FRAME: Frame Routing And Manipulation Engine

Submitted in fulfilment
of the requirements of the degree of

Master of Science

of Rhodes University

Sean Niel Pennefather

Grahamstown, South Africa
February 19, 2016

Abstract

This research reports on the design and implementation of FRAME: an embedded hardware network processing platform designed to perform network frame manipulation and monitoring. This is possible at line speeds compliant with the IEEE 802.3 Ethernet standard. The system provides frame manipulation functionality to aid in the development and implementation of network testing environments. Platform cost and ease of use are both considered during design resulting in fabrication of hardware and the development of Link, a Domain Specific Language used to create custom applications that are compatible with the platform.

Functionality of the resulting platform is shown through conformance testing of designed modules and application examples. Throughput testing showed that the peak throughput achievable by the platform is limited to 86.4 Mbit/s, comparable to commodity 100 Mbit hardware and the total cost of the prototype platform ranged between $220 and $254.

Acknowledgements

I would like to acknowledge the support received during this research. I would first like to give thanks to my supervisors Professor Barry Irwin and Anthony Sullivan as their guidance and support has been essential to the success of this research.

I am deeply indebted to my family for their continued love and support throughout the course of this degree, their care and aid has allowed me to focus on the completion of this thesis.

The financial assistance of the National Research Foundation towards this research is acknowledged. Opinions expressed in this thesis and the conclusions arrived at, are those of the author, and are not necessarily to be attributed to the National Research Foundation.

Finally I would like to acknowledge the financial and technical support of Telkom, Tellabs, Stortech, Genband, Easttel, Bright Ideas 39 and THRIP through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University.

Contents

List of Figures

Listings

List of Tables

Abbreviations

The following list records the various abbreviations presented in the remainder of this document. Further details relating to some abbreviations presented here are given on first use of the abbreviation in the text.

ALGOL	ALGOrithmic Language
API	Application Programming Interface
ARM	Acorn Risc Machine
ARP	Address Resolution Protocol
ASCII	American Standard Code for Information Interchange
BNF	Backus-Naur Form
CAT 5	Category 5 twisted pair cable
CGI	Common Gateway Interface
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DC	Direct Current
DDR	Double Data Rate
DRAM	Dynamic Random Access Memory
DSL	Domain-specific Language
EBCDIC	Extended Binary Coded Decimal Interchange Code
EBNF	Extended Backus-Naur Form
EDSL	Extensible Domain-specific Language
ESR	Equivalent Series Resistance
FCS	Frame Check Sequence
FIFO	First In First Out
FIFO	Frame Routing And Manipulation Engine
FPGA	Field-Programmable Gate Array
FTDI	Future Technology Devices International
GPIO	General Purpose Input Output
GPL	General Programming Language
HAL	Hot Air Level
HDMI	High-Definition Multimedia Interface
HTTP	Hypertext Transfer Protocol
IC	Integrated Circuit
ICMP	Internet Control Message Protocol
IP	Internet Protocol

JTAG	Joint Test Action Group
LAN	Local Area Network
LED	Light-emitting Diode
LLC	Logical Link Control
MAC	Media Access Control
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
NAT	Network Address Translation
NIR	Near Infrared
OSI	Open Systems Interconnection
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PCP	Frame transmission Priority
PHY	Physical Layer circuitry
PLL	Phase-locked Loop
RAM	Random Access Memory
RMII	Reduced Media-Independent Interface
RTT	Round-Trip Time
SATA	Serial AT Attachment
SD	Secure Digital
SDN	Software Defined Networking
SDRAM	Synchronous Dynamic Random Access Memory
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
SSH	Secure Shell
TCP	Transmission Control Protocol
TTL	Time To Live
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol
USB	Universal Serial Bus
VLAN	Virtual Local Area Connection
XML	EXtensible Markup Language

1 Introduction

1.1 Introduction

Research into Telecommunications Security and Cyber Defence are growing fields, particularly with regard to incident identification and remediation. Emerging software and hardware systems relating to these fields must undergo validation tests before deployment can occur. Validation is often performed by a test bed such as MX (Knežević *et al.*, 2010) and can be composed of both hardware and software components which must balance a trade-off between environment realism and cost (Sherwood *et al.*, 2010). When considering realism an important attribute is the ability to emulate non-ideal network environments which can have significant impact on network related performance. Unavailable to most researchers is an affordable hardware solution to create such network configurations, with conditions such as packet corruption and loss, in addition to being able to log and manipulate traffic during tests.

1.2 Problem Statement

As different approaches to identifying security threats and the techniques to resolve them are designed, suitable simulation environments must also be developed before testing of the conceptual systems can be performed. Testing environments should be flexible enough to allow for modifications and extensions by the researchers themselves as the requirements of the environment change. These environments would be comprised of a series of subcomponents each handling a particular domain of the system. One such area is the manipulation of network frames at wire level during transmission between hosts within the environment. Devices operating in this domain must limit delays relating to network frame processing to minimise the impact on network transmission speeds during operation. Flexibility for such a device can be realised though a soft-programmable hardware platform which allows for a rapid turnaround in the test phase of the systems development life cycle, along with rapid prototyping. While some of this could be done in software using traditional server platforms, researchers often need to introduce additional latency and have to deal with issues specific to network card drivers.

Commercial systems such as Packetstorm[1], BreakingPoint FireStorm[2] and Netropy[3] do exist which are capable of performing packet delay and loss; however, such systems are large, propriety and have limited extensibility in this domain. These limitations are not due to poor design but are simply not the primary functions the systems were developed for. A device capable of modifying network frames which is both flexible and can minimise the impact of processing latency on traffic throughput is thus necessary. By implementing this flexibility, researchers can quickly modify the device so that it can adapt to the changing requirements of the test bed in response to modifications applied to the system being tested. Researchers modifying the board should not be concerned with programming specific components of the platform, particularly with the hardware interfacing as such details should be suitably abstracted.

Similar approaches to developing a programmable frame processing platform have successfully been carried out with prototypes such as the Scalable Programmable Packet Processing Platform (SP4) (Gill *et al.*, 2012) being implemented. The SP4 uses a custom language based on Declarative Networking methodology (Loo *et al.*, 2009) with high level

[1]PacketStorm network emulators are designed to produce unfavourable network conditions in a controlled environment: http://packetstorm.com/

[2]The BreakingPoint FireStorm is a rack mounted device designed to perform large-scale network and security testing: http://www.ixiacom.com/products/firestorm

[3]Netropy 10G2 is a network emulation engine designed to simulate multiple network streams in a single device: http://www.apposite-tech.com/products/netropy-10G2.html

programming abstractions to provide the functionality to reprogram the system. Our research differs from this approach by developing a language that focuses on general frame manipulation as well as routing and delay within the platform rather than implementing rule sets.

1.3 Research Goals

The focus of this research was to design and develop a flexible system comprising both a hardware and software layer capable of processing Ethernet frames.

As its primary goals, the system should be capable of performing generic modifications to network frames on Ethernet, routing frames between output modules of the underlying device, and delaying received frames for a set period of time. The system should also be capable of logging specific attributes associated with received frames to be stored for later retrieval and use by the researcher.

Secondary goals included the production of a low cost device to act as dedicated hardware on which the proposed system could operate. Application development for the system was intended to be simple and intuitive and to not require an innate understanding of the underlying hardware while still remaining expressive enough to allow for flexibility.

Considering these goals, the approach taken by this research to investigate them were as follows:

1. Potential architectures which could act as the underlying hardware of the benchtop device were investigated. After investigation, the most appropriate candidates were selected to make up the device architecture. This selection formed part of the initial phase of the research as the resulting selection affected design decisions during the development and implementation stages of the system.

2. The design of the system began with determining a suitable structure for constructing network applications which could take advantage of features advertised by the selected architecture. Core components required for basic system operation were designed which would form the basis of any future network applications designed to use the system.

3. To account for ease of use, approaches into simplifying application development were investigated and implemented. Such approaches include developing API calls which

could be used in a more generic language such as C or C++ to develop applications. Alternatively, a custom language specific to the platform could be designed which would aid in the development of compatible applications.

4. Board design required further research into electrical characteristics such as signal propagation time and best practices on component placement. It was expected that a simplified version of the device should initially be developed to confirm proper design before development of the final product was attempted.

5. Both the board and the system underwent a series of conformance tests, which concluded functionality and determined feasibility of the system as a component responsible for frame manipulation and delay at wire speeds of up to 100 MBit/s.

1.4 Research Scope

This research remained in the scope of a functional prototype that can provide all of the intended functionality but allow room for future development and refinement. The focus of the research was divided between the design and development of a physical processing platform, and the design and implementation of the software layer. The physical platform was limited to supporting four network interfaces due to cost limitations and presenting a functional proof of concept rather than a production ready product.

The intention of the software layer was to develop and showcase the device performance by means of application examples showing potential uses of the device. For this research the focus of the software layer was on its functionality rather then completeness. As a result some features have been made available as a proof of concept and may need to be extended before their full functionality can be realised. Traffic manipulation was focused at the frame level, however, selecting frame data based on fields relevant to higher level protocols was implemented.

During an initial evaluation it was concluded that adding IPv6 support to the system would introduce unnecessary complexity for an initial prototype. As a result, the initial implementation of the system would only support the IPv4 protocol with IPv6 being reserved for future development. Transmission speeds were limited to 100 MBit/s primarily to minimise cost and to reduce complexity relating to Printed Circuit Board (PCB) design. Scaling to gigabit speeds could be addressed in a future revision of the device.

1.5 Document Conventions

For the remainder of this document, all URLs relating to websites, organisations, or software discussed will be available as a footnote associated with the term. This will avoid disrupting the flow of the document while still providing the reader with a reference for further reading.

Acronyms used throughout the remainder of this document are provided in a glossary at the beginning of the document. Furthermore, the first presentation of each acronym is preceded with the full definition of the term.

Owing to the length of some URLs related to other works referred to within this text, URL shortening was required to improve readability of the associated references. For all URL redirects, the URL shortening service provided by Google[4] was used. Appendix A associates each shortened URL with the original reference for the reader should the shortened URLs prove to be unsuitable.

Finally, this document uses appendices for presenting all code listings in excess of 30 lines and screen captures relating to the user interface produced as part of this research. These figures have been moved to the end of the document as they are considered too large to be placed in line with the associated text without distracting the flow of the document.

1.6 Document Structure

This dissertation consists of seven chapters followed by a list of appendices which contains supporting material. The remainder of the dissertation is structured as follows:

Chapter 2 focuses on providing the reader with the necessary background information relating to technologies used in this research. This includes an introductory discussion of the XMOS architecture relating to structure characteristics and the interprocessor communication methods currently supported. General compiler theory is also briefly discussed to give an overview of the language processor. Similar work to that undertaken in this research is also reviewed.

Chapter 3 covers the design of a prototype system which will be used to achieve the research goals. This design includes the selection of underlying hardware which the system is expected to operate on. The structure of a module is defined in Section 3.5.1

[4]Accessible at: https://goo.gl/

which discuses module communication requirements. Application examples are presented and evaluated to aid in the design of a language which can represent application functionality compatible with the proposed system before system components relating to user interaction and application compilation complete the chapter.

Chapter 4 presents the implementation of the designed system and the development of a Domain Specific Language (DSL). An investigation into transmission performance achievable by the selected network architecture is undertaken. The system implementation is divided into first constructing the subcomponents necessary to interface with external hardware before further defining the structure of a user-defined module. A DSL is designed and a supporting translator is implemented to aid in application development. The user interface designed in Chapter 3 is also implemented in Chapter 4.

Chapter 5 focuses on the design and fabrication of a hardware platform using the XMOS architecture. An investigation into power design is performed including sequencing characteristics required by the XMOS microcontrollers. An initial prototype is first designed as an intermediate step to help confirm design correctness and functionality of the proposed layout. This initial prototype is then scaled to produce the hardware component of the Network Layer for this research.

Chapter 6 discusses the testing of the implemented system. Pre-existing hardware is used to create an initial prototype of the underlying architecture and tests concerning the individual hardware components are performed to confirm functionality. Following this, internal modules designed to form part of user-defined applications are tested using simple application examples. More elaborate application examples are then designed and discussed to show potential uses of the research.

Chapter 7 concludes the research where the feasibility of designing a benchtop device to aid in traffic manipulation, within the constraints presented in Section 1.3, is reflected on. Following this, a discussion into future work that could be undertaken to extend the results of this research is presented.

2
Literature Review

2.1 Introduction

Network processing can be used in a wide range of research domains, from network simulation to network security, to assist in the generation of valid data for evaluation. As discussed in Chapter 1, the goal of this research was to create a network packet processing platform to aid in simulating a non-ideal network environment. This chapter addresses similar work in the field and presents literature to introduce some of the concepts discussed in this document.

As network processing requires knowledge in data transmission and some of the common network protocols, this chapter covers some general networking concepts. Section 2.2 gives a brief overview of the network OSI model with focus on the Link Layer.

Additional information relating to the XMOS XS1 is presented in Section 2.3. As this architecture forms core component of the network processor used in this research, this information is required to provide the reader with further background information relating to characteristics and communications model of the XS1 architecture. This information is later referred to in Chapters 3 and 4 along with more context specific information.

Table 2.1: OSI Seven Layer Reference Model (IEC, 1996)

7	Application Layer
6	Presentation Layer
5	Session Layer
4	Transport Layer
3	Network Layer
2	Link Layer
1	Physical Layer

To assist in giving the reader a better understanding into the DSL and its functionality, Section 2.4 covers literature relating to compiler theory at a broad level, followed by a more detailed discussion of DSLs generally. Tools used to aid in the development of the web interface for the base platform are briefly covered in Section 2.5.

Developing a network processor is not in itself a novel concept, and numerous implementations exist. Section 2.6 details some similar work in the field of packet processing, highlighting similarities with the goals of this research. From these works, common approaches to application development for packet processing can be extracted and analysed before development of a prototype device relating to this research can begin.

2.2 Networking

Prior to development, basic networking concepts need to be discussed to better understand how modern networks function. After a brief overview, network frames will be discussed in terms of the IEEE 802.3 family standards (IEEE, 2002). Literature relating to some common standards and protocols used in network transmission are also mentioned and discussed.

2.2.1 Networking Stack

To aid in the design of network protocols, communication can be structured into layers of services, with each service abstracting away the underlying services on which it operates. This conceptual model allows network protocols to be built on top of pre-existing protocols, delegating responsibility for lower-level operations. The Open Systems Interconnection (OSI) model (IEC, 1996) shown in Table 2.1 is a common conceptual model that represents the network as a hierarchical structure consisting of seven layers.

The Application Layer acts as the top layer of the model and represents the highest layer of abstraction available in the OSI model depicted in Table 2.1. This layer is used by protocols responsible for communicating between application endpoints and the underlying protocols used to achieve this are abstracted from services using an Application Layer protocol to communicate (Kurose and Ross, 2013).

The Presentation Layer represents the protocols responsible for handling data conversion and compression before transmission can occur. Protocols residing in this layer allow data to be translated between different formats such as EBCDIC and ASCII as well as data reduction and encryption. Data reduction is performed by compression protocols such as Joint Photographic Experts Group (JPEG) (Dubendorf, 2003) for images and Motion Picture Experts Group (MPEG) for video (Dubendorf, 2003).

Communication between network devices takes place within 'sessions', which consist of sequences of service requests and responses. Protocols residing in the Session Layer such as the Session Control Protocol (SCP) are used to manage and coordinate these sessions (Dubendorf, 2003).

The Transport Layer is responsible for the transmission of an entire message from the source host to the destination host. The Transport Layer is unaware of the context of any message it transmits: this is the responsibility of the layers that use this service. The two most common Transport Layer protocols used are TCP and UDP, both of which operate to transmit entire messages across the network (Forouzan and Fegan, 2007).

Protocols operating at the Network Layer are responsible for transmitting datagrams between hosts over a network. Datagrams conform to the IP protocol in terms of how the datagram fields are ordered to ensure compatibility with other routing devices also supporting the same protocol (Kurose and Ross, 2013). The layer responsible for transmitting data between the individual nodes that form part of the network path between hosts is the Link Layer. At this layer, the data transmitted is stored in frames which specify the destination address of the next physical node on the path (Dubendorf, 2003). This is not to be confused with the destination address at the Network Layer which specifies the host at the end of the route. The Link Layer can be divided into two components, the Media Access Control (MAC) Layer and the Logical Link Control (LLC) Layer.

The MAC Layer handles data access management for both access and transmission. This sublayer defines the MAC address which uniquely identifies each device within the network (Dubendorf, 2003). The LLC Layer is responsible for handling error checking and flow control of the Ethernet frame. Flow control is used to monitor and moderate data transmission between devices to ensure no device gets overwhelmed and risks dropping frames (Dubendorf, 2003). Error checking is necessary as frames can become corrupted during transmission due to interference on the physical medium. A common approach to handling error detection at the Link Layer is to include a redundant field, the Frame Check Sequence (FCS) (see Figures 2.1 and 2.2). The FCS is generated by performing a cyclic redundancy check (CRC) on the entire frame (excluding the FCS field). The result of the CRC is placed in the FCS field before transmission. The receiving interface then extracts the FCS and performs its own CRC on the entire frame. The CRC result is compared to the value in the FCS and if the values match the frame is considered uncorrupted (Forouzan and Fegan, 2007).

Finally, the Physical Layer is responsible for data transmission as individual bits across a specific medium. The protocol used to perform the data transmission is highly dependent on the physical medium being used as it must specify the mechanical and electrical characteristics of the medium (Forouzan and Fegan, 2007). For example, 100BASE-TX specifies how data bits are transmitted as a potential difference across two twisted pairs of a Category 5 (CAT5) cable, one for transmit and one for receive. Meanwhile, 100BASE-FX standardises the use of two fibre optic strands to transmit bits as near-infrared (NIR) light pulses in full duplex mode (IEEE, 2002).

2.2.2 Network Frames

As the focus of this research is to provide traffic manipulation at the Link Layer, a more in-depth discussion of network frames and Link Layer protocols is necessary. The Physical Layer is usually handled by dedicated hardware such as a physical network interface (PHY) which accepts network frames and handles their transmission and reception. Network frames must be submitted to the PHY in a specific format in order for them to be transmitted correctly. A network frame is described in RFC 1122 as a ‘unit of transmission in a Link Layer protocol, and consists of a Link Layer header followed by a packet’(Braden, 1989). The contents of the associated packet depend on the protocol under which the frame is being transmitted.

Figure 2.1 shows the basic structure of an Ethernet frame. Each frame begins with a preamble which is a 56-bit wide word of alternating 1 and 0 bits followed by a 1 byte start frame delimiter (SFD). The preamble is primarily used for synchronisation of the frame receiver with the SFD indicating the beginning of a frame. The second field holds the destination MAC address of the frame under transmission which all listening receivers can

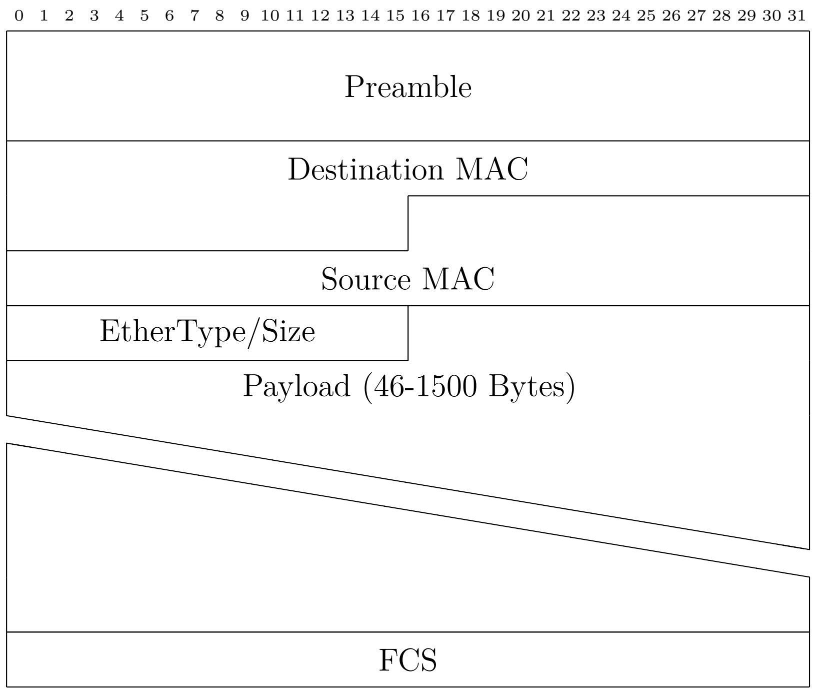


Figure 2.1: Layout of an IEEE 802.3 Compliant Ethernet Frame

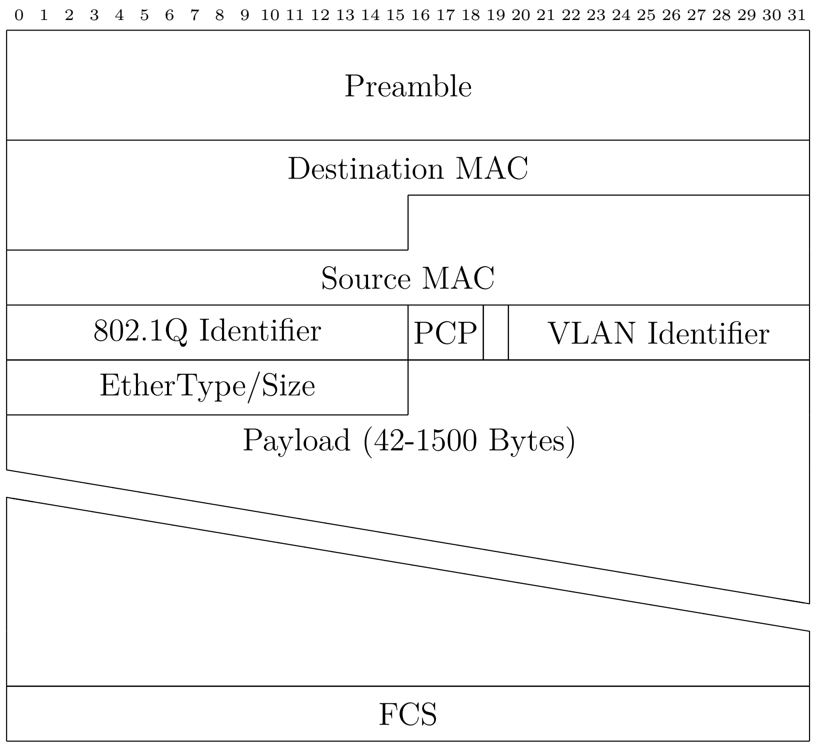


Figure 2.2: Layout of an IEEE 802.3 Compliant Ethernet Frame with IEEE 802.1Q (VLAN) Header (Bonaventure, 2011)

read to determine if the frame is addressed to them. The third field contains the MAC address of the PHY transmitting the frame onto the physical medium. The fourth field of the Ethernet frame contains either the frame size or the EtherType which indicates the protocol the frame is being transmitted under. To eliminate ambiguity as to whether this field is an EtherType or a size indicator, the IEEE 802.3 standard enforces that EtherType numbering must be greater or equal to 0x600 (1536). The reasoning is due to the maximum Ethernet frame size being limited to 1500 bytes for non-jumbo frames[1].

The VLAN protocol allows IEEE 802.1q switching devices to manage multiple Virtual LANs on the same physical LAN by maintaining a MAC table for each VLAN identifier recorded. Figure 2.2 shows an Ethernet Frame including a VLAN field or ‘tag’ which precedes the EtherType field. When a frame is received with an unknown or multicast destination address and includes a VLAN tag, that frame is only broadcast to ports associated with the VLAN identifier. If the VLAN tag is unknown or blocked, the packet is discarded (Bonaventure, 2011). To differentiate frames with a VLAN tag from those without, the IEEE 802.1q header must begin with the protocol tag identifier 0x8100, indicating the presence of a VLAN tag within the frame (Gwan-Su and Hong-Hee, 2005). The PCP field indicates the transmission priority of the frame. Following the PCP is a bit reserved to maintain compatibility between Ethernet and Token Ring networks (Bonaventure, 2011). The remaining 12 bits make up the VLAN identifier, allowing for up to 4094 VLANs.

2.3 XMOS XS1 microcontroller architecture

XMOS is a semiconductor company focused on developing multi-core, multi-threaded processors. This is apparent in the XS1 microcontroller architecture designed for event driven applications (May, 2009). The XS1 architecture is conceptually similar to the Transputer with improved IO capacity and the XC programming language which is an extended version of C (Dibley, 2014). The Transputer architecture was based on the concept of creating a range of processors that can operate as individual entities or be linked together to form a more complex parallel computer. Each Transputer contained its own RAM and IO controllers with dedicated channels for communication with other transputers (May and Shepherd, 1985).

The XS1 architecture is designed to provide parallel processing as well as determinism

[1] Jumbo frames can carry a 9000 byte payload http://www.ethernetalliance.org/library/whitepaper/ethernet-jumbo-frames/ .

and bounded latency for embedded systems. The architecture provides this functionality by including a number of XCore tiles in each microcontroller. Each tile contains 64KByte block of SRAM and up to eight ‘logical cores’. Each logical core contains 16 dedicated registers and is capable of executing a single thread (Martins *et al.*, 2012). The memory block is shared between each thread executing on a tile, provided all the threads obey the disjointness rules. These rules enforce read and write restrictions to prevent memory violations in a parallel context. For a thread T_x that contains a reference or modification to some variable in shared memory V the restriction is:

$$\{\ V \epsilon\ T_x\ \}\ \rightarrow\ \{\ V\ !\epsilon\ T_t\ \}\quad where\ \ t\ \neq\ x \tag{2.1}$$

This implies that, if some thread contains a reference or modification to a variable, no other threads operating in parallel may access that variable. However, concurrent threads may access a shared variable provided all access to the variable is read-only (May, 2009)

A scheduler is implemented to manage time slicing between threads during operation. The scheduler guarantees a minimum number of processor cycles associated with each thread based on the number of threads active (May, 2009). More accurately, the thread performance is dependent on two factors; the number of threads active on the tile [n], and the maximum pipeline depth [p]. The relationship between the thread performance $f(x)$ and these factors can be shown by:

$$f(x) = \begin{cases} \frac{1}{n}, & \text{if } n \leq p \\ \frac{1}{p}, & \text{otherwise} \end{cases} \tag{2.2}$$

This equation indicates a decline in thread performance as the number of threads increases with the maximum possible performance limited by the pipeline depth. Reducing the thread count beyond the pipeline depth results in no performance improvement. For an XCore logical tile operating at 500 MIPS[2], this approximates to a worst case operation speed of $\frac{500}{8}$ or 125 MIPS. Due to the independent registers available to a thread and the implementation of the scheduler, it is possible to view the threads executing on a single tile as logical processors with shared memory. The number of logical processors available for use on a given tile is defined by the number of logical cores implemented in its hardware design. The xCore tiles present on a XS1-L1 each contain eight logical processors. (XMOS, 2013d)

[2]Million Instructions Per Second.

The architecture allows threads to communicate within an XCore tile or between tiles within the microcontroller. An interconnect is implemented consisting of on-tile switches and xConnect[3] links between tiles. An interconnect is then implemented on this configuration which can allow data transfer of up to 250 Mbit/s and supports multiplexing. The interconnect acts as a medium for message passing between threads. The interconnect can also be used to allow a thread executing on one tile to access memory on a different tile (XMOS, 2012b).

2.3.1 Communication

A central goal of the XS1 architecture is the deterministic execution of concurrent tasks. Each XCore tile implements a communications infrastructure that enables concurrent tasks to exchange data and control messages This communication infrastructure is represented programmatically as a channel variable which can be declared and passed to two concurrent tasks prior to execution. During execution, either task may write or read data to/from the channel but only when the other task is expected to do the alternative otherwise a deadlock[4] situation could occur as the tasks will block on the channel until the data put into the channel buffer has been removed by the other task.

```
void f(chanend c){
  char val;
  c :> val; //Read a character from the channel buffer
}

void g(chanend c){
  c <: 'a'; //Put the character 'a' into the channel buffer
}

int main(void){
  chan c;  //Channel for threads to communicate over.
  par
  {       //Execute the following two statements in parallel.
    on stdcore[0]: f(c);
    on stdcore[1]: g(c);
  }
}
```

Listing 2.1: Sample Code to Pass a Single Character Over a Channel

[3]The xConnect is a scalable architecture that provides inter-tile communication (XMOS, 2013c).

[4]A situation where both processors are waiting on the other before being able to continue execution.

In Listing 2.1, a channel is used to pass the character 'a' from one thread to another before terminating. The logical core that is selected for a task to be executed on can be specified using the 'stdcore' command, otherwise a free logical core is selected within the tile. The '$f()$' and '$g()$' functions each specify a 'chanend' which data may be transmitted over or received from. In the main functions, a channel is created and handed to both functions. During execution, tokens are passed from function '$f()$' to the interconnect to establish a connection with the function '$g()$'. This token includes an address indicating the destination interconnect and destination logical core for communication. In this case there is only one interconnect so the tokens are directed to the logical core handling function '$g()$'. Once a connection is established, the data (in this case the character 'a') is transmitted before the connection is released.

At the program level, it is convenient to think of a channel as a pipe with each end attached to a core; at the hardware level this is not the case. On a tile, resources are shared between each core. These resources include channel ends which a tile has a finite number of. When a channel is declared in the program, the associated number of channel ends are then allocated to the relevant processors (XMOS, 2013c).

To further enhance communication between concurrent tasks, each tile supports routing traffic from the switch through a link, as shown in Figure 2.3[5]. Each link is associated with a specific set of pins on each device. Currently, there are two types of links that the switch can use to communicate; the serial XMOS link and the fast XMOS link. The serial XMOS link depicted in Figure 2.4 is also referred to as a two wire link and consists of four wires for full duplex operation. Each wire is associated with a state for a single bit so that during transmission the wires do not maintain the same signal unless transmission is complete or an error has occurred. The lines associated with the serial XMOS link are capable of achieving 200 million pin transitions a second with a byte being transmitted every 10 transitions. This results in a theoretical speed of 160 Mbit/s (May, 2009).

Alternatively, the fast XMOS link consists of 10 wires and has a theoretical duplex speed of 400 Mbit/s. Figure 2.5 depicts the Fast Link connection with five lines dedicated to data transmission in each direction. Using a five line interface, a nibble is sent by associating a value with each of the lines as shown in the state diagram (May, 2009). In practice however, achieving speeds in excess of 100 Mbit/s with the serial XMOS link interface or 250 Mbit/s with the fast XMOS link interface is difficult as this value does not take all control tokens into account. Furthermore, the theoretical speed assumes a perfect connection between the switches (XMOS, 2013c).

[5]https://www.xmos.com/files/images/xlinks.jpg

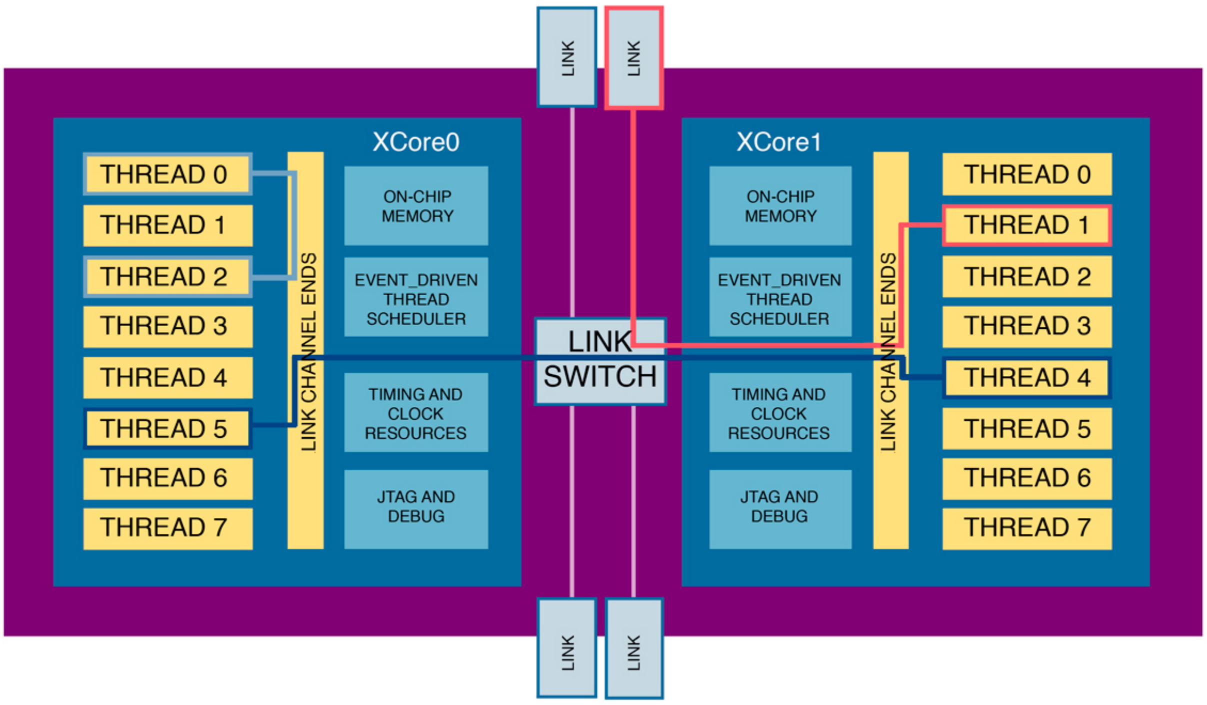
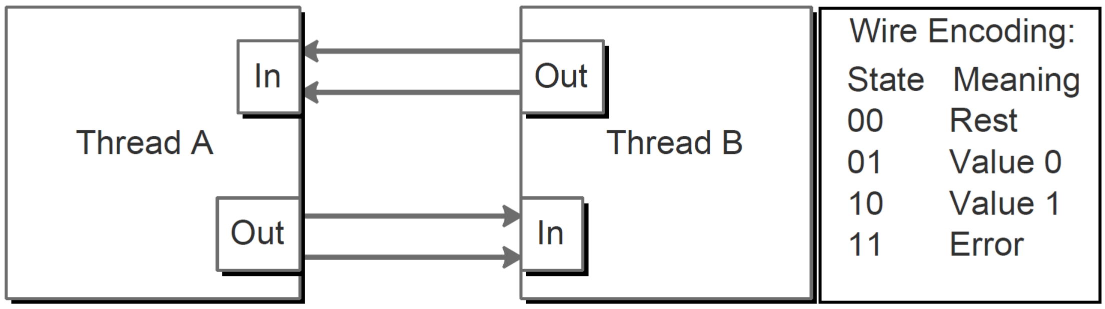
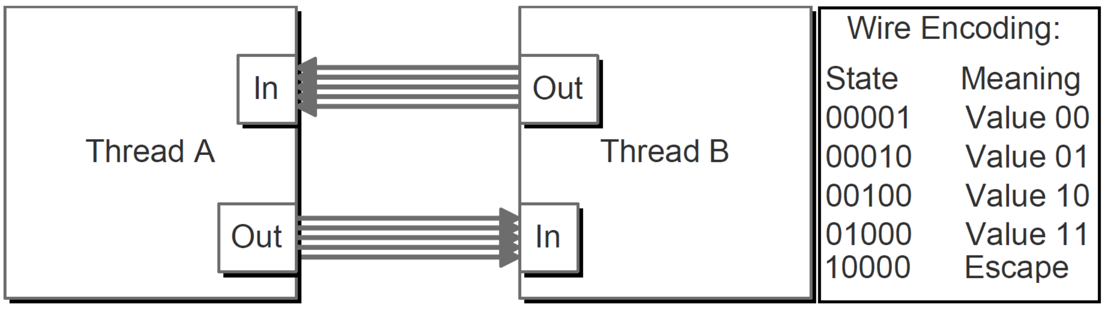


Figure 2.3: xLink Layout for an XMOS Controller Containing Two XCore Tiles



State	Meaning
00	Rest
01	Value 0
10	Value 1
11	Error

Figure 2.4: XMOS Serial Link Communication (May, 2009)



State	Meaning
00001	Value 00
00010	Value 01
00100	Value 10
01000	Value 11
10000	Escape

Figure 2.5: XMOS Fast Link Communication (May, 2009)

To manage communications between connections, each XMOS device contains a switch which handles the routing of channels relating to that tile. The switch supports up to four intratile channels and up to eight intertile channels, all duplex. However, the actual number of links present on an XMOS microcontroller depends on the package under consideration. In multi-tile packages, at least one fast XMOS link per tile is reserved for communication within the package while on smaller packages such as the XS1-L4 (XMOS, 2013i), only two links are available due to the reduced number of pins.

Associated with these switches is a series of tokens that are used to control the setup and tear-down of a route between two logical cores. Data may be transmitted across a channel either as packets or as a stream. When transmitted as a packet, the established route is torn down after each packet is transmitted and set up again before the next packet can be sent. This process is designed to include very little overhead and so does not cause any notable drop in throughput over the link. By enforcing that the communication is torn down after each packet is routed, the resources used by a route are made available for other communications. As a result, multiple logical cores may share the resources needed to perform communications.

Should a deterministic rate of throughput be required, an application can specify stream transmission, in which case the hardware resources for a connection between two tasks are reserved and maintained. Streams are routes between processors that are not torn down after each data packet is sent. In this situation, the connection is maintained and so the overhead relating to tear-down and setup is removed. Furthermore, as the resources are not shared, throughput of the streaming task is not dependent on whether other tasks need to communicate. The consequence is that no other processor can use the resources held by the route, reducing the number of connections the switch is capable of maintaining. If there are specific situations where the resources associated with a stream can safely be shared, a PAUSE token can be used to allow multiplexing on part of the route used by the stream. When this token is sent, the connection between the processor sending the token and the local switch is torn down while the other end of the connection is maintained. This allows another task on the same tile as the processor which sent the PAUSE token to use the route resources. Once done, the connection is set up again so that streaming of data can continue (XMOS, 2013c).

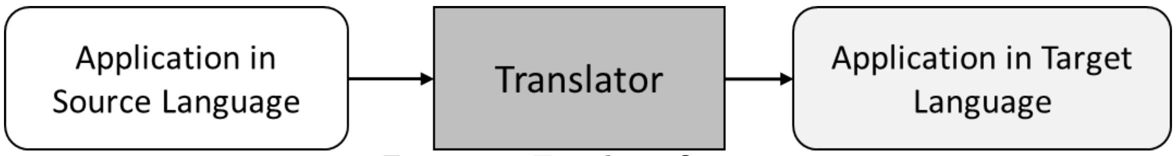

Application in Source Language
Translator
Application in Target Language

Figure 2.6: Translator Overview

Application in Source Language
Compiler
Application as an Executable Object

Figure 2.7: Compiler Overview

2.3.2 Programming

XMOS supports application development in high level languages such as C and C++. XMOS have also developed an extended version of C called XC which allows the programmer to take advantage of the concurrency and channel communication features of the architecture. For logic analysis as well as timing evaluation, XMOS provides the xTIMEcomposer toolchain. This toolchain can be used to analyse applications written for the XS1 architecture and verify that execution timing constraints are met for a given application / configuration. The toolchain also includes xSCOPE which is a software analyser that can monitor applications and memory usage during execution on the device (XMOS, n.d.b).

2.4 Compiler Theory

Programming languages are an essential tool for describing an application in a manner that is understandable to people and can be translated into a format more easily understood by machines. A program that performs this translation is termed a ‘translator’, and can be seen as a mapping function as shown in Figure 2.6 (Aho *et al.*, 2006). An equivalent view of a translator is a function whose domain is the source or input language and the range is the target or output language (Terry, 2005).

