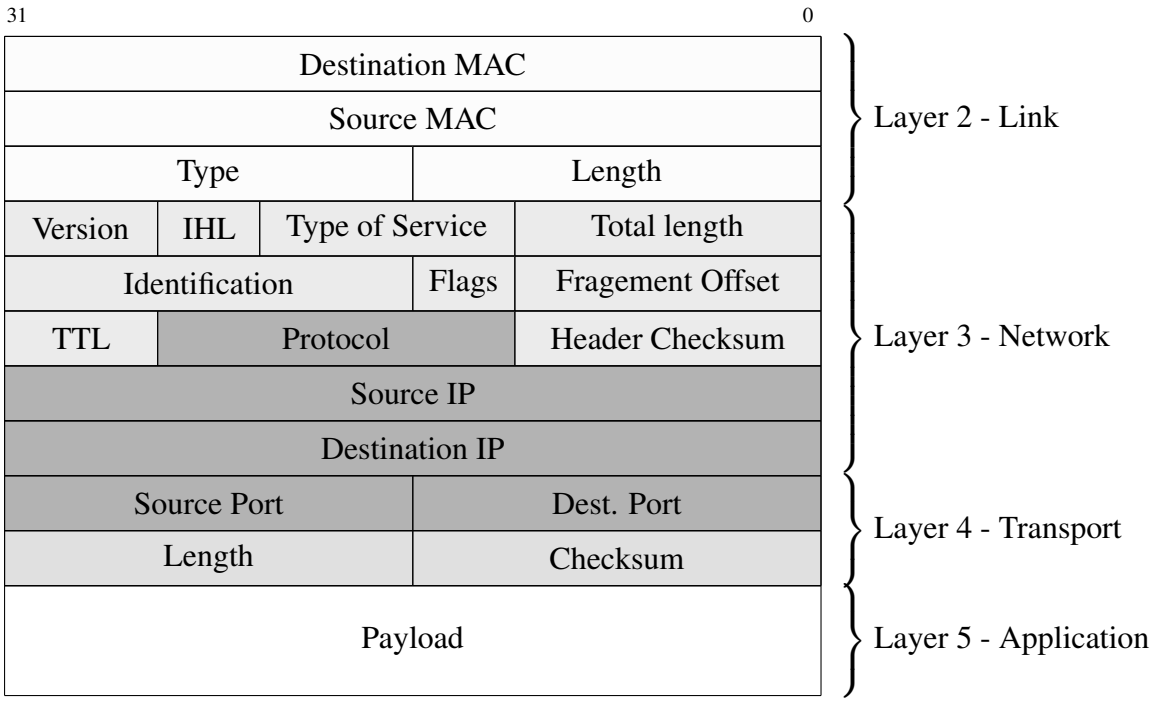


Figure 2.3: Network headers highlighting fields of interest of a UDP packet

2.5 Ethernet Media Access Control

First introduced in 1983, the IEEE 802.3 standard [26], 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 at layer two.

A core function of the Ethernet MAC is to attach the required MAC headers and preamble 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 2.4.

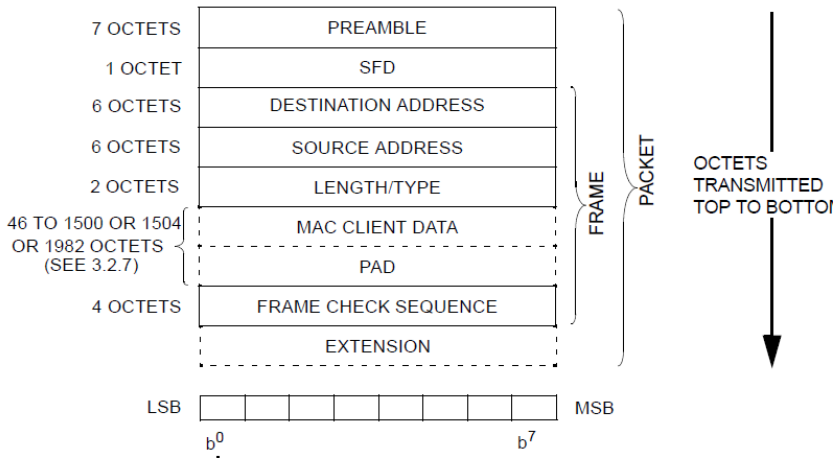


Figure 2.4: MAC layer headers [26]

The preamble and start frame delimiter (SFD) is used to synchronise the physical interface (PHY) and get it ready to send out data and is typically omitted when passing data up to higher layers such as for inspection with tools like Wireshark.

After the packet has been constructed, the data is forwarded to the PHY least significant bit (LSB) first [26]. Typically, a PHY management chip is used to handle the physical layer channel encoding and scrambling amongst other things. These PHY chips can often be interfaced with the media independent interfaces such as MII, RMII, GMII and RGMII [27]. The Reduced Media Independent Interface (RMII) is one of these standards defined in [26] and consists of a reference clock, 2 bit wide transmit (TX), 2 bit wide receive (RX) lines and a few other supplementary signals like defined in the LAN8720A datasheet, an example RMII PHY [28].

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 hardware and that parts of the MAC can operate independently [29] of each other. One key example is the calculation of the Frame Check Sequence (FCS in figure 2.4). The FCS for Ethernet is a 32bit cyclic redundancy check (CRC) [26]. CRC32 is not unique to Ethernet, but rather can be found in an extensive amount of applications. As such, prior research into parallelised architectures for the calculation have been made by others. Notably, Mitra and Nayak [30] 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, [27] [31] [32], can be found about implementing Ethernet MACs on FPGAs each with a slightly different approach and focusing on different properties. Fundamentally though, as best highlighted in [27], a simple way of implementing a MAC is by employing a finite state machine (FSM) like the one in figure 2.5 to set the required fields and send out the packet. Another technique found in these articles is to use first-in first-out (FIFO) buffers to cross clock domains as the PHY speed likely won't match the clock frequency of the rest of the system. 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 [31].

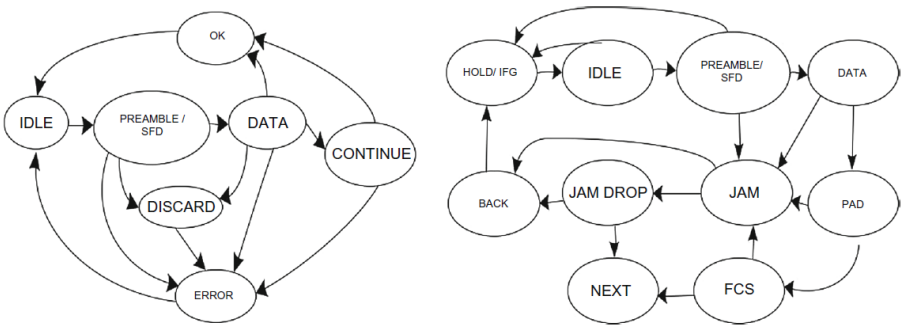


Figure 2.5: An example Ethernet MAC FSM [27]. Left: Receiver FSM, Right: Transmitter FSM

The source address field is typically filled in by hardware and is fixed as the MAC address is supposed to be globally unique to prevent address clashes on the same network. The destination address is typically set by software however as the ARP (Address Resolution Protocol) tables are typically stored in software due to them storing a mapping between the layer three IP addresses and the MAC addresses to forward a packet to.

The Length/Type field has a dual purpose, and it depends on the value. If a value is less than or equal to 1500, then it indicates the size of the packet. While if a value of greater than 1500, then it indicates the type of packet, called EtherType [1]. Common values are 0x0800 for IPv4 packets and 0x0806 for ARP. Like with the destination address, this is usually populated by software as the type is dependent on the above layers.

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].

2.6 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 5). As embedded platforms are resource limited, special precautions need to be taken into consideration when it comes to memory and resource usage [33].

Article [12] 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 [34] [33]. Often these papers run LwIP on real time operating systems (RTOS) such as FreeRTOS or Zephyr. These provide an abstraction to the hardware that allows for multitasking and brings other OS-Like features to embedded systems.

[1] See https://en.wikipedia.org/wiki/EtherType for the different types
[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

LwIP is not threadsafe and typically suffers from memory issues as found in [33]. However, FreeRTOS's own TCP/IP network stack called *FreeRTOS-Plus-TCP* provides a threadsafe Berkley sockets API and is newer. Consequently, less research can be found apart from existing documentation. These libraries typically implement multiple protocols such as DHCP, DNS, TCP, and UDP [35].

RFC2616 [36], defines the HTTP protocol used by all browsers and web servers and includes the formatting, allowed characters and the different request methods for each packet. HTTP1.1 and HTTP2 use TCP, typically on port 80, to communicate between the client and the server. However, HTTP3 uses the UDP protocol as defined in RFC9114 [37]. For the remainder of this thesis *'HTTP'* will be used to refer to HTTP1.1.

The HTTP specification, [36], defines the protocol as a stateless request-response type where the client first initiates a request to the server and the server responds with a status code and the requested data such as HTML data or images. *GET* and *POST* requests are the most common type and allow data to be retrieved and sent to the server. Since the browser renders the received HTML the data on screen, the webserver only needs to forward the raw HTML data (which can be stored on physical media) to the client.

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 [38]. 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.

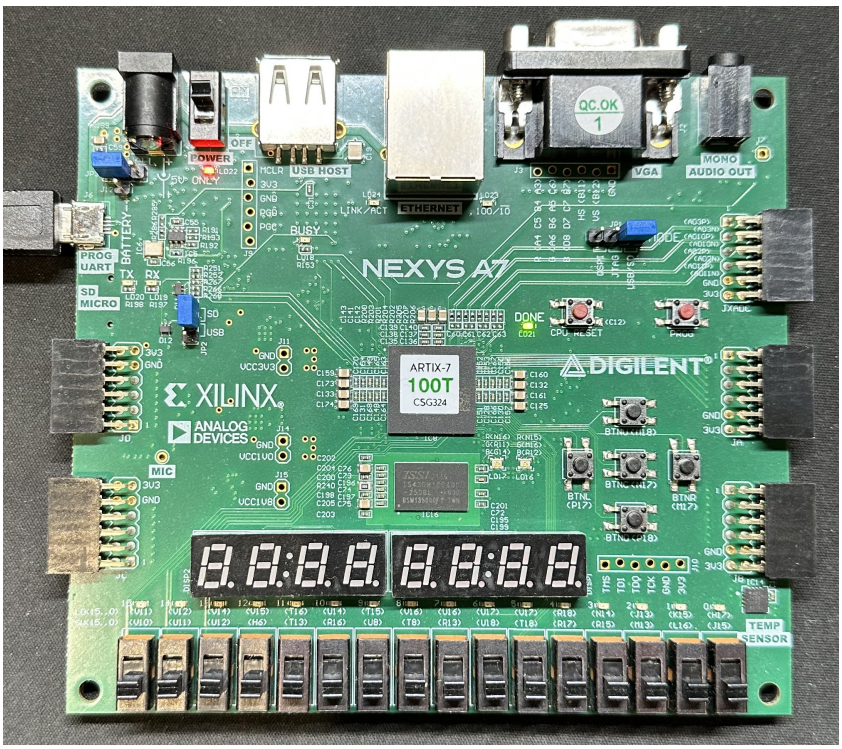


Figure 3.1: Digilent Nexys A7 FPGA development board.