A translator is a general term for a language processor, and these can be divided into different classes each focusing on a different type of translation. These classes are characterized by the input language they accept and the type of output they produce. Two classes of particular interest to this research are the compiler and the high level translator.

For both a compiler and high level translator, the goal of the software is to accept an

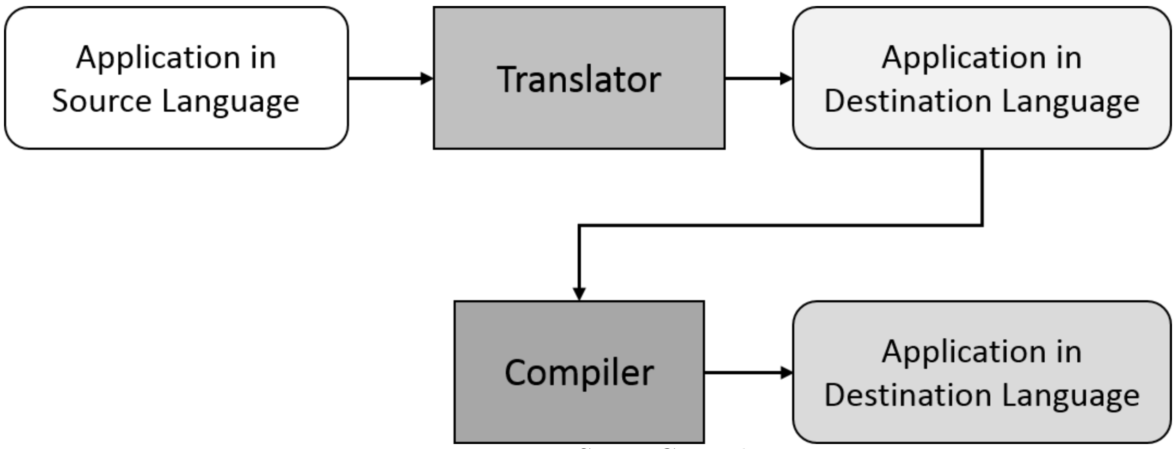

Application in Source Language
Translator
Application in Destination Language
Compiler
Application in Destination Language

Figure 2.8: Two-Stage Compilation

application written in the source language and translate it into a functionally equivalent application but written in the target language (Aho *et al.*, 2006). A translator can be classified as one or the other depending on how closely the target language relates to machine language that can be processed by the target platform.

In general terms, a compiler is a language processor designed to translate a source application from a human readable format into a language that can be executed on the target platform as depicted in Figure 2.7. A translator operates in a similar fashion except this language processor translates the source application code into either a language that is still considered high level and often still human readable or into some intermediate language. These translations are often used for two stage compilation such as in Figure 2.8 or bootstrapping (Terry, 2005).

Though depicted as a single entity in Figure 2.7, a compiler consists of a sequence of processes or phases, each of which perform a specific task in the translation process (Louden, 1997). These phases can be treated as distinct entities though often they are combined to form larger links. Louden defines this sequence to consist of six stages described in Figure 2.9. Associated with this sequence are three external components; the literal table, the symbol table, and the error handler (Louden, 1997).

2.4.1 Literal Table

The literal table is used to store constants and strings relating to the source code. The purpose of this object is to reduce the memory consumption of the final application on the target platform. This is achieved by ensuring only one instance of each constant or string is included and that these are included at the global scope. These constants and

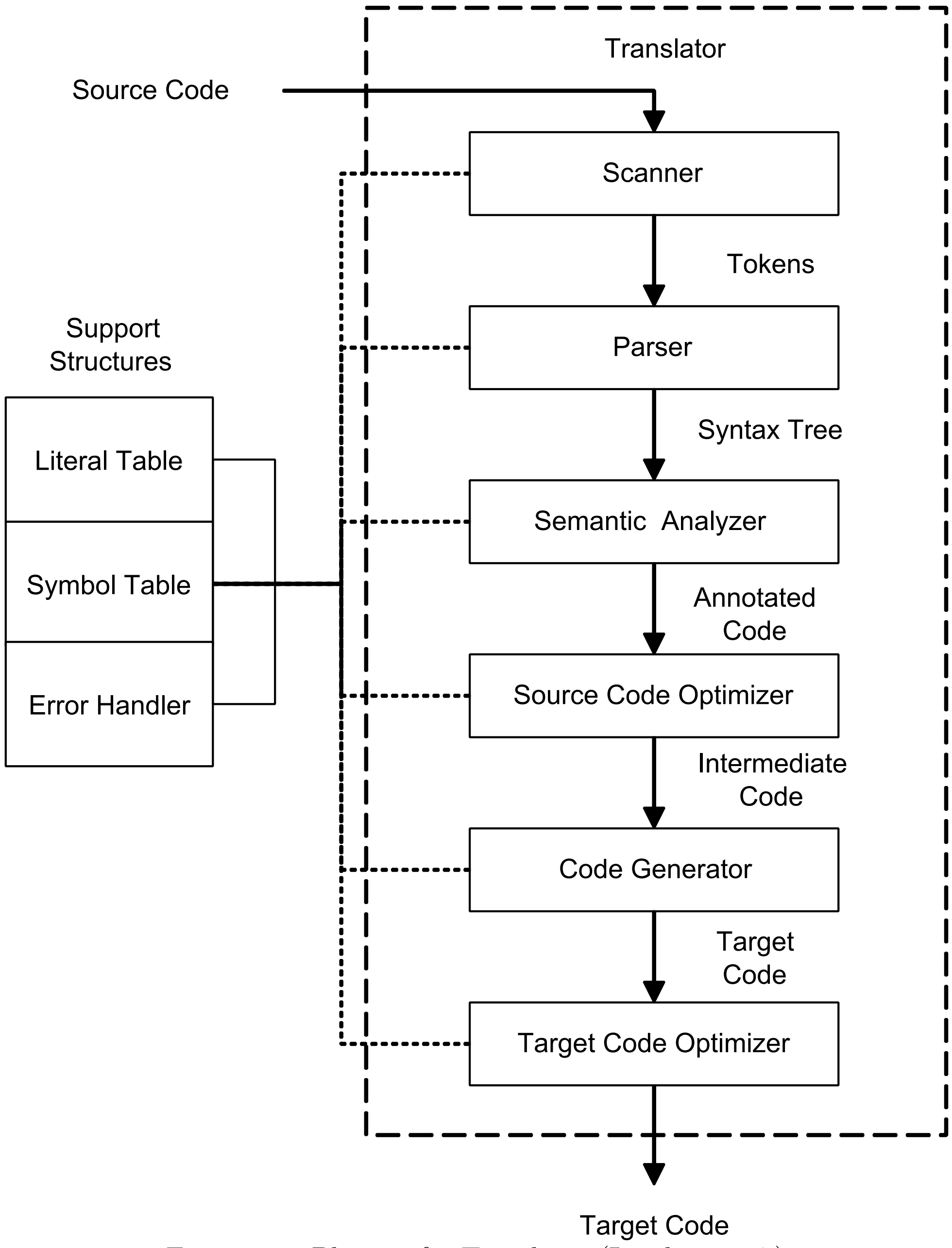


Figure 2.9: Phases of a Translator (Louden, 1997)

strings will then be accessible from all scopes within the application code (Louden, 1997). This is possible as strings and constants are immutable and so are not modified within the application code.

2.4.2 Symbol Table

The symbol table or name list (Grune *et al.*, 2012) is a record of all identifiers present in the source application code. Each record associates an identifier with all its collected properties such as its type, its memory requirements, value, or (in the case of function identifiers) input parameters (Terry, 2005).

2.4.3 Error Handler

Finally, to aid in the development of applications in the source language, a compiler should include an error handler to help accurately identify errors. The error handler works in two parts: error detection and error recovery. The errors a compiler can detect are syntax errors which occur when incorrect syntax is found during parsing (Grune *et al.*, 2012) or the symbol table indicates a type mismatch.

```c
//Simple main function written in C
void main(void) {
  int x,y = 7        //The ';' is forgotten here
  y = x + 2;
//^ Here is where the parser will detect an error.
}
```

Listing 2.2: Error Detection Example

Once an error has been found, error detection must then determine what error message to present to the user as often the location at which the error was detected may not be the location of the error itself (Grune *et al.*, 2012). For example, in Listing 2.2, the actual error is a missing ';', however the compiler only detects an error once it starts processing the next line. Error detection methods must then decide if the error is due to the missing ';' or if the user was indeed continuing the expression and is potentially missing an arithmetic symbol.

While error detection must try identify the error to present the appropriate message to the user, error recovery is responsible for enabling the syntax parsing of the source language to continue without falsely identifying errors as a consequence of detecting the initial error (Terry, 2005). This can be done through either error correction or non-correcting error recovery. Error correction substitutes the offending symbol with the most likely alternative and continues normally (Grune *et al.*, 2012) while non-correcting error recovery stops parsing the program using currently stored information and continues afresh for the remainder of the source code (Richter, 1985).

2.4.4 Domain Specific Languages

Programming languages can be grouped according to the level of generality of the applications they are designed to create. For languages such as C, C++, Java and C#, the range of possible implementations is very large. As a result, these languages are classified as General Purpose Languages (GPL) (Bentley, 1986). A GPL boasts a large set of features allowing for high functionality and application development in a large range of environments. Alternatively, a DSL is more limited in the features available which in turn limits the range of possible implementations of the language (Mernik *et al.*, 2005). In exchange, a DSL can be optimized to suit a particular domain and becomes very expressive for applications within that domain (van Deursen *et al.*, 2000).

As discussed by Cleveland and Uzgalis (1977) and van Deursen and Klint (1998), a detailed understanding of the problem domain is necessary before attempting to develop the DSL. To achieve this, it is recommended that the investigation and development are guided by the following steps:

- Identify the target domain.
- Investigate the domain and gather relevant knowledge.
- Use this knowledge to develop a set of semantic notations ans operations for the domain.
- Design a DSL that describes applications targeting that domain.
- Design an interpreter or compiler that transforms applications written in the DSL into the semantic notations and operations discovered.

2.5 Web Development Tools

As a component of this research includes the development of a web interface, it is necessary to discuss the tools used to render the interactive web pages. The three tools we discuss further are: Common Gateway Interface (CGI), JavaScript, and Asynchronous JavaScript and XML (AJAX).

CGI is a lightweight interface that allows for the use of external script to generate dynamic webpages (Cozens and Wainwright, 2000). The client makes a HTTP request to the server for a CGI script. The server responds by executing the CGI script as a separate process and passes parameters included in the request to the process. The output of the process is then collected and served back to the client as a HTTP query response (Gundavaram, 1996).

The primary concern with using CGI is that a new process is started to execute a CGI script every time the associated request is received from a client. For servers expecting regular CGI requests this approach is considerably more resource expensive than having a continuous application running (Rhodes and Goerzen, 2010). However, for this research, the webserver is only expected to serve a limited number of clients at any given time so using CGI scripts should not cause a significant impact on the webserver.

JavaScript was developed by Netscape Communications Corporation and was one of the first Web scripting language to be implemented (Moncur, 2000). Even so, it is still one of the most popular scripting languages in use and is a common way to include interactivity on served web pages. JavaScript programs execute within the browser and so allows for computation to be done by the client rather than the server (Thau, 2000). This improves responsiveness of the web page by eliminating the latency of communication between the client and server. Furthermore, by operating as a client based application, the server only has to transmit the bulk data and may delegate further processing to the client.

AJAX is a group of tools that are combined to create asynchronous applications by allowing the web application to transmit and retrieve data from the server in an asynchronous manner. This is achieved by separating the controller and view components of the web application such that communication between the controller and server can happen without interfering with the view. Whenever an AJAX related JavaScript call is initiated by the web application view, the associated data is collected and included in an HTTP request by the controller. This request is transmitted to the server and any response is received by the controller. Once processing of the request is complete the view is updated with the new HTML and CSS data (Holdener, 2008).

2.6 Related Work

As development of a network processor is not a novel concept in itself, it is necessary to review literature relating to similar works before research can progress. Considering similar research highlighted popular approaches that were considered during platform design.

2.6.1 Click Modular Router

The Click Modular Router[6] is a software architecture designed to build flexible routers. The architecture comprises elements that each represent a single unit or stage of processing. All actions taken by the designed router must be encapsulated in these elements which are then linked to create the routing application (Kohler *et al.*, 2000).

Click comes with a custom language to aid in router development, however this language was designed exclusively for declaring and connecting elements. The element classes themselves are written in C++ and a supporting collection of libraries to simplify the process. The routing application developed using Click is designed to run on the Linux kernel. Packet references are transmitted between elements, while the actual packet is stored separately until it is either dropped or transmitted off the platform. This greatly reduces overhead in communication between elements (Kohler *et al.*, 2000).

The modular approach taken by the Click modular router is both novel and effective in providing scalability to a processing platform. The Click modular router functionality is very similar to the goals that this research aims to achieve, with the most notable difference being the intention to develop a physical device dedicated to traffic processing rather then implementing the functionality as a kernel module. A second key difference is the focus on user friendliness, especially with users who have a limited exposure to programming. The elements in Click are written in either C or C++ which, in these languages, requires the user to have reasonable familiarity. This research attempts to address this by providing a more simplified approach to programming the platform, either by a minimal scripting language or connecting prexisting nodes to form an element.

[6]https://github.com/kohler/click/

2.6.2 NetFPGA

The NetFPGA[7] project revolves around the development and maintenance of open source hardware relating to the NetFPGA network adaptor. Supporting these adaptors is a collection of reusable modules designed to aid in the development of common network related tasks. Initially, the intent was to develop the NetFPGA network adapters as rack mounted switches with remote access for upload and debugging (Watson *et al.*, 2009). The initial NetFPGA network adapters went through two revisions, the first beginning in 2001 and the second in 2004. After testing, limitations of the initial revisions were recorded and addressed in the second revision. Chief amongst these limitations was the lack of a CPU on the NetFPGA network adapter. Developing applications to operate on the adapter allowed for processing to occur at line speeds however the device lacked resources for more data intensive processing. Furthermore, to account for the lack of a CPU, the NetFPGA network adapter had to be linked to a remote host to use pre-existing software which induced significant processing latency (Watson *et al.*, 2009). To account for this and other limitations, the second revision of the NetFPGA network adapter was developed as a PCI card that could be mounted in an existing host. This approach allowed the network adapter to use the PCI bus for power and communication with the host (Watson *et al.*, 2009). Packets can be received by the NetFPGA network adapter via the input module either from the four onboard gigabit network interfaces, from the host via the PCI interface, or from other NetFPGA network adapters connected using one of two onboard SATA interfaces (Lockwood *et al.*, 2007).

The intent of the NetFPGA project is to provide a open platform on which network related architecture can be rapidly developed. A basic IPv4 router is implemented using two software packages, one to handle routing and IP lookups, and one for the user interface. The hardware aspect of the router is broken into a series of pipelined stages where each packet is received from an onboard interface before being handed to the user datapath. The datapath performs the IP lookup and selects the appropriate output queue to move the received packet to, as well as performing all necessary modifications such as TTL checks and decrements. The output ports are then continually emptied by the associated network interface (Naous *et al.*, 2008). The NetFPGA network interface has also been used to create a packet generator that is capable of generating traffic at gigabit speeds by reading a previously recorded PCAP file and regenerating the associate packets while acutely observing the recorded packet delay. The generator also includes accurate timestamping for received traffic which can then be recorded in a PCAP-compatible format for

[7]http://netfpga.org/

later use (Covington *et al.*, 2009).

2.6.3 Netgraph

Netgraph[8] is a system designed to allow for the instantiation of kernel objects to perform networking functions in the FreeBSD kernel. As described in the FreeBSD man pages, netgraph allows for the inclusion of nodes in the kernel to aid in network processing. These nodes can communicate by using channels described as hooks (Elischer and Cobbs, 2005). Network packets are described as flowing through the nodes in a bidirectional manner. Each node processes a packet in some way before passing it to the next node. This approach of chaining elements is similar to the approach taken by the Click Modular Router.

2.6.4 SP4

Similar research at the University of Pennsylvania has developed the Scalable Programmable Packet Processing Platform (SP4) (Gill *et al.*, 2012). This research uses a declarative language to develop applications from prexisting modules which are then deployed onto the processing platform (Gill *et al.*, 2012). The SP4 is based on an optimized version of the networking engine called the RapidNet networking platform. RapidNet takes an approach to network protocol simulation called declarative networking which uses a query language called Network Datalog (NDLog) to to implement routing protocols (Muthukumar *et al.*, 2009). Though the research being undertaken on the SP4 is similar to the research discussed here, the SP4 focuses on providing a platform the rapid development of network protocols and as such differs from the goals of this research.

2.6.5 Switchblade

Similar research to that proposed in this thesis is the Switchblade platform (Anwer *et al.*, 2010). The Switchblade is a hardware platform designed to aid in the rapid development of new or customised network protocols. The platform is designed around an embedded FPGA architecture and has been implemented using the NetFPGA platform. The goals of the device are to provide rapid turnaround on network protocol development by providing a platform that can operate at wire speeds (Anwer *et al.*, 2010).

[8]https://www.freebsd.org/cgi/man.cgi?netgraph(4)

The Switchblade supports flexibility by incorporating a customizable hardware module which can be used to perform most tasks common to protocols, such as acquiring the destination address of a frame or calculating the associated checksum. These modules can then be included as part of a larger project that implements the protocol being developed. For data intensive processing better suited to implementation in the software layer, the Switchblade implements software-based interrupts allowing such processing to happen outside of the FPGA and be injected at a later stage. Switchblade is compatible with existing routing software such as Click (Anwer *et al.*, 2010).

2.6.6 Scout and Silk

Another software-based approach similar to this research is the Scout architecture which operates by segmenting the communication layer into paths or slices. Each slice is associated with a particular application or service allowing traffic to an application to be individually addressed. A design feature of Scout that is of interest to this research is its extensibility. The architecture was developed following a modular design making it easy to extend by including new components which can operate alongside existing components of the architecture (Montz *et al.*, 1994). Improving on this, SILK implements the Scout architecture as a networking module to effectively replace the current networking module in the Linux kernel. This module allows application running on the operating system to be classified into slices or paths which allows for individual addressing (Wehrle, 2005).

2.6.7 Frenetic

Frenetic[9] is a high level language designed to aid in the programming of software-defined networks (SDN) implemented across a distributed collection of switches. Frenetic provides an alternative solution to network controller software such as NOX (Gude *et al.*, 2008) and OpenFlow (McKeown *et al.*, 2008). The language operates using a declarative query syntax to perform network traffic classification and aggregation (Foster *et al.*, 2011). The implemented network query sub-language operates by installing a series of low-level rules on switches which allow applications implemented using Frenetic to observe the target network. Frenetic attempts to improve on existing languages in the domain by providing race-free semantics, modular design, and single-tier programming. Frenetic also assists in application development by informing the user on performance costs relating to specific

[9]http://www.frenetic-lang.org/

programming constructs (Foster *et al.*, 2011).

2.6.8 Nettle

Nettle[10] is a system designed to simplify development and configuration of SDNs. The system operates on OpenFlow-compatible network switches and provides a family of extensible domain specific languages (EDSLs) to aid in the development of compatible applications (Voellmy *et al.*, 2010). Each EDSL is designed to cater for specific operational domains. The goal of the overall system is to express network requirements in a clear and concise manner while also providing the mechanisms to implement network control. These mechanisms allow Nettle to capture and generate control messages associated with OpenFlow compatible switches (Voellmy and Hudak, 2011).

2.6.9 Procera

Procera is a control architecture for SDN that includes a declarative language aimed at supporting expressiveness of high level network policies and constructs to support common policy queries (Voellmy *et al.*, 2012). Procera supports flow constraint functions, which are used to limit the behaviour of low-level network controllers and can be implemented using high-level control functions. Procera is designed as a policy layer that operates on top of an existing network controller such as Frenetic (Foster *et al.*, 2011), NOX (Gude *et al.*, 2008), or Nettle.

2.7 Summary

This chapter presented literature relating to several areas of this research. Section 2.2 provided a brief overview to the OSI model before discussing the Link Layer and the structure of an Ethernet frame. Section 2.3 provided background information on the XMOS architecture and its characteristics, detailing the communication interfaces supported by the architecture. Section 2.4 discussed the theory relating to general compilers and the components commonly needed to construct them. Following this, DSLs were discussed. Section 2.5 briefly highlighted some common web development tools that were used in this research while Section 2.6 presented similar research in this field.

[10]http://haskell.cs.yale.edu/?post_type=publication&p=350

3 Design

3.1 Introduction

To achieve the research goals stated in Chapter 1, a prototype platform was designed and developed. Section 3.2 begins by describing the intended functionality of the platform which is later implemented as a benchtop device. Initial design work was published in Pennefather and Irwin Pennefather and Irwin (2014) which provides a brief overview of topics covered in this chapter.

For design purposes, the proposed platform is divided into two layers: the Network Layer and the Base Layer. The justification for separating the system into two subcomponents is due to the two distinct domains each layer must operate in. The first domain relates to the routing and manipulation of network frames as specified by the user. Subsystems operating within this domain are considered time critical to minimise latency incurred during frame processing. The second domain includes all auxiliary subsystems necessary for the successful operation of the platform. These subsystems include interfacing with the user, collecting and storing data from the application, and compiling and programming the application binaries for execution.

Section 3.3 describes two potential hardware candidates that could be used to act as the hardware base of the Network Layer. A brief overview of each candidate is given relating to architecture type, Ethernet functionality and development tools. The section concludes with the selection of a suitable candidate architecture to act as the hardware base.

Following the Network Layer selection, a similar investigation is performed in Section 3.4 to select an appropriate device to act as the Base Layer of the platform. This research opted to implement the Base Layer as an existing device capable of performing the tasks required rather than developing custom hardware to achieve the same goals.

After a suitable architecture has been selected for both layers of the proposed platform, the remainder of the chapter focuses on developing the software based on the proposed hardware configuration. Section 3.5 is divided into subsections to give a brief overview of the approach taken in this research to develop the application software. An application example is presented and discussed to highlight areas of interest that need to be accounted for during implementation.

The software that operates on the Base Layer is also described towards the end of Section 3.5, including a brief discussion on how the user interface will be presented. This chapter concludes with a summary reflecting on the system design proposed in this chapter.

3.2 Platform

Before design and development of the system could begin, it was first necessary to select appropriate underlying hardware on which to implement each of the platform layers. These architectures were then used to design and fabricate a benchtop device capable of executing the proposed system.

The completed device was expected to provide a hardware platform capable of parallel processing on which the Network Layer of the system could be executed. This platform must include physical interfaces for CAT 5 wire links as well as dedicated SDRAM for temporary frame storage.

The device must also support a hardware platform on which all components not directly related to network processing can run. These components include a user interface such as a web server and a database for data storage and management. For the benchtop device to become standalone, the Base Layer may also be required to include the compilation toolchain required to compile the application source code into an executable format for

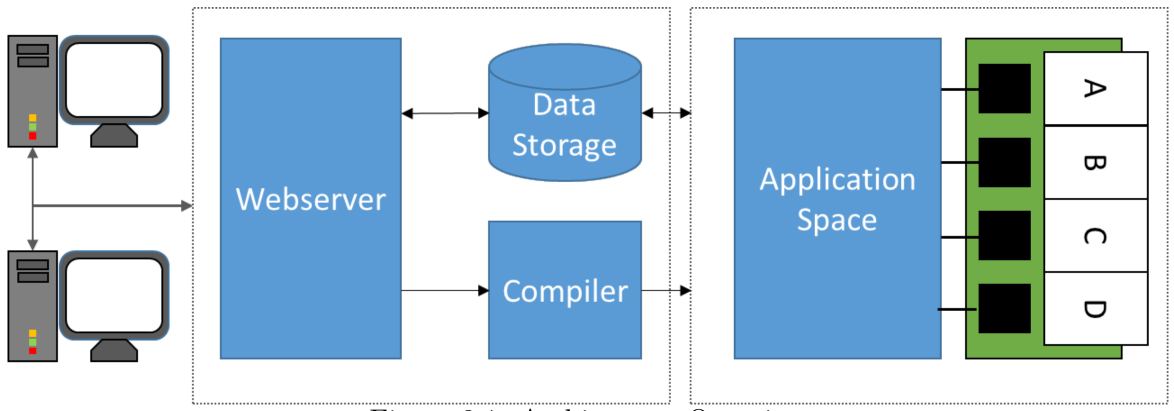


Figure 3.1: Architecture Overview

the Network Layer hardware.

By logically separating the system into two layers, separate architectures could be chosen for each to ensure optimal hardware configuration for the required functionality. Figure 3.1 provides a basic overview of the device which indicates the responsibilities of each layer. From the diagram it is clear that the layers will communicate in two situations: uploading of an application, and logging of data from the running application. Provided that the method the two layers will communicate in is standardised, the architectures used for each layer can be selected independently.

The Network Layer in Figure 3.1 is responsible for executing the client application and handling all network-related IO, with the exception of the user interface. One of the research goals described in Section 1.4 is to have the device operate at 100 Mbit/s and as a result, this layer must be able to execute time sensitive applications while maintaining network transmission speeds in this region. To maintain these speeds, all application logic should be suitably pipelined in an attempt to minimize processing impact on latency.

The Base Layer is responsible for allowing the platform to interface with other devices. This layer includes storage devices for storing log files generated and an I/O interface. This layer could potentially include an additional Ethernet interface for uploading applications to be run as well as accessing stored log files. This layer does not participate in the execution of the applications but rather provides the base of the platform responsible for collecting and presenting network related data to the user.

3.3 Network Layer

Applications operating on the Network Layer are intended to focus on performing frame manipulation and routing while minimizing the performance impact on transmission speed due to latency arising from frame processing. Considering this requirement, it is necessary to investigate potential architectures that could be used to implement this layer.

The architectures that are investigated in this chapter are the XMOS XS1 architecture and the Xilinx Spartan 6 FPGA architecture. Both architectures were selected owing to their low cost, ability to perform IO operations at speeds suitable for network communication, and their ability to perform multiple operations in parallel.

3.3.1 XS1 Architecture

Details relating to the XMOS XS1 architecture are presented in Section 2.3 which provides an overview of the architecture structure, communication, and programming environment. The proposed component used to represent this candidate is the XMOS XS1-L8A available for approximately $15[1].

I/O

Communication between the application executing on a logical processor and the attached hardware is achieved using ports. In the XC application code, these ports represent logical endpoints which are associated with a pin or group of pins, allowing the logic level to be read or written to.

Once specified, each port can be associated with a different timing block. A timing block specifies the frequency which in turn determines the sampling rate and latency of that pin block. A timing block can come from an external clock source or some variant of the system clock.

The XS1 architecture has the ability to delay the system clock used by applications while allowing the timing blocks associated with IO to operate normally. This functionality allows the XS1 to provide a larger window for sampling input signals which in turn enables the microcontroller to guarantee correct signal reception for an application clock

[1]As of August 2015 http://goo.gl/XRmUF1.

cycle. This is achieved by reducing the application clock frequency so that the application instruction and signal sampling are not done on the same clock cycle (XMOS, 2010).

Ethernet

Layer 2 network communication can be implemented by including the XMOS layer 2 Ethernet MAC component. This component comes in two versions: the full version and the light version (XMOS, 2013e). Both versions support the IEEE 802.3u standard (IEEE, 2012) but vary in the number of features supported. In both cases, the operational speed to the PHY is limited to 100 MBits/s.

The full version allows for multiple clients and independent buffering as well as more accurate time stamping and support for the IEEE 802.1 QAV standard (IEEE, 2009). Owing to the larger feature set, the full version also requires five logical processors to operate. Alternatively, the lite version is limited in features supported and only allows for one receive client and one transmit client. The advantage of this version when compared with the feature rich alternative is that it only requires two processors to operate (XMOS, 2013e).

Both versions of the component work by running a server process which can send and receive network frames using a RMII interface to the PHY. Transmission occurs by splitting the server into two subcomponents which are the transmit subcomponent (TX) and the receive subcomponent (RX). Each subcomponent then supports a series of queues which can contain buffers for storing frames received or to be transmitted. Each frame is stored in a buffer until it is either transmitted, dropped, or collected by the client application (XMOS, 2013e).

Development Tools

The XS1 supports application development in high level languages such as C and C++. XMOS have also released an extended version of C called XC which allows the programmer to take advantage of the concurrency and channel communication the microcontroller can support. For logic analysis as well as timing evaluation, XMOS provide the xTIMEcomposer development environment. This development environment can be used to analyse applications written for the XS1 and provide timing constraints that the microcontroller can guarantee will be met during execution. The development environment also includes xSCOPE which is a software analyser that can monitor applications and memory usage during execution on the device (XMOS, n.d.b).

3.3.2 Xilinx Spartan 6 LX25

An alternative architecture to investigate is the field-programmable gate array (FPGA). This architecture allows for integrated circuits to be designed and implemented after the FPGA has been manufactured. This is due to the design of an FPGA not being developed as a specific controller but rather a large collection of logic and RAM blocks. These blocks can then be configured and connected using signal lines to represent the designed circuit. (Wain *et al.*, 2006)

The Spartan 6 LX25 architecture allows for up to 358 User I/O pins, grouped into 4 I/O banks. The Spartan 6 family supports RAM block sizes of 18 Kb and the Spartan 6 LX25 contains 52 of these blocks, giving it a total of 936 Kb of internal RAM. The Spartan 6 family architecture provides a large collection of clock lines to support multiple clocks running at different frequencies for different circuit components. (Xilinx, 2011)

The component selected was the Spartan 6 LX9 available for approximately \$17[2].

Development Tools

In order to develop applications that can be loaded onto the Spartan 6 LX25, Xilinx maintains a design suite that supports development in both Verilog and VHSIC[3] Hardware Description Language (VHDL). These languages allow the user to design circuits using a programmatic syntax which is then simulated by the ISE design suite into a circuit diagram. To implement the designed circuit, the logic needs to be converted into a bit stream that can be uploaded to the device. Before this, the developer needs to specify a user constraints file (UCF) which maps input and output signal lines to the device pins (Xilinx, 2013). Once specified, the ISE then performs a place and route which maps out the circuit to best fit the block configuration of the target device. Finally the bit stream is generated which can be uploaded to the device (Xilinx, 2010).

For testing before bit generation, the ISE supports a simulator for both Verilog and VHDL. The simulator can be used by setting up a test bench which will allow the user to monitor

[2]As of August 2015 http://goo.gl/ExFzzn.

[3]Very High Speed Integrated Circuit

output signal for specific input signals (Xilinx, 2010).

To improve development with FPGAs, previously generated integrated circuits are available as modules called Intellectual Property (IP) Cores. These IP cores can be downloaded and included in the current project manually or, if the IP core is supported by Xilinx and the suitable licensing file is present, by using the IP wizard tool provided in the ISE. The components can then be called in the developer's code to take advantage of the previously generated circuits. Most IP cores from Xilinx require licensing to use but others are available under the GNU General Public License (GPL)[4] from websites such as Open Cores[5].

3.3.3 Network Layer Summary

Considering the requirements, both the Xilinx Spartan 6 and the XMOS XS1 architectures can be used to implement a suitable hardware platform on which to run network specific applications. Both architectures can operate at the necessary speed and both architectures come with previously developed software/circuits relating to Ethernet communication. The development tools in both cases are extensive and maintained which limits the motivations of which candidate should be selected to the actual languages used to develop on each and the cost of the representative components. Both Verilog and VHDL are hardware description languages which means that they need to be simulated into an actual circuit before being converted into a bit stream by some propriety software. Initial investigations into developing network related applications for both architectures was done. It quickly became apparent that developing network specific applications in VHDL would be a complex and lengthy process. This would result in the device becoming less user friendly.

Developing the equivalent applications in XC was found to take considerably less time though this is also due to XC being an extension of the C and C++ languages to include the parallel functionality offered by the XMOS XS1. More importantly, because it is an extension of the language, applications written in either C or C++ could be converted into XC which allows applications to be developed in any language that supports Application Programmers Interface (API) calls in C or C++. An example language is Python which can support extensions for both C and C++ by means of API calls[6]. With suitable API calls for the XMOS extensions and some restrictions on the types of applications users

[4]https://www.gnu.org/copyleft/gpl.html
[5]http://opencores.org
[6]http://docs.python.org/2/extending/extending.html

can write, using the XMOS XS1 could greatly simplify the software component of this research.

Considering the costs associated with the representative of each candidate, it initially indicates that the XMOS XS1-L8A would be the better device. Taking into account the hardware requirements however, it is apparent that multiple XMOS XS1 controllers would be required where a single Spartan LX9 would suffice. Assuming three XMOS microcontrollers are required, the cost for the XMOS candidate becomes $45 in total which is significantly more than the $17 required for the Spartan LX9.

In conclusion, it is proposed that the hardware platform for network specific applications be implemented on a XMOS SX1 microcontroller as, though both ease of use and cost are major goals of this research, ease of use was given a higher priority.

3.4 Base Layer

The Base Layer is responsible for all processing and communication that is not directly related to the network applications. This includes supporting a user interface, handling and storing log information, uploading application code to the Network Layer, and possibly be responsible for translating the application code into code compatible with the Network Layer.

Owing to the scale and complexity the development of a suitable device acting as the hardware base of this layer would introduce, it was decided to rather implement this layer using an existing device. Initial investigations identified two possible candidates: the BeagleBone Black and the Raspberry Pi. The BeagleBone White was initially considered as well but was decided against due to cost limitations imposed on the resulting prototype. During the course of the research however, the Intel Edison became available as a potential candidate and the platforms were re-evaluated accordingly.

3.4.1 BeagleBone Black

The BeagleBone Black is an embedded platform designed by Gerald Coley and manufactured by Circuitco LLC. It is an open source platform to allow for the development of systems and applications using an ARM Cortex A8 based processor. The board is currently at revision A5.2 as of February 2014 (Coley, 2013).

The BeagleBone Black can be interfaced with in numerous ways; two common approaches are as a standalone PC, and as a mass storage device. As a mass storage device, the board is connected to the PC via a USB cable which acts as the main communication bus as well as a power supply. Once power is supplied, the BeagleBone operating system will boot up and the device can be interfaced with on the host PC by means of a web browser.

Operating the BeagleBone Black as a standalone PC requires all the necessary I/O peripherals such as a keyboard and display. The device also needs to be powered. Both the display and audio are provided via the HDMI interface. An alternative is to run an SSH server on the BeagleBone Black and interface with it remotely (Coley, 2013).

Board specifications

The BeagleBone Black runs a 1GHz ARM Cortex-A8 processor which includes a 16KB level-1 cache and a 256KB level-2 cache (ARM, 2007). For memory, the BeagleBone comes with a 512MB DDR3 RAM module as well as a 2GB embedded Multi Media Card (eMMC) for storing an operating system. For interfacing with the BeagleBone Black, the board contains both Ethernet and USB. USB interfacing can occur via a mini USB port over which the board will emulate a client device in mass storage mode or via USB in which the device acts as the host controller. Ethernet communication is performed at speeds capable of 100 Mbit/s with its own dedicated communication lines rather then being implemented on the USB.

For external storage, the BeagleBone Black can accept a microSD card which can be interfaced to for increased storage capacity. Unfortunately, due to the presence of eMMC, the general pins on the board cannot directly address the microSD unless the reset pin of the eMMC is held high during transmission. Video and audio are both accessed via HDMI with an on-board HDMI Framer (Coley, 2013).

The BeagleBone Black allows for expansion boards, referred to as ‘capes’, to be stacked on top of it. These capes interface with the underlying board via the expansion headers present on the BeagleBone Black. The BeagleBone Black contains two expansion header sets, each containing 46-pin connectors. It is important to note that these pins operate with a logical high of 3v3 and are not 5v tolerant. There are eight different modes that the pins run in, each providing a different functionality such as SPI and I2C (Coley, 2013).

The creation of capes that are capable of interfacing with the BeagleBone Black is well documented in the BeagleBone Black Reference Manual. This documentation covers best practices and concerns relating to SD communication with the eMMC present (Coley, 2013).

3.4.2 Raspberry Pi Model B

The second potential device that could be used as the Base Layer is the Raspberry Pi. The primary motivation for considering this device was board cost compared with the BeagleBone Black. After initial tests, it was found that the GPIO interface of the board could only operate in the KHz range which was a major concern for proposing this device. Further investigation however, revealed that the speed limitation was a result of pin toggle speeds being controlled by the device operating system and not the ARM microcontroller itself. To account for this, the standard BCM2835 Linux kernel maps the peripheral registers relating to the BCM2835 microcontroller to the kernel address space. This allows applications running on the Raspberry Pi to directly access the microcontroller GPIO registers (Broadcom, 2012). Using this approach, it became possible to have the GPIO interface on the Raspberry Pi operate at speeds up to 20 MHz.

By exposing the register set of the microcontroller, it also becomes possible to enable SPI mode on select pins. This mode enables dedicated hardware which runs an SPI interface with two chip select lines. To further simplify interaction between the operating system and the microcontroller register sets, programming libraries such as the Python library `spidev`[7] exist which help to abstract away from some of the low level interfacing and thus simplify development.

Board specifications

This research adopted the Raspberry Pi model B due to availability and the device meeting all necessary requirements. The Raspberry Pi model B was first released on 29 February 2012 boasting a 700MHz ARM1176JZF-S processor with a Broadcom VideoCore IV graphics processor (Broadcom, 2012). The current version of the B+ contains 512 MB DDR2 RAM. Peripherals include an HDMI port, two USB 2.0 ports, and a 100 Mbit/s Ethernet controller (Raspberry Pi Foundataion, n.d.). The GPIO header on the Raspberry Pi contains fewer pins compared with the Beaglebone Black but still includes eight GPIO pins, a UART interface,an I^2C bus, a SPI interface with two chip select pins, power pins for both 3v3 and 5v, and a ground pin.

[7] `http://lxr.free-electrons.com/source/drivers/spi/spidev.c`

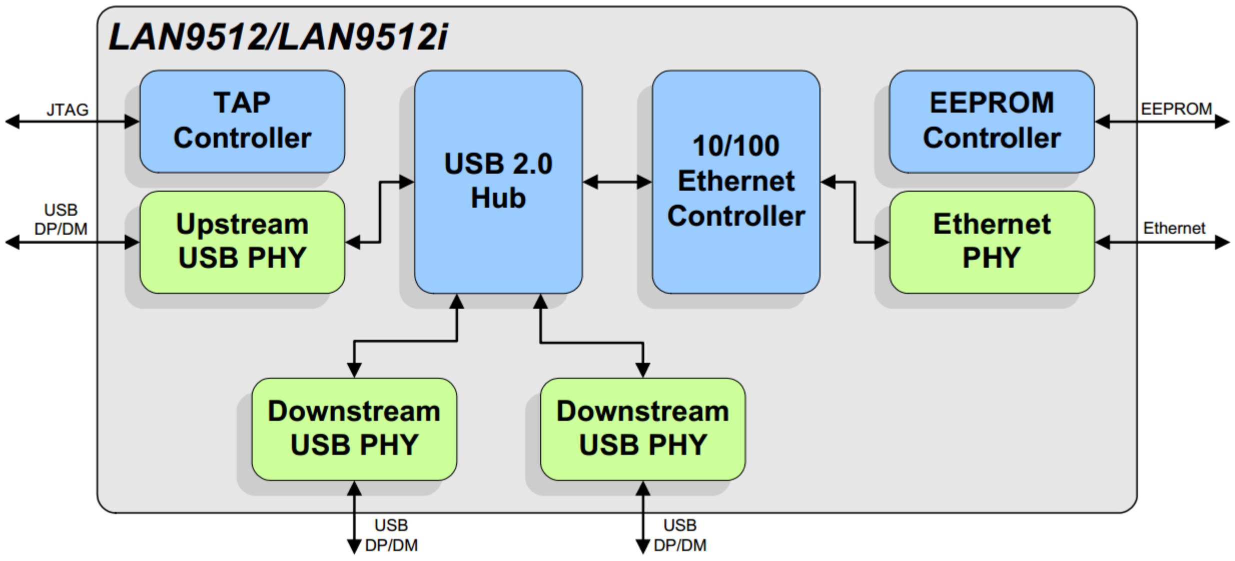


Figure 3.2: Internal Diagram of LAN9512 (SMSC, 2012)

There is no embedded multimedia card on the device but an SD expansion slot is present for external memory and the loading of an operating system. The device also does not support a real time clock and so the operating system must rely on either a time server or external hardware to acquire the current time after start up. (Sheffield Learning Community, 2013)

Ethernet communication is handled by a LAN9512 controller which manages both the on-board USB port and the Ethernet magjack. As illustrated in Figure 3.2 both the Ethernet controller and the USB interface use the same bus to communicate with the ARM processor (SMSC, 2012). Though usually this will have insignificant impact on either peripheral, it is important to note that significant traffic between one peripheral and the device could potentially reduce the performance of the other. As a consequence, it cannot be guaranteed that the Ethernet controller will always operate at 100 Mbit/s.

3.4.3 Intel Edison

The Intel Edison[8] is a compute module designed to have a small form factor and low power consumption. The compute module runs a Dual Core 32-bit Atom processor with each core operating at 500 MHz. The module also includes a separate Z34 microcontroller operating at 100 MHz. The Intel Edison supports 1 GB RAM and a 4GB onboard eMMC for data storage. Communication peripherals include both wireless and Bluetooth

[8]http://www.intel.com/content/www/us/en/do-it-yourself/edison.html

embedded modules as well as SPI and I2C interfaces. USB host support is also available via the compute module breakout pins (Intel, 2014c).

With regard to this research, the Intel Edison comes with two major drawbacks. The first is its small form factor. Though impressive, this was achieved by only presenting the available IO ports via a 70 pin Hirose connector[9] which, due to its small form factor, requires a base platform before any of the pins can easily be interfaced with. The second drawback is the lack of an Ethernet interface being broken out. This is intentional as the target domain of the Intel Edison is wearable devices where wireless communication is preferred over wired communication. For this research, wired communication is preferred.

To simplify interfacing with the Intel Edison compute module, two breakout boards have been released by Intel. Both boards provide two USB interfaces; one for serial communication via an FTDI chip, and one for the compute module host USB. Both breakout boards are also responsible for power management and provide a more accessible breakout of the compute module GPIO. The first breakout board is minimal in size and includes no additional features while the second includes logic translation, an SD interface, and a mounting point for Arduino compatible boards. Drivers exist for both boards to allow for Ethernet over USB and thus provide a wired network interface for the compute module (Intel, 2014b,a).

As both breakout boards resolve the main concerns relating to using the compute module in this research, the approach proposed is to include both the compute module and a breakout board as part of the platform. As an SD card is not required and no Arduino compatible devices formed part of this research, the proposed approach was to use the minimal breakout board and physically mount it on the prototype device.

3.4.4 Base Layer Summary

Prior to the release of the Intel Edison compute module, the proposed approach was to use the Raspberry Pi model B. The justification for this decision was the price difference between the two initial candidates with the Raspberry Pi being the cheaper device. Furthermore, since this research began, XMOS have released a new development platform called the XMOS Startkit which includes a Raspberry Pi GPIO expansion header to promote hobbyists to investigate interactions between the XMOS architecture and the Raspberry Pi (XMOS, 2013b).

[9]http://goo.gl/kWG3p2

After the Intel Edison compute module was released and investigated, it quickly became apparent that it would be a better candidate for the Base Layer, the primary reason being that it runs an x86 architecture natively while both the BeagleBone Black and Raspberry Pi model B use the ARM architecture. All the development toolchains released by XMOS are released as compiled binaries for the x86 architecture. The initial workaround to this was to emulate an x86 architecture on the Raspberry Pi but it was found to take approximately 30 minutes to compile the application code while the Intel Edison achieved the same result in approximately 30 seconds. The conclusion was to develop the Base Layer software to be compatible with both devices.

3.5 System Design

Once suitable architectures had been selected, the software component of the device was designed. This component should take advantage of the capabilities of the selected architectures and be responsible for communication throughout the device.

The selected XMOS architecture is capable of executing a set of concurrent processes in parallel while supporting interprocess communication. Applications that are developed to run on this device should be designed to take advantage of this parallel nature. Taking this into account, the approach adopted by this research is similar to that of the Click Modular Router discussed in Section 2.6.1, where a series of concurrent modules are developed, each to handle some specific aspect of frame processing. These modules can then be linked together to form an application chain. The chain of connected modules represents the path taken by network frames through the application.

3.5.1 Module Overview

To ensure that modules can be freely interconnected, each module must conform to a standard that will guarantee compatibility. The first requirement of this standard is that all compliant modules must be able to operate independently of one another in terms of memory and processing requirements. Processing independence of different modules can be handled by the architecture provided each module is run in a separate core. The architecture includes a scheduler and each core maintains its own registers and handles all scheduling tasks transparently from the application. Ensuring memory isolation can be achieved by reserving a block of memory for each process and guaranteeing that only

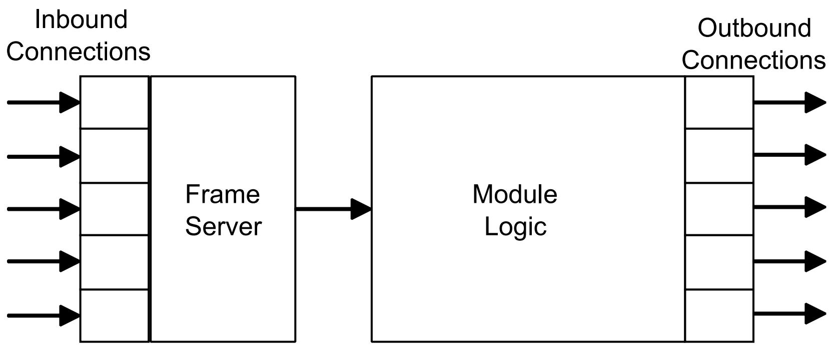


Figure 3.3: Overview of a Module

data copied into that memory block can be modified by the process. Enforcing this independence will guarantee that adding or removing a module will not impact on the functionality of other modules, provided that the application chain is reconnected appropriately. The second requirement is to define the format of data to be communicated between modules. All modules must receive and send data in a specific order with the exception of internal modules interfacing with external hardware. This can be enforced by defining a frame object type and an interface for handling its transmission across a channel between processes.

A secondary factor to consider with regard to communication is parallel communication. The advantage of being able to write an application by creating a chain of defined modules will simplify development however it limits the application to filtering. If we consider allowing parallel communication to and from modules, we allow for branching in the application chain and thus conditionals to be processed. To facilitate this, compliant modules should support many-to-many communication however care must be taken to eliminate any circular chains that could be created.

Considering these requirements, Figure 3.3 describes the first iteration of a generic module to be used in an application chain. All frames received from one of the input channels will be processed before being sent to the specified group of output channels. The specification of which channels the data is presented to should be determined by the processing logic which is further discussed during implementation in Chapter 4.

As platform flexibility is an objective of this research, these modules must be customizable. However, as simplicity is also important, some modules will need to be implemented without any user input. These internal modules will be preconfigured similar to the modules implemented by the Switchblade discussed in Section 2.6.5. However, where the Switchblade modules focus on performing a specific frame related task, these internal modules will focus on interfacing with external hardware to simplify further module development.

As a result, these internal modules can be communicated to and from but not modified. It is necessary to define such modules as communication with some external components such as SDRAM and the networking PHYs is essential for a viable application but may require specialized knowledge to implement. Furthermore, internal modules must be implemented on specific tiles within the architecture for access to the correct pins. These modules should be separately developed and attached to the module chain as dictated by the user.

For the first revision of the device, three different types of internal modules will be required. These are defined by the hardware components that the application may need to interface with. The interfaces that will be supported in the first revision of the software component are:

- An interface for the storing network frames in SDRAM.
- An interface for communicating back to the Base Layer.
- An interface for each of the four network interfaces.

The internal modules required for each of these components are further discussed in Chapter 4.

3.5.2 Application Example

To better elaborate on the approach of using modules to create applications compatible with the prototype device we designed a basic application according to a simple scenario. The presented scenario requires that the platform pass all frames received from one network interface to another (for the sake of this example we will label these portA and portB). All frames transmitted between these ports should be unaltered unless they are transmitted under the ICMP protocol from portA to portB. Should this be the case, the Time To Live (TTL) field of the ICMP packet should be set to zero before the frame is retransmitted. Furthermore, the source IP address of the ICMP packet should be recorded in a log file on the device for later evaluation.

Considering this scenario, we first note the internal modules that will be necessary for the application to function. As traffic will be streamed between ports A and B, two internal port modules are required to cater for this. To handle the exporting of IP addresses from the Network Layer to the Base Layer an SPI interface module needs to be included in the application.

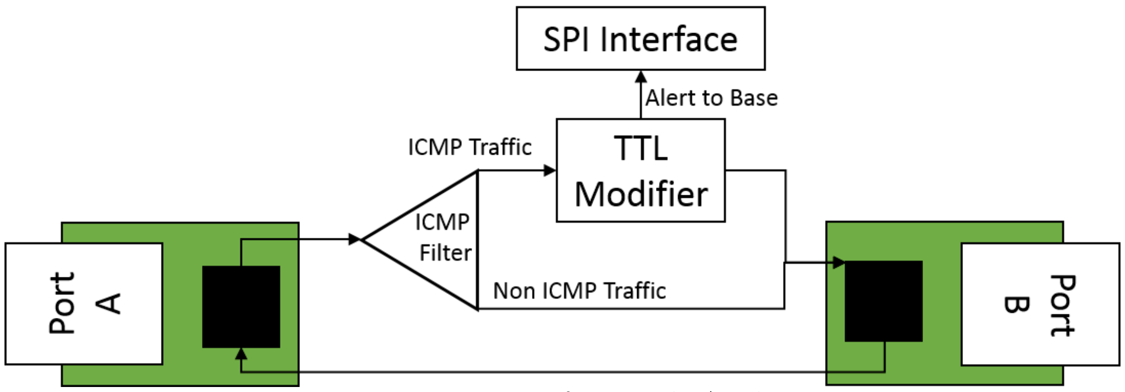


Figure 3.4: Diagram of Example Application

Now that communication between the networking layer and other systems has been considered, it is necessary to design the application logic. For this scenario there are two stages to the application; filtering ICMP packets and modifying ICMP packets. As traffic is received from port A it needs to be filtered of all ICMP traffic which is then routed to the modifier module while all other traffic passes to port B unaltered. The modifier module is then responsible for setting the TTL field to 0 before passing the frame on to port B. A secondary function of the modifier module in this example is to transmit the source IP address to the SPI internal module so that it can be recorded by the Base Layer. This could be handled by the filter module but for the sake of this example, it is the responsibility of the modifier module.

Figure 3.4 depicts how the modules would communicate as described in the above application which can act as an example to highlight potential areas of concern which should be investigated when implementing similar applications. The first point to note is that the communication between modules must be capable of operating at transmission speeds equal to the Ethernet interfaces or greater. Should the communication channels be slower, it will cause a backlog throughout the application which in turn will cause network frames to be lost as the port buffers overflow. To address this, a more thorough investigation of the different XMOS communication methods must be performed.

A second issue to note is that some internal modules will be interfacing with other components using communication protocols that might not be capable of transmitting data at 100 Mbit/s such as SPI. For these modules, special care needs to be taken to support buffer responsible for temporarily storing messages. The buffer is still susceptible to overflow but such an event will not affect transmission speeds relating to the rest of the application. When allocating the buffer size, it should be noted that multiple modules could be communicating to an internal module simultaneously.

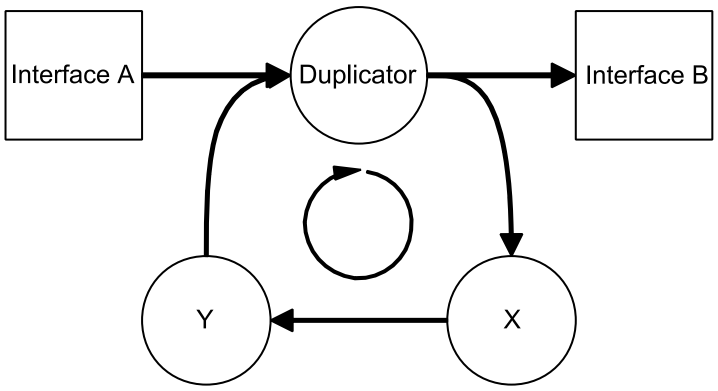


Figure 3.5: Example of a Circular Path

Figure 3.4 depicts both the ICMP Filter and the TTL Modifier as having two outbound connections, allowing for communication to multiple modules. Though this functionality is essential, it does invite the possibility of generating a circular path. Care needs to be taken to avoid such a situation where traffic gets trapped within the application and never reaches a port module or is dropped. Figure 3.6 shows an example of this where traffic from port A passes through a duplicator which sends the frame to both port B and module X. Module X passes the frame to module Y before it is returned to the duplicator module. The duplicator, as before, sends the frame to port B and module X thus creating a circular path. As more frames are received from port A, communication between modules X, Y, and the duplicator becomes blocked and the entire application halts.

3.5.3 Base Layer Software

To accommodate user interaction with the device, a user interface has been developed on the Base Layer. For this revision, the user interface was implemented as a simple dynamic webserver for user interaction. As information is transmitted from the Network Layer to the Base Layer, it is stored in log files on the device to be retrieved later. CGI scripts also have access to these log files and the recorded information can be served to the user via a console interface in the web application. Traffic information relating to active ports is logged by the Network Layer and transmitted to the base. This information is stored in a database which can then be accessed and visualized by the web application.

As the Base Layer is expected to handle all user interaction with the platform, this layer should be responsible for receiving and compiling application code to execute on the Network Layer. This procedure is highly dependent on the approach taken to provide ease of use which could include additional processing of submitted code to be compatible

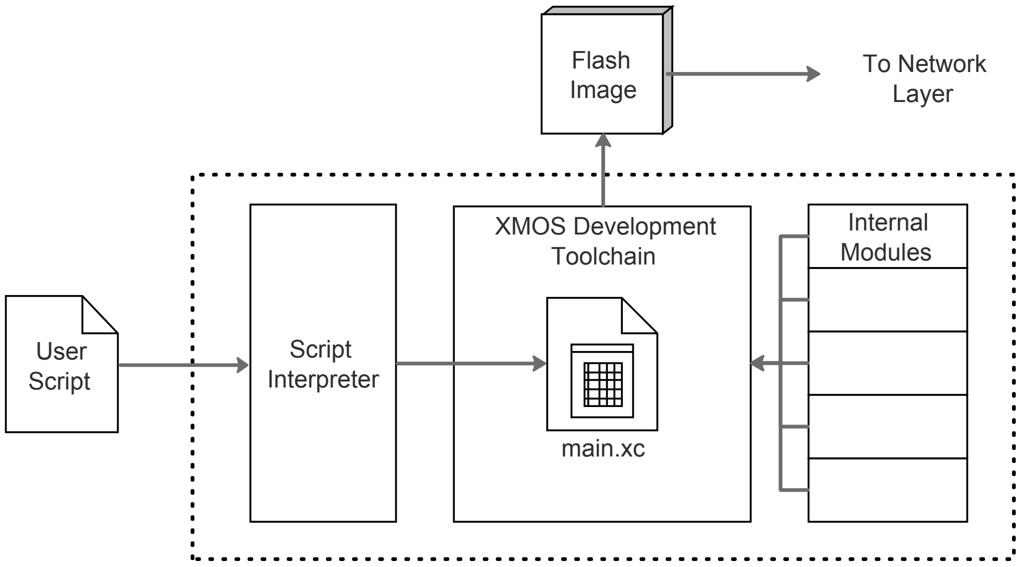


Figure 3.6: Overview of Uploading Script to Platform

with the XMOS development tool chain[10].

Figure 3.6 shows the procedure followed for executing user applications on the Network Layer. After the script is received via the web server, the application code is interpreted into a format that can be processed by the XC compiler. Associated with the resulting code are all relevant internal modules that will be required by the application during execution. The resulting project is then compiled to produce an application binary and later converted into an application image. This image is then written to the flash device embedded on the Network Layer of the platform.

3.6 Summary

This chapter identified both the hardware and software requirements of the prototype device which was developed in this research. The chapter began by separating the physical components of the device into two layers, the Network Layer and the Base Layer, to simplify hardware selection and development. Potential architecture candidates were then identified and compared before an architecture for each of the layers was selected.

The remainder of the chapter focused on investigating the software requirements of the system and designing the basic layout of an application module as well as describing how applications could be developed using them. Finally, software for the Base Layer was

[10]Details relating to the development tools required for compilation are described at: `http://www.xmos.com/products/tools`.

addressed with this layer acting as the user interface. It was concluded that the Base Layer should run a web server through which the user could interact with the platform for uploading applications and monitoring system status. The Base Layer would also be responsible for application code compilation and programming of the Network Layer. Chapter 4 introduces the implementation of the software architecture designed in this chapter to run on the selected hardware.

4 Software Implementation

4.1 Introduction

This chapter begins with an analysis of transmission speeds using the XMOS communication architecture discussed in Section 2.3.1. Comparisons between different communication types for intratile and intertile communication are highlighted. After an appropriate communication architecture has been selected, implementation of the proposed software architecture will begin by implementing each of the internal components described in Section 3.5. Finally, to associate the DSL with the internal modules, a translator responsible for producing XC source code from the DSL syntax is implemented.

4.2 Analysis of Transmission Speeds

As discussed in the application example presented in Section 3.5.2, before any implementation of the proposed system can be performed, data transfer speeds achievable by the XMOS architecture require further investigation. This investigation is necessary to gauge

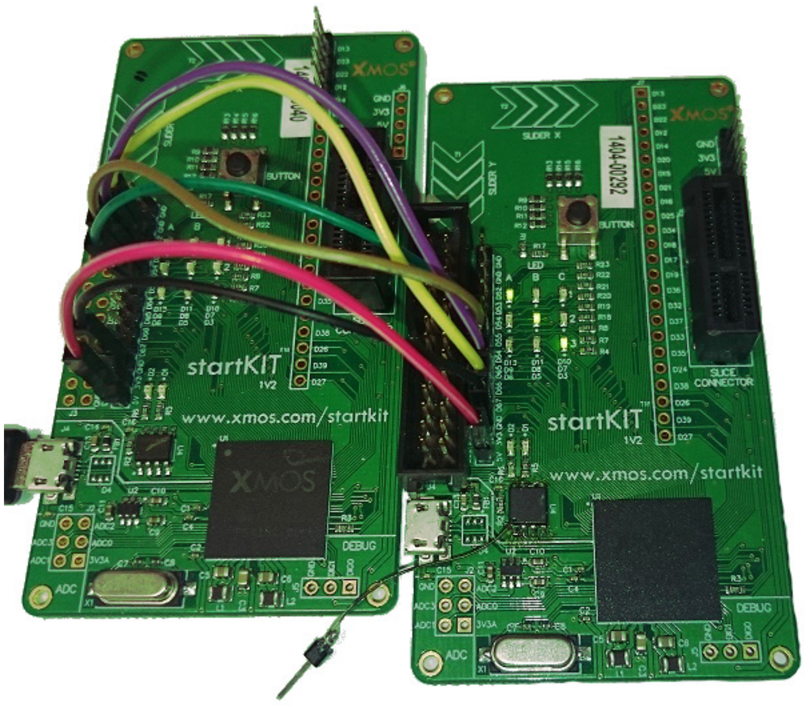


Figure 4.1: Implemented Platform for Testing XMOS Serial Link

the feasibility of producing applications operating on the XMOS architecture that can maintain network throughput comparable to retail devices that are IEEE 802.3u compliant (IEEE, 2012). As discussed in Section 2.3.1, the XMOS architecture supports two mediums for data transmission between tiles: the XMOS Fast Link, and the XMOS Serial Link. Both mediums could be suitable candidates for transmitting a network frame without incurring degradation of network throughput due to delay. To confirm the theoretical transmission speeds discussed in Section 2.3.1, two sets of tests were performed to analyse throughput achievable with each of the communication architectures.

To test the XMOS Fast Link, a test environment was designed to operate on an XMOS Slicekit development board (XMOS, 2013a) which supports an XMOS XS1-L16 microcontroller. This board was chosen as the test platform since the XMOS XS1-L16 microcontroller is composed of two tiles connected with a series of XMOS Fast Links (XMOS, 2013h). The testing environment was implemented as a simple traffic relay capable of passing network frames between the two tiles within the microcontroller.

To test the expected throughput of the XMOS Serial Link, the testing environment designed on the XMOS Slicekit was ported to a different XMOS development platform where the XMOS tiles are physically separated and communication limited to using an XMOS Serial Link. The second testing platform was composed of two XMOS Startkits (XMOS, 2013b) connected as depicted in Figure 4.1. The Startkit models were selected due to their low cost and availability while still being fully compatible with the testing environment previously implemented.

Once the two suitable testing environments had been set up, transmission analysis of network frames could begin. Tests implemented were designed to be compatible with both communication mediums so as to provide a meaningful comparison between results. The network frame was represented by a body of data composed of 1522 bytes and four additional bytes for representing the frame length. As the goal was to have the application run across multiple physically separated XMOS devices, tests performed were partitioned into two groups: intertile communication and intratile communication. Timings relating to intertile communication on the two Startkits were carried out using an oscilloscope to record the time taken to transfer a full frame between the two tiles. This was achieved by setting the oscilloscope to monitor a status pin associated with a Startkit which was set to toggle between transfers. Intratile related timings for both testing environments was recorded using a timer available on the development boards.

4.2.1 Intratile Delay

To accurately monitor the time taken for each task to complete on the XMOS development boards, the xTAG interface was used. This provided an interface for logging and debugging purposes communicating over USB. This communication was performed using a method similar to UART and so care was taken to ensure that communication with the host did not interfere with the task being tested (XMOS, 2009). Timings were done using a 100MHz clock which generated a timestamp directly before and after the task being tested was executed. A consequence of performing tests in this manner is that all timing results would have an associated uncertainty of up to 20 ns which should be taken into account when evaluating the testing results.

The first investigation performed was a transmission test when using the `memcpy`[1] function to duplicate a body of data on a single logical processor. This test was designed to evaluate the performance of `memcpy` itself. Both the time taken for `memcpy` to complete and the volume of data copied was recorded. For this test, 15000 different data sizes were sampled ranging from 0 bytes to 30000 bytes being copied. The results in Figure 4.2 show a clear linear relationship between volume copied and time taken. Using regression

[1]A function that copies a block of memory from one memory location to another memory location (The Open Group, 2013).

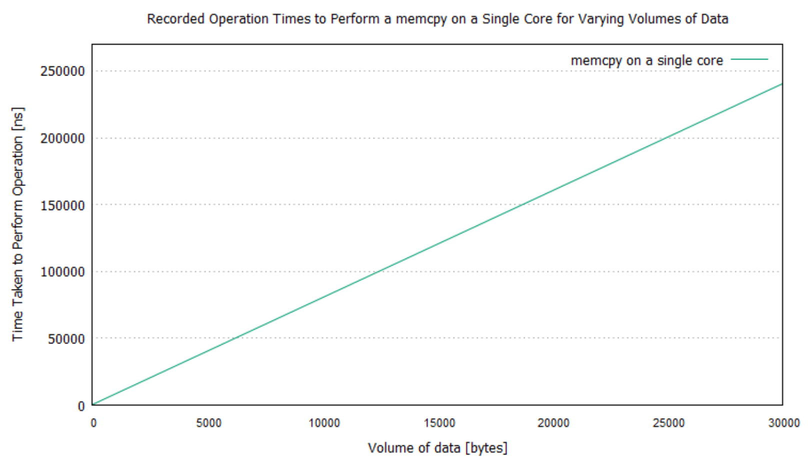


Figure 4.2: Graph Showing Relationship Between Time Taken by memcpy and Number of Bits Copied

analysis, this relationship is represented by Equation (4.1) with an R squared value[2] of 1. Using this equation, the latency for a 60byte frame was calculated to be approximately 515.2 Mbit/s which is close to the recorded latency of 545.5 Mbit/s.

$$t_{ns} = bits \times 8 + 414.22[ns] \tag{4.1}$$

To test transmission speeds between two logical processors within a single tile, the same test was performed but with the source memory array being accessed by a different logical processor. To conform with the previous **memcpy** test, 15000 different data sizes were sampled ranging from 0 bytes to 30000 bytes being copied. Regression analysis was again used to produce the linear Equation (4.2) which relates the time taken to perform the operation to the volume of data transmitted. As with Equation (4.1) the resulting equation was recorded to have an R squared value of 1.

$$t_{ns} = bits \times 6 + 927.27[ns] \tag{4.2}$$

The larger constant is a consequence of the delay relating to the channel communication but it was found that when operating over sufficiently large bodies of data, a **memcpy** between two logical processors became more efficient when compared to a **memcpy** within a single logical processor. For a network frame of 60 bytes, the transmission speed is approximately 372.88 Mbit/s.

[2]The R Squared is a value between 0 and 1 which indicates how closely the recorded data matches the model (Gujarati and Porter, 2008).

Considering the maximum frames size of 1522 bytes, performing a `memcpy` within a single core results in a throughput of 964.2 Mbit/s while performing the equivalent operation across two cores results in a throughput of 1210.4 Mbit/s. In both cases however, the recorded and predicted throughput of a data using `memcpy` well exceeds the maximum network throughput of 100 Mbit/s, minimizing the performance impact `memcpy` could have on network throughput.

```
<Link Encoding="5wire" Delays="0,1">
  <LinkEndpoint NodeId="SLICE_0" Link="XLG"/>
  <LinkEndpoint NodeId="SLICE_1" Link="XLF"/>
</Link>
```

Listing 4.1: Example XN Link Specification

4.2.2 Intertile Delay

To account for wire impedance and track length between the communicating microcontrollers, the developer can set delays which are associated with the link defined between two tiles. As discussed in the XN Specification document (XMOS, 2013g) and depicted in Listing 4.1, a link delay has two values associated with it specified by the label [`Delays="X,Y"`]. The first value (`X`) represents the intertile delay value while the second (`Y`) represents intratile delay. These delays indicate the number of cycles the switch should delay between signal transmissions so as to maintain a reliable link between the two tiles. Restrictions on this value are that the delay must be an integer value and cannot be negative. Furthermore, the intertile delay must be less than or equal to the intratile delay. During hardware design, it is important to minimise the length of wires used as a link so that the specified delay can be set as low as possible while still being reliable. The number of cycles by which the switch delays a transmission has a direct impact on the maximum throughput of that link.

XMOS Serial Link

To investigate actual speeds observed in the application, an oscilloscope was used to measure the duration of a bulk data transfer across the Serial Link. One of the goals of this research was to show that it would be possible to implement an XMOS network that could handle transmission at 100 Mbit/s while using the serial XMOS link rather than being required to use the XMOS Fast Link. The primary motivation for selecting the

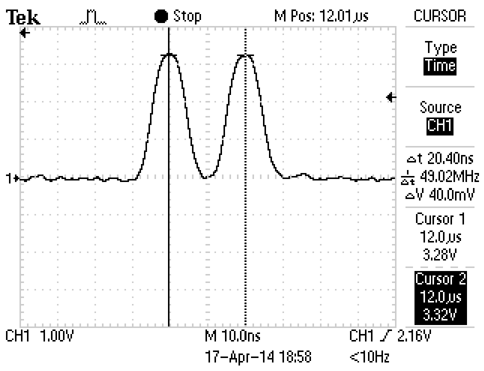
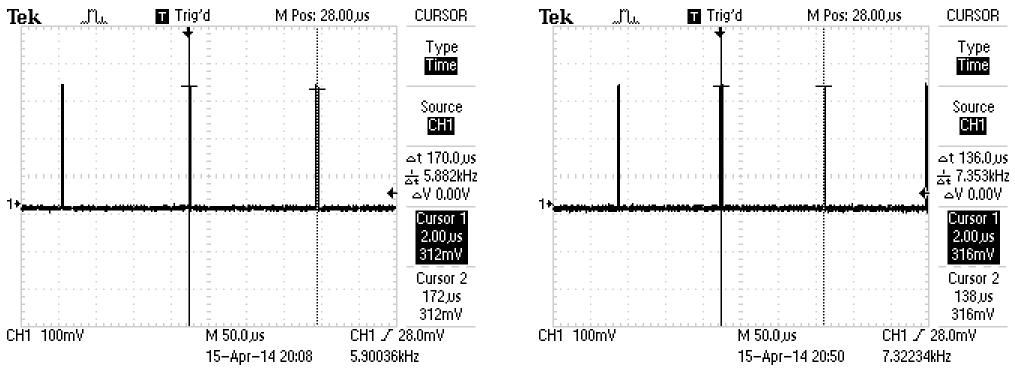


Figure 4.3: Pin Toggle Speed on XMOS Startkit



(a) Four Cycle Delay (b) Three Cycle Delay

Figure 4.4: Transfer of a Frame With Different Delay Times

Serial Link over the Fast Link is the reduction in pins required to instantiate an XMOS Serial Link, thus making it more readily available on low cost XMOS devices such as the XS1-L4 (XMOS, 2013i).

The time taken to transmit 1526 bytes between the two Startkits was recorded by toggling the output state of a separate one bit port after each bulk transfer was completed. Figure 4.3 graphs the time taken to perform a state transitions on the monitoring port from which the duration of a full cycle was measured to be approximately 20 ns. As a result, all recording related to this test included an additional 20ns of latency which needed to be accounted for.

Figure 4.4(a) graphs the time taken to transmit a single frame over the XMOS Serial Link. At the beginning of each transfer, the monitoring port was set low and remained in this state for the duration of the operation. As soon as the operation completed, the monitoring pin was set high again, indicating the operation was complete. The operation performed a bulk transfer of data by first sending a memory reference across the XMOS Serial Link to the destination tile. The `memcpy` operation was then called on this remote array reference to handle the bulk memory transfer. This process took 170 μs which resulted in a transmission speed of 71.62 Mbit/s.

The next attribute of the transmission to consider is the delay applied to the link between XMOS devices. For the above test, this delay was set to four cycles for both intertile and intratile communication. By reducing this delay to two cycles for intratile communication and three cycles of intertile communication, it was possible to record a reduced transmission delay. Figure 4.4(b) records this reduced delay to be 136 μs which implies a transmission speed of 89.53 Mbit/s. Attempting further reductions in delay resulted in errors in communication.

XMOS Fast Link

To investigate the XMOS Fast Link, the testing environment implemented on the XMOS Slicekit was used. As both tiles were contained in a single microcontroller, timings relating to the data transfer operations were recorded using internal timers rather than monitoring an external pin.

As with the testing of the XMOS Serial Link, the transfer operation was performed by first sending a remote memory reference of the array to the destination tile. The `memcpy` command was again used to copy the array contents to the destination tile. As the two tiles are contained within the same microcontroller, the delays relating to the XMOS Fast Link were set to a minimum of 0 cycle intratile and 1 cycle intertile. The process was recorded to take 27.96 μs which results in an approximate transmission speed of 435.47 Mbit/s, a significant increase on the speeds observed from the XMOS Serial Link.

To observe the effects of increasing the values associated with both the intertile and intratile delay would have on data throughput, the same test was repeated for multiple delay values, the results of which are depicted in Figure 4.5. The tests indicate a clear linear relationship between the number of cycles delayed and the transmission speed with a four cycle intertile and intratile delay resulting in a transmission speed of 175.42 Mbit/s.

After reviewing the results from initial tests on both the XMOS Serial Link and the XMOS Fast Link, it was concluded that platform implementation should opt to use the XMOS Fast Link exclusively. Though the transmission speeds of the XMOS Serial Link achieved results that were better than expected, data transmission between the logical processors within the XMOS platform must achieve a minimum throughput of 100 Mbit/s or else risk becoming a bottleneck for network traffic throughput thus reducing the throughput of the entire system.

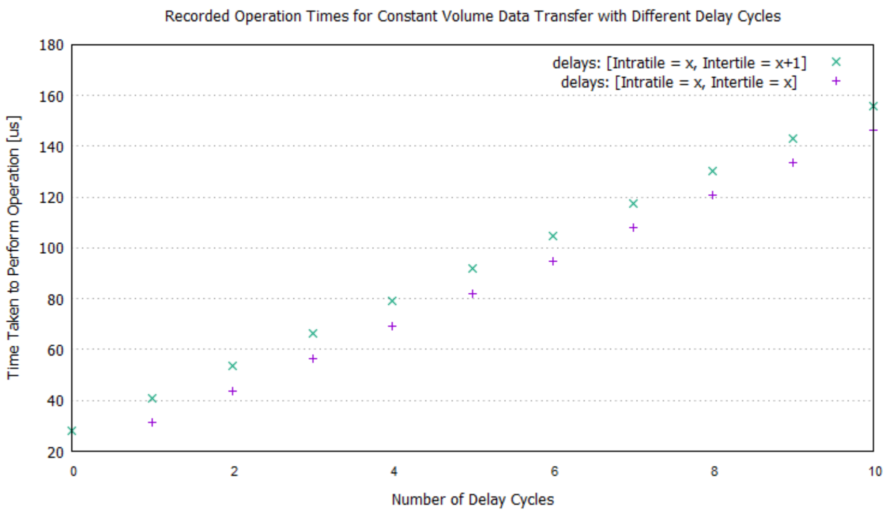


Figure 4.5: Graph Showing the Recorded Time Taken to Perform Data Transfer at Different Delays Using the XMOS Fast Link

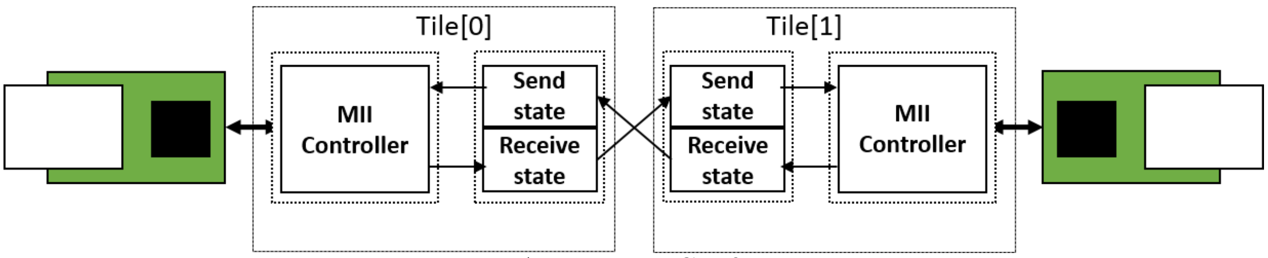


Figure 4.6: Application Configuration

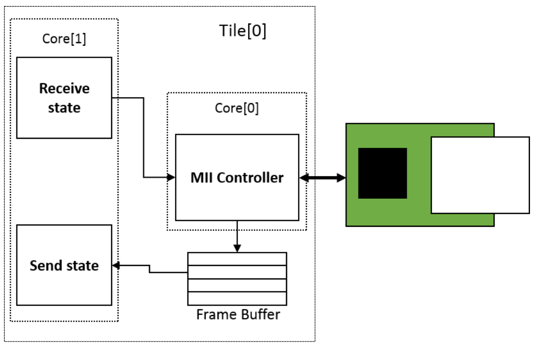
4.3 Implementation of Internal Modules

As discussed in Chapter 3, it was necessary to design and implement a series of specialized modules for interfacing with external hardware. These modules would be preconfigured and made unavailable to the user for modification. To ensure these modules had access to the correct ports on the XMOS microcontroller, they would be restricted to operate on specific tiles within the platform and only instantiated if required by the user application.

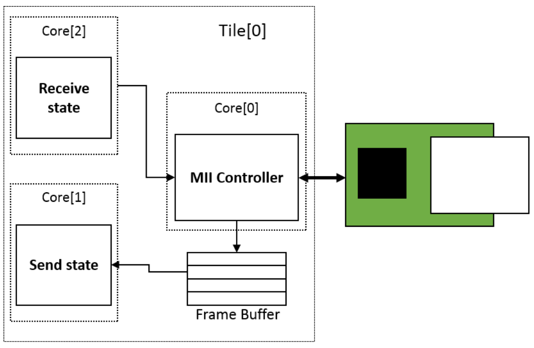
4.3.1 Port Module

The first internal module implemented was the port module which is responsible for all communication with the PHY. The construction of this module was based on the Ethernet module application project provided by XMOS (XMOS, 2013f) for communicating with the Ethernet breakout cards developed by XMOS.

An investigation was undertaken on an XMOS SliceKit with two Ethernet modules to



(a) Port module using two logical processors



(b) Port module using three logical processors

Figure 4.7: Ethernet Port Module Used to Interface with the Physical Transceiver

evaluate the performance of the module application. An Ethernet module was attached to each of the two tiles of the Slicekit development board so that all network communication would be restricted to using the connecting XMOS Fast Link.

To test functionality of the implemented configuration, the Ethernet module and an associated demo application were downloaded. Two variations of the Ethernet module are available from XMOS: a full version and a light version, both of which are further discussed in Section 3.3.1. As the focus of this research is on generic traffic processing, the additional features provided by the full Ethernet module were considered unnecessary and so the light version of the module was used. The light Ethernet module in Figure 4.7(a) was designed to operate on only two processors with the MII controller responsible for all communication between the PHY while the second processor handled all communication with the remaining application. Frames received from the network are stored in a buffer which is shared between the two processors.

The demo application[3] was only designed to support one Ethernet module and so had to be modified to allow for the use of both. The goal of the demo application was to act as a loop-back that reflected all frames received back onto the network. By changing the receive and send channels that the loop-back component interacted on to instead be connected to a second Ethernet module, the application was made to act as a frame relay as shown in Figure 4.6, passing all traffic seen between the attached endpoints.

Performance tests using this configuration were done to monitor the bandwidth the application was capable of supporting. Using `iperf`(Tirumala *et al.*, 2005), this was recorded to be 7 Mbit/s of bandwidth for a TCP communication. Traffic passed in a single direction via UDP allowed a throughput of up to 20 Mbit/s.

[3]Available from: `https://github.com/xcore/sc_ethernet`.

During these tests, it became apparent that there were problems with the traffic transfer between endpoints as occasionally the endpoints connected to the device would cease communication entirely until the device was restarted. A more detailed investigation into the the Ethernet IP module functionality revealed this event to be a deadlock where both Ethernet modules attempted to send a frame at the same time. In this situation, both server applications had entered the Send State shown in Figure 4.6 and put data into their respective channel buffers to transmit. Once in this state, the processor cannot transition into any other state until the receiving processor has removed the data from the buffer. As a result both server processors would lock up and no traffic could be transmitted.

After consideration, the approach implemented to resolve the deadlock issue was to split the server module across two different processors, one responsible for sending frames and another responsible for receiving them and relaying the associated data to the MII controller. This extension of the the module effectively resolved the deadlock issue at the cost of requiring an additional processor be reserved for each port module present in the application. Figure 4.7(b) shows the revised configuration of the Ethernet module with sending and receiving components of the server occurring in parallel. Due to the number of xCores available on most XMOS microcontrollers and the importance of removing the deadlock issue with minimal impact to transmission speed, the proposed solution is considered acceptable.