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 80MHz, giving a bandwidth of $32 \times 80 \times 10^6 = 2.56 \times 10^9 bit/s = 2.56 Gbit/s$. 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

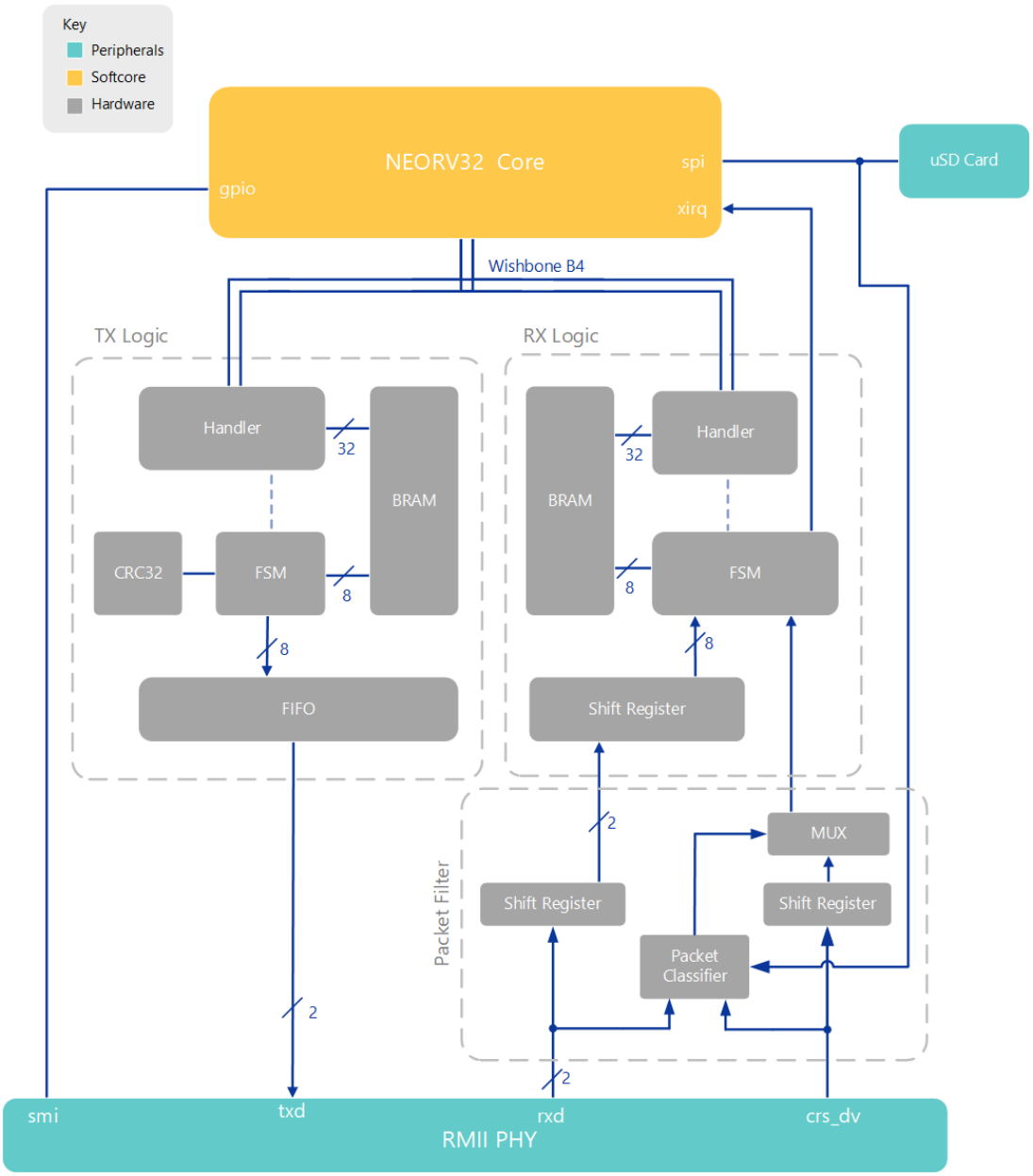


Figure 3.3: System on Chip high level architecture.

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. As there are four clock cycles per byte the needed register size is $50 \times 4 = 200$ 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.

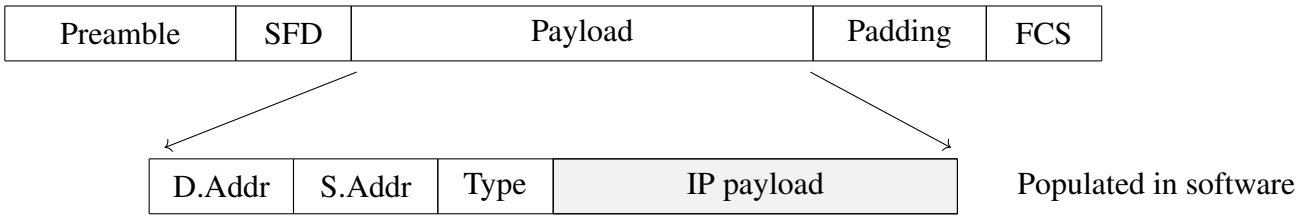


Figure 3.4: Format of frame in BRAM

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 80MHz clock signal, consequently the equivalent bit rate is 6/4 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.

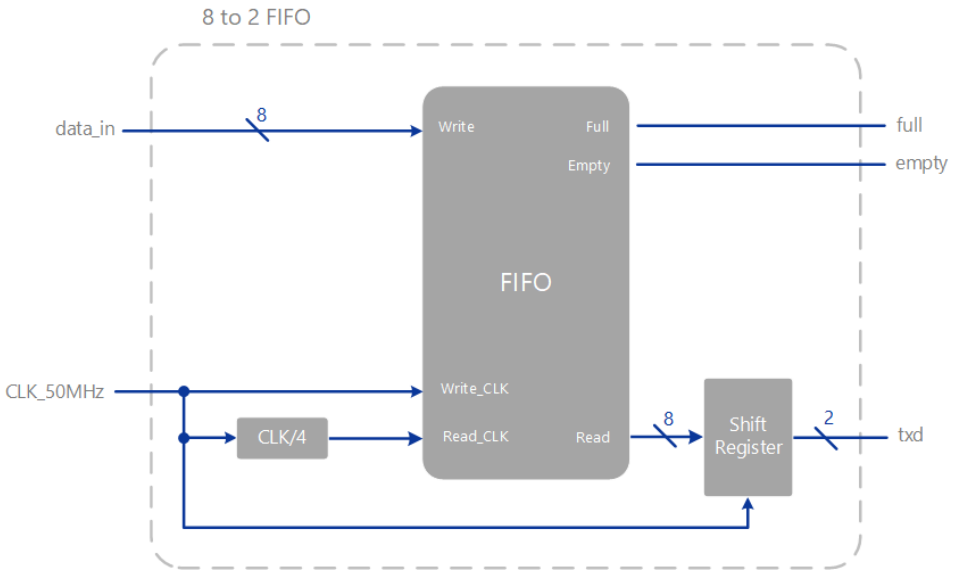


Figure 3.5: 8bit to 2bit FIFO used for RMII output.

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

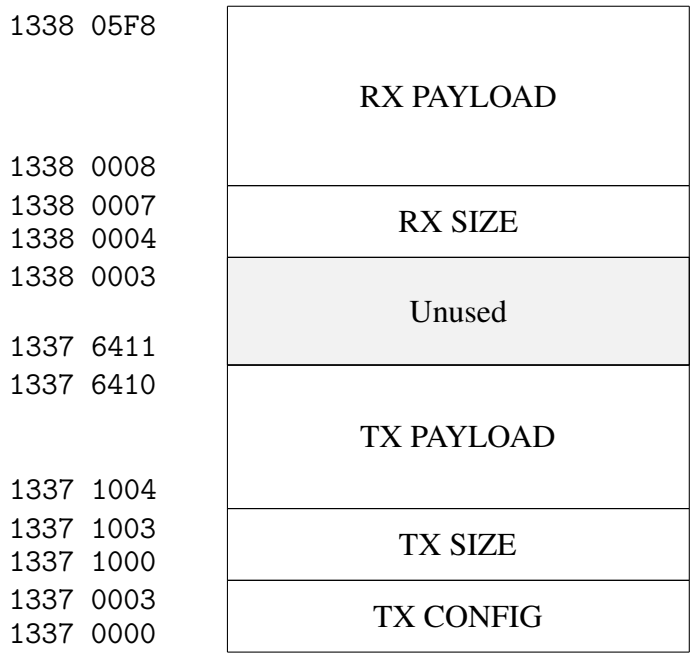


Figure 3.6: Memory Address Layout

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 3.8 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 [8] as multiple clock cycles would be needed to classify the headers. In theory

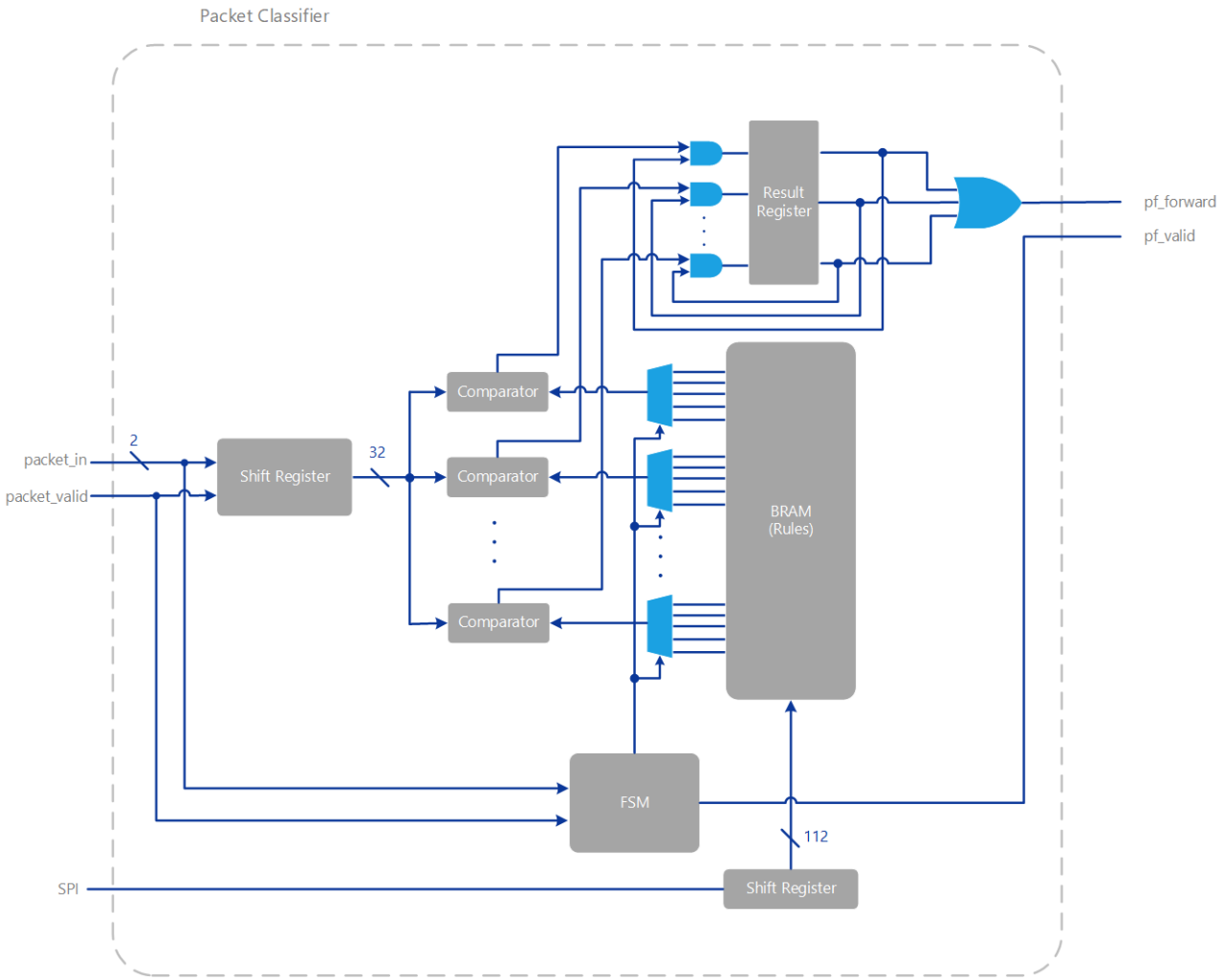


Figure 3.8: Packet classifier architecture. Clock signals have been omitted.

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, ([28]) 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.

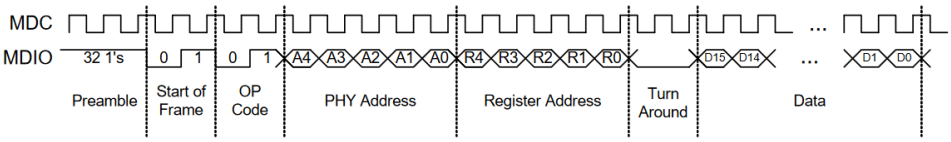


Figure 3.10: SMI Write message structure. [28]

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 80Mhz (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.

The network stack was configured to use DHCP to automatically aquire an IP address from a DHCP server on a network so that when it's transported between networks it can dynamically get a new IP address and can be accessed without needing to reconfigure the device. The device can then be

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

given a static IP through the DHCP server by creating a static mapping. This will then prevent the DHCP server from assigning the desired IP address to another device accidentally.

3.3.4 Web application

With the resource constraints of the RISC-V microcontroller in mind and other design requirements, Vue.js was chosen as the framework of choice for developing the web application in. Vue.js is a framework for developing componentised single page applications. Although technically single page, the javascript it compiles handles routing between web pages on the client side in javascript. This means that once the initial page has loaded, no additional server requests need to be made to change page. To make the website appear dynamic, an API can be used to fetch data and statistics from the device itself. These HTTP API requests are much smaller than the requests needed to load the webpage initally. This results in very low network traffic between the client and server, perfect for embedded systems.

For styling, Tailwindcss was used to help style the webpages to give the webapp a contempory feel.

The webapp consisted of 3 main pages, the index or home page, about page and config page. The home page, figure 3.11 showed statistics such as total and blocked packet count, and statuses for a few tasks on the microcontroller. These tasks, through an API call, can be toggled on and off to save resources. An uptime reading is also present. These values are updatable by clicking the 'refresh' button.

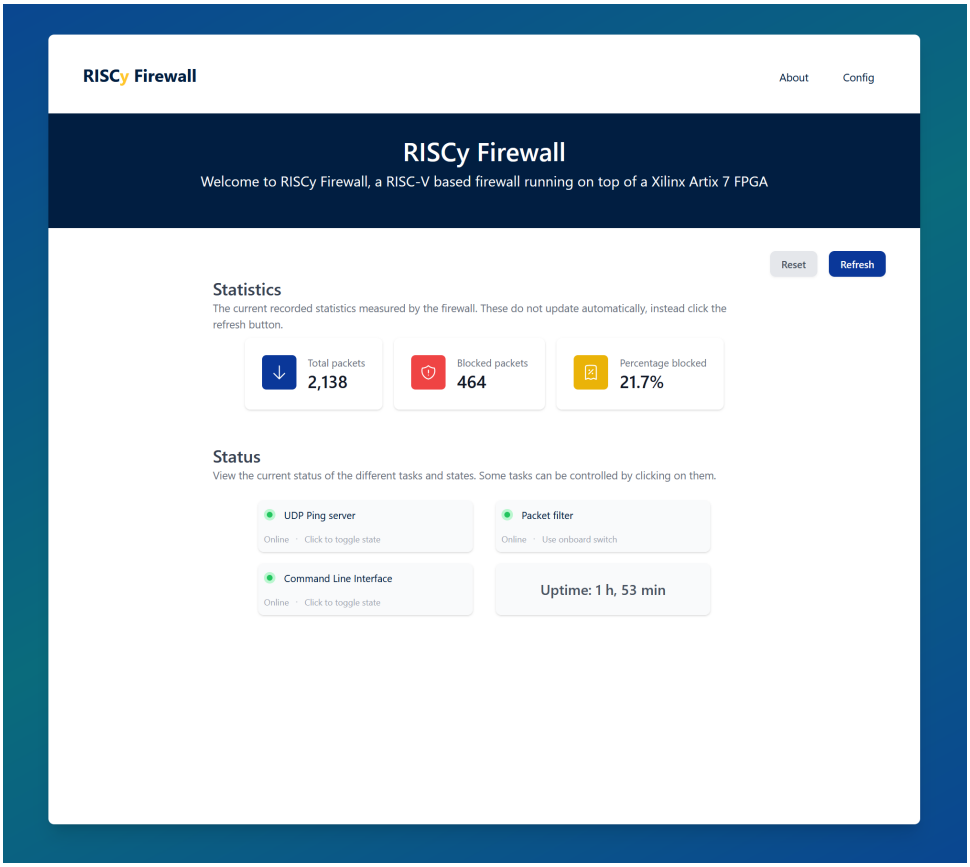


Figure 3.11: Screenshot of homepage on webapp.

The configuration page, (figure A.3 shown in appendix A.3) is where the user configures the firewall rules. The rules first need to be loaded which is requested through an API request. This API request reads the rules from the microsd card, sets the rules (for consistency) and then returns them in the response packet. When the user saves a set of rules, an API request is made which first applies the rules and then saves the rules to the microsd card so that the changes are persistant.

An about page (figure A.2 shown in appendix A.3) was also created for demonstration purposes and contained an image.

In addition, as this is all client side and compiles to html, css and javascript, no special webserver is required for things such as server side rendering.

3.3.5 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. All the TCP webserver has to do is determine the route requested by the client, based on this it knows whether to read a file from the micro SD card and send it over multiple TCP packets or if it needs to respond with raw data like in the case of the API. Regardless of the type of request, a response is formed in the standard HTTP 1.1 format given in RFC2616 [36].

3.3.5.1 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).

3.3.6 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 and Discussion

This chapter aims to demonstrate the practicality and efficacy of the designed hardware. Performance metrics such as latency, resource utilisation, timing, and power consumption are analysed to gauge the design's effectiveness. The design is also tested with a small sample of real life packets of different properties to ensure the packet filter can correctly block unwanted packets. Preexisting solutions are then compared with the design, providing insight into the design's relative strength and weaknesses in areas of latency, power and thermal performance.

4.1 Latency Performance

Reducing the latency of hardware packet filtering in embedded systems is one of the key objectives of this thesis. As such, verification of the latency added due to the filtering is essential.

4.1.1 Theoretical analysis

The design employs a 200-stage shift register to temporarily store the incoming packet with each stage being 2bits wide. Given a clock frequency of 50Mhz, the added latency can be calculated to be $200 \times \frac{1}{50 \times 10^6} = 4 \times 10^{-6} = 4\mu s$.

4.1.2 Measured analysis

The packet classifier's performance was measured with an Agilent MSO6054A MSO due to its high sampling rate of 4GSa/s. A PMOD pin was connected to output of an xor operation between the carrier sense line (crs_dv) from the PHY and the crs_dv post packet filtering.

This process allowed the added latency to be measured characterised by the time between the rising and falling edge of either pulse as shown in figure 4.1. The measured output can be seen to match the theoretical calculation of $4\mu s$.

The two distinct pulses are a result of the phase shift between the PHY crs_dv and the delayed crs_dv line. Additionally, the distance between the two pulses indicates the size of the packet.

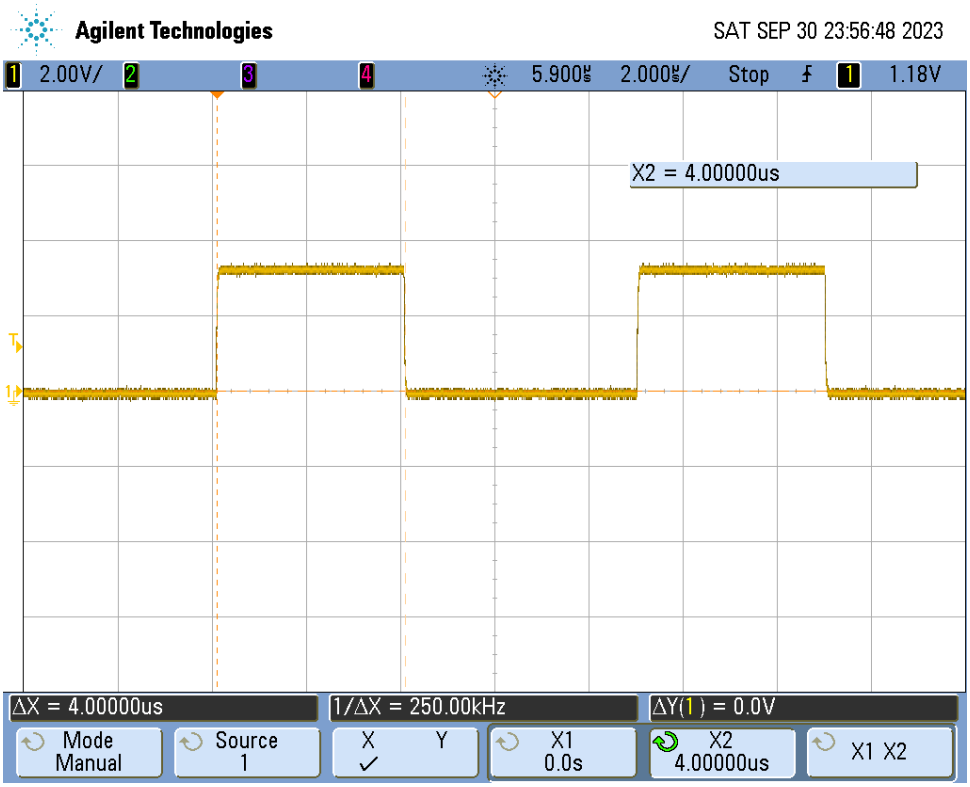


Figure 4.1: Added latency by packet filter waveform.

4.1.3 Improvements

A potential improvement on the design could be to integrate the packet classifier with the Ethernet MAC. This could reduce the added latency to zero by removing the need to store the packet in the additional shift register. The approach would involve storing the incoming packet and if the packet is later deemed to be blocked, the controlling FSM could be reset to ignore the packet.

4.2 Utilisation

Resource utilisation is an integral part in validating the feasibility of implementing the design on a particular FPGA or microchip. This section details the post synthesis resource utilisation of the design on the Nexys A7-100T FPGA using Xilinx Vivado 2022.2. Namely, the NEORV32, Ethernet hardware and packet filter are analysed.

The resource breakdown referred to in this section can be found in figure 4.2 while a more detailed breakdown of the resource utilisation including the primitives can be found in appendix A.2.

The design as a whole uses a total of 64.5% of the available slice LUTs, 12.9% of the available flip flops and 96.3% of the available BRAM.

4.2.1 NEORV32 processor

significant LUT usage was attributed to the 32-bit wide wishbone interface. Considerable LUT consumption relates to frame buffers used in Ethernet hardware, elucidated further in subsection 4.2.2.

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 4.2: Summary of the resource utilisation on XC7A100T FPGA.

The NEORV32 SoC consumed the majority (69.5%, or 28455), of the slice LUTs in the design. While many interfaces were enabled such as SPI, UART, GPIO, external interupts (XIRQ) and the true random number generator, a considerable amount of the LUTs were consumed by the 32bit wide wishbone interface. More specifically, 25992 slice LUTs, or 91.3% of the NEORV32 usage.

The large LUT utilisation relates to the frame buffers used in the Ethernet hardware, elucidated further in section 4.2.2.

DSP48 blocks were also used by the SoC to handle the multiply operations to free up LUTs.