After resolving the deadlock, focus shifted to optimising the throughput of the platform to achieve more suitable transmission speeds. Speed reductions are expected due to transmission delay through the platform, but should not reduce the bandwidth of a 100 Mbit/s line by 93%.

Numerous optimizations were performed to pipeline communication with the PHY and the rest of the platform. These optimizations reduced traffic latency from 200 ms to less than 10 ms. Raw channel communication was first replaced with transaction communication which removed synchronization requirements in transmission and this was then later replaced with interface communication. Interface communication is not bidirectional but optimizes transmission of data according to a specified type (XMOS, 2014). After optimizations were implemented, the bandwidth throughput of the development platform was increased to 80.8 Mbit/s.

4.3.2 SPI Module

Although it was initially decided that communication between the Network Layer and Base Layer would occur using an SPI interface, an alternate proposal was to develop a parallel interface between the two layers to take advantage of the large number of GPIO pins available on the selected device. As discussed in Chapter 3, the Raspberry Pi was initially selected as the Base Layer and is capable of handling frequencies in the region of 20MHz. In theory, a parallel interface operating at a speed of between 15MHz and 20MHz could be implemented between the Base Layer and the Network Layer. The implementation of a communication interface however, resulted in a considerable reduction in pin toggle speed with data being transferred at a rate of 4MHz. The slow down was due to additional processing required by the Raspberry Pi to read and write pin values rather than simply performing a continual write. Knowing this toggle rate and reserving 10 GPIO made it possible to generate a parallel port capable of transmitting data at 27 Mbit/s which would was considered suitable for a prototype device.

Corruption in larger data transfers was recorded which was concluded to be a result of cable length. Reducing the transmission speeds improved reliability on the connection but reduced throughput to a point where it was unsuitable even for prototype purposes.

Considering the possible stability and compatibility issues of the parallel interface, it was decided that an optimized communication medium between the Raspberry Pi and the XMOS device was not essential and the standard SPI protocol would be more suitable. Furthermore, by implementing the communication to operate using SPI, the process of replacing the Raspberry Pi with the Intel Edison later in the research was greatly simplified. An SPI module was developed for the XMOS architecture that would allow it to act as a slave device on an SPI communication channel. The module allows for standard bidirectional communication using the clock provided by the master device which in this case was the Base Layer.

4.3.3 Logging Module

Once the SPI communication module was implemented and tested, a logging module was implemented to transfer logged information from the Network Layer to the Base Layer. The logging module is split into two concurrent processes with a shared set of circular buffers between them. The first process acted as a logging server for the Network Layer that accepted internal connections from other modules. This process would then store

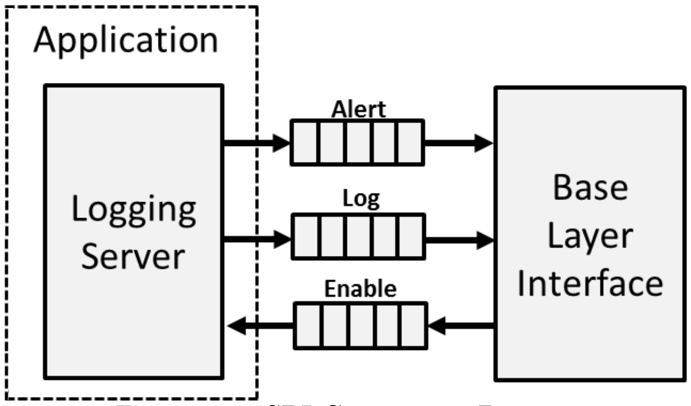


Figure 4.8: SPI Component Diagram

these records in the head of the circular buffer associated with that log object type before updating the buffer count. The second process would then use the SPI module to set up an SPI slave interface with the Base Layer. Whenever the base sends a data request command, this process would respond by fetching the tail element of a non empty circular buffer and transmitting it to the Base Layer before decrementing the buffer count. A single instruction character was sent prior to the log object to help the Base Layer identify the object type received. Numerical values were also preceded with an instruction character indicating that the next set of characters represent a number and are not an ASCII value. There are multiple number instruction characters implemented, each indicating a different number width in bytes.

Separating the SPI interface and the application interface into concurrent processes was necessary to minimize the performance impact communication over SPI would have on the running application. Transmission of data within the Network Layer must maintain a speed comparable with 100 Mbit/s while the SPI interface should be allowed to operate at speeds of 15 Mbit/s to 20 Mbit/s.

The implemented circular buffers will overflow if the transmission speed of the SPI interface cannot match the rate at which the application generates logs. This approach was decided on over trying to store all logs generated which would require considerably more dedicated memory. Figure 4.8 describes how the component functions by using memory buffers shared between the two processes. During the testing of this module it became clear that applications set up to generate logs for every frame would cause the buffers to overflow regularly which was expected but also resulted in a rare case where a log being recorded would become corrupt.

The corruption was due to both processes accessing the same buffer entry with the write occurring faster than the read. To account for this, a `buffer_in_use` flag was implemented to signal that the buffer was being modified by a process. Memory space for a single buffer entry was also reserved for the process interfacing with the SPI module to reduce time the buffer was held in use.

Whenever the Base Layer requests data, the `buffer_in_use` flag of a non-empty buffer is raised and the tail of the buffer is copied using `memcpy` into the reserved memory. The flag is then released before any data is actually transmitted to the base. The slower SPI can then operate on the allocated memory independent of the buffer.

Concerning data overflow, the current approach is to set up reasonably large buffers and then simply allow for frame loss to occur without impacting on platform performance. A potential indicator that could be added to the board would be an LED so the the user can be made aware when an overflow occurs but no further action will be taken by the prototype.

4.4 Programming and Flash

To allow the platform to operate without supporting hardware, the Base Layer must be capable of uploading the compiled user application to the Network Layer for execution. Currently this is done via the XMOS XTAG external programmer (XMOS, 2009). This programmer is based on the JTAG interface[4] which writes the compiled application image onto the XMOS tile it is connected to. Additional tiles are then programmed by the initial tile sequentially in a chain as specified by the application configuration file. Converting the Raspberry Pi into an XMOS programmer was briefly investigated which, if successful, would allow the Base Layer to upload applications to an XMOS board without any additional hardware. Investigations concluded that the complexity of the is approach was outside the scope of this research but also revealed a potentially simple alternative regarding the flash interface present on XMOS development boards.

The XMOS XS1 microcontrollers support the ability to retrieve application code stored as images on attached flash devices as discussed in the XMOS XS1-L16 datasheet (XMOS, 2013h). If the MODE[5] pins of the device are correctly configured, the microcontroller will start up and initialize an SPI master interface over four specific pins. The controller

[4]Joint Test Action Group (JTAG) is the common name for the IEEE Standard 1149.1 (Microsemi Corporation, 2012). It is a standard initially designed to aid in testing controllers already embedded on printed circuit boards but has grown to include on chip debugging and communication functionality.

[5]Boot configuration pins which must be set before the device is powered on. These pins determine the expected clock rate of the device and how it expects to be programmed on startup.

will then download the image from the attached flash device and program itself and all connected tiles according to the configuration file used when generating the image. The approach taken by this research then is to bypass the XMOS microcontrollers and access the flash device directly from the base platform. The generated flash image can then be written to a flash device on the Network Layer which the XMOS microcontrollers can access on startup.

The feasibility of this approach was tested using an application called flashrom (Flashrom, 2015) in conjunction with a Raspberry Pi. Attaching a flash device to the appropriate pins of the GPIO header of the Raspberry Pi allowed flashrom to write a binary image to the attached device. Initial attempts at performing this operation on the XMOS Slicekit were unsuccessful even with the SPI interface of the Raspberry Pi directly connected to the flash device. Removing the flash device from the development board and programming it externally proved to be successful and when reattached to the board, could be read and executed by the development platform. Reviewing the schematic for the Slicekit[6] revealed this issue to be due to switch chips which resulted in the flash device being physically disconnected from the development board when attempts were made to program it. Provided that accessibility of the flash device on the prototype device can be made available to the Base Layer, this will be the approach taken for programming application code to the Network Layer of the platform.

Due to compatibility limitations, it was found that flashrom (Flashrom, 2015) was not available for the Intel Edison, rendering the the initial revision of the upload application incompatible (Flashrom, 2015). Section 3.4.4 concluded that all Base Layer software should operate on both the Raspberry Pi and the Intel Edison and the upload software was reimplemented to achieve this.

Lower level API calls were used to interface with the SPI controller on both devices. Though these differ between devices, all hardware specific functions are wrapped to allow the application to be compatible with both architectures after minimal modifications.

4.5 Frame Delay Module

An important feature in simulating a network environments is the ability to implement a delay in network traffic. A delay can be used to simulate larger networks by only allowing

[6]Available from the XMOS website as part of the Slicekit documentation `https://www.xmos.com/support/boards?product=15825&secure=1`

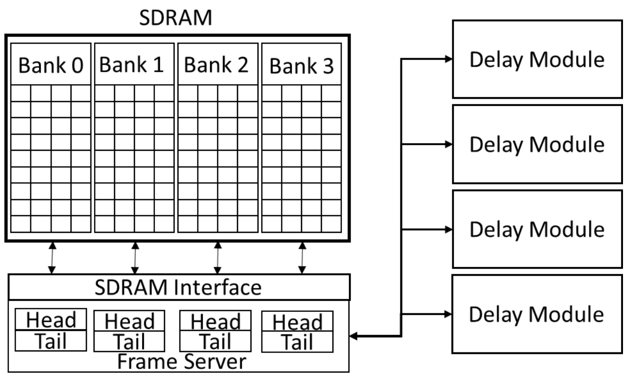


Figure 4.9: Functional Layout of the Delay Module

received frames to propagate forwards after a set duration of time has passed. This delay emulates the delay seen by network traffic traversing a link with multiple hops. Modifying the TTL field of relevant packets may also be required to simulate this in a more realistic manner.

For this research, network frame delay is implemented by including a Synchronous DRAM block on the platform which could be used to store frames during operation. The SDRAM chosen was the IS42S16400F which contains 64Mib of memory, segmented into four memory banks (ISSI, 2008). To interface with the block, an SDRAM interface provided by XMOS was used which allowed for read and write operations capable of achieving transmission speeds of 60MB/s or 480 Mbit/s (XMOS, 2012a). Interfacing with the application layer of the platform was done through a delay module which is functionally described in Figure 4.9. The SDRAM interface communicates with a frame server process which is responsible for maintaining the state of the four memory banks. Each memory bank is treated as a circular buffer that operate independently, allowing for up to four delay modules to be safely instantiated in an application. The frame server process maintains both the head and tail of each buffer as well as how each translates into the geometry of the SDRAM block. Each buffer can store up to 1024 Ethernet frames before overflow will occur which translates into a delay time of between 6.5ms and 168.4ms when operating at maximum throughput. The large variation in possible delay times reflect the volume difference between the largest and smallest Ethernet frames that could be transmitted under the IEEE 802.3 standard (IEEE, 2012)[7].

The delay modules were abstracted away from the implementation of the memory storage and treated the frame server as a FIFO buffer of a finite size that accepts and returns network frames. Details relating to the duration which a network frame should be stored

[7]Jumbo frames were not adopted by the official IEEE 802.3 standard, limiting the maximum Ethernet frame size to 1522 bytes.

Table 4.1: Statistical Summary of Read and Write Operations for SDRAM [in ns]

	Write [ns]	Read [ns]
MIN	39390	34320
Q1	39400	34320
AVG	39403	34325
Q3	39410	34330
MAX	39410	34330

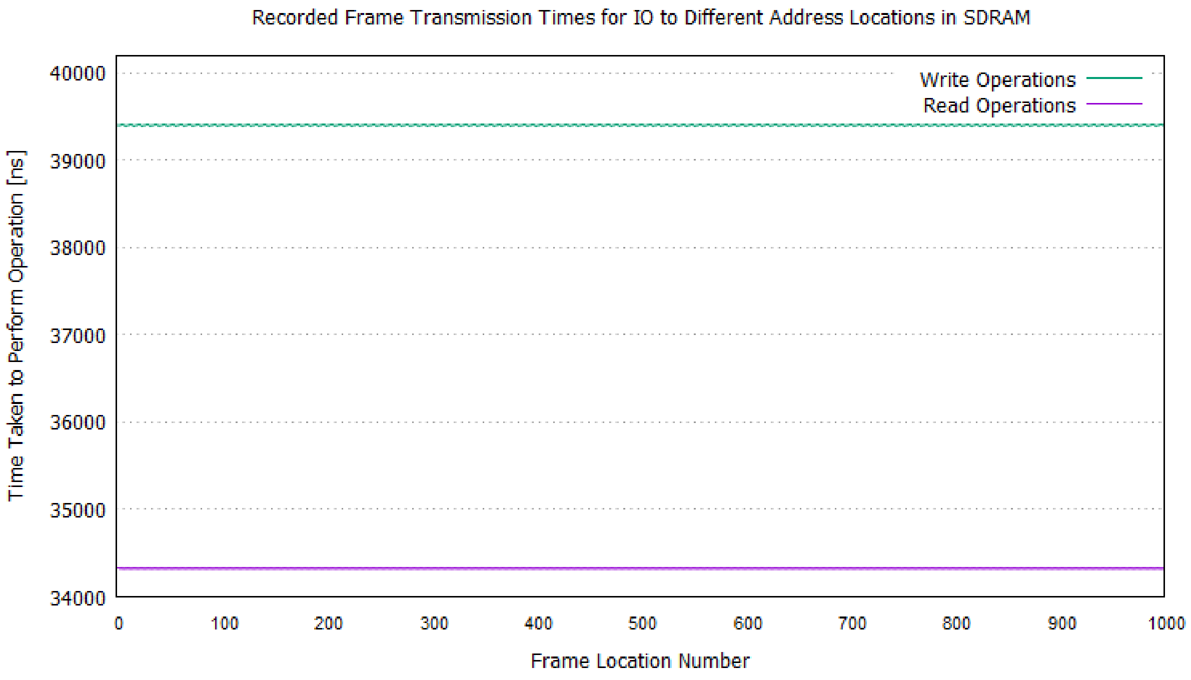


Figure 4.10: Recorded Timings for RAM IO with Frame of Non Varying Lengths

for as well as maintaining communication channels with other application level modules is the responsibility of the delay module.

4.5.1 SDRAM Timings

Before timing constraints could be determined for the delay module, the timings relating to bulk data transfer to and from the SDRAM had to be recorded. Once collected, these timings can be used to determine the minimum delay time the application module should allow. This minimum delay will account for the time taken to write and read a network frame from the SDRAM.

Table 4.1 shows the statical summary of time taken to transfer bulk data to and from the SDRAM. The bulk data transferred contained 1522 bytes of data which is the maximum allowable size of a network frame transmitted under the IEEE 802.3 protocol (Beili, 2007).

For both operations, 1000 iterations of the tests were performed and each data block was written to a different address in SDRAM in a sequential manner. Figure 4.10 describes the times taken to write and read the bulk data for sequential locations in SDRAM. From this figure it is clear that the timing overhead relating to read and write operations is independent of the location in SDRAM addressed.

Considering the results summarised in Table 4.1, the variation recorded can mostly be attributed to the 10ns resolution of the timer used to record the timings and the operations can be considered to only be dependent on frame size. Tests relating to variable frame sizes show a linear relationship between volume of data to be transmitted in 32-bit integers and time taken to perform the operation. Considering the largest frame size that could be delayed, the expected maximum time taken to write and read a single network frame from SDRAM is 0.074ms.

Table 4.2: Statistical Summary for 1000 Frame Transfer Timings in Nanoseconds

	Without Delay Module [ns]	With Delay Module [ns]
MIN	535010	645510
Q1	535010	645520
AVG	535016	645522
Q3	535020	645520
MAX	535020	645940

Accounting for additional overheads such as transferring the network frame from the source module to the delay module and then later from the delay module to the destination module was tested by implementing the delay module in a test system using a source and destination module with controlled input and output. As a baseline, two tests were performed, the first was a transfer between the source and destination modules directly while the second test included the delay module as an intermediate step. A statistical summary for both tests is shown in Table 4.2. Comparing the average time taken for each test, including a delay module with a wait time of 0ms in the frame transmission path resulted in an increase of 0.11ms in transmission delay. As the current intended resolution of the delay module is 1ms, it was concluded that no minimum delay would be enforced on the delay module. The DSL would compensate for the transmission latency by reducing defined delays by 0.11ms and simply not including a delay module of 0ms.

After implementing the delay module into the DSL, transmission speeds were tested for different latency durations. Table 4.3 lists the latencies that were tested along with the average RTT recorded using the ping utility application found on most operating systems.

To evaluate throughput and different latencies, iperf (Tirumala *et al.*, 2005) was used to

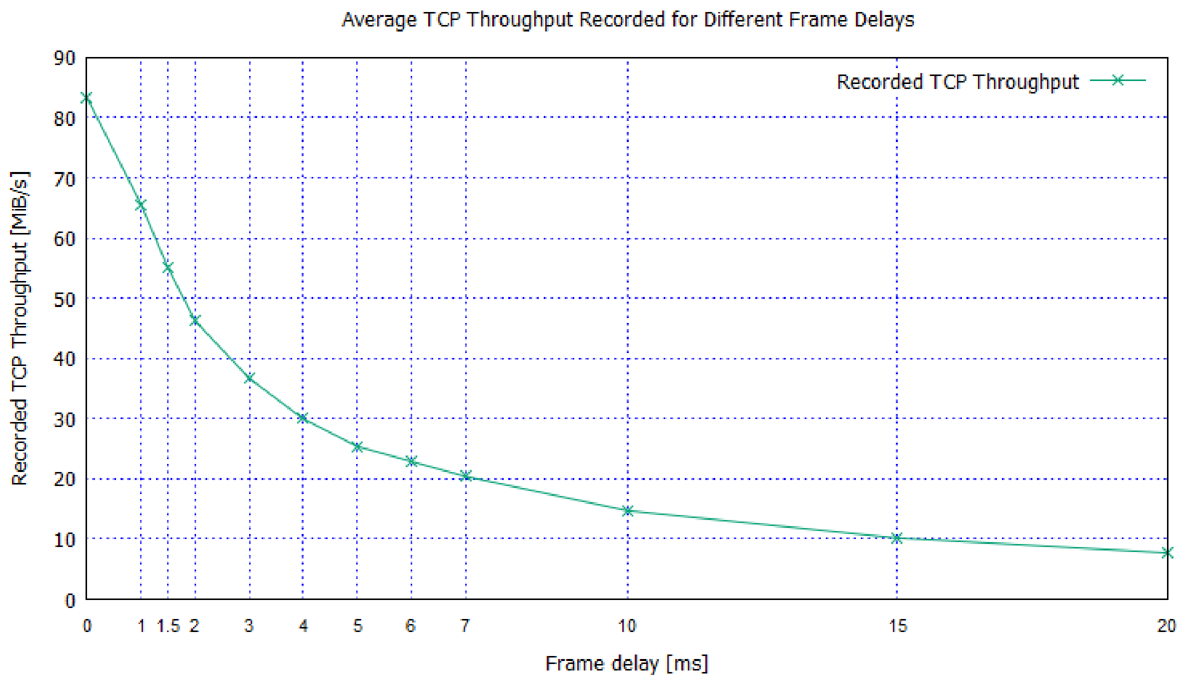


Figure 4.11: Frame Throughput For Varying Latencies

Table 4.3: Recorded RTT Times for Different Delays [20 Iterations Each]

Test No.	Delay [ms]	Average RTT [ms]
1	5	5.2
2	10	10.3
3	20	20.3
4	25	25.2
5	30	30.2
6	50	50.2

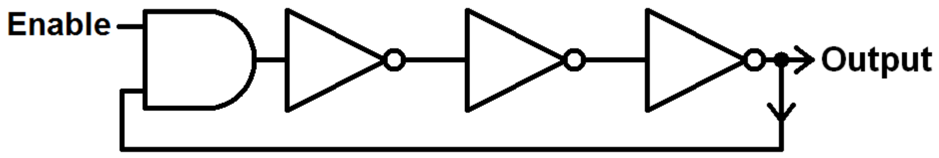


Figure 4.12: Logical Representation of a Ring Oscillator

move a bulk volume of data between end hosts through the functional platform implementing the delay module. Figure 4.11 shows the observed throughput for each latency. As expected, increasing the latency reduces throughput as it directly impacts the maximum TCP throughput as described in Equation (4.3) where $window_r$ is the window size advertised by the receiving host (Jain *et al.*, 2003).

$$Throughput \leq \frac{window_r}{RTT} \tag{4.3}$$

4.6 Random Number Generation

To support the inclusion of entropy when developing applications, the prototype device must be capable of generating random numbers. To facilitate this, a module responsible for random number generation should be developed and made available for the inclusion in user-defined applications.

During an investigation into random number generation, it was found that the XMOS microcontrollers all contain four ring oscillators which are designed to operate independently of each other. Furthermore, these oscillators are sensitive to heat and current draw which can effect oscillation speed (XMOS, 2015a). Ring oscillators are usually implemented in hardware as a chain of logical negations that can be conceptually described by Figure 4.12. A ring oscillator is started by applying a signal to the first logic gate of the chain, the negation of which is fed into the second logic gate. This process is repeated until the signal reaches the last logic gate in the chain. The signal produced by the last logic gate acts as the output signal of the oscillator and is then fed back into the beginning of chain causing the process to repeat. As the number of logic gates in the chain is always odd, the output signal will always be the logical negation of the input signal. If the procedure is allowed to operate continuously, the output signal will oscillate. The period of the oscillation is dependent on the time taken for a signal to propagate through the chain which is affected by the number of logic gates present and external factors such as heat and current draw (Ricketts and McNeill, 2009) . Due to their sensitivity to these external

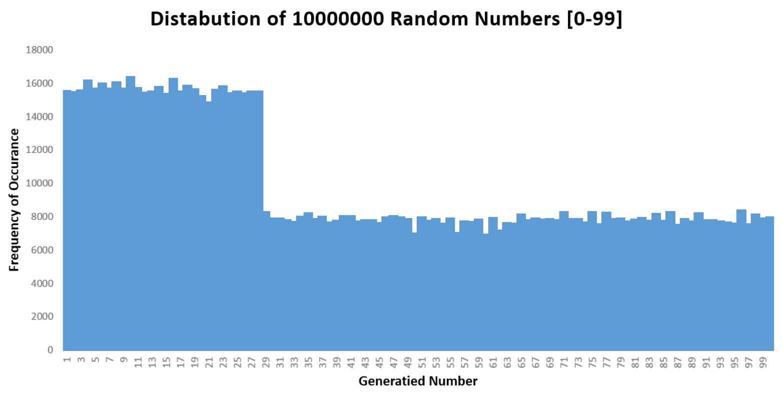


Figure 4.13: Frequency Distribution Showing Bias of Modulo Operation

factors, ring oscillators provide a good source of entropy required for random number generation. As the oscillators run independently of the system clock, they do not require a dedicated thread to operate.

Random number generation was implemented by using the frequency differences between different ring oscillators, the result of which was then combined with a pseudo-random number generator. This was achieved by reading the current values of the four oscillators to be used as parameters in a polynomial which is fed into a CRC check to produce a codeword to act as the generated random number. This operation can be performed by the thread that requires a random number rather then communicating with a dedicated process. The only requirement is that the ring oscillators are initialized before the dependent application begins. The CRC check can be done in a single clock cycle and the only additional overhead is reading of the ring oscillators.

To test the implemented module, 10 million random numbers were generated in the range from 0-99 inclusive. The recorded results are graphed in Figure 4.13 which indicates a significant issue with the generation where some number ranges are favoured more than others. This issue arises from the method used to reduce the 32-bit random number to fit in the specified range. The implemented method is to simply perform the modulo operation on the 32-bit random number and return the remainder as the result. Equation (4.4) depicts a minimal example used to describe how for certain ranges, some values have a higher probability of occurring as there are more situations in which the modulo operation will resolve to them.

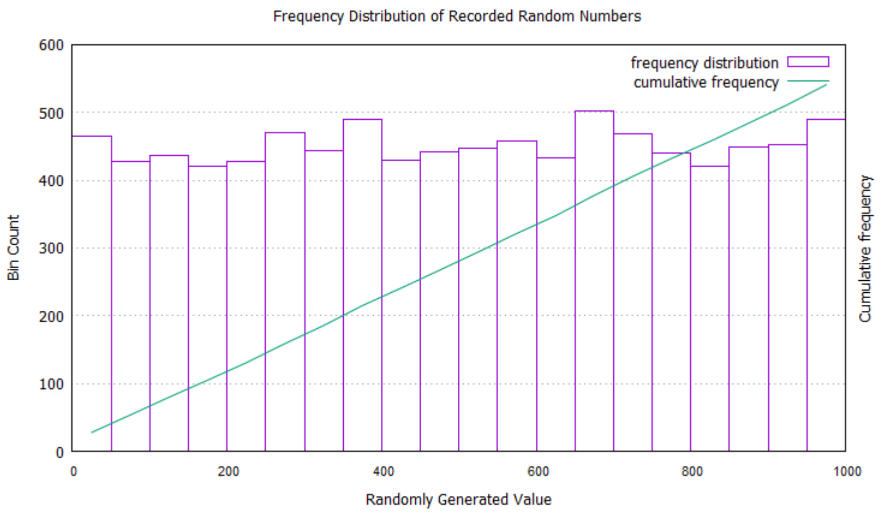


Figure 4.14: Frequency Distribution of Generated Numbers

$$random(2^4) \mod 5 = \begin{cases} 0: & 0,5,10,15 \\ 1: & 1,6,11 \\ 2: & 2,7,12 \\ 3: & 3,8,13 \\ 4: & 4,9,14 \end{cases} \tag{4.4}$$

A proposed solution is to mask the 32-bit number to the least significant bit which will still contain the target range[8] and then repeat the process until the masked value lies within the target range. This does produce a more evenly distributed result which is depicted in Figure 4.14 but at the cost of becoming non-deterministic. The cumulative frequency of the generated numbers was also computed which shows an even distribution in numbers generated within the specified range.

Considering the trade off between using a deterministic and non-deterministic approaches to number generation presented here, both were implemented. The user can then select between using a deterministic but biased generation approach and the non-deterministic but unbiased approach. A final important note is that the deterministic generation is highly recommended when the desired range is of the form 2^n where n is in the range 1 to 32 as these ranges are not affected by the bias seen in the deterministic generation.

Providing the proposed system with tools capable of performing random number generation all allow developed applications to include non-deterministic behaviour. Random

[8]For example, the value 100 would be completely contained in 128 or 2^7.

number generation achieves this by allowing the developer to associate a probability of occurrence with an operation or set of operations within the application.

4.7 User-Defined Modules

User-defined modules are classified into two types according to the output states the module can achieve. The first type discussed is a modifier module which must output all received frames to modules connected to it's output regardless of the logic processed by the module. The second module type is the switch module which can select between two groups of modules connected it's output depending on internal conditional logic. The user-defined modules must still conform to the standard discussed in Section 3.5.1 to ensure compatibility with all modules instantiated within an application.

The intention behind a user-defined module is to provide the structure required to communicate with other modules while allowing the user to develop the majority of the internal logic to meet the requirements of the intended application. Unlike internal modules, a user-defined module is independent of any specific hardware and can operate on any available xCore on the Network Layer of the platform.

4.8 DSL Devlopment

In Section 3.5.2 the requirements relating application development to the scope of this research were highlighted. The first factor to consider was ease of use as application development relating to the proposed platform should be achieved in a user friendly manner without requiring any specific hardware knowledge.

The sensitivity of the internal modules described in Section 4.3 must also be considered. Implementation of these components was performed under the assumption that they would be inaccessible for modification by the user. To confirm conformance of user-defined modules and ensure correct interfacing between these modules and required internal modules, the solution proposed by this research is to develop a Domain Specific Language (DSL).

Development of a DSL would allow the research to design a syntax which is both simple and expressive in the domain of the prototype platform. The semantic notations relating to the DSL could also provide the necessary abstractions away from hardware specific knowledge such as communication between physical components that form part of the platform. Applications developed using the DSL should be easy to read and concise.

Considering the proposed design of applications compatible with the platform, the DSL should be divided into two stages: the declaration stage and the link stage. The declaration stage would be responsible for the design of any user-defined modules which are required as part of the intended application. Following this, the link phase would be responsible for describing the connectivity of both user-defined modules and internal modules. These connections would ultimately describe the flow of network frames through the Network Layer between port nodes.

As the internal modules described in Section 4.3 were implemented in XC, applications written in the DSL must be translated into a language compatible with the XMOS development tool chain. The translated code can then be compiled with the necessary internal modules as first discussed in Section 3.5.3.

The final component that needs to be discussed before example applications can be investigated is the ability to record or log data and receive commands from the Base Layer during application execution. Due to memory limitations of the XMOS platform, all data management and storage will be passed to the Base Layer for handling. To achieve this, it is necessary to create and maintain a connection between the executed application and the Base Layer. As concluded in chapter 3, the implemented interface will use the SPI protocol with the Base Layer acting as the master and the Network Layer acting as the slave.

To incorporate the logging functionality into the DSL, two approaches were considered. The first was to define the logging as a separate component that could be made available to the user with multiple input channels and an output channel. Other modules with characteristics to be logged would then write to this channel and details relating to communication with the base platform would become the responsibility of the link exclusively.

The second approach was to allow both the modifier and switch modules the ability to define when a characteristic should be logged. In this case, each defined characteristic would have two attributes associated with it, the value to be compared or updated, and the condition on which a record relating to that characteristic should be logged to the base.

After testing both approaches in a series of applications, it was expeditiously discovered that defining a separate component to act as a logger would not improve the functionality

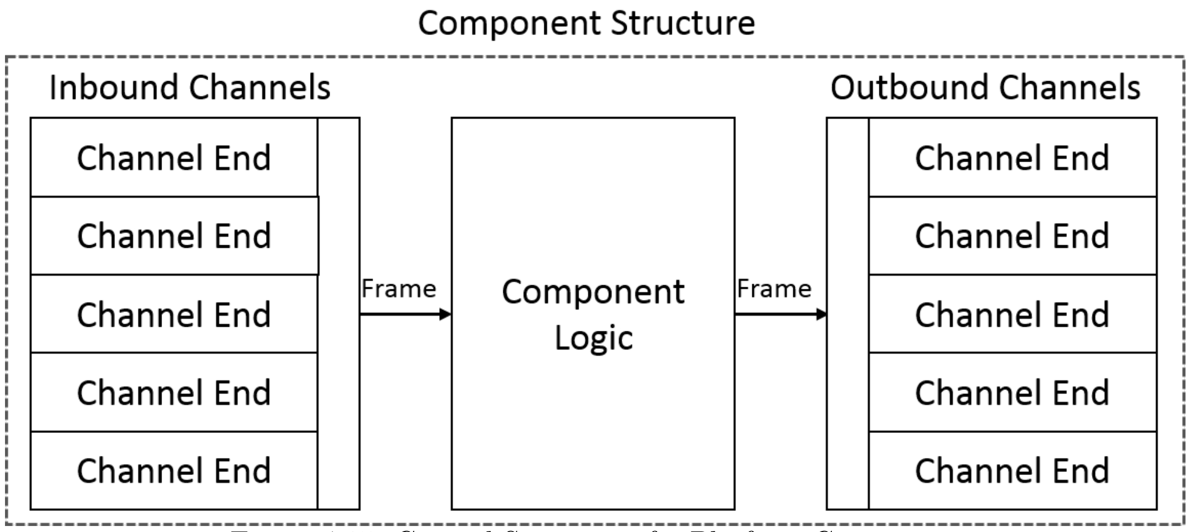


Figure 4.15: General Structure of a Platform Component

of the code for the user as the logging does not require any user interaction to define and set up. The approach taken by this research was to extend the functionality of current components to allow for the logging of characteristics. Should a component of the application then require logging, a background process would be created to handle the communication between the platform and the base.

4.8.1 Defining a Module

To simplify the translator and enforce compatibility, all user-defined modules should conform to a standard as discussed in Section 3.5.1. Figure 4.15 shows the input and output constructs of a module. These constructs are set up to allow for multiple input and output channels to be handled so that each module can connect to multiple other modules. A condition that must be applied to every module is that each channel, both inbound and outbound, must be connected to another module. Furthermore, a module can only connect to another module once across both inbound and outbound channels as duplication could result in circular connections as highlighted in Section 3.5.2.

4.8.2 Example Target Applications

To better understand how the platform will function once developed, a series of simple target applications were developed and investigated. These applications are minimal as they focus on showing how the platform could be used.

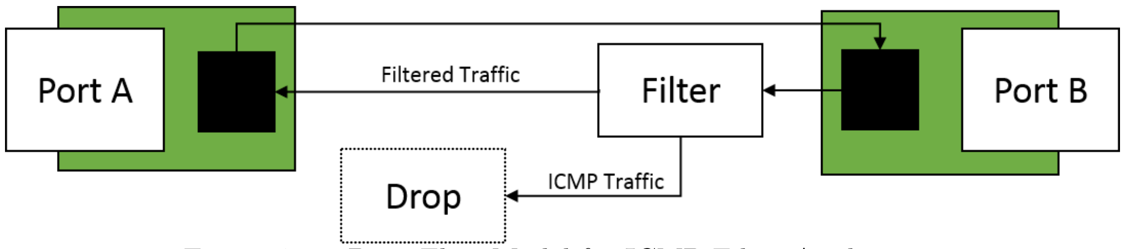


Figure 4.16: Data Flow Model for ICMP Filter Application

Application Example: ICMP Blocker

The first example application is an ICMP blocker. The goal of this application is to allow traffic to pass freely from port A on the platform to port B. Traffic travelling in the opposite direction however will be monitored and all ICMP requests will be dropped. Figure 4.16 depicts the traffic flow model of this application where the directed lines represent the direction of data flow.

This application is made up of three modules, Port A, Port B, and the Filter. In XC, each of these components represents a concurrent process or group of processes that are executed in parallel on the XMOS architecture. For this application, both Port A and Port B are internal port modules which has previously been developed and cannot be modified by the user. The filter module however is the responsibility of the user to develop as its functionality is dependent on the application requirements. In this application, the filter link will contain a single condition that it will check against every packet it receives on its input channels. The condition to check is if the IP layer protocol associated with the frame is of type ICMP. To perform this check it is first necessary to determine if the frame received is being transmitted under the IPv4 protocol. If the correct underlying protocol is being used and the ICMP protocol tag is present, the packet is dropped immediately and the system returns to the idle state (waiting for incoming traffic). Should the check fail, the packet is then broadcast to all outbound channels attached to the filter link. For this application, the filter only has one inbound channel and one outbound channel.

From this example it is clear that the only component of the application that needs to be defined by the user is the filter link. Furthermore, only the characteristics of the frame to be filtered and how the channels between links are connected require user specification. The rest of the application such as how frames are passed over channels or how links manage multiple inbound and outbound connections is not a concern of the user and should be abstracted away by the DSL.

The second point highlighted by this example is the evaluation of frame related charac-

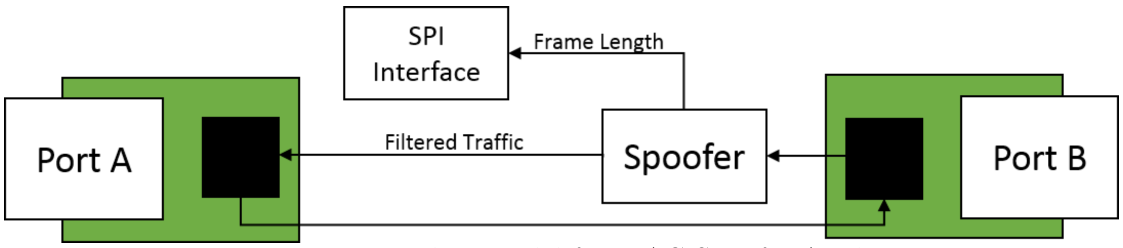


Figure 4.17: Data Flow Model for MAC Spoofer Application

teristics. To check a characteristic relating to a specific protocol, it is first necessary to confirm that the compatible lower level protocol is also present in the frame. This additional check is not a concern of the user and the interpreter associated with the DSL should be responsible for insuring that this check is performed.

Application Example: MAC Spoofer

A second example application that could be executed on this platform is a MAC spoofer which also logs the length of each frame sent. In this application all traffic received from port A will be passed unaltered through to port B. However, all traffic from port B will have the source mac address altered to some defined value before being passed through to port A. Figure 4.17 shows how the frames will traverse the application between the two ports.

As in the previous example, the application specifies two Port modules which will handle communication between the application and the physical network medium. In addition to these modules, this example includes a modifier module. The modifier composition is very similar to that of a filter except all traffic received from an inbound channel will be broadcast to all outbound channels in a modifier as the modifier link cannot drop frames. The purpose of a modifier is to alter attributes of each frame received as specified by the user. This is done by replacing the value in the frame that represents the characteristic to be modified with the value specified by the user.

Unlike the previous example however, this application is also required to log the length of each frame sent to the base platform. To do this, an additional background process must be created which can maintain a connection to the Base Layer for data transmission.

To allow the modifier component to communicate with this background process and send data, it is necessary to specify an additional channel type that the component can communicate through. Figure 4.18 shows the updated structure of a component object which now includes an optional channel for communicating with the SPI process.

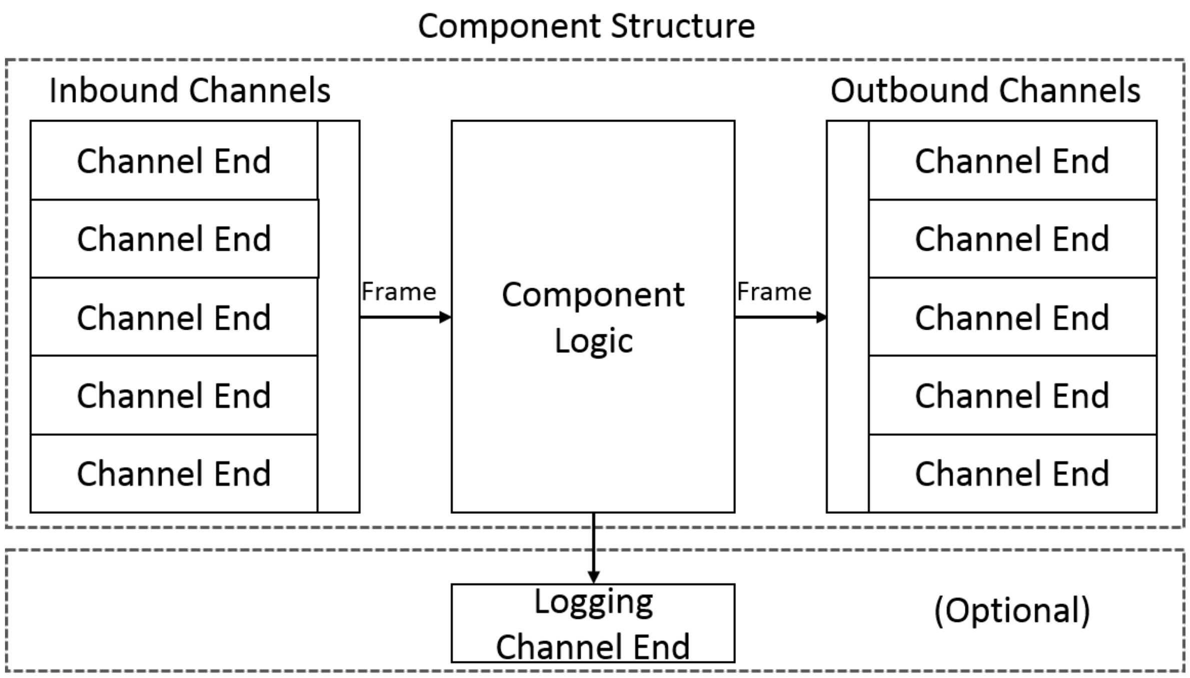


Figure 4.18: Updated Structure of a Platform Component

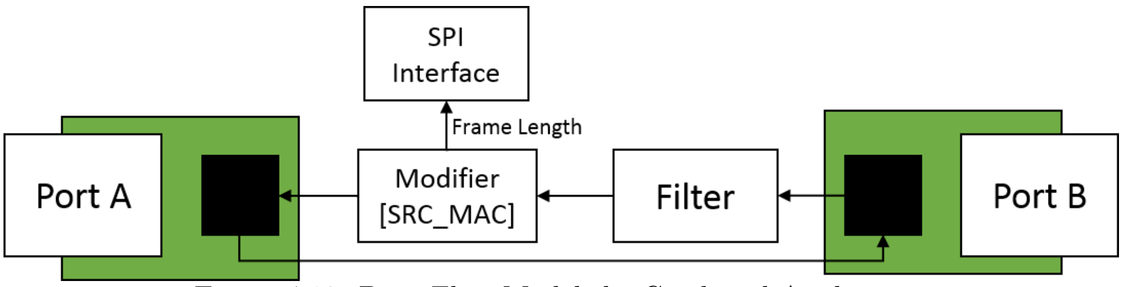


Figure 4.19: Data Flow Model the Combined Application

Application Example: Combined Example

A third example is to combine the functionality of the previous two examples into a single application. Figure 4.19 depicts the data flow of this new application with a filter component and modifier links chained together. For this the component indicating that ICMP traffic is dropped has been removed so as to not imply that channel communication is required to to perform this action.