The instruction memory (IMEM) and data memory (DMEM) sizes were configured to optimise the remaining BRAM blocks for increasing flexibility in firmware. Specifically, IMEM was allocated 256KB while the DMEM (acting as RAM) amounting to 168KB.

4.2.2 Ethernet hardware

Comparatively, the Ethernet hardware accounted for 11738 slice LUTs and 12505 slice registers, most of which is consumed by the transmit logic (78% LUT and 99% registers). The considerable LUT utilisation in the transmit logic and NEORV32 Wishbone interface is due to the manner in which the frame buffer is written to. The complex operations such as address validation and array modification for writing to the frame buffer can be seen in listing 4.1. Notably, the address validation specifically implements a 32bit wide comparator and the need to write/modify to the frame buffer array causes large LUT utilisation.

```
1 if wb_i_addr >= x"13371004" and wb_i_addr <= x"13376410" then
2     -- Subtract 4100 from the address to get the virtual address
3     virtAddr := to_integer((unsigned(wb_i_addr(15 downto 0)) - 4100));
4     FRAME_BUFFER(8 + virtAddr) := wb_i_dat(31 downto 24);
5     FRAME_BUFFER(9 + virtAddr) := wb_i_dat(23 downto 16);
6     FRAME_BUFFER(10 + virtAddr) := wb_i_dat(15 downto 8);
7     FRAME_BUFFER(11 + virtAddr) := wb_i_dat(7 downto 0);
8 end if;
```

Listing 4.1: Code for writing to the frame buffer

This design choice also influenced the critical path delay as detailed further in section 4.3. Optimisations to this area would be imperative to drastically reduce the resource consumption and improve the critical path delay.

By contrast, the receive logic consumed far less resources of 12% of the Ethernet MAC's LUTs and only 1% of the registers. This is attributed to the simpler design which stores the incoming packets with the correct endianness in the frame buffer before subsequently triggering an interrupt upon the packet's arrival.

The rmii_init logic acts as the glue logic between the PHY and the MAC. It facilitates the clock and bus domain crossings between the 8bit wide 80Mhz MAC and the 2bit wide 50Mhz RMII PHY.

4.2.3 Packet filter

The packet classifier consumed a total of 571 slice LUTs and 1145 slice registers, suggesting a potential to increase the number of rules. Though, the fan-in and fan-out will likely need to be considered due to the nature of the implementation. The synthesiser's decision to use registers over BRAM for rule storage is likely due to the rules being small in size and that the BRAM blocks are better used elsewhere.

Alternatively, the minimal resource utilisation indicates that the design is suitable for implementation on smaller FPGAs or ASIC designs and will have negligible impact on the overall silicon area.

4.3 Timing Summary

Timing analysis of the design provides insight into the critical path delay and the maximum frequency of the design. Basic results are presented in this section and are primarily around the Wishbone interface due to its role in the critical path. Importantly, the Wishbone interface operates at the same clock frequency as the NEORV32 processor which directly impacts the speed of the CPU for software based operations.

Initially, a single 50Mhz clock sourced from the onboard MMCM was used. A post-synthesis analysis indicated a positive slack in the design, allowing for the clock frequency to be updated to 80Mhz. This was done to increase the performance of the CPU so that it could process network packets faster in software.

After resynthesising, a worst negative slack for setup time of -2.634ns was reported with a total of 82 endpoints failing the constraints. Additionally, a worst hold slack of -0.021ns was reported. The identified critical path originates from the Wishbone interface to the BRAM block within the Ethernet hardware, shown in figure 4.3. Due to the NEORV32's implementation of the Wishbone bus, the critical path cannot be easily improved and is the leading limitation in the speed of the design.

While in practice these timing constraints do not crucially impact the design's operation, caution is advised in future adaptations or usage. Other paths were also encroaching on a slack of zero and hence

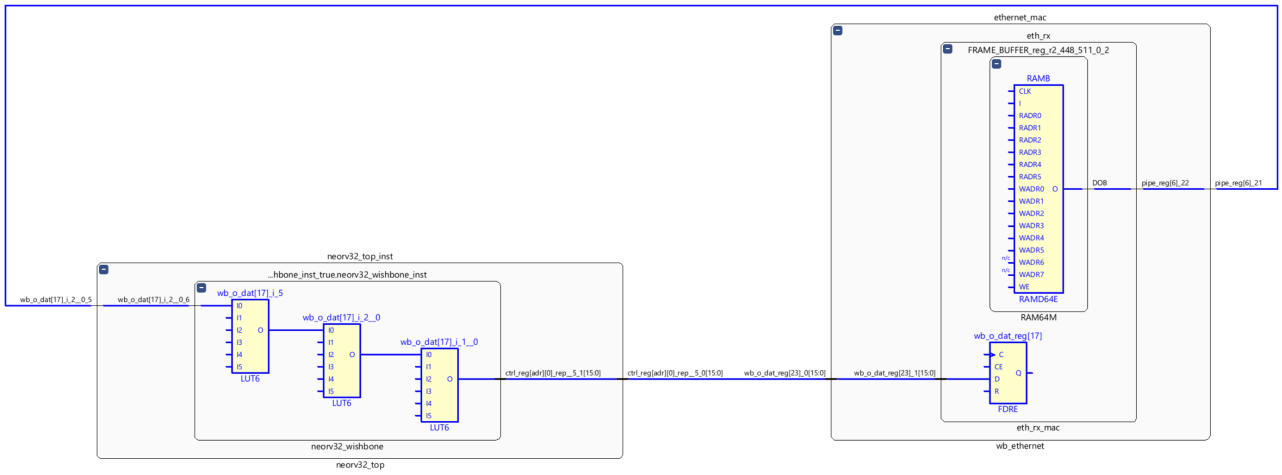


Figure 4.3: Critical path in SoC design.

the design was limited to 80Mhz. Testing at 90Mhz was conducted, however, the design was found to be unstable and had a large increase in the number of failing endpoints.

More accurate timing constraints would need to be set to acquire the absolute maximum frequency of the design. Though, this was outside the scope of this thesis.

4.4 Filtering performance

Blocking unwanted packets is imperative to a firewall's operation with any packets bypassing the filter rendering it useless. Testing all possible permutations of bits going through the firewall is infeasible due to the sheer number of packets needed, which is on the order of 2^{104}. For example, a complete testing suite for all source IP addresses alone would require 2^32 (about 4.3 billion) packets to be sent to the device. Therefore, a small sample of packets, shown in table 4.1, were tested to ensure the filter is working as intended.

4.4.1 Test setup

Four hosts distributed across two networks were used to test the device as depicted in the network diagram in figure 4.4. The network consisted of three Raspberry Pi 4s and an x86 based Ubuntu machine. The specifics characteristics of these devices are irrelevant to this thesis, given that they all adhere to the TCP/IP standards. More specifically they were chosen due to their capability to run python scripts and are accessibility over Ethernet. A common script[1] was used on all devices to test each of the services available on different ports and protocols.

The rules shown in table 4.1 shows the rules that were loaded on the packet filter. While these test cases is only a subset of the total coverage, a mix of different combinations of ports, protocols and IP addresses were chosen. More specifically the rules were chosen to test the following cases:

- All devices can access the device in some way (eg, over ICMP),

[1]The python *test* script can be found at https://github.com/matty0005/thesis-tools

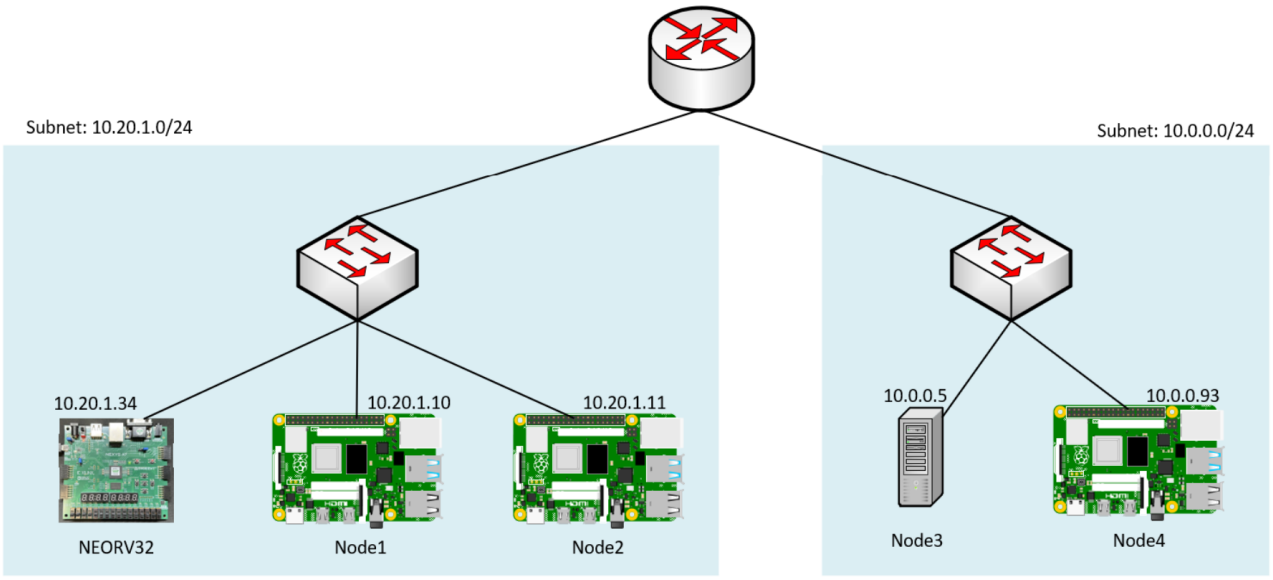


Figure 4.4: Network architecture for firewall tests.

- All machines could access at least two services,
- Identical ports over different protocols can be filtered (eg, port 1337 over TCP and UDP),
- Ports can be blocked on a device even though the device can access other services with the same protocol,
- The wildcard operator works in each field, and
- Any port that is not explicitly allowed is blocked.

By default, as the packet filter is a default-deny, any packet that doesn't fit these constraints is blocked and can be verified in the tests summarised in 4.2.

Table 4.1: Firewall configuration for testing

Source IP	Destination IP	Source Port	Destination Port	Protocol
any	10.20.1.34	n/a	n/a	ICMP
10.0.0.5	10.20.1.34	any	any	UDP
10.0.0.5	10.20.1.34	any	80	TCP
10.0.0.93	10.20.1.34	any	1337	TCP
10.0.0.93	10.20.1.34	any	9999	UDP
10.20.1.10	10.20.1.34	any	1337	any
10.20.1.10	10.20.1.34	any	any	UDP
10.20.1.11	10.20.1.34	any	9999	UDP

Typically, firewalls have a second interface that connects to a local network, allowing packets from multiple IP addresses to be processed and forwarded on. However, in this setup, a single device is behind the firewall, requiring the *destination IP's* to be the same (the NEORV32's IP address, 10.20.1.34).

Additionally, the *source port* was configured to allow any source port through. This is important due to the client arbitrarily choosing the source port at random. Therefore, a wildcard was used as this makes the source port impossible to predict. It's worth noting that some niche programs operate with predicable ports, though none of the programs used in this test were of this nature.

4.4.2 Test results

A summary of the results can be seen in table 4.2 which show the firewall blocking and allowing the respective packets correctly.