As described the previous two application examples, all network traffic traversing from port A to port B will pass through the application unaltered. Traffic traversing from port B to port A however, will be be altered according to the application chain depicted in Figure 4.19. The first module in this chain depicts a filter taken from the application example described in Section 4.8.2. This filter is responsible for removing all ICMP related frames from the network traffic traversing from port B to port A. The filtered traffic is then processed by the second module in the application chain, the MAC spoofer. This module, discussed in Section 4.8.2, will alter the source MAC address of all filtered network traffic to a previously determined value. A secondary function of this module is to log the frame length of every modified frame to the Base Layer via the SPI interface.

4.8.3 Semantic Notations

After considering the issues highlighted by the example applications and concerns raised in Section 3.5.2, a basic syntax that represents each component from the perspective of the user was developed. This notation focused on simplicity by removing all factors relating to the application that should not be a concern of the user. The syntax was designed to operate in two distinct phases: the declaration phase and the linking phase.

During the declaration phase, modules that are to form part of the application must be declared and defined. Due to the limited number of module structures available, there is very little syntax related to this stage of the application design. Once the module type has been declared, it must be given a name so that it can be addressed later during the linking phase. A list of the characteristics associated with the module would then be applied to every frame received. The syntax used to develop a list was designed to be independent of the module type to allow for compatibility.

These attributes are defined locations in the data collected off the network port and are effectively indexes into an array which represents the network frame. Associated with each characteristic is a value that the frame attribute should be compared against. To show this syntax quantitatively , the declaration phase of the first application shown in figure 4.16 is presented in Listing 4.2.

```
Switch ICMP_Filter
  {
    on Condition(ethertype == ICMP): FAIL;
    Default : PASS;
  }
```

Listing 4.2: Declaration Phase for Application Example: ICMP Blocker

The syntax presented in this listing declares the defined module to be of type Switch. A switch type module requires two output states indicated as PASS and FAIL. Each statement included in a switch module must resolve into a boolean expression which is associated with one of the two proposed output states. Finally, all switch modules require that the last statement listed is `Default` which will be the action performed by the module if all other statements resolve to false.

Following the declaration phase, the remainder of the script is reserved for the linking phase. In this phase, the modules defined by the user are used to create an application chain. This chain can also include modules not defined by the user provided they form part of the internal module group.

Each chain must end with a terminal module which acts as the sink of the application chain. For the proposed revision of this research, only the internal port modules may act as a sink. The linking phase following the declaration phase in Listing 4.2 is presented in Listing 4.3.

```
Port_A --> Port_B
Port_B --> ICMP_Filter<on PASS> --> Port_A
```

Listing 4.3: Linking Phase for Application Example: ICMP Blocker

The '–>' syntax indicates a channel between the two links within the application. All switch modules need an additional parameter of either ¡on PASS¿ or ¡on FAIL¿ which indicate whether the chain should be linked to the PASS output group or the FAIL output group of the switch module.

A more advanced application to evaluate is the example discussed in Figure 4.19. As this is an application that can be run on the platform, the syntax must be capable of representing it. In this case, the length characteristic of each frame must be logged which is achieved by including the LOG statement. This indicates the length characteristic of the current frame should be written to the log interface. The completed code in the proposed syntax for the third example application is presented in Listing 4.4.

Table 4.4: Frame level attributes available in the DSL

Characteristic	Value	Available to Modifier
dst_mac	array of hex	No
src_mac	array of hex	No
length	unsigned integer	Yes
ethertype	unsigned integer	Yes
VLAN_tag	unsigned integer	Yes
frame_data	array of hex	Yes

```
Switch ICMP_Filter
  {
    ipv4_ethertype = ICMP;
  }

Modifier Spoofer
  {
    [LOG <: length] /*Log the length of the frame*/
    src_mac = x00:x3D:x45:xA7:xE5:xFF;
  }

Port_A —> Port_B;
Port_B —> ICMP_Filter —> Spoofer —> Port_A;
```

Listing 4.4: Complete Application example: Combined Example

To allow the user to easily retrieve or modify frame characteristics, the DSL will support a selection of attribute names. The user can then interact with the frame characteristics by using the associated name rather then explicitly selecting the relevant data. For example, should the user require that the frame length attribute is incremented by ten, they can achieve this by simply adding the statement [`length = length + 10`] to a modifier type module. Any attribute appearing on the right of a statement will be resolved into its numeric value as it currently appears in the frame. Table 4.4 shows an initial revision of the exposed characteristics purely at the frame level. The third column of Table 4.4 indicates whether a modifier link can use the characteristic as part of a numeric operation.

Additional characteristics that the DSL will include are dependent on the Ethertype field included in the frame. Table 4.5 presents a sample of the additional characteristics that will be supported by the first revision of the DSL. The fourth column in the table indicates the Ethernet type that is required for the characteristics to be relevant. Later extensions to this research will include extending the availability of user modifiable characteristics

Table 4.5: Ethertype specific attributes available in the DSL

Characteristic	Value	Available to Modifier	Required Ethertype
ipv4_src_ip	array of char	No	IPv4
ipv4_dst_ip	array of char	No	IPv4
ipv4_ttl	unsigned integer	Yes	IPv4
ipv4_version	unsigned integer	Yes	IPv4
ipv4_protocol	unsigned integer	Yes	IPv4
ipv4_flags	unsigned integer	Yes	IPv4

to include attributes associated with different protocols. The user can bypass the requirements of selecting a characteristic and address the data directly using the [`frame_data`] keyword.

When adding conditions based on IP level Ethernet types, if a frame does not form part of the appropriate underlying Ethernet type, the condition will be ignored entirely. It is recommended that this situation be avoided by linking a filter between the modifier and the frame source to ensure that only frames of a particular Ethernet type reach the modifier.

After describing a functional syntax capable of representing minimal applications in a concise manner, the prototype language was named Link with each application being comprised of interconnected node.

4.9 DSL Interpreter

For the proposed language to be functional, a translator capable of accepting valid application scripts written in Link and producing equivalent XC source was necessary. Rather than develop such an interpreter from first principles, it was proposed that a compiler generator be used to simplify the process.

Due to the structure of the application examples, Link can be broken into the declaration and linking phase with the requirement that the declaration phase occurs first in the produced script. By enforcing this requirement, it is possible to ensure that any node used in the linking phase must have already been defined either by being an internal module or by being previously defined during the declaration phase. The resulting language can be modelled as an L-attributed grammar which is formally defined by Mauer Wilhelm (Wilhelm *et al.*, 2013) as:

An attributed grammar AG is L-attributed, if for every production of the grammar p : $X_0- > X_1...X_p$, all inherited attributes a $\in$ Inh(Xi) for i $\in$ $\{1, ..., p\}$ only depend on inherited attributes of Xj for $1 \leq j < i$.

This allows for a top down parsing of the source code in a single pass by the translator while maintaining a list of known variables. As a result, generating a translator can be performed using a compiler generator such as Coco/R.

Coco/R was developed in Obreon in 1989 and is a derivative of a table driven parser called Coco which in turn was developed in 1983 (Terry, 2005). Coco/R accepts an attributed grammar and generates a scanner and parser designed for that language. The developer is also required to supply some additional support components such as a code generator or table handler to call the generated parser.

A requirement of Coco/R is that the productions of the DSL grammar be written in Extended Backus-Naur Form (EBNF) form. John Backus, a member of ALGOL[9], developed a notation for describing the syntax of programming languages. Impressed by the notation, Peter Naur used this meta language to completely describe the ALGOL language syntax, thus proving the significance of the meta language. As a result the syntax became known as Backus-Naur Form (BNF) (Patti, 2005).

This meta language was then extended by adding operations to allow for conditions and repetition within the production rules of the language grammar (ISO, 1996). Productions in EBNF are equivalent to productions written in BNF but can be written in a more compact form by not having to use recursion to allow for a sequence of symbols to repeat.

4.9.1 Generated Code

The goal of Link is to be used in conjunction with a high-level translator responsible for converting the application script into into source code compatible with the XMOS development toolchain. The resulting code can then be compiled with additional libraries to produce the application image. The code is structured such that each node declared in the application chain will be reserved for a separate logical processor on the platform. Because of this, the number of modules that an application can include is dependent on the number of logical processors available on the Ethernet platform. For an XS1-L16, up to 16 modules can be implemented. It is important to consider that the number of channels that can be declared in an application will also be limited as it is a finite resource of the device.

[9]ALGOrithmic Language, an early high-level programming language

The designed translator works in to two stages, first the script is parsed and once complete, the source code is generated. During the parsing phase of translation, module objects are created for each user-defined module. Associated with these objects are lists indicating the attached upstream and downstream modules. All internal modules referred to in the script are also recorded so that the translator can include the initialisation code for them at a later stage. A counter of the number of logical processors is kept to ensure that the implemented Network Layer supports enough logical processors to run the application.

Once the script has been parsed, checks are performed on all generated objects to ensure every module present in the linking stage of the script has been properly defined. All user-defined modules are renamed to ensure they do not conflict with the names of internal modules which could case the resulting application to fail later compilation. Semantic checks are also performed during parsing and any errors detected are announced along with the line on which they were detected. No attempt at error recovery is done and the translator will simply terminate at that point without producing any output file. Once the source is successfully parsed, generation of the application code in XC can begin.

As indicated in Figure 4.18, a module is required to support multiple inbound channels which will need to be monitored for incoming frame data. To support this, XC offers a new statement type called the `select` statement. In a programming context, the `select` statement works in a similar fashion to a `switch` statement common in C and C++ in that it consists of a series of cases which which are evaluated upon entering the statement block. The difference between the two statements arises from the context of each case. Instead of comparing some input variable against a predetermined value, the cases of a `select` statement are associated with events such as channels, ports, or timers. The first case in which an event occurs is the case the select statement transfers control to (Watt, 2009). `select` Statements allow threads written for the XS1 architecture to respond to multiple inputs without having to revert to polling. Appendix B.1 shows an example of this functionality where two threads are executed in parallel. The first thread sequentially sends a character over four different ports which the second thread prints out. If the second thread does not receive anything from the first thread for a specified duration, it times out.

To improve readability and reduce repetition in the generated code for `select` statements, XC provides an optional extension to the `select` statement called a `replicator`. The code listing in appendix B.2 is logically equivalent to the code listing in appendix B.1 except now the second thread operates using a `replicator` rather then explicitly specifying each case.

4.9.2 Module Control

A proposed feature of the this research would be to allow the user to interact with an application by enabling or disabling user-defined modules during execution. The approach taken by this research to achieve this is to provide a listing of user-defined modules associated with the active application on the user interface identified in Section 3.5.3 and further discussed in Section 4.10. Associated with each element should be a check box to allow the user to enable or disable a specific module.

To facilitate this, the interpreter is required too produce a secondary file listing all user-defined nodes and an identification number associated with each. The secondary file produced by the translator should then become available to the web interface and be used too populate a control tab for user interaction.

The purpose of the identification number is to allow the executing application to distinguish which user-defined module is to be enabled or disabled. As communication of the enable commands must operate through the internal module responsible for logging, the identification numbers associated with each module would simply be the channel number associated with that node in the internal logging module.

The consequence of this approach however is that the internal logging module becomes compulsory and must be included in all applications, reducing the number of available logical processors for user-defined modules by two. Given the potential functionality this could add to the executing applications by allowing user control without reloading the target application, this cost was considered acceptable.

4.9.3 Supporting Function Design

During an evaluation of semantic notations in Section 4.8.3, a series of frame related characteristics were tabulated. Allowing the user to address these and other protocol specific characteristics is essential in achieving the research goal of providing frame manipulation.

Tables 4.4 and 4.5 provide an initial listing of characteristics relating to the frame and IP protocol which applications developed in Link should have access to. As the language is extended to support more protocols, the list of available characteristics should increase accordingly.

From the perspective of the Link grammar, incorporating such extensions can be performed by simply including the new attribute names. Every modification to the grammar however, must be reflected in the accompanying translator. For the current revision of the translator, each characteristic resolves into one of two function calls depending on whether the characteristic is being retrieved or modified. For the generated code to be compiled, each function being called must have been previously defined in a supporting library.

The characteristics currently defined in the supporting library are listed in appendix B.4 which can be extended during future development of the translator. Each function listed is responsible for insuring all relevant underlying protocols relating to the target characteristic are present before modifying for returning the characteristic value.

4.9.4 Compilation and Uploading

Once the translator produces the application source file, it is included as part of an existing application project. The project is then compiled with relevant libraries into a binary using the XMOS XCC compiler[10]. Included in the project is a configuration file for the Network Layer of the platform. This file tells the development tools how the target platform is structured.

Once the binary is generated, the `xflash`[11] application is called to convert the binary into an image file which needs to be uploaded to the Network Layer flash as discussed in Section 4.4. This is done via an SPI connection to the flash device dedicated to storing the application code. Once flashed, the Network Layer of the platform is reset and the images are loaded from flash onto the device.

4.10 Platform User Interface

To accommodate user interaction with the device, a user interface is provided via the Base Layer. The user interface is presented as a web application that serves an interactive web page to provide monitoring of the Network Layer and the current application being executed. The webserver is a simple Apache server that uses CGI scripts to populate served pages via AJAX calls. JavaScript is then used on the served pages to render the information recorded by the platform to the user as a series of charts. This data is periodicity updated as information is recorded from the Network Layer.

[10]https://goo.gl/22Bie5
[11]https://goo.gl/WVV8aF

Currently, monitoring is done by default on all enabled ports of the device. Each active port records basic information on the frames received or transmitted and pushes this meta information to the Base Layer via the logging interface. For the current implementation, this information is limited to volume of bytes, number of frames, and a count of ICMP and IP frames handled. The information recorded in this manner is done as a proof of concept with more informative data to be recorded in later revisions of the platform.

This recorded data is periodically received by the Base Layer and stored in a database. This database exists for the current application and is only active for the application lifespan. When the application is restarted or a new application is run, a new database file is created to associate with it. Old database files are stored for later retrieval.

To make the recorded information available to the user during application runtime, a set of CGI scripts are run periodically on the database. These scripts retrieve the last window of records from the database and transmit the processed data via the webserver to all active client interfaces. The window size is currently set to 100 entries which are processed into coordinate values and ratios to be rendered by the client interface as shown in Appendix C.

An additional function of the client interface is to allow the user to upload their application scripts onto the device. Once uploaded, the Base Layer compiles the script and uploads it to the Network Layer as discussed in Section 4.9.4.

4.11 Summary

Prior to implementation, Chapter 4 focused on the analysis of transmission speeds through the architecture chosen for the Network Layer. From this investigation, it was concluded that the XMOS architecture was a suitable candidate for implementation though it was recommended that only the XMOS Fast Links be used for intertile communication. The internal modules described in Chapter 3 were implemented and tested before further application examples were investigated which utilise these internal modules. The findings from the application examples were then used to design a DSL called Link which could be used to represent each application in a simple manner.

Following the language design, an interpreter was developed for translating applications written in Link into source code compatible with the XMOS architecture. The user interface and supporting applications relating to the Base Layer were also implemented with the discussion focusing on user interaction and the application components required to facilitate them.

5 Hardware Design

5.1 Introduction

In Chapter 4, the internal software modules required to support frame manipulation were implemented and the format of general modules was covered. The scripting language Link was designed to aid in the generation of platform compatible software which includes custom components supplied by the user. The supporting software required to interface with the Network Layer was implemented to operate on both the Intel Edison and Raspberry Pi, each of which included an SPI interface for programming the platform flash.

This chapter presents the design and fabrication of a dedicated hardware platform to act as the underlying hardware for the Network Layer of the system. The initial stage of Section 5.2 focuses on the design of a stable power supply for the platform while the second stage covers general design characteristics related to board development such as acceptable PCB track lengths and capacitor placement. Section 5.3 focuses on the fabrication and testing of a prototype board containing a subset of the hardware expected to be supported by the final device. This preliminary board is intended to test the operation of the on-board power supply and designs for the network interfaces.

Section 5.4 uses conclusions drawn from previous sections for the design and fabrication of the dedicated hardware for the network platform of the system. Section 5.4.2 reviews the resulting device, identifying all errata detected. Actions taken to remedy the errors detected are also discussed. Modifications to the DSL, allowing for the use of the new board in application development are discussed in Section 5.4.3 before the chapter concludes with a summary in Section 5.5.

5.2 Hardware Design Theory

In Section 3.3 the XMOS XS1 architecture was selected to act as the underlying hardware of the Network Layer associated with the platform. As stated in Section 1.3, an intended outcome of the research is to fabricate dedicated hardware on which the Network Layer of the platform can operate. Owing to the complexity of the intended board it was decided that an intermediate prototype should first be fabricated to test component and subsystem design, and highlight any issues that could impede the functionality of the final hardware configuration.

A significant component of the proposed device was the power supply as the XMOS microcontroller requires both 1v and 3v3 to operate. Furthermore, the XMOS microcontroller requires proper sequencing of the power regulators and reset line to ensure correct booting procedure (XMOS, 2015c). To account for these requirements, a detailed evaluation of a suitable power supply is required followed by an analysis of some transient effects which should be considered to ensure smooth current distribution throughout the designed device.

5.2.1 Power Supply Design

To achieve the output voltage V_{out} required for the prototype platform from a given input voltage V_{in}, a conversion circuit is required. Two common approaches to conversion are linear regulation and switched mode conversion.

5.2.2 Linear Regulation

Linear regulation can only produce an output voltage where $V_{out} < V_{in}$ as it operates in a fashion similar to a potential divider shown in Figure 5.1, varying the resistance Z_1

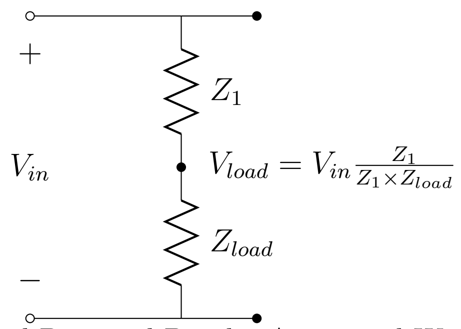


Figure 5.1: Theoretical Potential Divider Associated With a Linear Regulator

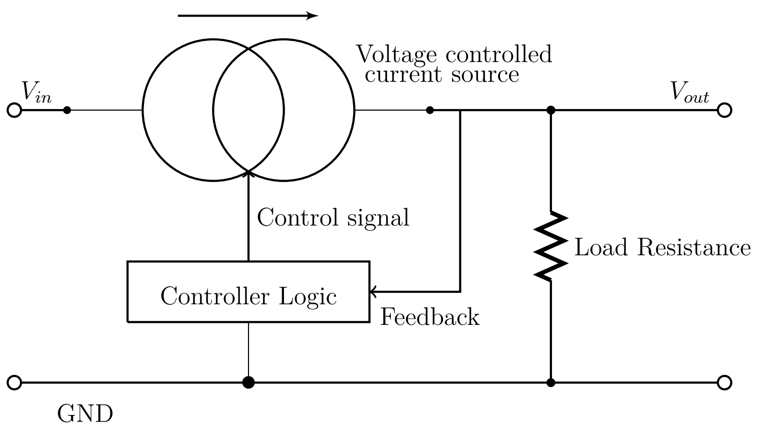


Figure 5.2: Function Diagram of a Linear Regulator (Simpson, 2011)

to maintain a constant V_{out}. Differences between V_{out} and V_{in} are then dissipated across Z_1, usually as heat (Erickson and Maksimovic, 2001, ON Semiconductor, 2014). As more current is required, the voltage drop across the load resistance Z_{load} will begin to change. The series resistance Z_1 must then change accordingly to maintain the correct voltage drop across Z_{load} (Wens and Steyaert, 2011).

An alternative representation of a linear regulator is depicted in Figure 5.2 which describes a basic implementation consisting of two components: a voltage controlled current source, and control circuitry attached to a feedback loop. The voltage controlled current source is responsible for presenting a fixed voltage at the output terminal of the linear regulator. To achieve this, the control circuitry uses a feedback loop and error amplifier to monitor the voltage on the output terminal and adjust the current source to account for variations in current draw and input fluctuations (Zhang, 2013). The error amplifier is commonly composed of an operational amplifier along with some reference voltage. The operational amplifier will then exaggerate the difference between the potential divider output and the reference voltage, amplifying the difference. The control circuitry uses this exaggerated result to determine how the voltage controlled current source should react to maintain V_{out} at the intended voltage. A common characteristic of linear regulators is the time taken to react to a change in the output voltage. These variations are usually owing to

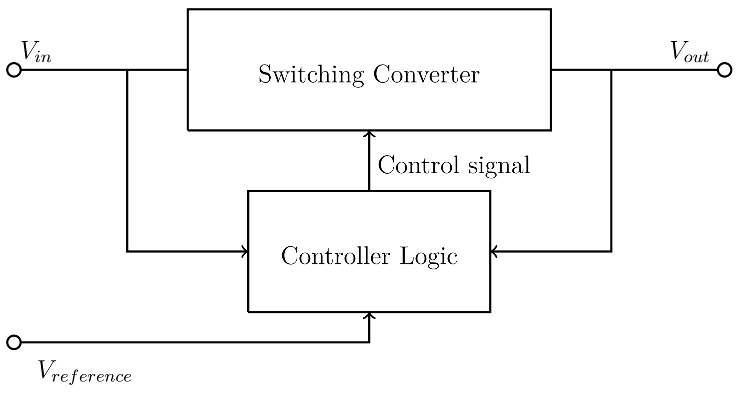


Figure 5.3: Function Diagram of a Switching Converter (Erickson and Maksimovic, 2001)

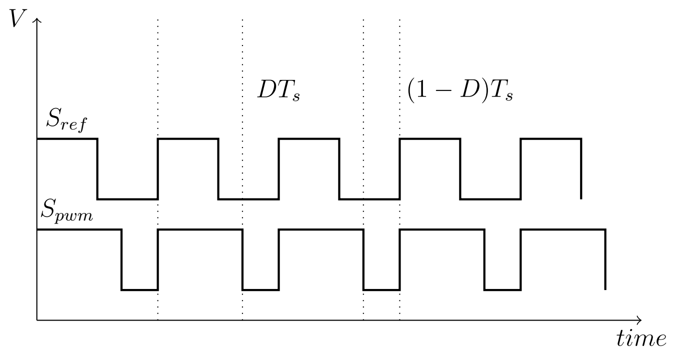


Figure 5.4: Switching Controller Output Voltage.

a sudden increase or decrease in current draw or a variation on the input voltage. This delay time is called the transient response and describes the time taken for the linear regulator to return to a stable state in response to a disruption (Simpson, 2011).

Owing to the power factor of linear regulators, their efficiency will drop according to the difference between the input and output voltages, resulting in an efficiency of approximately 20-60% (Rashid, 2011). This inefficiency results in excess heat being generated by the circuit which needs to be extracted to maintain a stable ambient temperature for neighbouring hardware. A significant advantage of linear regulators when compared to switched mode conversion circuits is the little to no electrical noise generated by the regulator (Rashid, 2011). Furthermore linear regulator circuits are relativity cheap and simple, requiring little or no additional circuitry to operate.

5.2.3 Switched Mode Conversion

Switched mode conversion is a procedure involving a switching converter which is used to produce a controlled output potential V_{out} from a given input V_{in} (Erickson and Maksimovic, 2001). Figure 5.3 describes a basic switching converter regulating an output V_{out}

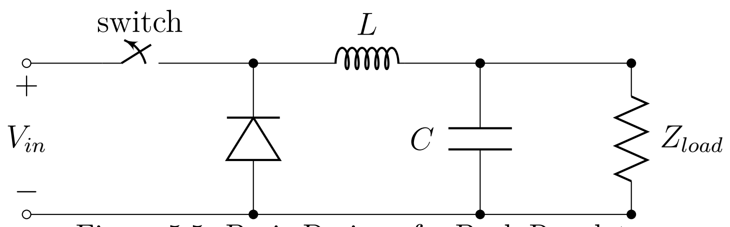


Figure 5.5: Basic Design of a Buck Regulator

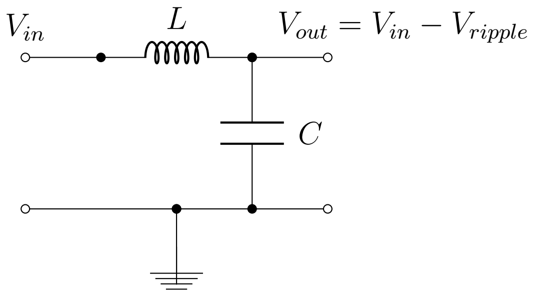


Figure 5.6: An LC Low Pass Filter

as defined by a control signal. Controller logic commonly associated with a switching converter allows for the monitoring of both input and output terminals. This additional logic enables the switching converter to react to voltage fluctuations so as to maintain a constant output voltage according to some reference.

The operation of a switching power supply is dependent on the topology of the switching converter. A common topology is the Buck regulator which can produce an output potential V_{out} from a given potential V_{in} provided $V_{out} < V_{in}$ (Kazimierczuk, 2008). A basic Buck regulator topology, depicted in Figure 5.5, operates by continuously switching between two states: on, when the input terminal is connected to the inductor; and off, then the input terminal is disconnected from the inductor(Erickson and Maksimovic, 2001, Texas Instruments, 2012). Pulse width modulation is used to vary the duration D the switch remains in each state for a given period T_s (Kazimierczuk, 2008) where D is then the duty cycle of the switching converter. Figure 5.4 describes the resulting waveform generated on the output terminal where D is the duty cycle and T_s is the switching period (Erickson and Maksimovic, 2001). The DC component of this generated waveform is given by (5.1) (Erickson, 2001):

$$V_{out} = \frac{1}{T_s}\int_0^{T_s} v_s(t)\,\mathrm{d}t = DV_{in} \text{ when: } I_c = I_{out} \tag{5.1}$$

It is important to note that, owing to imperfections in switching elements of the switching converter, a ripple voltage is also generated which results in an actual output voltage of (5.2)(Erickson and Maksimovic, 2001).

$$v(t) = V + v_{ripple}(t) \tag{5.2}$$

To account for this, a low pass filter such as in Figure 5.6 must be inserted between the Switching element and the output terminal to reject the high frequency ripple generated by the switching elements (Erickson, 2001). The filter however, will also include imperfections and as a result, not all harmonics are cleaned from the signal resulting in a possible output ripple regardless (Kursun *et al.*, 2003). If the induced ripple is significantly small, it can be neglected and the output terminal is approximated to be just the DC component. This approximation is referred to as small-ripple approximation or linear-ripple approximation (Erickson and Maksimovic, 2001).

A consequence of the low pass filter is the effect it has on the output current supplied by the switching regulator. The inductor will filter out the high frequency harmonics present in the output voltage but in doing so, also acts as a current source which adds a ripple current I_{ripple} to the regulator output. To minimise I_{ripple}, care must be taken to match the inductor to the intended output voltage. The selected inductance should induce a ripple current that is less than 20% of the average current supplied (Vaucourt, 2004). The expected ripple current I_{ripple} can be calculated using (5.3) (Richtek, 2010) which relates the ripple current to the switching frequency f and inductance L value of the low pass filter.

$$I_{ripple} = \frac{V_{out}}{f \times L} \times \left(1 - \frac{V_{out}}{V_{in}}\right) \tag{5.3}$$

Minimising the ripple current will help reduce power losses owing to ESR of capacitors which is discussed shortly. When the current demand of the power regulator fluctuates suddenly, the transient response time of the error amplifier used to monitor the output voltage must be minimised to allow the switching regulator to react accordingly. Should the transient response time be too large, current stored in both local and global board capacitance must be able to supply the increased demand in current until the regulator can respond.

Selecting a smaller inductance for the low pass filter reduces this transient response time, allowing the buck converter to react sooner to current fluctuations (Vaucourt, 2004). The actual inductor should be selected to have a saturation point well above the output current the buck converter can provide (Richtek, 2010). Reaching the saturation point of a ferrite core inductor will cause the inductance of the core to drop abruptly, changing the

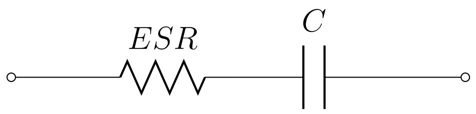


Figure 5.7: Representation of a Real World Capacitor.

characteristics of the low pass filter.

Proper selection of capacitance is also important for optimal operation of the converter. Increasing the capacitance value will reduce the voltage ripple resulting in a smoother output voltage (Zhang, 2013). Increasing the output capacitance however will also increase the transient time taken by the error amplifier to react to a sudden change in load current demand (ON Semiconductor, 2013).

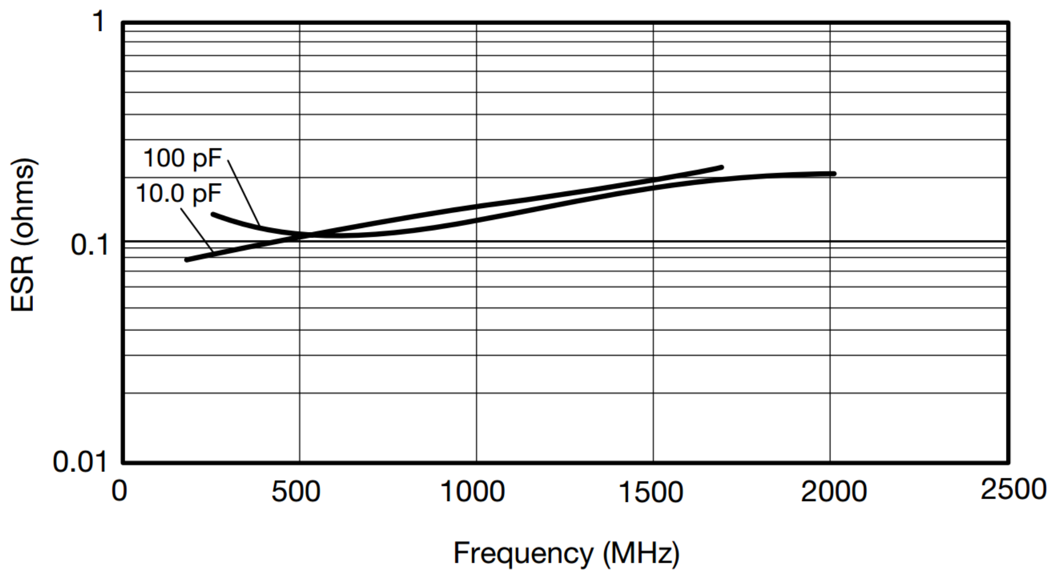
Capacitors also include a resistance owing to imperfections in the physical component which can magnify the voltage ripple V_{ripple}. This resistance varies with frequency and is referred to as the Equivalent Series Resistance (ESR) which represents the total lossiness of the capacitor during current fluctuations (QuadTech, 2003). This resistance can be approximated by treating an imperfect capacitor as a perfect capacitor in series with a small resistance which is represented in Figure 5.7 (Franklin, n.d.). The effective impedance of the capacitor is then given by Equation (5.4) where R is the ESR, and jX_c is the complex impedance of the capacitor given by Equation (5.5).

$$Z = R + jX_c \tag{5.4}$$

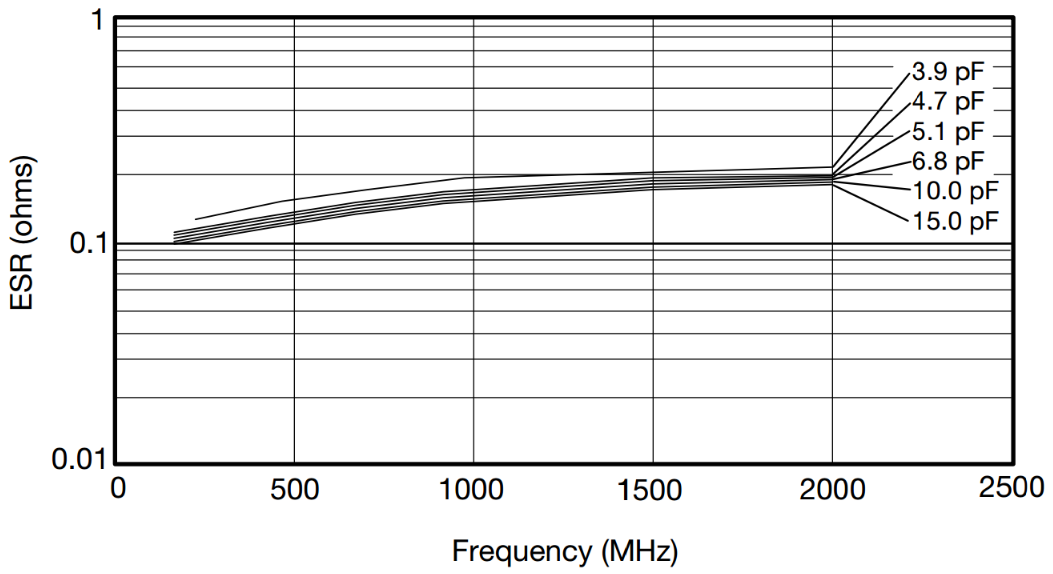
$$X_c = \frac{-1}{wC} \tag{5.5}$$

The ESR is then the difference between theoretical impedance X_c and the measured impedance Z of the capacitor at a known frequency. It is important to note that the ESR is affected by frequency and so would be more accurately represented as $R(w)$. Figure 5.8(a) and (b) describe the effect of frequency on the ESR of a capacitor which shows a non-linear increase in resistance at higher frequencies. The effect of the output capacitor on the output voltage ripple V_{ripple} can be determined using Equation (5.6) (Richtek, 2010) where the ESR is dependent on the type of capacitor used.

$$V_{ripple} \leq I_{ripple} \times \left(ESR + \frac{1}{8fC_{out}}\right) \tag{5.6}$$



(a) ESR Value vs Frequency for 0805 Capacitors



(b) ESR Value vs Frequency for 0603 Capacitors

Figure 5.8: ESR vs. Frequency for Different Capacitor Sizes (AVX Corperation, n.d.).

Table 5.1: Characteristics of the PCB Panel Used to Manufacture the Initial Prototype

PCB Material	FR4
PCB Thickness	62.99 mil
Copper Plate	35 μm
Finish	Lead-free HAL

5.2.4 PCB Track Characteristics and Signal Propagation

Another aspect to consider during board design is the PCB tracks in terms of signal propagation and resistance. The prototype board is expected to operate at 100 Mbit/s but only tracks between the port magnetics and the PHY are expected to handle switching frequencies in this region. The majority of the board is only expected to handle signal speeds dictated by the on-board clocks, all of which operate at 25 MHz. As a result, the board can be considered to be operating at low frequencies when considering track resistances and signal propagation (Flamm, 2004). Table 5.1 lists the characteristics provided by the manufacturer of the PCB panel used to manufacture the prototype platform (PCB Pool, 2015). All tracks on the board are etched from a copper plate and will thus have a thickness of 35 μm or approximately 1.38 mil. The fabricated PCB tracks can be approximated by a rectangular conductor depicted in Figure 5.11 (Hall *et al.*, 2000). The track cross-sectional area is then $A = TW$ where T is the track thickness and W is the track width. The resistivity of a copper track is described by Equation (5.7) (Sharawi, 2004, Thierauf, 2004) where A is quoted in thousandth of an inch squared (mil^2) and L is quoted in thousandth of an inch (mil).

$$R_{DC} = \frac{A \times 4.47 \times 10^{-7}}{L} \tag{5.7}$$

However, this only accounts for the DC component of the track resistance when current will flow in a uniform manner through the cross-sectional area of the track. An AC current on the other hand tends to flow closer to the surface of the track. This behaviour of the AC current is referred to as the skin effect (Popovic and Popovic, 2000). As the frequency of the AC current increases, the current flow will begin to migrate further away from the center of the track as described by Equation (5.8) (Popovic and Popovic, 2000). For Equation (5.8), δ is referred to as the Skin Depth, ρ is the resistivity of the material, μ is the permeability of the system, and f is the signal frequency. Relating the Skin Depth to the resistance of a PCB track can be approximated using Equation (5.9) (Sharawi, 2004).

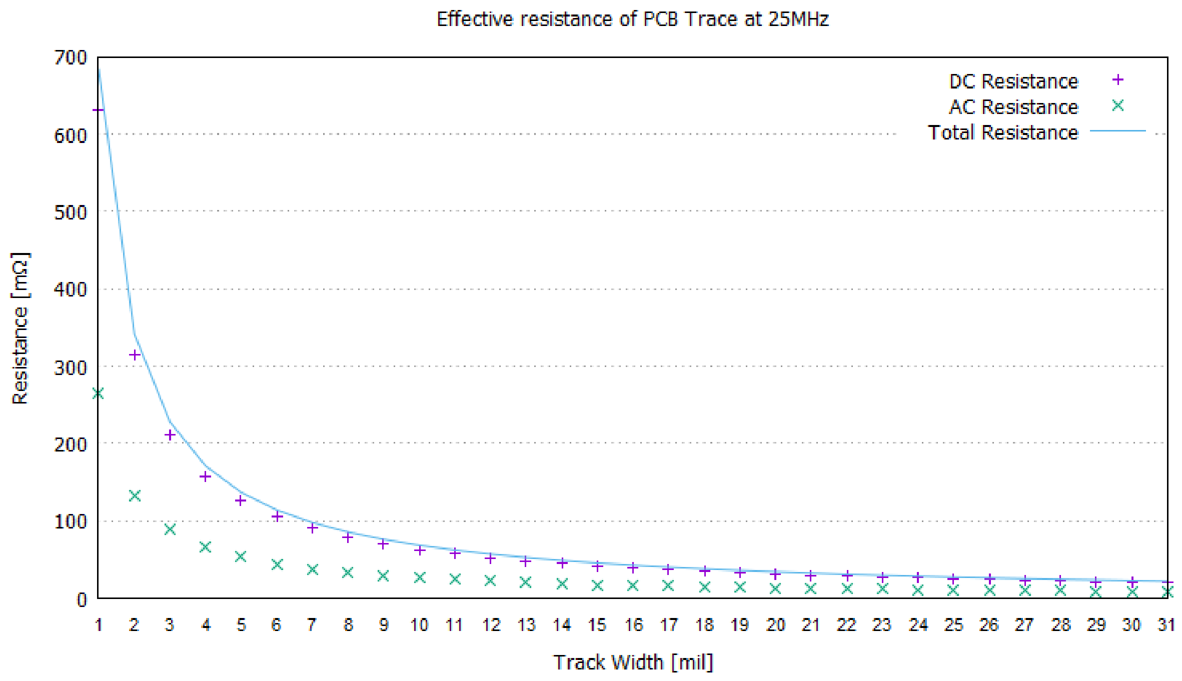


Figure 5.9: AC and DC Components of Track Resistance for Different Track Widths

$$\delta = \sqrt{\frac{\rho}{\pi f \mu}} \tag{5.8}$$

$$R_{ac} \approx \frac{\rho}{W\delta} = \frac{\sqrt{\rho\pi\mu f}}{W} \tag{5.9}$$

Finally, the net resistance of a PCB track can be approximated as $\sqrt{R_{DC}^2 + R_{AC}^2}$ (Sharawi, 2004) where at low frequencies the R_{DC} is dominant but falls away in favour of R_{AC} at much higher frequencies. Working with the upper bound frequency expected to be handled by the prototype platform of 25 MHz, Figure 5.9 shows the resistance per inch expected for tracks of varying widths.