Table 4.2: Summary of packets including expected and actual outcomes

Device	Dest. IP	Protocol	Src. Port	Dest. Port	Expected Outcome	Actual Outcome
Node1 (10.20.1.10)	10.20.1.34	TCP	Any	1337	Allow	Allowed
Node2 (10.20.1.11)	10.20.1.34	ICMP	-	-	Allow	Allowed
Node3 (10.0.0.5)	10.20.1.34	UDP	Any	9999	Allow	Allowed
Node4 (10.0.0.93)	10.20.1.34	TCP	Any	1337	Allow	Allowed
Node1 (10.20.1.10)	10.20.1.34	UDP	Any	1337	Allow	Allowed
Node3 (10.0.0.5)	10.20.1.34	TCP	Any	80	Allow	Allowed
Node3 (10.0.0.5)	10.20.1.34	TCP	Any	1337	Deny	Denied
Node2 (10.20.1.11)	10.20.1.34	TCP	Any	1337	Deny	Denied
Node4 (10.0.0.93)	10.20.1.34	UDP	Any	1337	Deny	Denied
Node4 (10.0.0.93)	10.20.1.34	TCP	Any	80	Deny	Denied

The detailed output from the script on each host is available in appendix A.7. While these tests don't formally prove the firewall is correctly working, they do provide substantial evidence of the design's correctness. It's important to note that additional rigorous testing, which wasn't formally documented, was conducted throughout the development of the design. Such an example test case involves spamming ping packets from one host while sending valid packets from another.

These results stipulate the devices suitability for use in a real world environment. The filter exhibits capacity to filter out the significant majority of unwanted packets, making it ideal for low security applications. However, formal verification and meticulous testing is imperative to suitability for high-security applications. With the absence of these in the above tests, the design would necessitate further validation to be deployed in high security environments.

4.5 Comparison to preexisting solutions

To ensure the effectiveness of the designed hardware, some comparisons to preexisting solutions have been made. Three other devices that featured *Fast Ethernet* (100Mbit/s) connectivity were compared. These devices were the WIZ5500 Pico [1], Nucloe-F767ZI [2] and MilkV-Duo [3]. The WIZ5500 pico board uses a *Raspberry Pi RP2040* as the MCU at 133Mhz, while it uses the *WIZ5500* IC for handling the Ethernet traffic. The WIZ5500 additionally handles layers 1 to 4 onboard and is interfaced over SPI. The Nucleo-F767ZI (referred to as just F767ZI) is powered by the *STM32F767* MCU and features an onboard *LAN8742A* PHY chip onboard. Like the Nexys board, it is also connected over an RMII interface, making it a logical comparison to the Nexys A7. The STM32 has hardware support for the RMII interface. Unlike the Nexys A7, it utilised the LwIP network stack instead of FreeRTOS-Plus-TCP. Finally, the *MilkV-Duo* is the most recent RISC-V based board which features the 64-bit*CVITEK CV1800B* processor. The CV1800B operates at a 1Ghz, includes 64MB of RAM and provides the MAC and PHY inside the chip itself.

4.5.1 Network latency tests

A small UDP ping program was made on the NEORV32 to receive a UDP packet on port 9999 and reply to the client to test the round trip time (RTT). This test also had the benefit of quantitatively determining the software overhead on the platform. The round trip time was then averaged over 1,000 transactions and has been presented in figure 4.5. To determine the amount of software overhead the TCP/IP stack had on the RTT, two packets of different payloads was tested. This is because the time difference in hardware between the packet size is in the order of nanoseconds, whereas the software processing time is several orders of magnitude larger due to moving more data around and processing it. The two UDP payload sizes were 8 bytes and 256 bytes.

A table of the results can be found in appendix A.4. Testing on the Nexys A7 observed a larger difference compared to the competing products. This indicates that the software overhead on the Nexys A7 is larger. The difference in delay at the hardware level can be calculated by knowing the number of bits and the clock frequency. Fast Ethernet has a throughput of 100Mbit/s, which when sending 248 bytes results in a delay of $\frac{248\times8}{100\times10^6} = 19.84\mu s$. This is likely due to the FreeRTOS-Plus-TCP stack

[1] See: https://www.wiznet.io/product-item/w5500-evb-pico/
[2] See: https://www.st.com/en/evaluation-tools/nucleo-f767zi.html
[3] See: https://milkv.io/duo

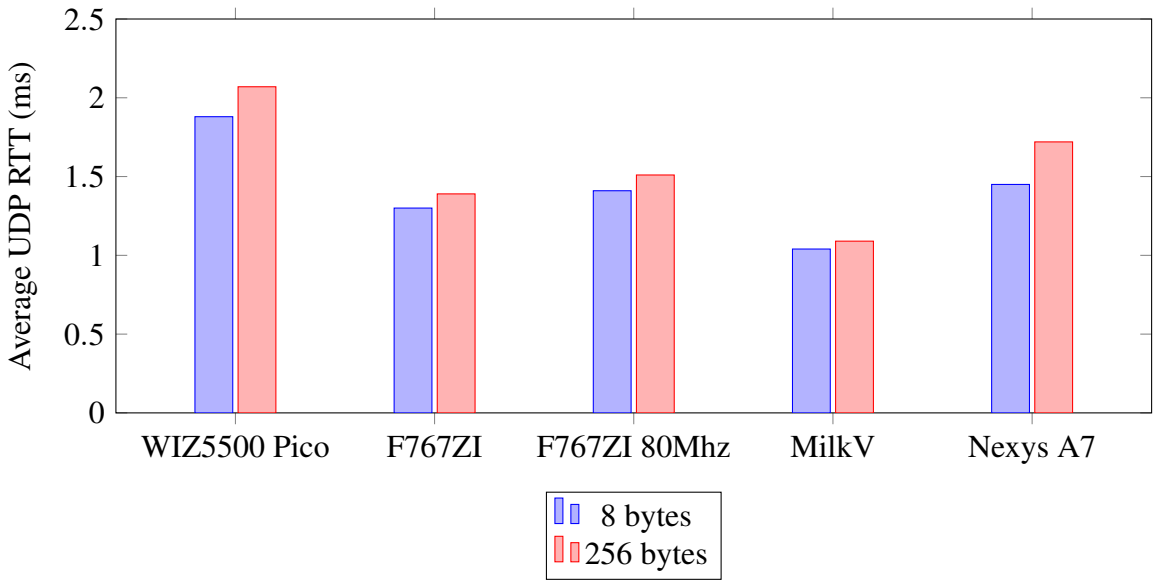


Figure 4.5: Average UDP RTT for different devices and payload sizes.

using more computations per packet than LwIP on the F767ZI. The MilkV not only runs linux, it operates at 12.5 times the frequency, 1Ghz.

This test results indicate that the device is relatively comparable to preexisting solutions in terms of networking latency. To further compare the network performance, network throughput tests were conducted.

4.5.2 Network throughput tests

In this section, only the F767ZI, MilkV Duo and Nexys A7 were tested. The WIZ5500 Pico was not tested as no direct support for Iperf 3 was available. Iperf 3 was used to test the bandwidth of the network between two devices. The client device was a desktop connected at gigabit speeds to the network. Iperf 3 was chosen for it's a universally accepted throughput measuring tool and that both FreeRTOS and LwIP had Iperf 3 server implementations. Table 4.3 summaries the bitrates of the tested devices.

Table 4.3: Bitrate of various embedded devices

Device	Bitrate
Nexys A7	1.32Mbits/s
F767ZI	7.11Mbits/s
MilkV	92.4Mbits/s

It came at no surprise when the MilkV Duo with it's 1Ghz processor saturated the 100Mbits/s *Fast Ethernet* interface. This test is consistent the idea that the software on the Nexys A7 is the bottleneck as alluded to in the latency tests. It is also of the belief that the FreeRTOS task priorities and interrupts on the device are not optimal which is also decreasing the performance. This was evident when changing the tick frequency in FreeRTOS from 500Hz to 1000Hz as it gave more time for the packet to be processed before switching tasks. This resulted in a 41.8% drop in performance. The

F767ZI was able to achieve a higher throughput than the Nexys A7 which is likely due to using LwIP over FreeRTOS-Plus-TCP and having more optimised drivers that handle interrupts and better task scheduling.

While the Nexys A7 provides comparable performance in the UDP ping tests, the throughput is lacking, but can be improved with optimisations in the firmware of the device. However, in having a hardware firewall, the design in this thesis has an advantage in security, which is discussed in the next section.

4.5.3 Security analysis

Only the Nexys A7 designed in this thesis employed a hardware packet filter. The other devices required a software based packet filter. The issue with software based firewalls in embedded systems is that they may be susceptible to power-glitch attacks. Power-glitch attacks are a common vulnerability in embedded systems which consists of switching off and on the power very rapidly at a critical point in the code to essentially bypass a certain instruction. While this is highly unlikely and often hard to implement, it is an advantage nonetheless for the hardware firewall.

As the packet filter is designed in hardware and is directly reading the bits from the PHY and doesn't operate in the same way as a software implementation, the hardware packet classifier in this design should not susceptible to power-glitch attacks. Formal testing of this however was not part of the scope of this thesis. Another benefit of the design created in this thesis is the latency properties, which is discussed in the next section.

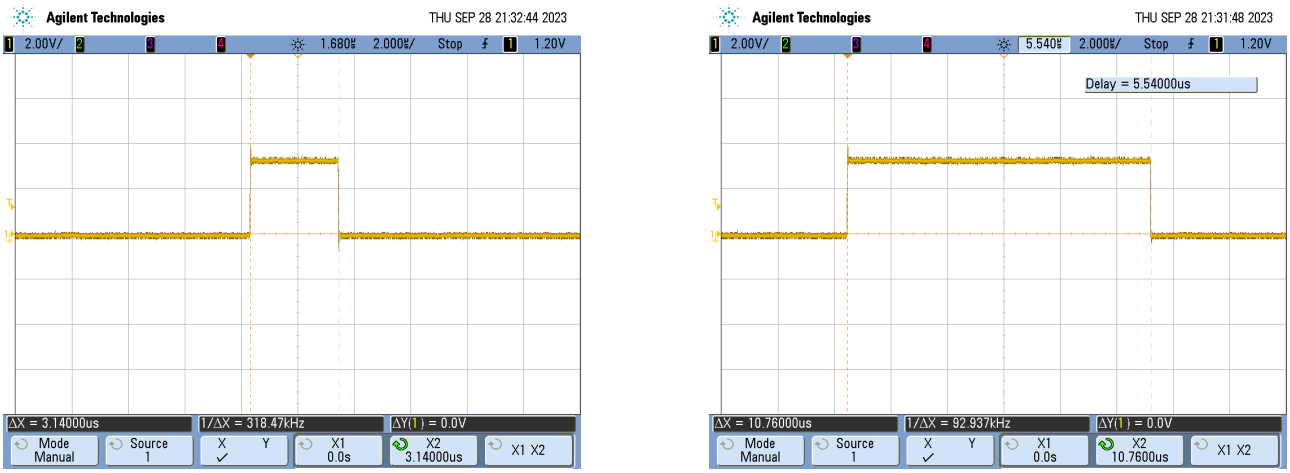
4.5.4 Firewall latency

To measure the latency of the software firewalls accurately, a GPIO pin was configured to toggle when the packet first enters the packet filter and then again after it has been filtered. It is assumed that the latency of setting the pin high cancels out with the latency of setting the pin low and that it simply results in a phase shift and does not introduce any additional delay between the start and end.

A software based implementation of the firewall was created on the Nucleo-F767ZI board with eight rules to make for a fair comparison as the FPGA design is limited to eight rules. Using an oscilloscope to measure the time, the measured classification delay depended on the number of rules and where a valid rule would be matched, if at all, at the start or end of the sequence.

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

The lower delay in the ideal software case is due to the small number of rules needed to be considered reducing the number of comparisons needed to be made. In practice, it is expected that the first rule is not always matched, and hence the average case should be used for an amortised latency. Importantly these delays unlike the FPGA one, do impact throughput as the processor is limited to filtering the packets and not doing other things like receiving another packet. In a low bandwidth environment, there is little benefit other than security over a hardware firewall in an



(a) Best case scenario

(b) Average case

Figure 4.6: Software packet classifier timings

embedded system.However, if latency is critical, like it is in real time robotic control systems and bandwidth is high or even if the highest security is needed such as in secure access control systems, a hardware firewall provides a better alternative to a software implementation. Another difference between hardware and software is efficiency of the design, which will be detailed in the next section.