Figure 5.9 indicates that the AC component R_{AC} contributes less than a third to the PCB track resistance for this implementation and, as Equation (5.9) is proportional to the signal frequency, R_{AC} will contribute even less for signals operating at lower frequencies. An alternate perspective of this data is depicted in Figure 5.10 which shows the track length (in inches) required to induce a resistance of 1Ω for varying track widths. These figures do not take into account any sources of interference owing to surrounding circuitry and are used primarily as indicators of possible upper bounds on track dimensions.

At very high frequencies additional factors come into play and the capacitive and inductive

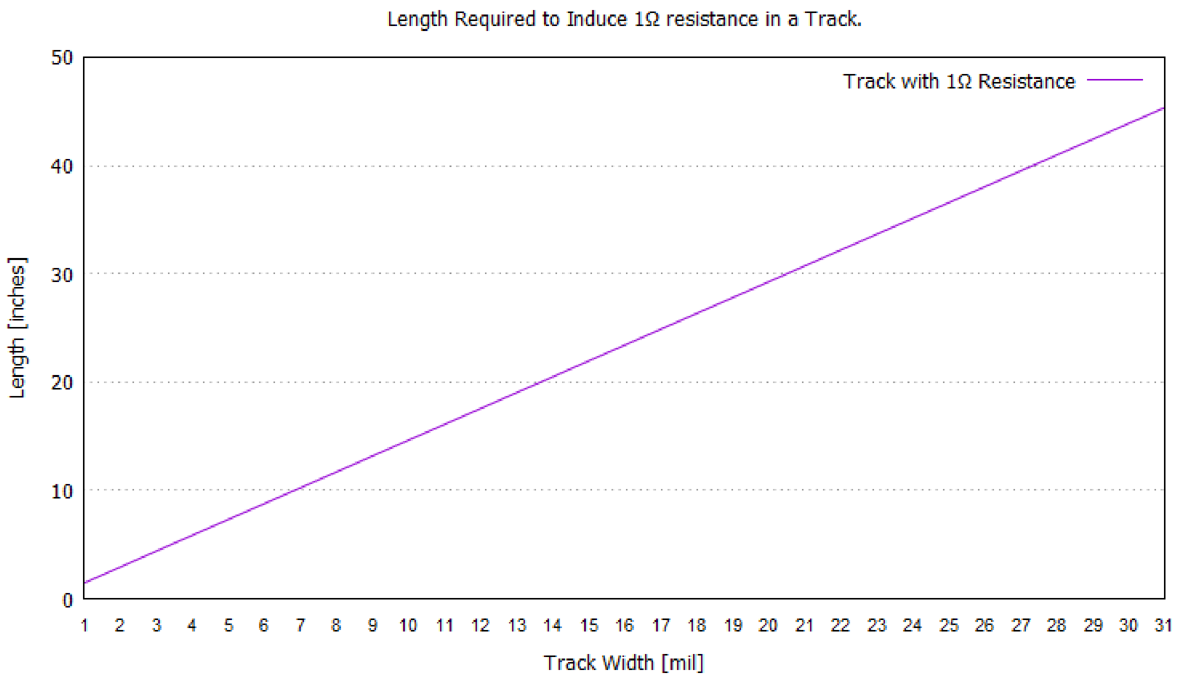


Figure 5.10: Trace Dimensions Contributing a Net Resistance of 1 Ohm

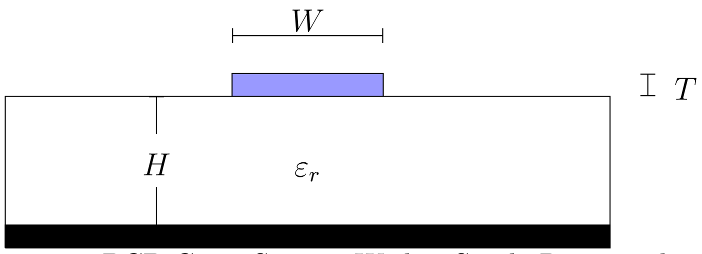


Figure 5.11: PCB Cross-Section With a Single Rectangular Trace

characteristics of the track become the dominant source of impedance, especially if there is a ground plane directly beneath the track. In this situation the effective impedance of the track is better approximated by $Z_0 = \sqrt{\frac{L}{C}}$ (Hall *et al.*, 2000) and its characteristics can be discussed in terms of a microstrip. Fortunately frequencies of 25 MHz are well below the frequency where such characteristics become significant and so can be neglected.

The next effect considered is the impact the PCB track could have on signal velocity. Signals in a perfect vacuum would propagate at the speed of light c which is a velocity of approximately $3 \times 10^8 m/s$ (Weiler and Pakosta, 2006). The PCB track and surrounding material is not a perfect vacuum, however the transmission speed can still be reasonably approximated as waves propagating through a homogeneous dialectic using Maxewll's equations which result in Equation (5.10) (Thierauf, 2004).

$$V_p = \frac{1}{\sqrt{\mu_0 \mu_r \varepsilon_0 \varepsilon_r}} \tag{5.10}$$

Where ε_0 is the permittivity of free space, ε_r is the relative permittivity of the dialectic, μ_0 is the permeability of free space, and μ_r is the relative permeability. In a perfect vacuum, μ_r and ε_r are defined to be 1 which simplifies Equation (5.10) to:

$$V_p = \frac{1}{\sqrt{\mu_0 \varepsilon_0}} \text{ where: } \varepsilon_0 = \frac{1}{\mu_0 c^2} \tag{5.11}$$

Equation (5.11) simplifies to c which implies that the propagation of a signal through a perfect vacuum is the speed of light as expected. Substituting this result back into Equation (5.10):

$$\frac{V_p}{c} = \frac{\sqrt{\mu_0 \varepsilon_0}}{\sqrt{\mu_0 \mu_r \varepsilon_0 \varepsilon_r}} = \frac{1}{\sqrt{\mu_r \varepsilon_r}} \tag{5.12}$$

Thus the signal propagation velocity through a dialectic of permittivity ε_r and permeability μ_r is:

$$V = \frac{c}{\sqrt{\mu_r \varepsilon_r}} \tag{5.13}$$

However the relative permeability of a non magnetic substance is 1 and so μ_r falls away (Sheng and Varadan, 2007), simplifying Equation (5.13) to:

$$V = \frac{c}{\sqrt{\varepsilon_r}} \text{ and } T_d = \frac{\sqrt{\varepsilon_r}}{c} \tag{5.14}$$

Where ε_r is the effective dielectric constant of the PCB track and surrounding mediums. As a PCB surface track is in contact with at least two mediums (air, PCB material), this value is composed of multiple ε_r values (Weiler and Pakosta, 2006). An approximation of the effective dielectric constant can be found using the diagram 5.11 where part of the signal field will exist in air with an approximate dielectric constant of 1 and the remainder will propagate through the substrate material (Semtech, 2006). Calculating ε_{eff} can be done using formulas (Collin, 2000) :

$$\varepsilon_{eff} = \frac{\varepsilon_r + 1}{2} + \frac{\varepsilon_r - 1}{2}(1 + \frac{12H}{W})^{-\frac{1}{2}} + F - \frac{0.217(\varepsilon_r - 1)T}{\sqrt{WH}} \tag{5.15}$$

$$F = \begin{cases} 0.002(\varepsilon_r - 1)(1 - \frac{W}{H})^2 & \text{when } \frac{W}{H} < 1 \\ 0 & \text{when } \frac{W}{H} > 1 \end{cases} \tag{5.16}$$

Where ε_r now refers to the dielectric constant of the substrate material. Using the values associated with the prototype platform, the signal propagation velocity can be found to be:

$$V = \frac{3 \times 10^8}{\sqrt{4.013}} = 1.498 \times 10^8 \text{ m/s or } 169.559 \text{ ps/inch} \tag{5.17}$$

Considering the physical dimensions of the prototype platform, no PCB track is expected to exceed 25.4 cm in length with signals propagating at the calculated velocity achieving this distance in 1.7 ns. As a result, signals switching under a frequency of 588.25 MHz such as the prototype platform should not be concerned with propagation speed limitations imposed by the PCB material.

5.2.5 Decoupling Capacitor Placement

During operation, the power draw by the device will fluctuate depending on processing load and the activity of device components. This power draw will result in a fluctuation in current demand described by the simple formula (5.18).

$$P = IV \tag{5.18}$$

As the voltage is not expected to vary, the load current must change to account for power fluctuations. This increased demand in load current should be readily available from a local source such as a decoupling capacitor. Should insufficient current be locally available, the current will be drawn from the general capacitance of the board which can cause voltage fluctuations on the power tracks referred to as power supply noise (Popovich *et al.*, 2011). Many Integrated Circuits such as the XMOS XS1-L8 require that the voltage supply be kept within a tolerance band of 10% (XMOS, 2015c) so the net voltage ripple V_{ripple} cannot exceed these bounds. Power supply noise is usually high frequency owing to the rate at which load current requirements can fluctuate. These fluctuations can be reduced by applying a surface mount ceramic capacitor close to each source of current fluctuation (Analog Devices, n.d.). For a microcontroller such as the XMOS XS1-L8, this implies multiple ceramic capacitors, one for each of the power pins on the package. It is recommended that four 100nF ceramic capacitors and a bulk 22 uF capacitor be placed close to the microcontroller for each of two power supplies to act as transient current sources. These capacitors can supply a power pin during an abrupt increase in current draw without impacting surrounding components (XMOS, 2015c). Placing the decoupling capacitor close to the source of the current demand also helps reduce the return current flow. The density of the return current flow is greatest closer to the source track through which the load demand is supplied (Silicon Labs, n.d.). In the case of a ground fill, the current density will be at its greatest directly adjacent to or beneath the source track.

5.3 Initial Prototype Design

To test the design of supporting circuitry required by hardware components such as the Ethernet PHYs and XMOS microcontrollers, it was proposed that an initial prototype board be fabricated. This prototype would include switching regulators designed to handle all power requirements of the Network Layer and confirm stability of the produced outputs. Once fabricated, this prototype could also be used to test interactions between components such as the SPI flash and XMOS microcontroller. Any errors identified during the development of the prototype could then be corrected during final board design.

5.3.1 Board Power Consumption

The platform will require three different voltage regulators for the proposed hardware. The Intel Edison will require an input voltage of 7-28V to act as its source power supply which it will then manage internally. Most IO and message passing on the platform operates at 3.3v logic, including the XMOS microcontrollers (XMOS, 2015c) and the Ethernet PHYs (Texas Instruments, 2015). These devices require that the 3.3v potential be supplied externally.

Table 5.2: Approximate Power Consumption Breakdown for Hardware Platform

Component	Voltage	Power	Count	Total Power	Total Current
Ethernet PHY	3v3	267mW	4	1068mW	323.64mA @ 3v3
Additional Circuitry	3v3	100mW	1	100mW	30.30mA @ 3v3
SDRAM	3v3	170mW	1	170mW	51.51mA @ 3v3
XMOS Core	1v	250mW	3	750mW	750mA @ 1v

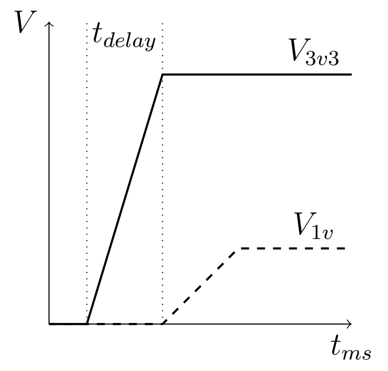
Table 5.3: Approximate Power Consumption Breakdown for Hardware Platform

Supply Voltage	Supply Current
3v3	405.45mA
1v	750mA

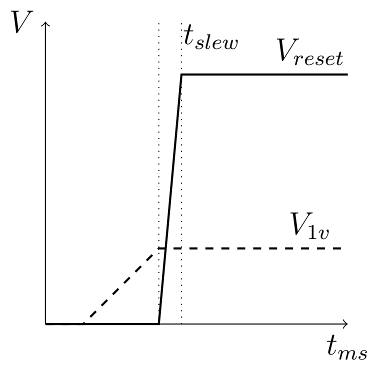
Table 5.2 lists the predicted power requirements for the Network Layer of the proposed platform. These figures are taken from the respective datasheets and indicate the peak power draw of each device. Platform power supplies should be chosen to exceed these requirements so as to ensure ample current is supplied during operation. The "Additional Circuitry" entry in Table 5.2 includes components such as the on-board flash, the 25 MHz clock, and LEDs which do not draw enough power to be individually assessed. Table 5.3 lists the total current requirements for the 1v and 3v3 power supplies that will be required to power the Network Layer of the platform.

5.3.2 XMOS Power Sequencing

An additional requirement of the XMOS XS1-L8 microcontroller is proper sequencing of the power supplies and reset signal to ensure stable power during device power up. First, the XMOS XS1-L8 requires that the 3.3v power supply rise from 0v to 3.3v in a monotonic manner before the 1v has reached 0.4v (XMOS, 2015c). To ensure this, it is recommended

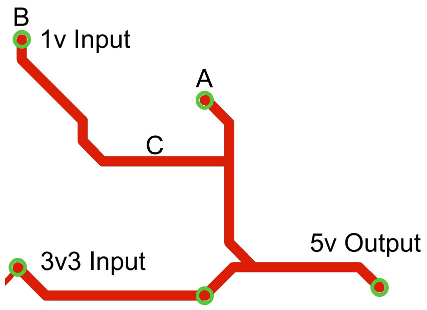


(a) Sequencing of V_{3v3} and V_{1v} power supplies

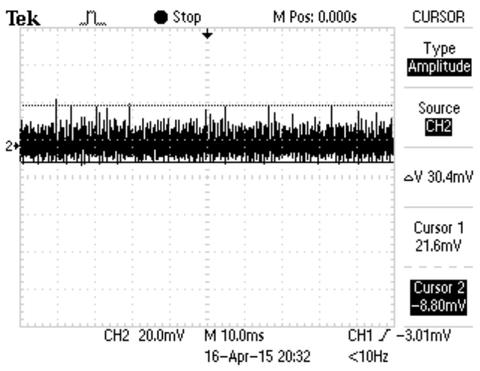


(b) Power on Reset

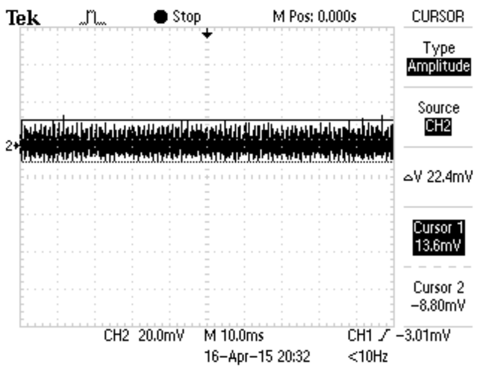
Figure 5.12: Required Power and Reset Sequencing for XMOS Microcontroller



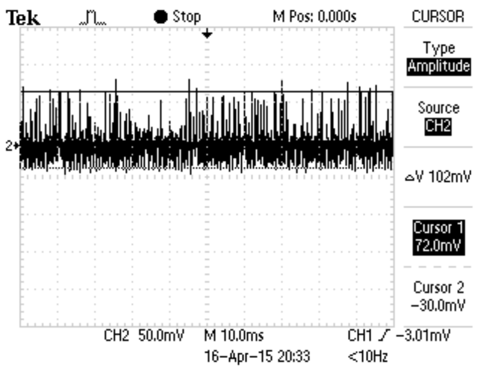
(a) PCB track for the 5v supply



(b) 5v Output



(c) Test point A



(d) Test point B

Figure 5.13: Recorded Voltage Ripple at Test Points on 5v Track.

that the 1v regulator only be enabled after the 3.3v supply has reached is expected output voltage of 3.3v $\pm$ 10% as depicted in Figure 5.12(a).

Once the 1v supply has reached its expected output voltage of 1v $\pm$ 10%, the reset signal must be pulled low for 100ns. This allows time for the PLL input voltage to stabilise before the microcontroller begins its boot procedure. The theoretical sequencing that the hardware design will attempt to achieve is depicted in Figure 5.12(b).

5.3.3 Power Supply Testing

After PCB design and fabrication was complete, the process of board population could begin. This process was broken into three phases with testing performed at the end of each stage. The first stage was populating the components responsible for board power supply which included circuitry required for correct power sequencing and power on reset. Once populated, a voltmeter was used to confirm that each power supply was generating the correct voltage.

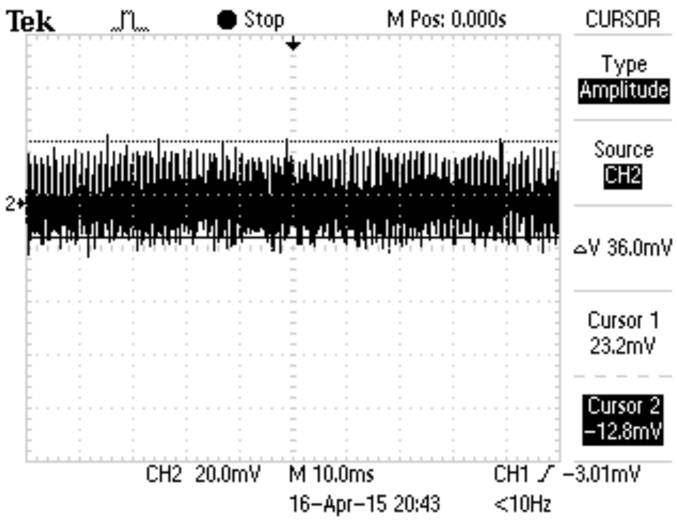


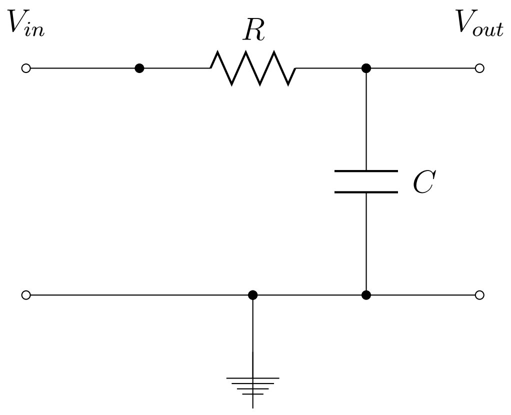
Figure 5.14: Recorded Voltage Ripple at Test Point A After Adding an Additional Capacitor

As discussed in Section 5.2.3 however, switch mode power supplies can induce a ripple voltage. This ripple voltage needed to be checked to ensure the power rails were within the 10% tolerance band required by the XMOS microcontroller for correct operation. As the output ripple voltage for both 1v and 3v3 supplies could be affected by the stability of the input supply, it was important the V_{ripple} in the 5v supply be suitably minimal as well. The oscilloscope capture shown in Figure 5.13(b) shows the ripple observed on output voltage from the 5v power supply. In this supply V_{ripple} was recorded to be approximately 30.4mV or 0.61% of the expected DC output voltage.

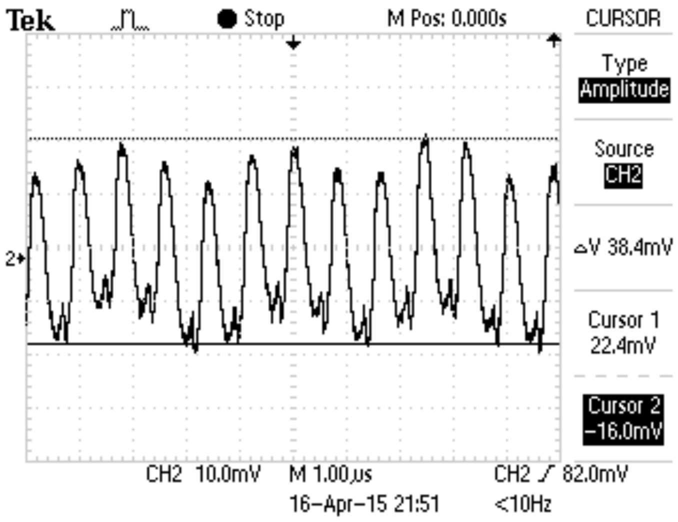
Monitoring the same track at the input location to the 1v regulator described a concerning variation. The 5v track was monitored for voltage ripple at two locations between the 5v regulator and the input to the 1v regulator. The 5v PCB track is depicted in Figure 5.13(c) with the voltage ripple observed at location A depicted in Figure 5.13(c) and the ripple voltage at location B depicted in Figure 5.13(d). From the oscilloscope readings it was clear that a significant amount of noise was being introduced to the 5v track between test points A and B. On the underside of the PCB, care was taken to ensure components were not placed close to the 5v track however, inspecting the topside of the PCB revealed that the 47uF capacitor used to smooth the voltage ripple of the 3v3 regulator was positioned directly above the 5v track at position C in diagram 5.13(b) introducing the power supply noise seen at test point B.

Resolving this increase in V_{ripple} was done by adding an additional 4.7uF capacitor at the 1v input. Figure 5.14 depicts the V_{ripple} at test point B after the additional 4.7uF capacitor was added to the board indicating a significant improvement.

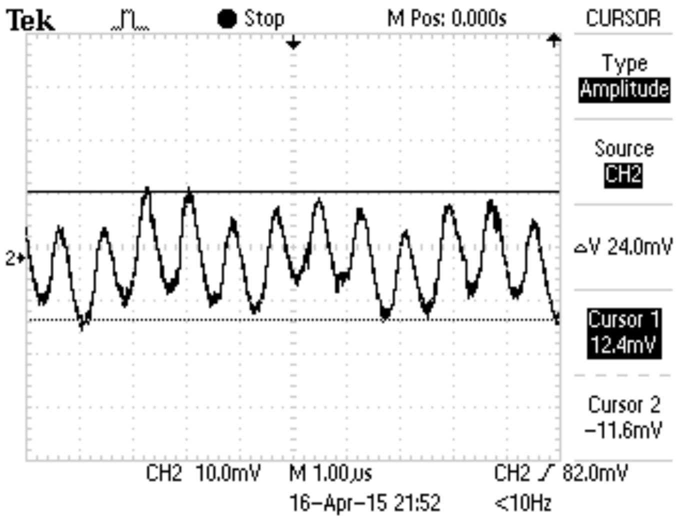
Finally, the XMOS microcontroller had a dedicated power input for the PLL which re-



(a) Diagram of PLL low pass filter



(b) Recorded V_{ripple} just before low pass filter (approximately 3.84% of DC voltage)



(c) Recorded V_{ripple} after low pass filter (approximately 2.4% of DC voltage)

Figure 5.15: Comparing Effects of the Low Pass Filter on the Voltage Ripple Present on the 1v Power Supply

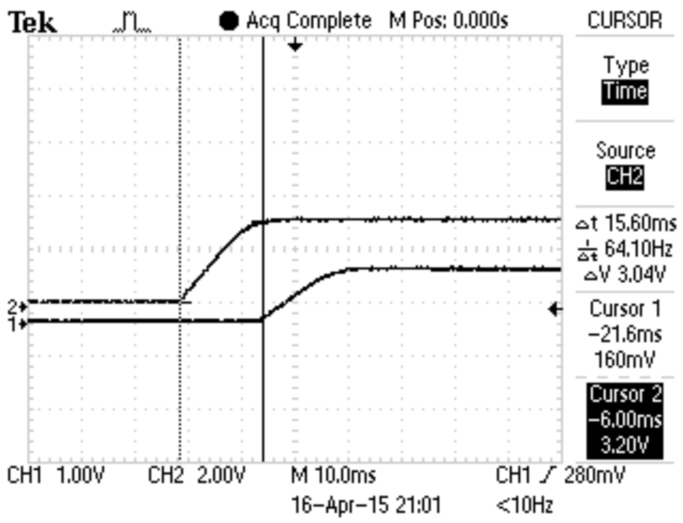


Figure 5.16: Recorded Sequence in Which the 1v and 3v3 Power Supplies are Initiated.

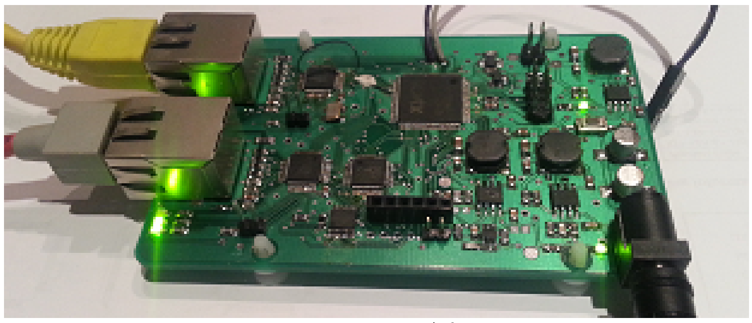


Figure 5.17: Device Prototype After Board Population

quired a very clean input voltage. To achieve this, a low pass filter on the input power supply was recommended (XMOS, 2015c). The implemented low pass filter consisted of a 4.4Ω resistor and a 100nF capacitor as shown in diagram 5.15(a) and was placed close to the PLL input supply pin. The effect of the low pass resistor can be seen by comparing the noise on the 1v power supply before [depicted in Figure 5.15(b)] and after [depicted in Figure 5.15(c)] the low pass filter.

5.3.4 Board Testing

As discussed in Section 5.3.2, the XMOS power supplies had to be correctly sequenced before correct booting of the device can be insured. During power up of the prototype, both the 3v3 and 1v tracks were monitored. Figure 5.16 depicts the recorded sequence in which the power supplies were initialized which follows the theoretical sequence described in Figure 5.12(a).

Once fabrication of the initial prototype shown in Figure 5.17 was complete, testing of the device was performed to ensure hardware functionality. An XMOS xTAG programmer was used to access the XMOS microcontroller over the JTAG interface. Test applications were then written to the board to test communication with the Ethernet PHYs and the SPI flash. Errors were identified however these were owing to incomplete soldering and were suitably corrected. Further details relating to this initial prototype were published in the Southern Africa Telecommunication Networks and Applications Conference (SATNAC) 2015 (Pennefather and Irwin, 2015)

5.4 Final Board Design

In Section 5.3.1, peak power consumption of the final proposed board was discussed and a suitable power supply was designed and tested. Design of a network interface and communication with the XMOS microcontrollers were also tested as part of the initial board design. The intent of the final prototype was to include four network interfaces and an additional two XMOS XS1-L8 microcontrollers. In addition, SDRAM and related components were also added to the final design. To provide communication between the SPI flash, XMOS microcontrollers and base platform, two SN74 (Texus Instruments, 2007) switching chips were added to the board. These switching chips allowed the necessary isolation of components to avoid interference during operation and programming of the board.

The final design was completed and the resulting device was named the Frame Routing and Manipulation Engine (FRAME). An annotated diagram of the completed board is depicted in Figure 5.18 while the Eagle design files are presented in Appendix D.1.

5.4.1 Fabrication Costs

As stated in Section 1.3, a secondary goal of this research was to develop a device that was comparatively low cost when compared with market equivalent products. To evaluate this, the costs relating to fabrication of the FRAME board must be recorded. These costs are quoted in US dollars ($) with all components purchased in rands (ZAR) being converted at the current exchange rate of 12.73 ZAR to the dollar.

PCB printing was performed by PCB Pool (PCB Pool, 2015) which prices PCB fabrication according to board dimensions with a minimum size of 100cm^2. As the FRAME board dimensions are 137cm^2, the PCB fabrication cost was $86.93. This price does not include

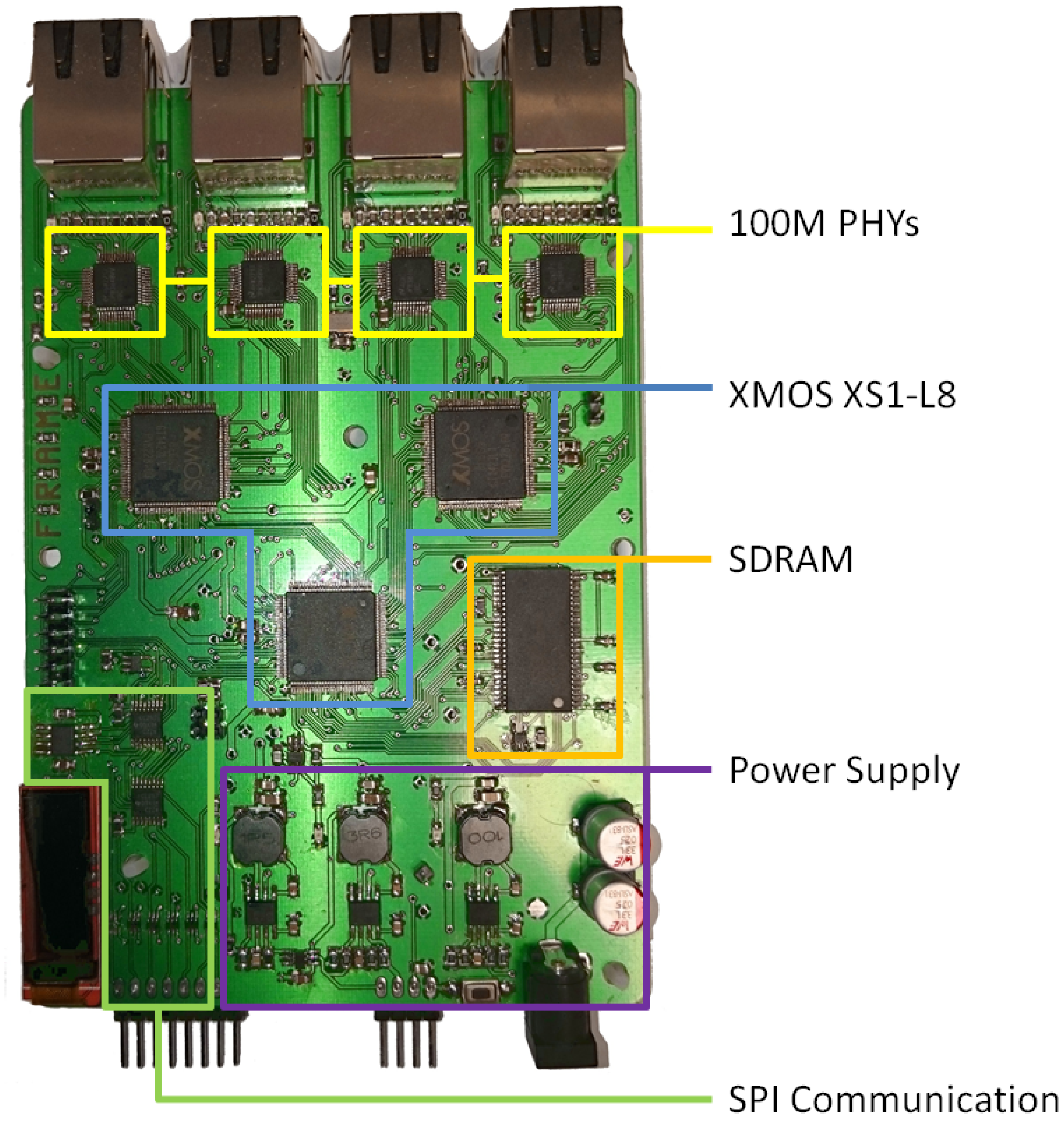


Figure 5.18: Annotated Image of the Fabricated FRAME Board

Table 5.4: FRAME Board Component Costs

Component Name	Unit Price [$]	Unit Count	Total Price [$]
XMOS XS1-L8	14.5	3	43.5
SDRAM	4.28	1	4.28
PHY	4.99	4	19.96
Switching Regulator	0.82	3	2.46
Ethernet	0.57	4	2.28
Inductors	0.64	3	1.92
I2C OLED	8	1	8
Additional Components	22	1	22
Sum Total			**82.4**

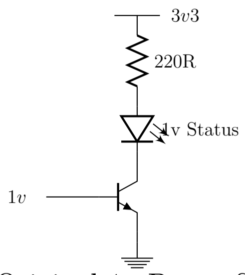


Figure 5.19: Original 1v Power Status Circuit

a solder mask to minimize costs. All component soldering was done by the researcher to further minimise costs.

All board components were purchased in dollars with an shipping cost of $60 and a import duty fee of 30% (shipping cost inclusive). Table 5.4 shows the component costs recorded in dollars, excluding shipping and import duty related fees. Including shipping and import duties, the cost for board components totals to $185.12. Total fabrication cost for the FRAME board is then $272.05.

5.4.2 Errata

During testing of the FRAME board, a series of design errors were identified and the impact of each was evaluated. The majority of significant errors relate to the communication interface between the XMOS controllers and the base platform, though design errors which directly impacted board functionality were successfully resolved with minor modifications.

The first design error detected relates to the power indicator LED for the 1v regulator. The intent was for this LED to indicate when the 1v power supply was active. The implemented circuit described in Figure 5.19 used a transistor to create an open drain when the 1v line was powered which caused the status LED to illuminate. Unfortunately the voltage threshold at the base of the transistor was too high to be affected by the 1v regulator resulting in the status LED remaining off regardless of the 1v power status. This design error was considered minor as it did not affect device performance and no action was taken to resolve it. A proposed solution for future board revisions is to either use a MOSFET to act as an open drain or operational amplifier to drive the status LED.

The second design error detected related to the link and power status LEDs associated with each of the four network interfaces. In all cases, these two LEDs had been swapped resulting in the power LED only illuminating when a connected device could sync at 100 Mbit/s while the sync LED remained permanently on. Considering possible solutions, it was concluded that no action would be taken to rectify the issue for the current revision as it did not impact actual board functionality. This error should be rectified in future board revisions however as it could lead to misinterpretation of connection speeds by the user.

The third error detected relates to reprogramming of the FRAME board. During design it was believed that toggling the reset line connected to all XMOS controllers would cause a board reset and application code to be read from the SPI flash. During testing this was found to not be the case and a full power cycle would be required. Fortunately this could be resolved by utilising the enable pin present on the RT8250 switching power supply (Richtek, 2010) responsible for producing the regulated input supply for the 1v and 3v3 regulators. The program reset line was disconnected from the microcontroller reset line and instead connected to the enable pin of the RT8250 via a jumper cable. This allowed the board to continue to function as intended.

The final error related to the switching chips used to switch the SPI interface between flash, XMOS microcontrollers and base platform. This switching was necessary, as during reset the XMOS microcontrollers would hold the SPI lines at a logical 1 causing communication between the base platform and the SPI flash to fail. The error identified was that one select pin was not routed to the communication header resulting in the base platform being unable to disconnect the SPI interface from the XMOS microcontrollers. To remedy this, a jumper wire was soldered to the select pin and an additional breakout header was attached to the board.

After the modifications noted above, the only impact the acknowledged errata had was to limit the base platform to the raspberry pi. This limitation stems from differing voltage levels with the Intel Edison requiring a voltage translator to communicate with the FRAME board. The voltage translator was built into the designed board but modifications to select and reset pins bypassed it and require a voltage level of 3v3 to operate.

5.4.3 Interpreter Modifications

Owing to architectural differences between the functional prototype and the FRAME board, a series of modifications had to be made to the DSL to ensure compatibility with the new hardware. These modifications resulted in no changes to the language itself but focus on minor alterations to the translation process.

The first modification required was to change the number of cores available to the application from 32 to 24. The reduction results from the new device supporting three XMOS tiles while the functional prototype could support four. The decision to support fewer tiles stems from component costs and availability of equipment and expertise required to mount components with a smaller form factor. Pin layouts for the port modules were updated to match the layout of the FRAME board and the XMOS tiles each module was associated with was also updated.

5.5 Summary

This chapter detailed the development of a dedicated hardware platform for network processing. An initial prototype was developed in Section 5.3 after a detailed evaluation of power requirements was performed. After functionality was confirmed, the initial design was scaled to meet the Network Layer requirements and the FRAME board was fabricated.

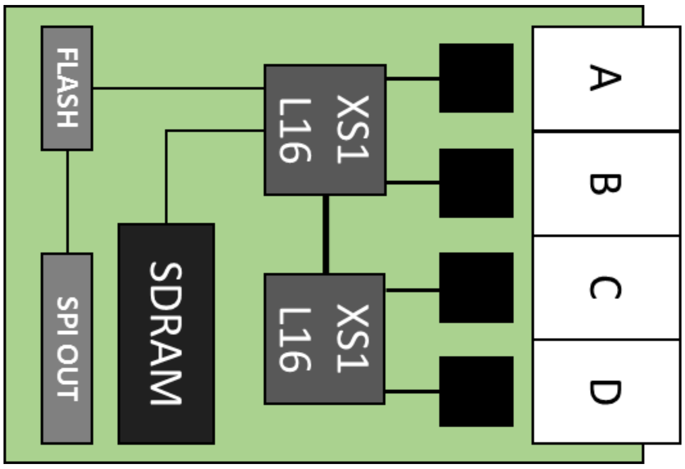
Errors relating to board development which varied form primarily cosmetic to affecting board functionality were highlighted and discussed in Section 5.4.2. Solutions to errors that impacted functionality of the board were implemented while indication and cosmetic errors were noted to be corrected in future board revisions. In Section 5.4.3, modifications to the developed translator associated with the DSL implemented in Section 4.9 were discussed and performed to ensure compatibility between the FRAME board and designed applications.

6 Testing

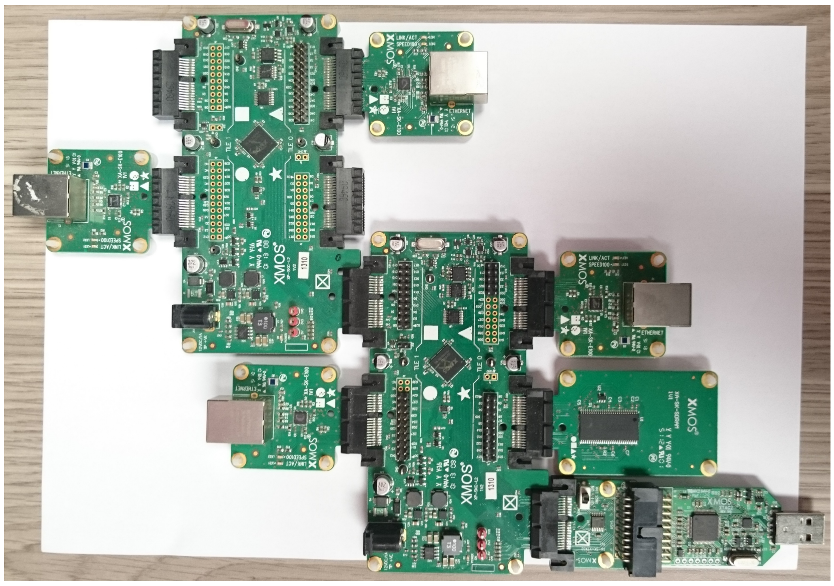
6.1 Introduction

This chapter presents testing procedures and results concerning system performance and conformance testing. Chapter 5 presented the design and fabrication of the underlying hardware platform on which the Network Layer could operate. Hardware design was performed in conjunction with the first phase of testing described in this chapter. Once fabrication was complete, testing resumed using the newly implemented FRAME board.