4.5.5 Power consumed between boards

No measurable difference in current consumption was observed when using the packet filter in hardware over it being disabled. This is likely due to the measurment equiptment not featuring a high enough resolution. However, when implementing a software firewall, a relative current increase of 1mA was observed which is due to running more comparisons per loop.

The remainder of this section compares the absolute current measurements recorded across the devices which can be seen in figure 4.7. In this figure, four measurements were made, Idle, Busy, No Ethernet and Clean state. The Idle is just the design loaded, but idling and not receiving any packets. Busy is the measurement when the device is receiving packets and processing them. This is typically lower due to the flashing status LED providing more of an impact than the additional power consumed by the hardware. No Ethernet is the measurement when the Ethernet cable is unplugged and the device is idling. This is an important measurement as the PHY chips tend to implement a low power sleep state when no Ethernet cable is connected. Clean state is the measurement when the device is idling and with no design loaded. This is effectively the quiescent current consumed by the device.

In Appendix A.5 figure A.5 shows the same data in table form presenting numerical values.

From figure 4.7, the difference between the clean state and the busy states can be learned. The Nexys A7 had the largest difference at 99.27mA, compared to the 87.94mA difference witnessed in the F767ZI board. What must be taken into consideration is that the measurement for the Nexys A7 is without the hardware design (both Ethernet MAC, packet filter and RISC-V core) applied. Another reading of 283.75mA was observed when the design and hardware had been loaded (no firmware). What this means is that even though the Nexys A7 has the highest quiescent current amoungst the

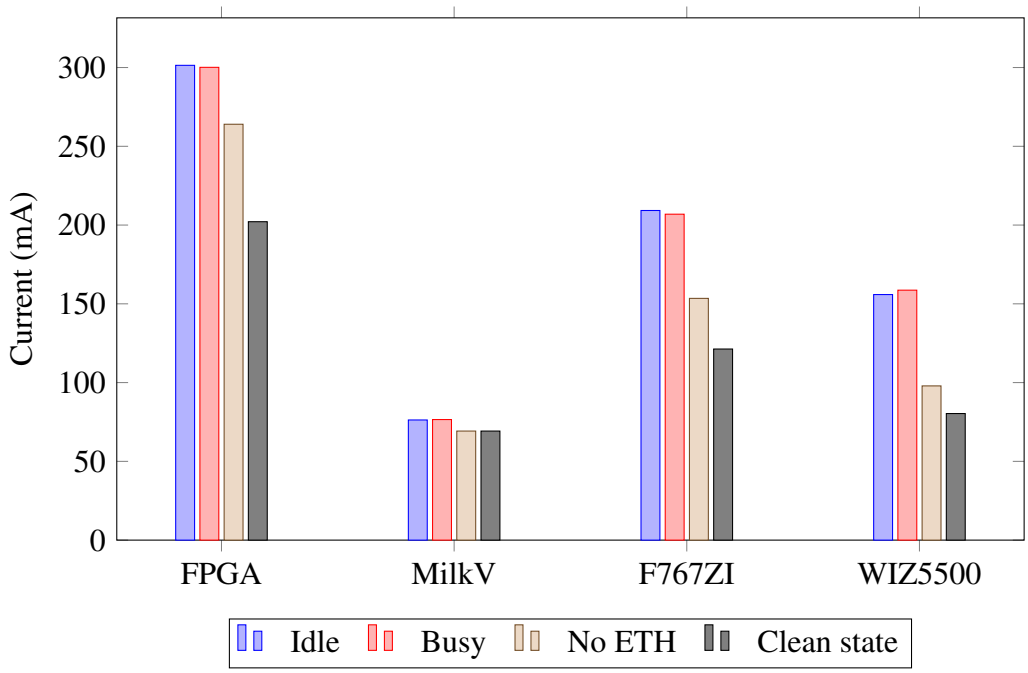


Figure 4.7: Comparison of Idle, Busy, and No eth currents for devices

devices, it still has the larger design current for the project itself. Power optimisations would be needed to be made in both the Ethernet and RISC-V core to reduce this current to perform better than the preexisting solution. However, this was not a focus of this thesis.

As it stands, this result indicates that the Nexys A7 board is not ideal to be used in a low power environment due to its large quiescent current. The purpose of the Nexys A7 board isn't to be power efficient, but rather be used as a platform for development. In a production environment, the board design would not have unused parts and may also feature more power efficient components or FPGAs. This is not the scope of this thesis, but is an important consideration for future work.

A byproduct of current draw is heat output. By using a thermal camera, one can learn where the majority of the power is being drawn and as such is discussed in the next section.

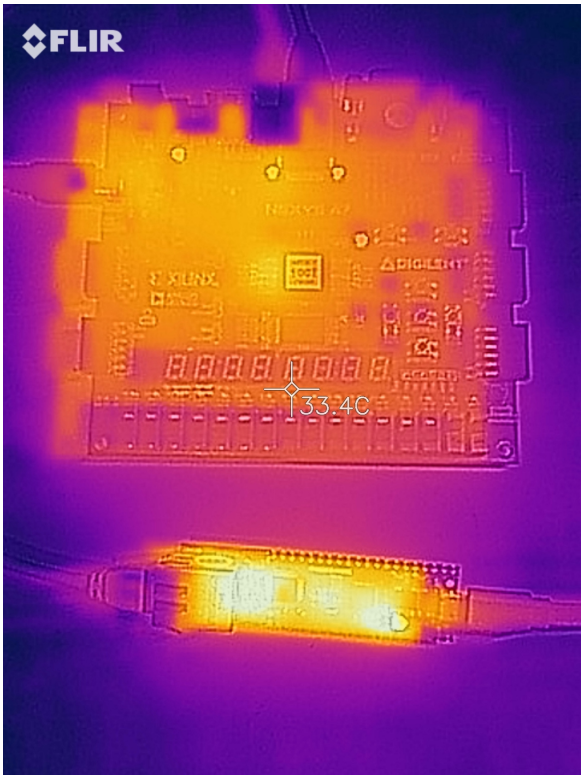
4.5.6 Thermal analysis

A Flir One thermal camera was used to record the temperatures periodically with a summary of results presented in table 4.4. Throughout the test, an ambient room temperature of 24.8°C was maintained. Measurements were made after 5, 10, 30, 60 and 120 minutes to track how the PCB board impacted the thermal properties of the chips. This is important due to the larger PCBs having a larger thermal mass, skewing the results. A table of results can be found in appendix A.6. No major differences were observed in the thermal images after 60 minutes, hence results halted at the 120-minute mark.

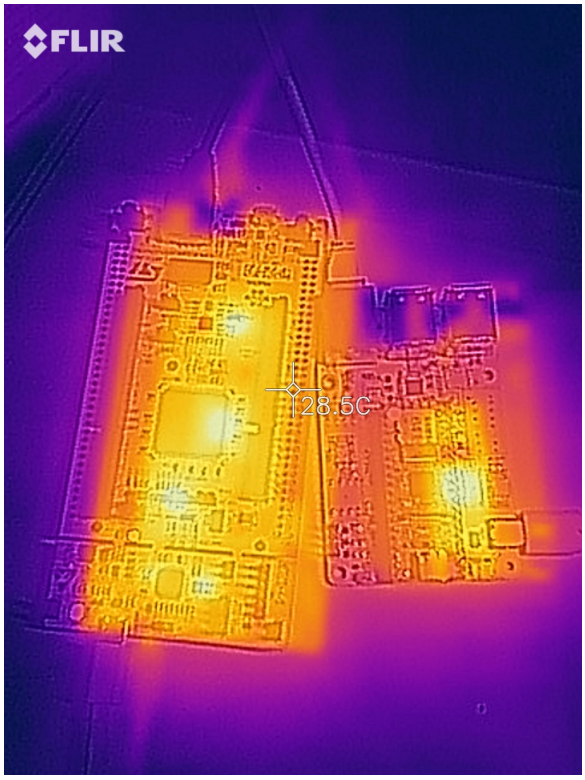
The thermal distribution on each of these boards after two hours can be seen in figure 4.8. It is important to note the measurements cannot be considered as accurate, but rather should be interpreted as an indication to the hotspots and the relative temperatures of the chips to everything else on the board. Variations due to the self calibration procedure and aiming of the thermal camera to record the hottest part are expected.

Table 4.4: Temperature comparison of different chips during the test

Time	Chip	Temperature ($°C$)
5 min	WIZ5500	58.0
	FPGA	38.0
	MilkV	36.7
	F767ZI	35.9
2 hours	WIZ5500	56.8
	FPGA	40.4
	MilkV	38.1
	F767ZI	36.8



(a) Nexys A7 (top) and WIZ5500 Pico (bottom)



(b) Nucleo board (left) and MilkV Duo (right)

Figure 4.8: Thermal images of boards under test after two hours

From the figure 4.8a, the Nexys A7 seems to have other hotspots on the board other than the FPGA itself. This is what is likely causing the higher quiescent current in the previous section. Relatively compared to the other designs, the Nexys A7 shows minimal heat output, indicating a more efficient design. It is important to remember the physical sizes of the chips and the thermal mass behind them.

The WIZ5500 seems to reach the hottest out of the boards tested, while the MilkV board seems to output the least amount of heat radiated. The F767ZI board records a lower temperature, but is significantly larger in chip size and PCB size. In addition, the F767ZI has an additional PHY chip which also adds to the thermal output. The MilkV in comparison only has one small chip with the MCU, MAC and PHY in the same package. These results indicate that the WIZ5500 is inefficient and may not be suitable for use in temperature sensitive environments such as recording temperature measurements in a room, wheras the MilkV or FPGA on a different board may be more suitable.

Table 4.5: Power consumption of components.

Name	Total (W)
neorv32	0.158
clk control	0.097
ethernet_mac	0.097
packet classifier	0.002

While these thermal tests don't provide a great deal of useful information, it does help support the idea that a lot of the current drawn by the Nexys A7 is not from the FPGA itself and that the device is relatively efficient. The following section delves deeper into the power consumption aspects of the design.

4.6 Power analysis

4.6.1 Theoretical analysis

Vivado provides a post synthesis power analysis summary for the design. While these are depenant on a lot of variables, they should provide a basis as to what to expect and help with optimisation of power within the design. The design was observed to take a total of 0.487W (figure 4.9) where 0.383W of that is dynamic and depends on the operations taking place.

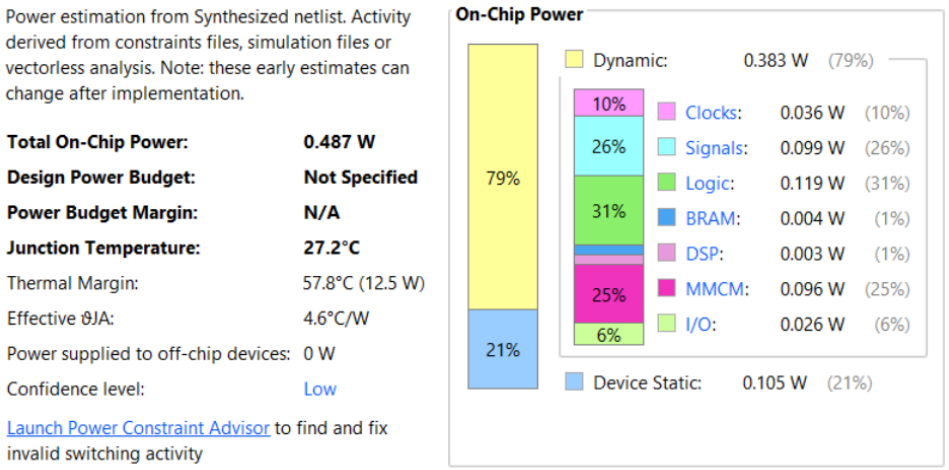


Figure 4.9: Post synthesis power summary for design.

Vivado further breaks down the design into the different hierarchical components shown in table A.5. Notably most of the power consumption in the design is a result of the RISC-V processor in the design. Notably the Ethernet hardware and packet filter consumes under 100mW.

4.6.2 Measured 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 (1W at 5V) 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. Multiplying this by 5V gives us a power draw of 1.51W. To factor in the quiescent current for the other devices on the Nexys A7 and if it is assumed that the 200mA reading solely for the other components on the Nexys A7, it can be found that the design draws around 0.51W. This is rather close to what the synthesis tool calculated.

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 4.10. 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 4.10 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 4.10 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.

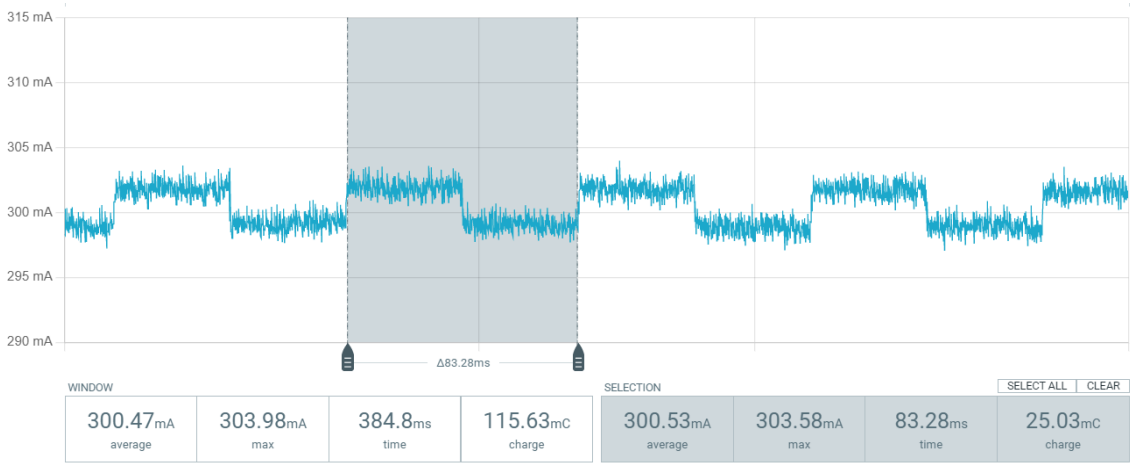


Figure 4.10: Zoomed in current consumption for ICMP pings.

The next test was done when accessing the webserver. Figure 4.11 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.

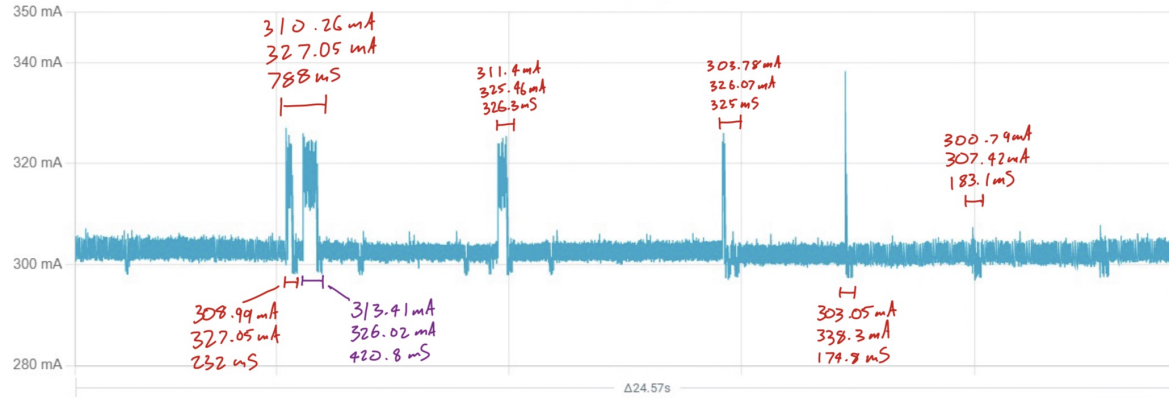


Figure 4.11: Current consumption of Nexys A7 with HTTP requests.

4.7 Modifications

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

4.7.1 Limitations

4.7.1.1 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 4.12) and restricted the use to just one interface - the onboard PHY. A new development board with two PHYs would be needed.

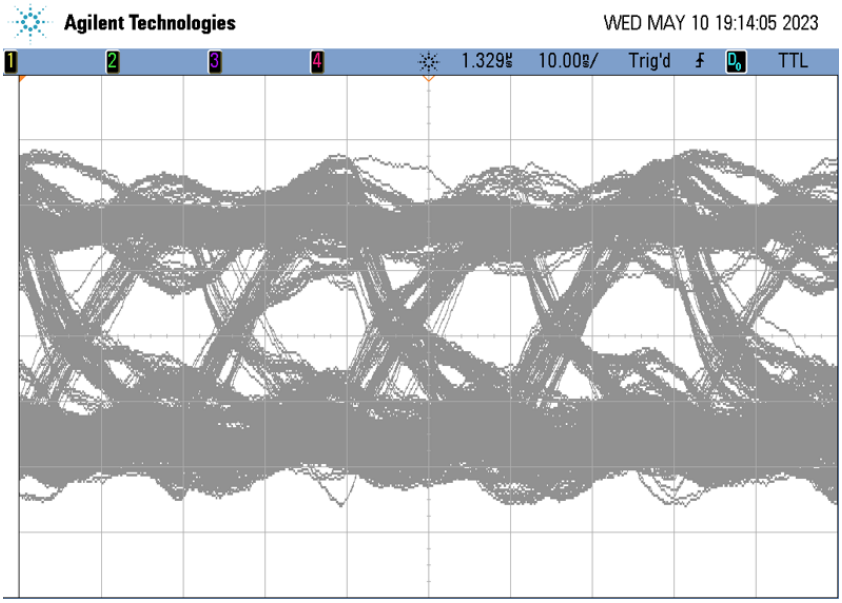


Figure 4.12: Eye diagram of TXD through PMOD interface.

Chapter 5

Conclusion

Conclude your thesis.

5.1 Limitations and improvements

- Bidirectional filtering - currently only doing incomming filtering
- Support IPv6
- Redesign of the transmit logic to consume less resources while not losing on speed/performance - can reduce number of buffers.
- Add a second interface - would need a new pcb and would need onboard switch if webserver was also wanted.
- Utilise the DDR2 RAM onboard the Nexys a7 board for RAM to free up BRAM.
- Use another FPGA board with only the essential peripherals (eg dont need USB, Audio, temperature sensor etc on Nexys board)
- Support certificates for HTTPS.
- Support faster media interfaces, eg RGMII for 1Gbit/s or XGMII for 10Gbit/s as an example
- Make efficiency considerations in the ethernet hardware and RISC-V core.

5.2 Recommendations

Add dedicated hardware packet filtering to SoC based computers - safer - faster, efficient and lightweight

Reduce number of transmit buffers needed - consolidate them.

Use an FPGA with more BRAM as this would help with filter sizes, buffers and also ram needed for the RTOS and webservers.

Consider Hybrid SoC FPGA such as the Xilinx Zynq lineup.

Hardware ethernet and packet filter reduces latency, but is bottlenecked by CPU.

Use a different bus which the processor can handle - wishbone is overkill 2.5Gbit for 100Mbit connection.

If power efficiency is desired, consider something like the MilkV duo. 128mA at 94mbit/s

The NeoRV32 is a solid softcore processor, but it's size grows exponentially when you use the wishbone interface. It's feature set is growing and while good, is not a great idea in production devices.

While Freertos plus TCP provides a good library, it's documentation and community support is lackluster in comparison to LwIP. But is relatively easy to use.

Using Vue.js to write the website in is a good idea as it reduces the network traffic between the device as routing is handled on the client side. So for lightweight applications, it is ideal. No need to have a special webserver for it as the site is static. However a separate API is needed, but easy to implement.

Configure the DHCP server so that it statically assigns the IP address to the firewall for easy configuration.

5.3 Sustainability

Ethernet standards have been in place since xxxx and the IP, TCP, UDP and other layer 4 protocols have been in service for decades and have not changed. Typically if new features do need to be added in these areas, they are added ontop of and not modify the packet structure itself. Such event was in 1998 with the introduction of IPv6 [39].

While malicious bad actors come up with new and innovative ways to breach a system, the core of this packet filter is solid and fundamentally should not need to change in future. Instead other additions to the architecture of a system should be added to better protect a host.

Using the neorv32 while is feature-rich, as it's in its developement process and new features are continually getting added and existing features moved around, it doesn't make for the best choice for ease of updating, but it does get regularly updated so if there was a security vulnerability, odds are that it will get patched.

Perhaps a larger issue with sustainability is the lifecyle of a Vue.js webapp. When updates are needed, potentially new versions of the toolchain may be needed thus contributing to longer development times.

Choice of riscv makes this project easy to commercialise since there is no licences needed for the risc-v cores are royalty free and the specific variant, neorv32 is open sourced under the BSD-3-Clause license. [1]

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

This design was deployed on a Xilinx Artix 7 FPGA board with the LAN8720A PHY. As the LAN8720A chip uses the standardised RMII interface, there should be no issues porting the project to another FPGA or FPGA board so long as there is sufficient LUTs and BRAM available on the FPGA and uses an RMII phy. The hardware could be updated to RGMII or XGMII without too many issues with the main difference being in the input and output FIFO being able to support the different clock rates and bit widths.

Bibliography

[1] S. J. Shapiro, "The strange story of the teens behind the mirai botnet," Sep 2023.

[2] 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.

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

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

[5] 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.

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

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

[8] 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.

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

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

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

[24] W. Odom, *CCENT/CCNA ICND 1 100-105 Official Cert Guide*. Cisco Press, May 2016.

[25] "Internet Protocol." RFC 791, Sept. 1981.

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

[27] 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.

[28] 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).

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] H. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1." RFC 2616, June 1999.

[37] M. Bishop, "HTTP/3." RFC 9114, June 2022.

[38] "7 Series FPGAs Data Sheet: Overview (DS180)." `https://docs.xilinx.com/v/u/en-US/ds180_7Series_Overview`, Sept. 2020. Standard.

[39] B. Hinden and D. S. E. Deering, "Internet Protocol, Version 6 (IPv6) Specification." RFC 2460, Dec. 1998.

Appendix A

Appendix

Write your appendix here. Following two are examples.