Section 6.2 focuses on combining pre-existing hardware components to create a functional prototype. This revision is intended to test the functionality of the software components designed for the platform and does so by designing and implementing a series of frame modification tests. Section 6.3 continues testing system components with the FRAME board replacing the functional prototype in the system. Finally, the chapter concludes in Section 6.4 giving a reflection of board development and testing.



(a) Connection Layout of the Functional Prototype



(b) Image of Assembled Functional Prototype

Figure 6.1: Produced Functional Prototype Used in System Testing.

6.2 Implementation of a Functional Prototype

The hardware used to construct a functional prototype of the network platform consists of two XMOS Slicekits (XMOS, 2013a), four Ethernet breakout modules and one SDRAM breakout module. The selected breakout modules were designed to be combinable with the XMOS Slicekit for rapid creation of testing hardware for applications targeted at the XMOS architecture (XMOS, n.d.a). Guidelines are provided for requirements relating to component interconnection as well as documentation of the ports made available on each connector interface[1].

[1]A list of available breakout modules is available at: https://goo.gl/gU1TP2 .

Table 6.1: Recorded Throughput on Functional Prototype

Source	Destination	Average	Standard Deviation	Test Count
A	B	83.1	0.17	20
C	D	83.0	0.12	20
A	C	83.4	0.15	20
B	D	83.2	0.22	20
A	D	83.2	0.15	20
B	C	83.2	0.26	20

6.2.1 Hardware Setup

The hardware components are connected as illustrated in Figure 6.1 with two network interfaces per microcontroller. These microcontrollers are then connected using the XMOS Fast Link, which is discussed in Section 2.3.1. Both SliceKits are powered separately using standard power supplies producing 12 V @ 500 mA.

6.2.2 Component Testing

Before testing of the designed modules could be performed, testing of the individual hardware components comprising the functional prototype was necessary to ensure proper operation. For each component, an application was designed and executed on the functional prototype. These applications were intentionally kept simple and included only the relevant internal modules to ensure hardware compatibility.

Ethernet Breakout Modules

The application used to test the Ethernet breakout hardware components was based on the Ethernet internal module demo application discussed in Section 4.3.1. This application allowed for the testing of two Ethernet ports simultaneously by relaying traffic between the selected ports in a full-duplex manner. Two end hosts were attached to the relevant Ethernet ports and traffic was exchanged accordingly. Six configurations were set up to test traffic communication between the functional prototype ports labelled A,B,C,D as indicated in Figure 6.1(a). Table 6.1 describes the transmission speeds recorded for each of the six configurations tested. These results show little variation between the interfaces.

SDRAM Breakout Module

Communication with the connected SDRAM was previously evaluated in Section 4.5.1, which focussed on providing an analysis of timings relating to communication with this component. The hardware configuration used to perform these tests was equivalent to the hardware configuration of the functional prototype. These tests were however, performed again to confirm conformance, but no significant variations were recorded.

6.2.3 Frame Modifications

The primary focus of the system and the associated DSL was to provide the user with the necessary tools to modify network traffic at the frame level in a manner that is simple and intuitive. To achieve this, Link provides a selection of pre-built functions intended to assist the user in modifying frame characteristics. A full list of the currently supported operations is shown in Appendix B.4. Though all were tested, only a sample of modification functions are discussed here to show system functionality.

The functions were divided into three groups according to the type of action performed. The first group of functions prefixed with `is_`, were used to identify the presence of a specific characteristic within the network frame. The second group were functions associated with retrieving frame characteristics to be used in the network application. The third were assignment related functions responsible for updating an associated frame characteristic with a specified value. For both retrieval and assignment function groups, underlying checks were performed to ensure the characteristic being retrieved or replaced was present in the current frame as discussed in Section 4.8.2.

To show functionality, three tests were performed using functions from each group. A conditional function was used to identify the target type of network frame with a characteristic form that frame being selected by using a retrial function. The assignment function was used to replace a targeted frame characteristic with a modified version of the characteristic extracted.

```
0000   70 54 d2 ab 4b 5e 00 22 68 58 3b fc 08 00 45 00
0010   00 28 00 01 00 00 40 06 e7 16 c0 a8 89 66 c0 a8
0020   89 01 00 14 13 88 00 00 00 00 00 00 00 00 50 02
0030   20 00 e8 8d 00 00 00 00 00 00 00 00
```

(a) Frame Header Before MAC Replacement

```
0000   70 54 d2 ab 4b 5e 70 54 d2 ab 4b 5e 08 00 45 00
0010   00 28 00 01 00 00 40 06 e7 16 c0 a8 89 66 c0 a8
0020   89 01 00 14 13 88 00 00 00 00 00 00 00 00 50 02
0030   20 00 e8 8d 00 00 00 00 00 00 00 00
```

(b) Frame Header After MAC Replacement

Figure 6.2: Hexdump of Ethernet Frame Before and After MAC Modification

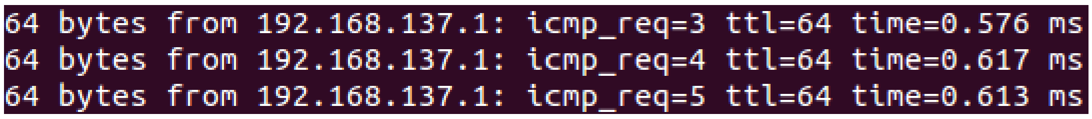
MAC Swap

```
//———Declaration Phase———
Modifier Swapper
{
    on Condition ( ip_protocol == TCP )
    {
      frame_source_mac = frame_destination_mac;
    }
}
//————Linking Phase————
PortA —> Swapper —> PortB;
PortB —> PortA;
```

Listing 6.1: MAC Swap Written in Link

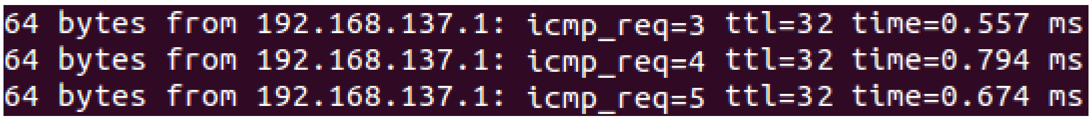
The first test performed modified the source MAC address of a network frame supporting the TCP protocol by setting it to the value of the destination MAC address. Figure 6.1 describes the application required to perform this task, which was then compiled and executed on the functional prototype. This application only modified traffic directed from port A to port B while traffic traversing in the opposite direction remains unaffected.

To show application functionality, a crafted packet was transmitted from the host connected to port A (host A) to a host connected to port B (host B) through the system. Figure 6.2(a) shows a hexdump of the packet header as seen by host A while Figure 6.2(b) shows a hex dump of the packet received by host B. As the highlighted fields in both figures indicate, the source MAC address has been replaced with the destination MAC address.



```
64 bytes from 192.168.137.1: icmp_req=3 ttl=64 time=0.576 ms
64 bytes from 192.168.137.1: icmp_req=4 ttl=64 time=0.617 ms
64 bytes from 192.168.137.1: icmp_req=5 ttl=64 time=0.613 ms
```

(a) Without TTL Field Modification



```
64 bytes from 192.168.137.1: icmp_req=3 ttl=32 time=0.557 ms
64 bytes from 192.168.137.1: icmp_req=4 ttl=32 time=0.794 ms
64 bytes from 192.168.137.1: icmp_req=5 ttl=32 time=0.674 ms
```

(b) With TTL Field Modification

Figure 6.3: Ping Utility Output With and Without TTL Field Modification

```
0000   00 22 68 58 3b fc 70 54 d2 ab 4b 5e 08 00 45 00
0010   00 54 4a bf 00 00 40 01 5c 31 c0 a8 89 01 c0 a8
0020   89 66 00 00 bd e2 15 c2 00 0a d2 11 aa 55 00 00
0030   00 00 e4 16 0d 00 00 00 00 00 10 11 12 13 14 15
0040   16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25
0050   26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35
0060   36 37
```

(a) Frame Header without TTL Modification

```
0000   00 22 68 58 3b fc 70 54 d2 ab 4b 5e 08 00 45 00
0010   00 54 4b 70 00 00 20 01 9b 80 c0 a8 89 01 c0 a8
0020   89 66 00 00 01 d7 15 f3 00 09 6f 12 aa 55 00 00
0030   00 00 0f f2 00 00 00 00 00 00 10 11 12 13 14 15
0040   16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25
0050   26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35
0060   36 37
```

(b) Frame Header with TTL Modification

Figure 6.4: Hexdump of Ethernet Frame TTL Modification

TTL MOD

```
//———Declaration Phase———
Modifier ttl_decrementer
{
  on Condition (ip_protocol == ICMP)
  {
    ip_ttl = ip_ttl/2;
  }
  ip_update_checksum;
}
//————Linking Phase————
PortA —> Swapper —> PortB;
PortB —> PortA;
```

Listing 6.2: TTL Decrement Application Written in Link

As a second test, the Time To Live (TTL) field of an ICMP packet was modified to half its original value. Again, a code snippet from Link is presented in Listing 6.2 which produces an application implementing the functionality discussed here. The command `ip_update_checksum` tells the interpreter that the checksum of the IP header should be updated to reflect the modified attribute. The origin of the ICMP request is a host connected to Port B and as a result, modifications were only applied to traffic responding to the ping request.

Functionality of the application was shown by comparing Figure 6.4(a) and (b). Figure 6.4(a) represents the ICMP response without the modification while Figure 6.4(b) represents the response packet with modifier application implemented. The highlighted fields indicate the updated TTL values and updated checksum. Comparing output snippets shown in Figure 6.3(a) and (b) as seen by host A further confirmed this functionality.

6.2.4 Timing of Supporting Functions

Section 6.2.3 used simple application examples to show the functionality of modification functions that could be used as part of user applications. Including these functions as part of an application however, would contribute to the timing delay associated with each network packet. Tests were undertaken to determine the impact each function could have on the system by performing timing analysis on each of the supporting functions associated with the TCP, IP, and ARP protocols listed in Appendix B.4.

Timing tests were performed using the XMOS Timing Analyzer (XTA)[2] which evaluates each operation associated with a function to determine the time required to perform each. These times were then accumulated to give the worst and best case timing paths through the evaluated function. All analysed functions were confirmed using the XMOS timers, which support a 10ns resolution.

Figure 6.5 graphs the recorded timings for each function tested. These graphs are divided according to protocol and include both the read and write related function calls available to the user as part of the DSL. Comparing the timing costs per function, it is apparent that some operations such as those relating to the TCP protocol required significantly more time to complete. This however, was expected owing to the hierarchy of tests, which were evaluated prior to each TCP related operation. These tests ware performed by an internal function called by all TCP related functions to confirm the presence of the TCP protocol in the current frame. This internal function was timed to take 1890 ns which, once deducted from the TCP function timings, brought the timing costs to below those seen for the IP header related functions.

This internal function performed a three-step evaluation of the frame. The first step determined if the VLAN tag was present so that the reminder of the frame could induce the appropriate offset. Following this, a test was done to detect the presence of the IPv4 protocol in the frame. If present, the TCP protocol identifier was checked before the

[2]http://www.xmos.com/products/tools/xta

function could determine if the frame formed part of a TCP transmission. As the IP protocol related function must include the VLAN and IPv4 evaluation, it was expected that the TCP function timing without the internal check would be less.

In graphs (a) and (b) in Figure 6.5, the longest timed functions relate to performing checksum recalculation, which is essential for producing valid modified network frames. These functions significantly exceed the timings relating to the remaining graphed function calls with the TCP update checksum function call taking taking 4.73μs to complete.

Though not as apparent as in Figure 6.5(a), Figure 6.5(b) and (c) indicate a distinct difference in time required to read a frame characteristic when compared with the time taken to write the equivalent characteristic. The recorded exception to this was the getting and setting of MAC addresses which involved operations on more than a single 32-bit integer of the target frame.

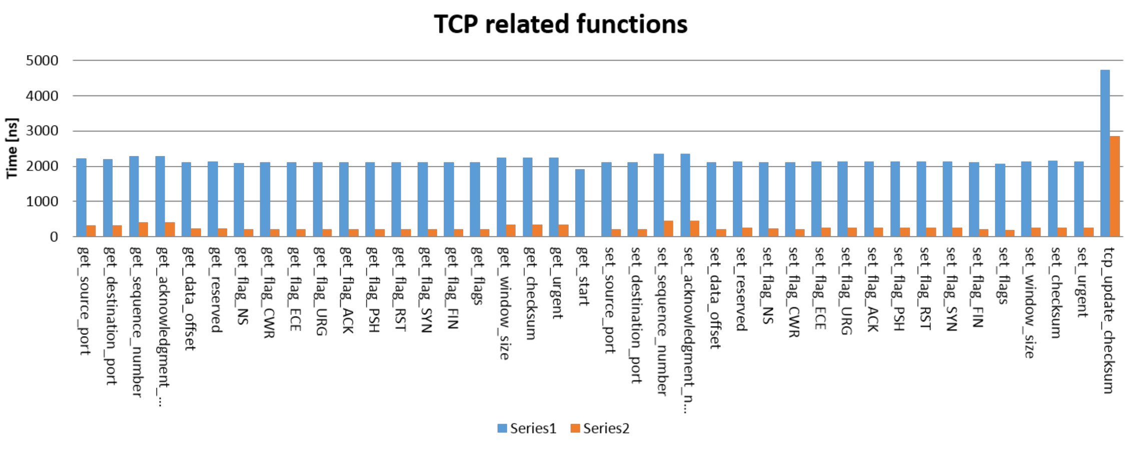
6.3 System Testing

After the board functionality had been confirmed, system testing including the new device could resume. The aim of the remaining tests was to confirm functionality of individual system components when executed on the new platform. Two application scenarios were also constructed and implemented on the FRAME board as part of the testing process.

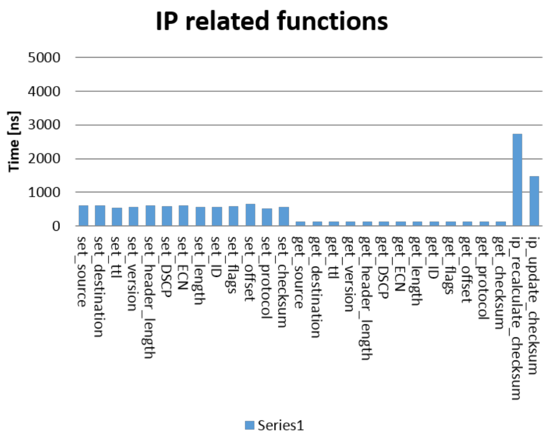
6.3.1 Delay Module

To confirm functionality of the delay module included as part of this system, tests that highlighted expected component operation in implemented applications were designed and undertaken. The aim of the delay module was to provide the system with the ability to delay the propagation of network frames by providing access to an SDRAM interface capable of storing the traffic for a specified duration as discussed in Section 4.5.

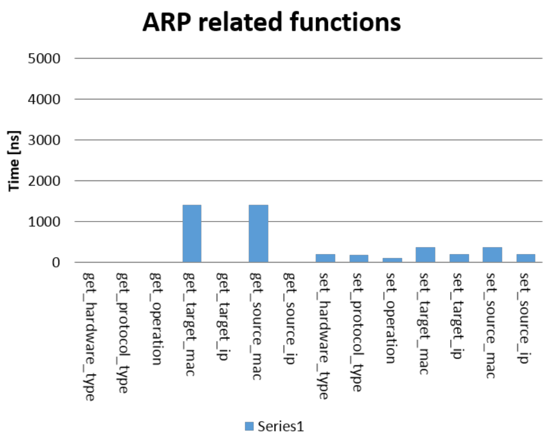
To test this feature, the ICMP protocol was used along with the ping utility common to most operating systems. The test comprised of three stages required to produce comparable data necessary to prove component operation. The first stage involved performing a sequence of ping requests without any delay implemented to determine the general delay associated with traffic transmission when using the device. Table 6.2 shows the average RTT delay to be 0.174 ms with a standard deviation of 6.5%.



(a) TCP Related Function Timings



(b) IP Related Function Timings



(c) ARP Related Function Timings

Figure 6.5: Recorded Timings Relating to the Execution of Supporting Functions

Table 6.2: Average RTTs With No Delay

Average	0.174 ms
Standard Deviation	0.011 ms
Test Count	1506

Table 6.3: Average RTTs With 0ms Delay

Average	0.349 ms
Standard Deviation	0.011 ms
Test Count	115

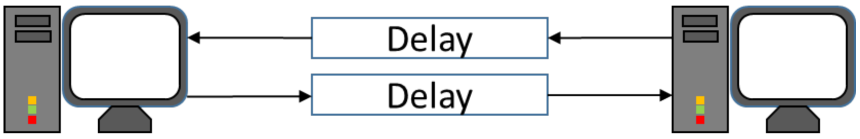


Figure 6.6: Functional Overview of Implementing Delay

The second stage repeated the same testing procedure, but with a delay of 0 ms implemented on the device. This delay was implemented on network traffic passing in either direction through the device requiring two delay modules to be instantiated as depicted in Figure 6.6. The results of the repeated tests are recorded in Table 6.3, which gives an average RTT of 0.349 ms with a standard deviation of 3.3%.

$$\frac{0.346 - 0.174}{2} = 0.0875\text{ms} \tag{6.1}$$

Comparing the two results, the inherent delay associated with using the delay module is calculated in Equation (6.1) to be 0.0875 ms. The implications of this are that for the FRAME board, a minimum delay of 87.5 μs would be required, which is an increase of 15.42% on the 74 μs delay associated with the testing performed on the functional prototype discussed in Section 4.5.1. The increase in delay is due to the different microcontroller families used by the platforms, but could potentially include additional factors that would impact the delay time. The interpreter was updated to include the delay before the third stage of testing could begin. The interpreter was also updated to handle delays less than 0.0875 ms by simply not including a delay module. Should the user attempt to include such a delay, the interpreter would issue a warning and trim the operation from the resulting application.

To show the delay module in operation a simple application written in Link is shown in Figure 6.3. This application adds a 10ms delay to the upstream and downstream traffic, which includes the updated delay offset. Table 6.4 gives a brief summary of the results

Table 6.4: Average RTTs With 20ms Delay

Average	20.2 ms
Standard Deviation	0 ms
Test Count	300

```
//————Linking Phase————
PortA—> Delay1(10ms)—> PortB;
PortB—> Delay2(10ms)—> PortA;
```

Listing 6.3: Delay Application Written in Link

Table 6.5: Statistical Summary of TCP Throughput on FRAME Board

Path	Average	Minimum	Maximum	Standard Deviation	Test Count
A.B	84.44	83.9	84.9	0.19	61
A.C	82.80	82.3	83.4	0.35	61
A.D	82.90	82.3	83.4	0.32	61
B.C	82.93	82.3	83.4	0.33	61
B.D	82.92	82.3	83.9	0.32	61
C.D	84.54	84.4	84.9	0.23	61

produced using the application, showing the expected increased delay of 20 ms.

6.3.2 Throughput

In Section 4.2, analysis of network transmission speeds was evaluated to determine the feasible data throughput of the system. These tests however were performed using components later combined to form the functional prototype and did not include the completed application layer. As a result, the conclusions drawn during the development stage needed to be re-evaluated. The tests undertaken during development the stage were revisited and a new series of tests were constructed for the complete system.

To determine achievable network transmission speeds while using the FRAME board, throughput tests were performed that took full system functionality into account. Testing was performed in three stages. The first stage involved monitoring network throughput of the FRAME board while executing a minimal application that performed no network processing. Background processing, including the monitoring of network traffic received at each port was included as this was now compulsory for all applications. Any additional overhead such processing could produce had to be included to determine realistic traffic throughput. As the FRAME board consists of four network interfaces, testing was repeated for each feasible path between two ports on the device described by Equation (6.2). This resulted in six permutations to test as listed in Table 6.5. The network transmission speeds, recorded using iperf (Tirumala *et al.*, 2005), are listed in Table 6.5.

$$\binom{n}{2}\text{, where } n \text{ is the set } \{\text{portA, portB, portC, portD}\} \tag{6.2}$$

By reviewing the results, it quickly becomes apparent that two of the four defined paths achieved a higher overall throughput. Graphing the recorded data further highlights the variation presented in Figure 6.6. To understand the source of the varying throughput

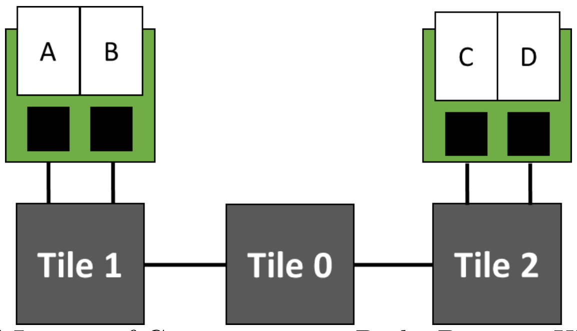
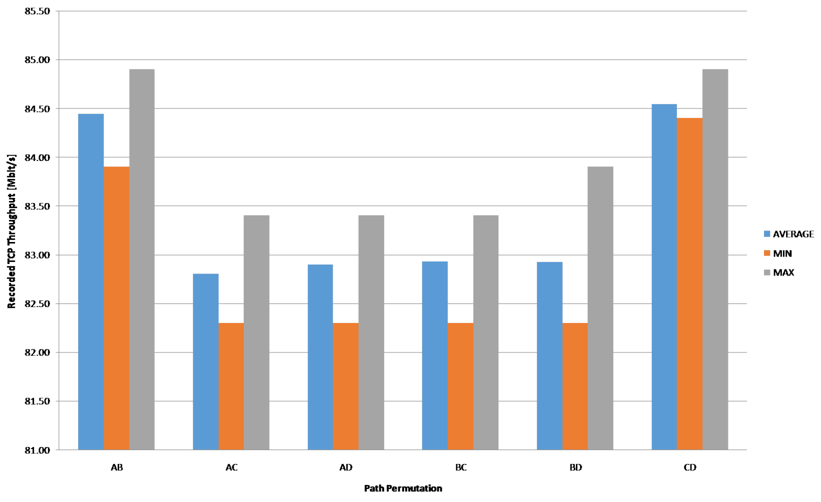


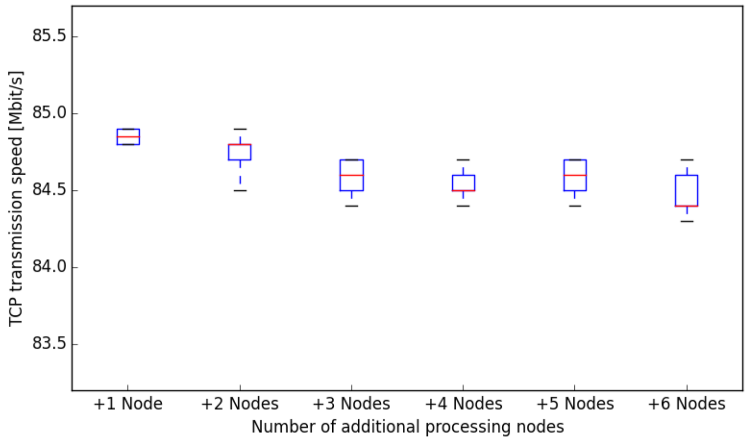
Figure 6.7: Functional Layout of Communication Paths Between XMOS Microcontrollers

Table 6.6: Statistical Summary of TCP Throughput on FRAME Board

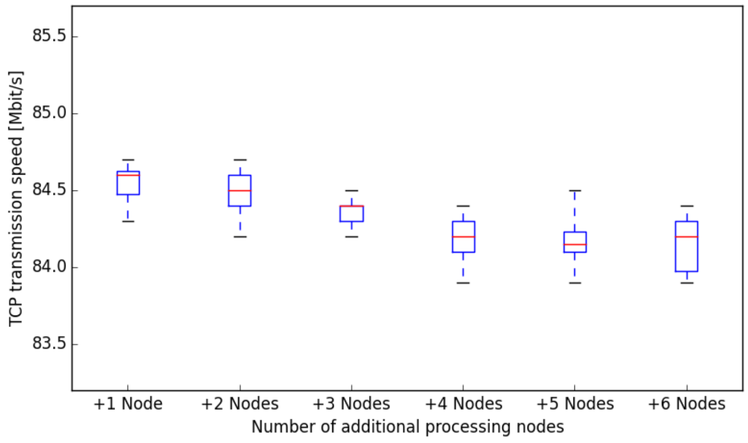


seen during testing, the physical layout of the FRAME board was reviewed. Figure 6.7 represents the FRAME board as a functional diagram, indicating connections between microcontrollers. For ports A and B to communicate, no intertile communication is required; the same situation exists for ports C and D. For the remaining four permutations, data must pass through the intermediate microcontroller labelled as Tile 0. These additional two steps are suspected to be the cause of the reduced throughput seen in Table 6.5

The second testing stage was to determine the performance impact on TCP throughput when including intermediate nodes within an application. Six applications were designed with an increasing number of intermediate nodes. Initially, these nodes contained no processing logic and simply piped all incoming traffic to the next processing element in the chain. Permutations $B \cdot C$ and $C \cdot D$ were selected from Equation (6.2) to perform tests on, which would determine the effects on intertile communication. Owing to the tiles of the selected XMOS microcontrollers supporting eight processors, only two free processors were available on Tile 1 and Tile 2. Tests exceeding two additional processing units will have the chain extended to Tile 0. The reduced number of available cores was due to the



(a) Empty Nodes



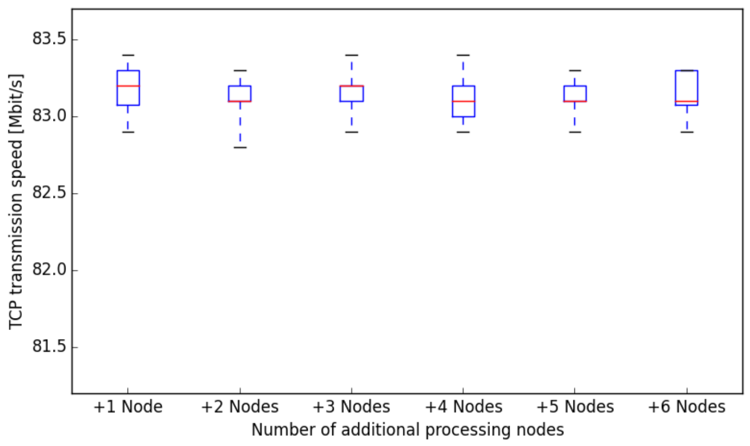
(b) Checksum Nodes

Figure 6.8: Recorded TCP Throughput for a Path Between Port C and Port D

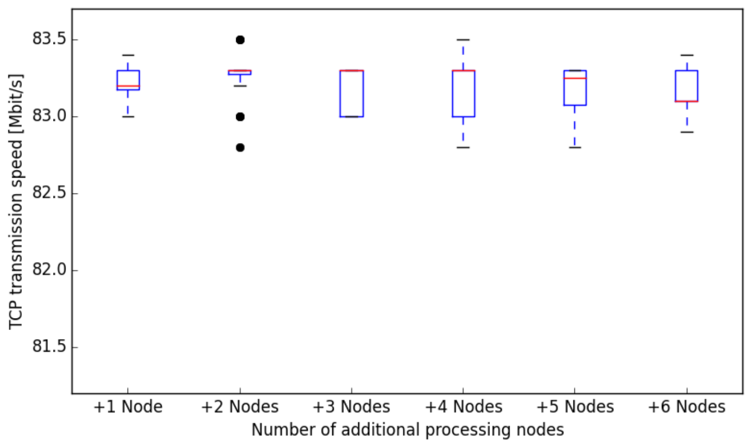
implementation of two port interface modules, each of which required three processors to operate. For these tests, all four port interface modules were instantiated so that the impact of offloading some processing to an adjacent XMOS tile could be observed.

Figure 6.8(a) and Figure 6.8(b) depict the recorded results of TCP throughput tests recorded on the functional prototype for permutations $B \cdot C$ and $C \cdot D$. Throughput tests for each plotted permutations were performed multiple times and the results are summarised as a box-and-whisker diagram per iteration. The top and bottom of each box presents the first and third quantiles of the recorded test results while the inner band presents the average recorded throughput. The whiskers present the maximum and minimum recorded TCP throughput for each iteration.

Figure 6.8 presents the recorded TCP throughput or communication between ports C and D. Increasing the number of intermediate cores added to the chain without any



(a) Empty Nodes



(b) Checksum Nodes

Figure 6.9: Recorded TCP Throughput for a Path Between Port B and Port C

additional processing logic is shown in Figure 6.8(a) while Figure 6.8(b) depicts the results of including some simple logic in each additional node. For these tests, the additional logic served to perform a complete recalculation of the IP checksum and replace the current checksum with the updated (though unchanged) value. For both graphs, increasing the number of intermediate nodes caused a gradual decrease in TCP throughput. This minor reduction in throughput was attributed to the increase in processing time related to the additional nodes. When comparing the overall performance of the two graphs, including the additional logic to recalculate the IP checksum, an average throughput reduction of 0.3 Mbit/s was recorded.

Figure 6.9 presents the same set of tests preformed in Figure 6.8 but with the network path instantiated between port B and port C. Recorded results are summarised in Figure 6.9 which indicates an overall throughput reduction of approximately 1 Mbit/s. This result correlates with the performance reduction seen in Table 6.6 for network throughput between ports B and C.

Unlike in Figure 6.8 however, Figure 6.9(a) and (b) shows a shallower decline in TCP throughput as additional nodes are added. Furthermore, the average throughput reduction incurred by the IP checksum logic is 53 Kbit/s which is 85% less than the recorded average difference when using ports C and D.

Raw Throughput

Relating TCP throughput to raw throughput was done by comparing the difference between the number of bytes transmitted onto the physical medium and the subset of these bytes relating to the TCP payload. Considering the actual bytes transmitted without VLAN tagging present, each frame begins with a preamble composed of seven bytes, which is followed by the start of a frame byte (Cisco Systems, 2003). Following this is a 14-byte frame header and frame payload of between 46 and 1500 bytes. The frame ends with a 4-byte checksum and an inter-packet gap of 12 bytes.

$$T_{raw} = T_{tcp} \times \frac{1538}{1588} \tag{6.3}$$

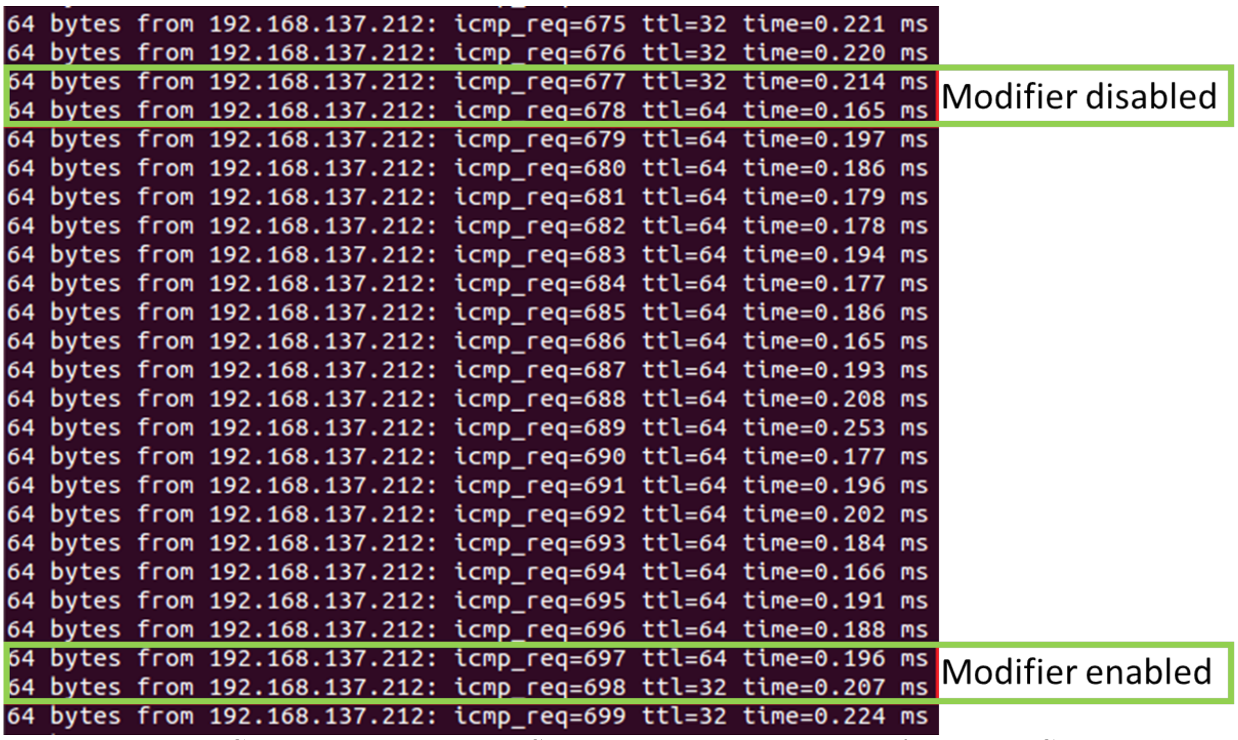
Assuming all 1500 bytes of the frame payload are used, a TCP throughput of 84.54 MBit/s requires a minimum of 6900 frames to be transmitted. Each of these frames includes an overhead of 38 bytes to be transmitted on the wire which, using Equation (6.3), results in a raw throughput of 86.68 MBit/s.

6.3.3 Module Enabling

As discussed in Section 4.9.1, a feature of the implemented system is to allow for the enabling and disabling of user-defined modules that form part of the active application. This feature allows for frame processing to be enabled or disabled without restarting the device and uploading a new application.

Testing of this feature was done using the same application described in Listing 6.2. All user-defined modules for the active application are listed in the enable tab on the control page shown in Appendix C.1.

The application was once again executed on the FRAME board with two hosts connected. The ping utility was used to send ICMP requests and responses through the platform



```
64 bytes from 192.168.137.212: icmp_req=675 ttl=32 time=0.221 ms
64 bytes from 192.168.137.212: icmp_req=676 ttl=32 time=0.220 ms
64 bytes from 192.168.137.212: icmp_req=677 ttl=32 time=0.214 ms
64 bytes from 192.168.137.212: icmp_req=678 ttl=64 time=0.165 ms
64 bytes from 192.168.137.212: icmp_req=679 ttl=64 time=0.197 ms
64 bytes from 192.168.137.212: icmp_req=680 ttl=64 time=0.186 ms
64 bytes from 192.168.137.212: icmp_req=681 ttl=64 time=0.179 ms
64 bytes from 192.168.137.212: icmp_req=682 ttl=64 time=0.178 ms
64 bytes from 192.168.137.212: icmp_req=683 ttl=64 time=0.194 ms
64 bytes from 192.168.137.212: icmp_req=684 ttl=64 time=0.177 ms
64 bytes from 192.168.137.212: icmp_req=685 ttl=64 time=0.186 ms
64 bytes from 192.168.137.212: icmp_req=686 ttl=64 time=0.165 ms
64 bytes from 192.168.137.212: icmp_req=687 ttl=64 time=0.193 ms
64 bytes from 192.168.137.212: icmp_req=688 ttl=64 time=0.208 ms
64 bytes from 192.168.137.212: icmp_req=689 ttl=64 time=0.253 ms
64 bytes from 192.168.137.212: icmp_req=690 ttl=64 time=0.177 ms
64 bytes from 192.168.137.212: icmp_req=691 ttl=64 time=0.196 ms
64 bytes from 192.168.137.212: icmp_req=692 ttl=64 time=0.202 ms
64 bytes from 192.168.137.212: icmp_req=693 ttl=64 time=0.184 ms
64 bytes from 192.168.137.212: icmp_req=694 ttl=64 time=0.166 ms
64 bytes from 192.168.137.212: icmp_req=695 ttl=64 time=0.191 ms
64 bytes from 192.168.137.212: icmp_req=696 ttl=64 time=0.188 ms
64 bytes from 192.168.137.212: icmp_req=697 ttl=64 time=0.196 ms
64 bytes from 192.168.137.212: icmp_req=698 ttl=32 time=0.207 ms
64 bytes from 192.168.137.212: icmp_req=699 ttl=32 time=0.224 ms
```

Figure 6.10: ICMP Response Listing Seen by the Initiating Host of the Ping Command

continuously. After allowing the application to run for a a duration of appropriately five minutes, the `ttl_decrementer` element on the enable tab was unchecked. The application was left in this state for a short duration so that the impact on the TTL could be recorded. After the application was allowed to continue for approximately 20 seconds, the box on the enable tab was again checked. The recorded output for this sequence of events is depicted in Figure 6.10, which lists the TTL response messages received. The highlighted fields show where the TTL value changes from the modified value of 32 to the unmodified value of 64 and vice versa. These transition points indicate when the user-defined module has been disabled and enabled again.

To show system logging of network traffic and frame specific characteristics, a series of live captures are presented in Appendix C.2. These captures were taken during testing of the system to show the prevalence of TCP, UDP, and ICMP traffic during operation.

6.3.4 Application Examples

Finally, to test operation of the complete system, two application examples were created with the problem domain related to network frame manipulation. A script was written in Link for each application to address the application requirements. This script was then

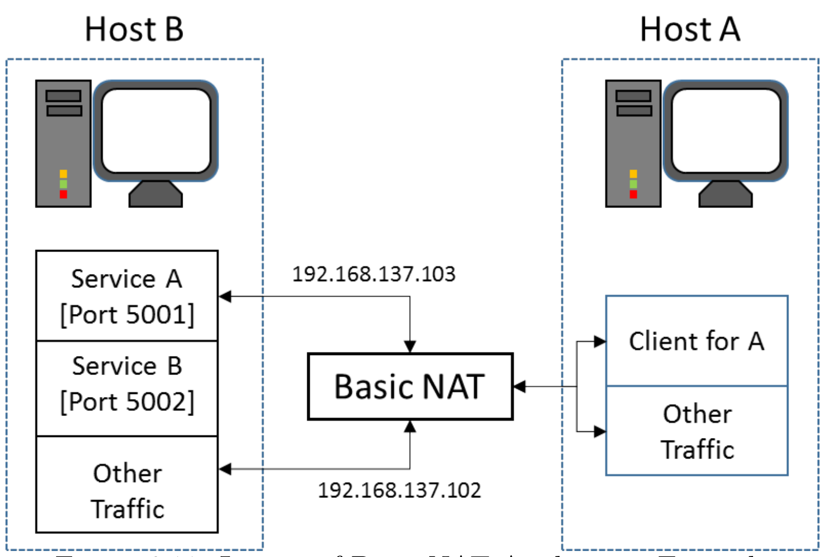


Figure 6.11: Layout of Basic NAT Application Example

compiled into an application compatible with the XMOS architecture and executed on the FRAME board.