A.1 Neorv32 memory address space layout

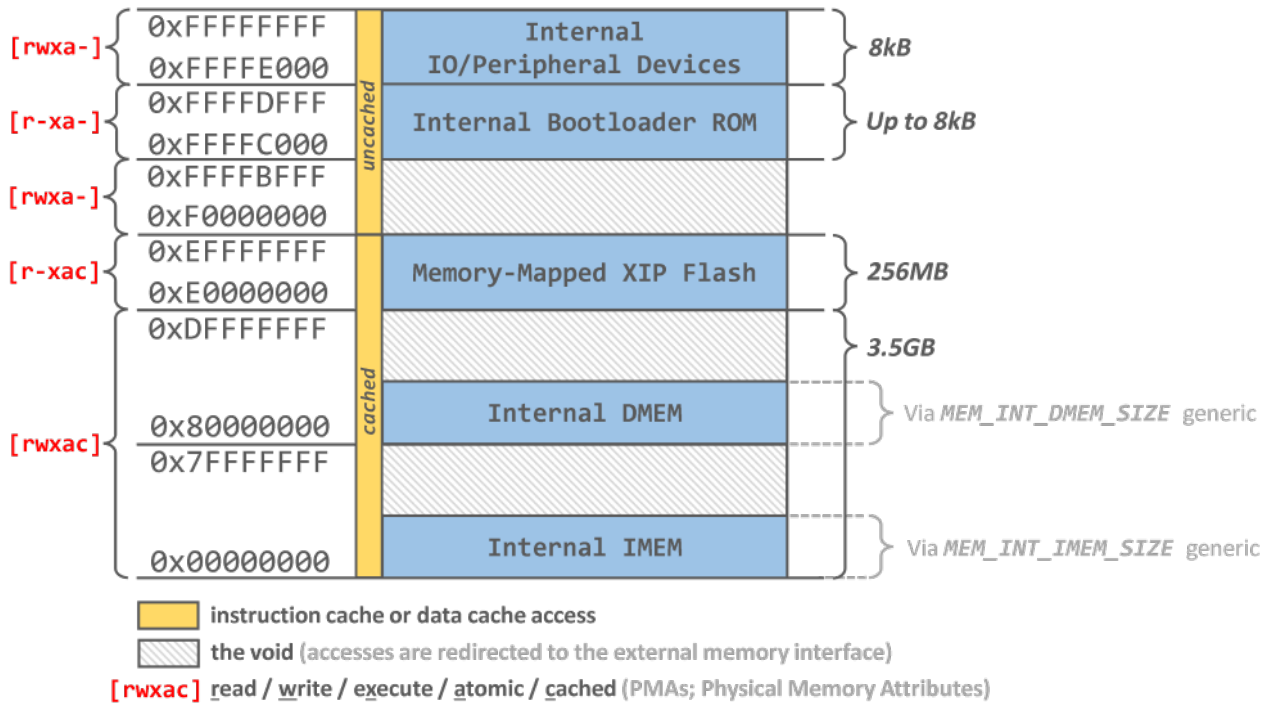


Figure A.1: Neorv32 Memory Address space.

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

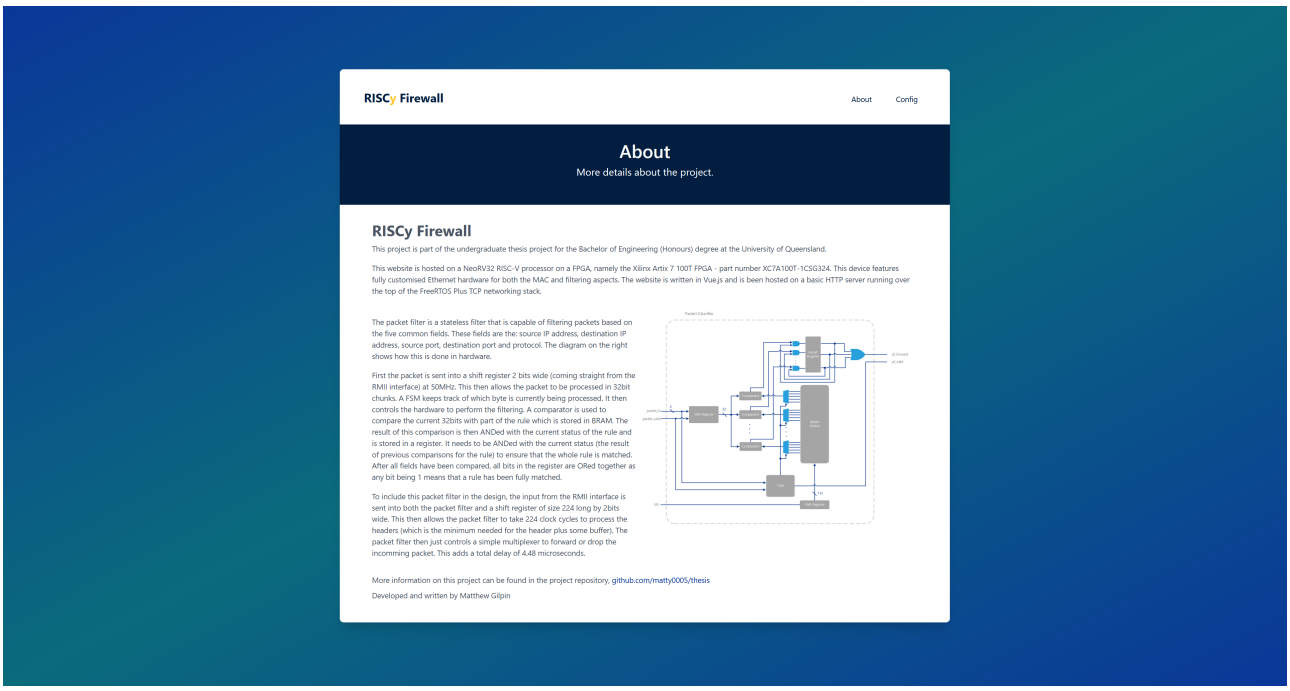


Figure A.2: Screenshot of the about page in the webapp.

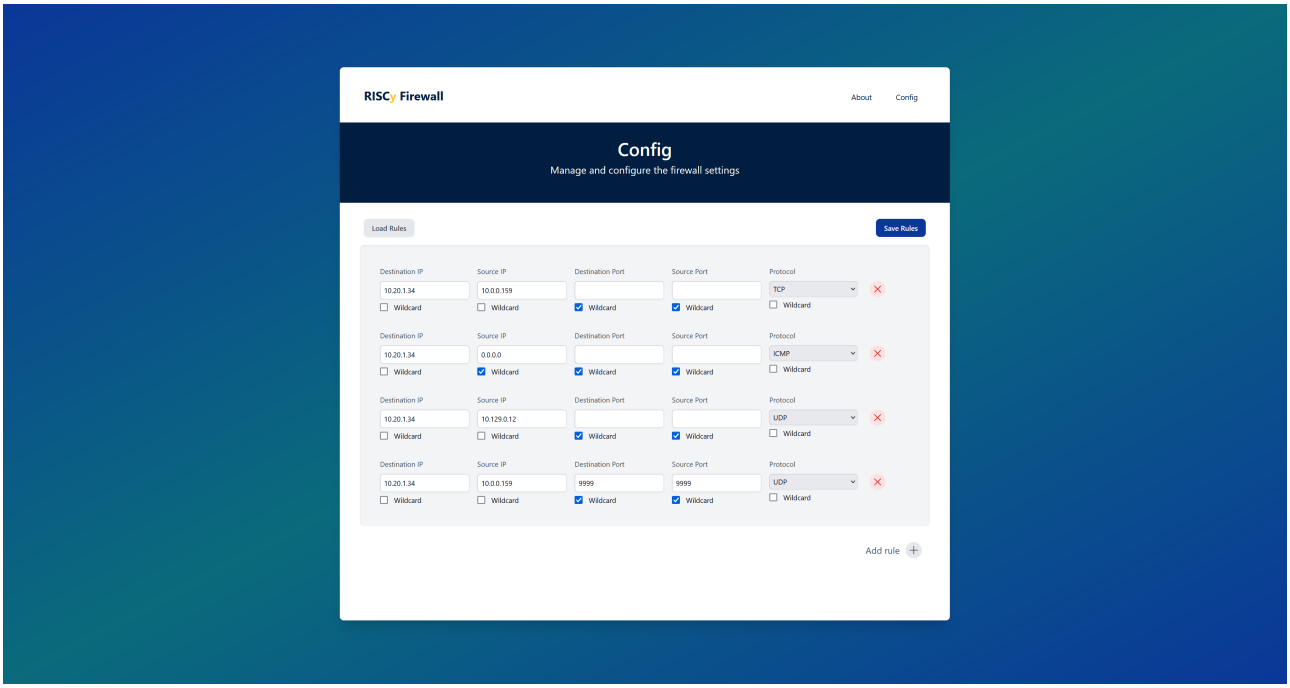


Figure A.3: Screenshot of the config page in the webapp.

Table A.4: Average UDP RTT for different devices and payload sizes.

Device	8 bytes (ms)	256 bytes (ms)
WIZ5500 Pico	1.88	2.07
F767ZI	1.30	1.39
F767ZI 80Mhz	1.41	1.51
MilkV	1.04	1.09
FPGA board	1.45	1.72

A.2 FPGA primitives utilisation

A.3 Additional webpages built into the webserver.

A.4 UDP ping times between boards

A.5 Current measurments from boards

Table A.5 shows the current measurments. Notably Diff1 is the difference between the Idle and No ETH fields while Diff2 is the difference between the Idle and Clean states.

Table A.5: Device power consumption data (all values in mA)

Device	Idle	Busy	Average	No ETH	Clean	Diff1	Diff2
FPGA	301.4	300.13	300.765	264	202.13	37.4	99.27
MilkV	76.27	76.53	76.4	69.23	69.23	7.04	7.04
F767ZI	209.25	206.9	208.075	153.46	121.31	55.79	87.94
WIZ5500	155.88	158.67	157.275	97.9	80.35	57.98	75.53

A.6 Thermal measurements for boards

Table A.6: Device measurements over time using FLIR One thermal camera, all measurements in degrees Celsius.

Device	5min	10min	30min	1h	2h
FPGA	38	38.2	38.9	39.1	40.4
MilkV	36.7	39.8	35.5	39.1	38.1
F767ZI (STM)	35.9	38.9	35.8	37.5	36.8
F767ZI (PHY)	38	38.8	35.2	37.2	36.2
WIZ5500 (RP2040)	46.6	53.1	54.6	54.5	53.2
WIZ5500 (PHY)	58	59	58.4	58.5	56.8

A.7 Testing the firewall with 4 nodes.

Node 1

```
pi@node1:~/thesis-tools $ python3 test.py
Testing from IP: 10.20.1.10
Ping to 10.20.1.34 took 0.522 ms
TCP (10.20.1.34, 1337) received response in 1.02ms: e5 07 00 00
UDP (10.20.1.34, 1337) received response from ('10.20.1.34', 1337) in
   0.83ms: egasseM llaweriF yCSIR
UDP (10.20.1.34, 9999) received response from ('10.20.1.34', 9999) in
   1.41ms: ok
Failed to retrieve content from http://10.20.1.34/
```

Node 2

```
pi@node2:~/thesis-tools $ python3 test.py
Testing from IP: 10.20.1.11
Ping to 10.20.1.34 took 1.12 ms
TCP: (10.20.1.34, 1337) Error occurred: timed out
UDP: (10.20.1.34, 1337) Error occurred: timed out
UDP (10.20.1.34, 9999) received response from ('10.20.1.34', 9999) in
   1.08ms: ok
Failed to retrieve content from http://10.20.1.34/
```

Node 3

```
matt@Node3:~/thesis-tools$ python3 test.py
Testing from IP: 10.0.0.5
Ping to 10.20.1.34 took 0.979 ms
TCP: (10.20.1.34, 1337) Error occurred: timed out
UDP (10.20.1.34, 1337) received response from ('10.20.1.34', 1337) in
   1.45ms: egasseM llaweriF yCSIR
UDP (10.20.1.34, 9999) received response from ('10.20.1.34', 9999) in
   1.70ms: ok
HTTP received from http://10.20.1.34/:80 in 11.95ms
```

Node 4

```
matt@Node4:~/thesis-tools $ python3 test.py
```

```
23 Testing from IP: 10.0.0.93
24 Ping to 10.20.1.34 took 1.23 ms
25 TCP (10.20.1.34, 1337) received response in 1.11ms: e5 07 00 00
26 UDP: (10.20.1.34, 1337) Error occurred: timed out
27 UDP (10.20.1.34, 9999) received response from ('10.20.1.34', 9999) in
      1.10ms: ok
28 Failed to retrieve content from http://10.20.1.34/
```