6.3.5 Basic NAT

The first proposed application example was to perform basic Network Address Translation (NAT) during communication between two hosts. NAT is a common procedure used to translate the IP addresses of devices in a local network to be represented by a single outward facing IP address (Kurose and Ross, 2013). This procedure is often used when locally assigned IP addresses require Internet access. Though NAT usually refers to a many-to-one translation, basic NAT provides a one-to-one translation between two IP addresses (Srisuresh and Holdrege, 1999).

In this example two hosts communicated without interference except in a specific case as described in Figure 6.11. From the perspective of host A, there was only one other host on the network; host B. Host B provided two services that could be accessed by connected network devices. From the perspective of host B however, there were two other devices on the network, each with their own unique IP address.

To achieve this configuration, an application was developed that could selectively perform basic NAT on network traffic relating to a particular service on host B. To simplify the problem, each service opened a known port for communication. Service A operated on port 5001 and service B on port 5002. The application was designed to monitor traffic from host A for network frames addressed to port 5001 on host B. These frames had the source IP address fields translated to the fictitious host IP 192.168.137.103. To translate returning traffic from service A, network frames addressed to this fictitious host had the destination address translated to the IP address of host A.

```
//———Declaration Phase———
Modifier Recieve_Translator
{
  on Condition ( ip_destination == 192.168.137.103 )
  {
    ip_destination = 192.168.137.102;
    ip_update_checksum;
    tcp_update_checksum;
  }
  on Condition( arp_target_ip == 192.168.137.103 )
  {
    arp_target_ip = 192.168.137.102;
  }
}
Modifier Send_Translator
{
  on Condition ( tcp_destination_port == 5001 )
  {
    ip_source = 192.168.137.103;
    ip_update_checksum;
    tcp_update_checksum;
  }
  on Condition( arp_source_ip == 192.168.137.102 )
  {
    arp_source_ip = 192.168.137.103;
  }
}
//————Linking Phase————
PortA —> Recieve_Translator —> PortB;
PortB —> Send_Translator    —> PortA;
```

Listing 6.4: Basic NAT Application Source

The application associated with the designed application is shown in Listing 6.4. This application used two modifier modules to translate source and destination IP addresses of network traffic relating to service A. During testing it was found that some hosts would attempt to confirm the existence of the fictitious host by using an ARP request and would ignore the connection if no valid response was received. To remedy this, both modules

```
int process_Drop_Switch(Frame &frame)
{

  if(gen_random(100) <= 10)
  {
    return 0;
  }
  else
  {
    return 1;
  }
}
```

Listing 6.6: Function Not Operating Correctly

```
int process_Drop_Switch(Frame &frame)
{
  int random_num = gen_random(100);
  if(random_num <= 10)
  {
  return 0;
  }
  else
  {
  return 1;
  }
}
```

Listing 6.7: Function Operating Correctly

were extended to apply address translation to any ARP frame related to the fictitious IP address.

```
Server [Address, Port]: [192.168.137.100, 5001]
Client [Address, Port]: [192.168.137.103, 6828]
==================Payload====================
Client [Address, Port]: [192.168.137.102, 6828]
```

Listing 6.5: Server Application Output

Testing was performed using a minimal client server application, the code of which is given in Listing B.3. The server application recorded and printed the server address and port as well as the address of any connected client. The client would send its host address, which was also printed out by the server as part of the TCP payload. Figure 6.5 depicts the output of the server application, which records the client IP address to be 192.168.137.103 instead of 192.168.137.102.

6.3.6 Packet Loss

As a second application example, the introduction of packet loss to an ideal network was implemented. Packet loss is usually caused by network congestion, faulty hardware, or misconfiguration of network components and is often quoted as a percentage related to the number of packets dropped relative to the total number of packets transmitted.

The hardware configuration used in this example was two hosts communicating through the FRAME board described in Figure 6.12. For this example all traffic transmitted from host A to host B was unaltered while ICMP related traffic traversing in the opposite direction had a 10% probability of being dropped. Implementing this logic on the FRAME

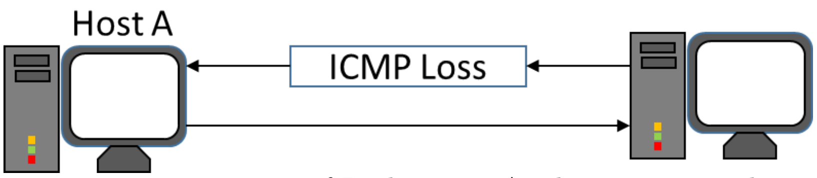


Figure 6.12: Layout of Packet Loss Application Example

board was achieved using a minimal application described in Listing 6.8.

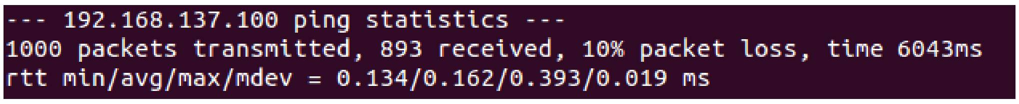
```
//———Declaration Phase———
Switch ICMP_drop
{
  on Condition (ip_protocol == ICMP)
  {
    on Random(10%) : FAIL;
  }
  Default : PASS;
}

//————Linking Phase————
PortA —> PortB;
PortB —> ICMP_drop<on PASS> —> PortA;
```

Listing 6.8: Basic NAT Application Source

During testing of the implemented application, unexpected behaviour was recorded where initial results indicated a 0% packet loss of network traffic transmitted through the platform. This behaviour implied that the random number test always evaluated to true and the frame was never dropped. Listing 6.6 presents a snippet of code generated from the implemented application. Further testing on this code revealed that the expression [`gen_random(100) ¡= 10`] always evaluated to true. Restructuring the conditional expression to operate as a two-stage operation depicted in Figure 6.7, resulted in the expected operation. The translator was updated to account for this behaviour by having the result of a random number generation stored in a temporary variable before comparison.

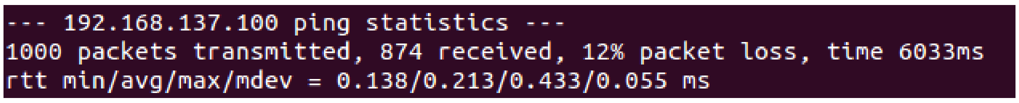
As discussed in Section 4.6, there are currently two variants of the random number generation: one deterministic, but suffering from bias, and the other eliminating bias but suffering from an unknown time cost. For this application example, both approaches were tested.

Using the ping utility, 1000 ICMP requests were performed between two hosts operating through the platform, Figure 6.13(a) depicts the results using the non-deterministic ap-



```
--- 192.168.137.100 ping statistics ---
1000 packets transmitted, 893 received, 10% packet loss, time 6043ms
rtt min/avg/max/mdev = 0.134/0.162/0.393/0.019 ms
```

(a) Recorded ICMP Results Using Non-Deterministic Random Number Generation



```
--- 192.168.137.100 ping statistics ---
1000 packets transmitted, 874 received, 12% packet loss, time 6033ms
rtt min/avg/max/mdev = 0.138/0.213/0.433/0.055 ms
```

(b) Recorded ICMP Results Using Deterministic Random Number Generation

Figure 6.13: Recorded Results for Traffic Loss Application Example

proach, which shows an expected 10% packet loss. To compare the biased and unbiased random generation approaches, the same test was performed using a deterministic approach. Figure 6.13(b) depicts the results of this test showing a 12% packet loss. When comparing the recorded average time taken by each of the tests over 1000 iterations, it is clear that, for an ICMP request, the overhead introduced by the non-deterministic random generation function was minimal with the maximum RTT recorded being less than its deterministic counterpart. This indicates that the time fluctuation resulting from other factors is more dominant when compared with the additional time induced by the random generation methods.

6.4 Summary

This chapter confirmed the functionality of the implemented system on a functional prototype constructed using existing hardware. Timing analysis of supporting functions was performed to determine the impact these could have on user applications. Once development of the FRAME board discussed in Chapter 5 had been completed, system testing was resumed on the new device. Network transmission using the FRAME board showed a recorded raw network throughput of 86.68 MBit/s. The chapter concluded with two application examples designed to confirm system functionality further.

7 Conclusion

The aim of this research was to develop a frame processing platform that could be used in conjunction with existing systems to augment environments set up to test network related applications. This chapter summarises the work performed as part of this research and draws conclusions from the results recorded in Chapter 6. Section 7.1 begins by reiterating the goals this research aimed to achieve before continuing with a brief summary of each of the three major components developed as part of the platform.

Section 7.2 discusses protocol support and implications of recorded timings associated with the supporting functions. A summary of the network throughput testing is discussed in Section 7.3, which highlights the peak raw throughput achieved by the implemented platform. The throughput summary is followed by an evaluation of the research goals presented in Section 7.1, before concluding the chapter with a discussion of future extensions to the platform in Section 7.5.

7.1 Key Aspects

The research goals, initially set out in Section 1.3, are the following:

- Selection of suitable hardware candidates, which when combined, produce a hardware platform for network frame processing.
- Proposal of a structure that can be used in conjunction with internal modules to produce applications compatible with the platform.
- Design and production of a prototype DSL to aid the user in application development related to the proposed platform.
- Investigation of the hardware design necessary to produce the FRAME board, which will act as the Network Layer hardware of the platform.
- Conformance testing of completed system and evaluation of application examples to confirm functionality.

In short, this research attempted to develop a platform for generic frame manipulation at wire speeds in the region of 100 MiB/s. Associated with this platform is the development of a suitable environment to simplify application development and a user interface for logging and monitoring purposes. The hardware components of the system should be relatively low cost to fabricate or use existing low cost devices.

The remainder of this section provides a brief summary of each of the three primary areas of development for this research. Application development covers the research related to the design and implementation of internal modules and DSL syntax. Following this, a summary of all hardware development leading to the fabrication and testing of the FRAME board is covered. The section concludes with a brief summary of the user interface and related development.

7.1.1 Application Development

During the initial phases of the research, approaches to providing the user with an intuitive and simple method for designing applications to operate on the system were discussed. It was concluded that the approach taken by this research would be to design and implement a DSL, which could be translated into a language compatible with the selected hardware.

Application design began in Section 3.5 after the appropriate hardware had been selected. The first stage was to determine how applications would be structured to take advantage of the XMOS architecture while accounting for requirements identified while evaluating application examples in Section 3.5.2. From this limited evaluation it was concluded that a series of internal modules would be implemented to handle communication with lower level components which could be called by the user-defined applications. Implementation of these individual modules was presented in in Section 4.3, which covered the development process and individual testing.

DSL development was covered in Section 4.8 where a module was defined. Through the evaluation of simple target applications, a basic set of notations was developed which could be used to express the functionality of a module. These notations were subsequently used to construct an interpreter targeted at the XMOS architecture. Rather than compile the semantic notations into a low-level language that could be executed natively by the XMOS architecture, the translator converted the semantic notations into equivalent XC code, which could be compiled using the internal modules as discussed in Section 4.9.1.

Testing of the designed semantic notations and associated translator was performed in conjunction with general system testing in Section 6.3, which concluded with application examples used to show platform functionality.

7.1.2 Hardware Development

In Section 3.3, an investigation into potential hardware architectures that could be used as part of the system was performed. After an initial evaluation, two candidates were proposed and investigated further. Considering component costs and development environments, the XMOS XS1 architecture was selected to act as the hardware base for the network processing layer of the system. A similar procedure was followed in Section 3.4 for the base layer underlying hardware. It was concluded that either the Intel Edison or Raspberry Pi could perform this role and so further development would be compatible with both.

Section 4.2 investigated and confirmed the feasibility of connecting multiple XMOS microcontrollers together to form an internal network for frame processing. This would allow the number of processing cores advertised by the underlying hardware to be extended between board revisions without requiring a complete rewrite of the associated software layer.

After consideration, communication between the Network Layer and the Base Layer was standardised to use an SPI interface in Section 4.3.2. By limiting communication to operate over a known protocol, the different hardware layers could be interchanged more easily.

As discussed in Section 5.2 an initial prototype of the underlying architecture for the

Network Layer was designed and fabricated. Section 5.2 covered an investigation of PCB characteristics and power supply design required to manufacture an operational device. Once the initial prototype was fabricated and tested, the design was extended in Section 5.4 to produce the FRAME board design. The FRAME board was fabricated and errors identified relating to the resulting device were discussed in Section 5.4.2. The final board is depicted in Appendix D.1.

7.1.3 User Interface

To maintain compatibility between the Raspberry Pi an Intel Edison, Section 3.5.3 concluded that the user interface should be constructed as a web-based application. Implementation of the described interface was discussed in Section 4.10 where the different components of the interface were presented.

7.2 Protocol Support

Section 4.9.3 discussed the design and implementation of supporting function calls for the detection and manipulation of protocol specific frame attributes. The implemented revision of the supporting libraries, which form part of the compiled applications, is capable of extracting header specific fields relating to the ARP, ICMP, TCP, and IP protocols. Testing of these functions was performed during implementation and further confirmed during general testing in Section 6.2.3.

Timings relating to these functions were recorded and presented in Section 6.2.4. These results indicate the performance impact some functions, such as recalculating a TCP checksum or modifying the MAC address, have on generated applications when compared with simply reading or writing a single integer. Such timing impacts should be considered when attempting to construct an efficient application to minimise the use of more costly operations. Considering Listing 6.4 in Section 6.3.5, the `Send_Translator` function incurs up to 9.024 μs additional delay on traffic transmitted from port B to port A owing to the included functions.

7.3 Throughput Evaluation

In Section 4.3.1, initial development of a port module was performed by extending the existing Ethernet module provided by XMOS to allow for bidirectional communication. The resulting module is capable of achieving 80.8 MBit/s which is a marked improvement on the initial 20 MBit/s throughput achieved with the initial implementation of the port module recorded in Section 4.3.1.

Through extensions to the initial implementation during research, Section 6.3.2 recorded an average TCP throughput of between 82.8 and 84.54 MBit/s depending on the path defined through the FRAME board while the functional prototype described in Section 6.2 recorded an average TCP throughput of 83.2 MBit/s.

7.4 Evaluation of Research Goals

This section aims to draw conclusions relating to the success or failure of this research by reflecting on the goals described in Section 1.3. As these goals were stated in numerical order, this section refers to each goal by its associated number.

1. Initial investigations were performed on a range of architectures before selecting two candidates for further evaluation. The XMOS architecture was finally selected as the underlying hardware of the Ethernet layer while both the Raspberry Pi and Intel Edison were selected to act as suitable platforms for the Base Layer.

2. Considering the multicore architecture provided by XMOS, application development focused on constructing a chain of nodes to perform network processing in incremental steps allowing for pipeline processing of network frames. Features such as the XMOS channels, timers, and ring oscillators were also incorporated into application design and implementation.

3. After consideration, a syntax was designed and used to describe network applications that would be compatible with the XMOS architecture. The syntax was intended to be both simple and expressive so as to be easily understood by the user. An interpretor was also designed as part of this research to translate applications written in the DSL into code that could be compiled for the hardware using the XMOS development toolchain. A user interface was also successfully implemented to allow system control and monitoring by the user.

4. An evaluation of stable board power supply and PCB characteristics that could affect signal propagation were investigated before the FRAME board was successfully fabricated.

5. Testing proved system functionality, however traffic manipulation at speeds of 100 Mbits/s was not achieved as discussed in Section 6.3.2, with a recorded raw throughput of only 86.68 Mbit/s, 13.31 Mbit/s below the 100 Mbit/s goal. Furthermore, increasing processing logic would further reduce throughput owing to the additional processing time incurred.

In closing, this research attempted to create a platform on which to perform frame manipulation at wire speeds of 100 MBit/s while considering ease of use and cost. While a system was successfully developed and tested, the achieved throughput is substantially lower than the proposed output. Ease of use was achieved through the use of a DSL, which is based on a syntax designed to express applications compatible with the system in a user friendly manner.

Section 5.4.1 recorded the component costs associated with the FRAME board totaling $185 per device. Additional costs include the Base Layer hardware of either a Raspberry Pi at $35.00 [1] or an Intel Edison at $69.00 [2], resulting in a total hardware cost of between $220 and $254 for the platform. It is noted that these costs will fluctuate depending on the price of the selected base platform and the number of boards to be fabricated.

7.5 Future Work

A future revision of the implemented FRAME board is highly recommended to account for errors identified in the current revision as detailed in Section 5.4.2. These errors have been accounted for in the design files made available as part of this research. Additional aspects to consider in future revisions are:

- An architecture upgrade from the current generation of XMOS microcontrollers to the new XS2 family (XMOS, 2015b). This newer generation supports an increased number of processing cores and is capable of traffic transmission at speeds up to 1 Gbit/s.

[1]Taken from Allied Electronics on 1 September 2015 http://www.alliedelec.com/raspberry-pi-raspberry-pi-model-b/70229569/

[2]Taken from Allied Electronics on 1 September 2015 http://www.alliedelec.com/intel-edi2bb-al-k/70526557/

- The addition of a secondary microcontroller responsible for controlling the on-board switching chips and potentially providing USB support for the FRAME board. USB would allow the FRAME board to be tethered to an existing host which could operate as the Base Layer of the system and allow for board integration into existing systems through the construction of an API library.
- Embed the base platform onto the FRAME board such that the underlying hardware for the entire platform is combined into a single device. This will reduce the risk of incorrectly wiring the separate components together, potentially damaging the hardware.
- Extend the number of both ports and processing cores on the FRAME board to allow the device to scale and support more complicated test bed configurations.

The DSL developed as part of this research is still limited with several potential areas for improvement. Extensions to the language are proposed as follows:

- As part of this research, a translator responsible for converting applications developed in the DSL into applications compatible with the XMOS development toolchain was implemented. A potential extension would be to construct additional translators targeting different architectures.
- During this research, support for the ARP, ICMP, IP, and TCP protocols were added to the DSL by constructing supporting functions responsible for reading and writing the relevant frame attributes. The language set should be extended to support more protocols including higher level protocols such as HTTP by constructing the required supporting functions in the target language.
- The current DSL does not allow the user to instantiate variables. This limitation restricts modules developed in the DSL to be stateless. Support for the creation of variables should be included to allow the user to include state such as counters, which could improve the range of application solutions the DSL can be used to describe.
- Addition of networking specific data structures such as IP tables would be of great use to the researcher when developing applications related to frame manipulation. Future work could incorporate such structures into the modules at the request of the user.

References

Aho, A., Lam, M., Sethi, R., and Ullman, J. *Compilers, Techniques, and Tools.* ISBN-10: 0321486811. Pearson Education, New York, USA, Second edition, September 2006.

Analog Devices. *Decoupling Techniques: What is Proper Decoupling and Why is it Necessary?* Tutorial Revision 0.03, Analog Devices, n.d. Accessed on: 11 April 2015. URL `http://goo.gl/094GgN`

Anwer, M. B., Motiwala, M., Tariq, M., and Feamster, N. *SwitchBlade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware. SIGCOMM Comput. Commun. Rev.*, 40(4):183–194, August 2010. ISSN 0146-4833. doi: 10.1145/1851275.1851206.

ARM. *Cortex-A8 Technical Reference Manual.* Technical Reference Revision: R2P1, ARM, November 2007. Accessed on: 18 Feb 2014. URL `http://goo.gl/nIQakV`

AVX Corperation. *Multilayer Ceramic Chip Capacitor.* Component Catalogue S-MCC20M299-C, AVX Corperation, n.d. Accessed on: 14 April 2015. URL `http://www.avx.com/docs/masterpubs/mccc.pdf`

Beili, E. *Definitions of Managed Objects for IEEE 802.3 Medium Attachment Units (MAUs).* Request for Comments 4836, Actelis Networks, April 2007. Accessed on: 8 November 2014. URL `https://tools.ietf.org/html/rfc4836`

Bentley, J. *Programming Pearls: Little Languages. Communications of the ACM*, 29(8):711–721, August 1986. ISSN 0001-0782. doi:10.1145/6424.315691.

Bonaventure, O. *Computer Networking: Principles, Protocols, and Practice.* ISBN: 978-1-304-97342-9. The Saylor Foundation, Washington, USA, October 2011.

Braden, R. *Requirements for Internet Hosts: Communication Layers.* Request for Comments 1122, Internet Engineering Task Force, October 1989. Accessed on: 8 November 2014.
URL `https://tools.ietf.org/html/rfc1122`

Broadcom. *Broadcom BCM2835 ARM Peripherals.* Device reference manual, Broadcom, Cambridge, UK, February 2012. Accessed on: 2 May 2014.
URL `http://goo.gl/pMBtbX`

Cisco Systems. *Internetworking Technologies Handbook.* Networking Technology. Cisco Press, Indianapolis, USA, Fourth edition, September 2003. ISBN: 1587051192.

Cleveland, J. and Uzgalis, R. *Grammars for Pogramming Languages.* ISBN 10: 0444001999. Elsevier Science Ltd, Amsterdam, Netherlands, December 1977.

Coley, G. *BeagleBone Black System Reference Manual.* System Reference Manual A4, Texas Instruments, January 2013. Accessed on: 18 February 2014.
URL `http://www.farnell.com/datasheets/1685587.pdf`

Collin, R. E. *Foundations of Microwave Engineering.* ISBN: 978-0-7803-6031-0. Wiley, New York, USA, Second edition, 2000.

Covington, G. A., Gibb, G., Lockwood, J. W., and Mckeown, N. *A packet generator on the NetFPGA platform. IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 235–238, April 2009. doi:10.1109/FCCM.2009.29.

Cozens, S. and Wainwright, P. *Beginning Perl.* ISBN: 978-1-118-01384-7. Wrox Press Ltd., Birmingham, UK, 2000.

Dibley, J. *An Investigation of the XMOS XSl Architecture as a Platform for Development of Audio Control Standards.* Masters thesis, Rhodes University, 2014.

Dubendorf, V. *Wireless Data Technologies.* John Wiley & Sons Ltd, Chichester, England, 2003. ISBN: 978-0-470-84949-1.

Elischer, J. and Cobbs, A. *All About Netgraph.* Application website, FreeBSD, 2005. Accessed on: 7 December 2014.
URL `http://people.freebsd.org/~julian/netgraph.html`

Erickson, R. W. *Power Converters.* John Wiley & Sons, Inc., 2001. ISBN 9780471346081.

Erickson, R. W. and Maksimovic, D. *Fundamentals of Power Electronics.* ISBN-10: 0792372700. Springer, Second edition, January 2001.

Flamm, J. *Best Practices for High Speed Digital PCB Design. International Cadence Usergroup Conference*, (24), 2004. Abstract Reference Number: 24. Accessed on: 6 April 2015.

Flashrom. *Flashrom 0.9.8.* Application site, Flashrom, March 2015. Accessed on: 14 May 2015.
URL `http://flashrom.org/Flashrom/0.9.8`

Forouzan, B. and Fegan, S. *Data Communications and Networking.* ISBN-10: 0073376221. McGraw-Hill, New York, USA, Fourth edition, 2007.

Foster, N., Harrison, R., Freedman, M. J., Monsanto, C., Rexford, J., Story, A., and Walker, D. *Frenetic: A Network Programming Language. ACM SIGPLAN Notices - ICFP 2011*, 46(9):279–291, September 2011. ISSN 0362-1340. doi:10.1145/2034574.2034812.

Franklin, R. W. *Equivalent Series Resistance of Tantalum Capacitors.* Tecnical information, AVX Limited, n.d. Accessed on: 2 April 2015.
URL `http://www.avx.com/docs/techinfo/eqtant.pdf`

Gill, H., Lin, D., Sarna, L., Mead, R., Lee, K. C., and Loo, B. T. *SP4: Scalable Programmable Packet Processing Platform. ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 75–76, August 2012. doi:10.1145/2342356.2342367.

Grune, D., Bal, H., Jacobs, C., and Langendoen, K. *Modern Compiler Design.* Springer, Second edition, 2012. ISBN 978-1-4614-4698-9.

Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. *NOX: Towards an Operating System for Networks. SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008. ISSN 0146-4833. doi:10.1145/1384609.1384625.

Gujarati, D. and Porter, D. *Basic Econometrics.* McGraw-Hill Education, 5th edition, October 2008. ISBN: 978-0073375779.

Gundavaram, S. *CGI Programming on the World Wide Web.* O'Reilly Media, 1996. ISBN: 978-1-56592-168-9.

Gwan-Su, K. and Hong-Hee, L. *A Study on IEC 61850 Based Communication for Intelligent Electronic Devices.* In *Science and Technology, 2005. KORUS 2005. Proceedings. The 9th Russian-Korean International Symposium*, ISBN: 0-7803-8943-3, pages 765–770. IEEE, June 2005. doi:10.1109/KORUS.2005.1507898.

Hall, S. H., Hall, G. W., and McCall, J. A. *High-Speed Digital System Design: A Handbook of Interconnect Theory and Design Practices.* ISBN: 978-0-471-36090-2. Wiley, October 2000.

Holdener, A. *Ajax The Definitive Guide.* ISBN 978-0-596-52838-6. O'Reilly Media, Inc, First edition, January 2008.

IEC. *Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model.* International Standard 7498-1, International Electrotechnical Commission, June 1996. Accessed on: 25 February 2015.
URL `http://goo.gl/nmjkGD`

IEEE. *802.3 Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications.* Standard 802.3, IEEE, New York, USA, March 2002. Accessed on: 22 January 2015.
URL `http://goo.gl/vHqvYP`

IEEE. *IEEE Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams. IEEE Std 802.1Qav-2009 (Amendment to IEEE Std 802.1Q-2005).* Standard, IEEE, May 2009. Accessed on: 22 February 2015.
URL `http://goo.gl/22mwd0`

IEEE. *IEEE Standard for Ethernet.* Standard, IEEE, 3 Park Avenue, New York, USA, December 2012. Accessed on: 2 February 2015.
URL `http://goo.gl/CMOcHa`

Intel. *Intel Edison Breakout Board - Hardware Guide.* Hardware Guide 004, Intel, October 2014a. Accessed on: 4 December 2014.
URL `https://communities.intel.com/docs/DOC-23252`

Intel. *Intel Edison Kit for Arduino - Hardware Guide.* Hardware Guide 005, Intel,

December 2014b. Accessed on: 4 December 2014.
URL `https://communities.intel.com/docs/DOC-23161`

Intel. *Product Brief: Intel Edison.* Product brief, Intel, 2014c. Accessed on: 11 December 2014.
URL `http://download.intel.com/support/edison/sb/edison_pb_331179001.pdf`

ISO. *Information Technology – Syntactic Metalanguage – Extended BNF.* Standard, International Organization for Standardization, December 1996. Accessed on: 6 May 2014.
URL `http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf`

ISSI. *IS42S16400F Synchronous Dynamic RAM* . Component Reference Manual A, Integrated Silicon Solution, Inc., March 2008. Accessed on: 7 May 2015.
URL `http://www.issi.com/WW/pdf/42S16400F.pdf`

Jain, M., Prasad, R. S., and Dovrolis, C. *The TCP Bandwidth-Delay Product Revisited: Network Buffering, Cross Traffic, and Socket Buffer Auto-Sizing.* Technical report, Georgia Institute of Technology, 2003. Accessed on: 15 May 2015.
URL `http://hdl.handle.net/1853/5920`

Kazimierczuk, M. K. *Pulse-width Modulated DC-DC Power Converters.* ISBN: 978-0-470-77301-7. Wiley, September 2008.

Knežević, N., Schubert, S., and Kostić, D. *Towards a Cost-effective Networking Testbed. ACM SIGOPS Operating Systems Review*, 43(4):66–71, January 2010. ISSN 0163-5980. doi:10.1145/1713254.1713269.

Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. *The Click Modular Router. ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, August 2000. ISSN 0734-2071. doi:10.1145/354871.354874.

Kurose, J. and Ross, K. *Computer Networking, A Top-Down Approach.* Pearson, Sixth edition, 2013. Accessed on March 2014.

Kursun, V., Narendra, S., De, V., and Friedman, E. *Analysis of Buck Converters for On-ChipOn-CIntegration with a Dual Supply Voltage Microprocessor. Very Large Scale Integration (VLSI) Systems, IEEE Transactions*, 11(3):514–522, June 2003. ISSN 1063-8210. doi:10.1109/TVLSI.2003.812289.

Lockwood, J. W., McKeown, N., Watson, G., Gibb, G., Hartke, P., Naous, J., Raghuraman, R., and Luo, J. *NetFPGA - An Open Platform for Gigabit-rate*

Network Switching and Routing. IEEE International Conference on Microelectronic Systems Education, June 2007. doi:10.1109/MSE.2007.69.

Loo, B. T., Condie, T., Garofalakis, M., Gay, D. E., Hellerstein, J. M., Maniatis, P., Ramakrishnan, R., Roscoe, T., and Stoica, I. *Declarative Networking. Communications of the ACM*, 52(11):87–95, Novemeber 2009. ISSN 0001-0782. doi: 10.1145/1592761.1592785.

Louden, K. C. *Compiler Construction Principles and Practice.* PWS Publishing Company, 1997. ISBN: 978-0534939724.

Martins, G., Moses, A., Rutherford, M. J., and Valavanis, K. P. *Enabling Intelligent Unmanned Vehicles Through XMOS Technology. The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, 9(1):71–82, November 2012. doi:10.1177/1548512910388197.

May, D. *XMOS XS1 Architecture.* XMOS Limited, First edition, September 2009. Accessed on: February 2014.
URL `https://www.xmos.com/published/xs1_en`

May, D. and Shepherd, R. *OCCAM and the Transputer.* In *Proceedings of the IFIP WG 10.3 Workshop on Concurrent Languages in Distributed Systems: Hardware Supported Implementation*, pages 19–33. Elsevier North-Holland, Inc., New York, USA, 1985. ISBN 0-444-87635-9. ISBN:0-444-87635-9.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. *OpenFlow: Enabling Innovation in Campus Networks. ACM SIGCOMM Computer Communication Review*, 38(2):69–74, April 2008. ISSN 0146-4833. doi:10.1145/1355734.1355746.

Mernik, M., Heering, J., and Sloane, A. M. *When and How to Develop Domain-specific Languages. ACM Computing Surveys*, 37(4):316–344, December 2005. ISSN 0360-0300. doi:10.1145/1118890.1118892.

Microsemi Corporation. *IEEE Standard 1149.1 (JTAG) in the SX/RTSX/SXA/eX/RT54SX-S Families.* Application Note AC160, Microsemi Corporation, May 2012. Accessed on: 25 February 2015.
URL `http://goo.gl/Jw3CxY`

Moncur, M. *Teach Yourself JavaScript in 24 Hours.* Sams Publishing, Carmel, USA, Second edition, 2000. ISBN: 978-0672336089.

Montz, A. B., Mosberger, D., O'Malley, S. W., Peterson, L. L., Proebsting, T. A., and Hartman, J. H. *Scout: a Communications-Oriented Operating System.* In *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V) (Cat. No.95TH8059)*, pages 58–61. May 1994. doi:10.1109/HOTOS.1995.513455.

Muthukumar, S. C., Li, X., Liu, C., Kopena, J. B., Oprea, M., and Thau Loo, B. *Declarative Toolkit for Rapid Network Protocol Simulation and Experimentation.* In *SIGCOMM (demo)*. SIGCOMM, ACM, Barcelona, Spain, August 2009.

Naous, J., Gibb, G., Bolouki, S., and McKeown, N. *NetFPGA: Reusable Router Architecture for Experimental Research.* In *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '08, pages 1–7. ACM, New York, USA, 2008. ISBN 978-1-60558-181-1. doi:10.1145/1397718.1397720.

ON Semiconductor. *LC Selection Guide for the DC DC Synchronous Buck Converter.* Applicaion Note 9135, ON Semiconductor, Literature Distribution Center for ON Semiconductor. Colorado, USA, April 2013.
URL `http://www.onsemi.com/pub_link/Collateral/AND9135-D.PDF`

ON Semiconductor. *Switch-Mode Power Supply Reference Manual.* Component Reference Manual Revision 4, ON Semiconductor, Denver, Colorado, USA, April 2014. Accessed on: 7 April 2015.
URL `http://www.onsemi.com/pub_link/Collateral/SMPSRM-D.PDF`

Patti, R. E. *EBNF: A Notation to Describe Syntax.* Online Book Chapter 2, October 2005. Accessed on: 15 May 2014.
URL `http://www.cs.cmu.edu/~pattis/misc/ebnf.pdf`

PCB Pool. *PCB-Pool Information: Technique.* Website, Beta Layout, Cape Town, South Africa, 2015. Accessed on: 5 April 2015.
URL `https://www.pcb-pool.com/ppza/info_technique.html`

Pennefather, S. and Irwin, B. *Design of a Network Packet Processing Platform.* In *Southern Africa Telecommunication Networks and Applications Conference (SATNAC) 2014* , pages 143–148. SATNAC, August 2014. In proceedings.

Pennefather, S. and Irwin, B. *Design and Fabrication of a Low Cost Traffic Manipulation Hardware Platform.* In *Southern Africa Telecommunication Networks and Applications Conference (SATNAC) 2015*. SATNAC, 2015. In proceedings.

Popovic, Z. and Popovic, B. D. *Introductuctory Electromagnetics.* ISBN-10:

0201326787. Prentice Hall, 2000.
URL http://goo.gl/6QGX0y

Popovich, M., Mezhiba, A., Köse, S., and Friedman, E. *Power Distribution Networks with On-Chip Decoupling Capacitors.* ISBN 978-1-4419-7870-7. Springer-Verlag, New York, Second edition, 2011.

QuadTech. *Equivalent Series Resistance (ESR) of Capacitors.* Application note, QuadTech Incorporated, July 2003. Accessed on: 5 April 2015.
URL http://www.low-esr.com/QT_LowESR.pdf

Rashid, M. H. *Power Electronics Handbook. Devices, Circuts, and Applicaions.* ISBN: 978-0-12-382036-5. Elsevier, Burlington, USA, 3rd edition, 2011.

Raspberry Pi Foundataion. *Raspberry Pi Model B+ Specification Sheet.* Technical flyer, Raspberry Pi Foundataion, n.d. Accessed on: 20 February 2015.
URL https://www.adafruit.com/datasheets/pi-specs.pdf

Rhodes, B. and Goerzen, J. *Foundations of Python Network Programming: The comprehensive Guide to Building Network Applications with Python.* Apress, Second edition, 2010. Accessed on: 11 April 2014.

Richtek. *RT8250: 3A, 23V, 340kHz Synchronous Step-Down Converter.* Component Reference Manual DS8250-03, Richtek, Hsintien City, Taiwan, July 2010. Accessed on: 5 April 2015.
URL http://goo.gl/zC1A74

Richter, H. *Noncorrecting Syntax Error Recovery. ACM Transactions on Programming Languages and Systems*, 7(3):478–489, July 1985. ISSN 0164-0925. DOI: 10.1145/3916.4019.

Ricketts, D. and McNeill, J. *The Designer's Guide to Jitter in Ring Oscillators.* ISBN: 978-0-387-76526-6. Springer Science and Business Media, New York, USA, 2009.

Semtech. *RF Design Guidelines: PCB Layout and Circuit Optimization* . Applicaion Note 1200.04, Semtech, 2006. Accessed on: 29 April 2015.
URL http://www.semtech.com/images/datasheet/rf_design_guidelines_semtech.pdf

Sharawi, M. S. *Practical issues in high speed PCB design. IEEE Potentials Magazine*, pages 24–27, April 2004.
URL http://web.cecs.pdx.edu/~greenwd/PCB_SI.pdf

Sheffield Learning Community. *Raspberry Pi.* page 1, 2013. Accessed on: April 2014. URL http://goo.gl/MW5Q8H

Sheng, Z. and Varadan, V. V. *Tuning the Effective Properties of Metamaterials by Changing the Substrate Properties.* *Journal of Applied Physics*, 101(1), 2007. ISSN 00218979.

Sherwood, R., Gibb, G., Yap, K.-K., Appenzeller, G., Casado, M., McKeown, N., and Parulkar, G. *Can the Production Network be the Testbed?* In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6. USENIX Association Berkeley, Berkeley, CA, USA, 2010. Accessed on: 5 March 2014.
URL http://dl.acm.org/citation.cfm?id=1924943.1924969

Silicon Labs. *AN203: 8-bit MCU Printed Circuit Board Design Notes.* Applicaion Note 203, Silicon Laboratories, Austin, USA, n.d. Accessed on: 11 April 2015.
URL https://goo.gl/NMNsbJ

Simpson, C. *Linear and Switching Voltage Regulator Fundamentals.* Power Management Applications SNVA558, Texas Instruments, Dallas, USA, 2011. Accessed on: 16 April 2015.
URL http://www.ti.com/lit/an/snva558/snva558.pdf

SMSC. *LAN9512/LAN9512i - USB 2.0 Hub and 10/100 Ethernet Controller* . Component Reference Manual, SMSC, February 2012. Accessed on: 20 Feb 2015.
URL http://goo.gl/xf1bEl

Srisuresh, P. and Holdrege, M. *IP Network Address Translator (NAT) Terminology and Considerations.* Request for Comments 2663, Lucent Technologies, August 1999. Accessed on: 30 July 2015.
URL https://tools.ietf.org/html/rfc2663

Terry, P. *Compiling with C# and Java.* Pearson Education, Boston, USA, First edition, October 2005. ISBN: 978-0321263605.

Texas Instruments. *Switching Regulators.* Technical Report, Texas Instruments, 2012. Accessed on: September 2015.
URL http://www.ti.com/lit/an/snva559/snva559.pdf

Texas Instruments. *DP83848C PHYTER Commercial Temperature Single Port 10/100 Mb/s Ethernet Physical Layer Transceiver.* Component Reference Manual SNLS266E,

Texas Instruments, March 2015. Accessed on: 11 April 2015.
URL `http://www.ti.com/lit/ds/symlink/dp83848i.pdf`

Texus Instruments. *SN74CBTLV3257 Low-Voltage 4-Bit 1-of-2 Multiplexer/Demultiplexer*. Component Reference Manual, Texus Instruments, October 2007. Accessed on May 2014.
URL `http://goo.gl/ApborU`

Thau, D. *The Book of JavaScript*. No Starch Press, 2000. ISBN: 978-1593271060.

The Open Group. *memcpy*. Function Specification 7, The Open Group Base Specifications, 2013. Accessed on: 15 June 2015.
URL `http://pubs.opengroup.org/onlinepubs/9699919799/`

Thierauf, S. C. *High-Speed Circuit Board Signal Integrity*. Artech House, Norwood, MA, USA, 2004. ISBN: 978-1580531313.

Tirumala, A., Qin, F., Dugan, J., Ferguson, J., and Gibbs, K. *Iperf: The TCP/UDP Bandwidth Measurement Tool*. Applicaion website, Lawrence Berkeley National Laboratory, 2005. Accessed on: 3 April 2015.
URL `https://iperf.fr/`

van Deursen, A. and Klint, P. *Little Languages: Little Maintenance*. *Journal of Software Maintenance: Research and Practice*, 10(2):75–92, March 1998. ISSN 1040-550X.
URL `http://homepages.cwi.nl/~paulk/publications/JSM98.pdf`

van Deursen, A., Klint, P., and Visser, J. *Domain-specific Languages: An Annotated Bibliography*. *SIGPLAN Notices*, 35(6):26–36, June 2000. ISSN 0362-1340. doi:10.1145/352029.352035.

Vaucourt, C. *Choosing Inductors and Capacitors for DC/DC Converters*. Applicaion Report SLVA157, Texas Instruments, February 2004. Accessed on: 5 April 2015.
URL `http://www.ti.com/lit/an/slva157/slva157.pdf`

Voellmy, A., Agarwal, A., Hudak, P., Feamster, N., and Burnett, S. *Don't Configure the Network, Program It! Domain-Specific Programming Languages for Network Systems*. Research Report YALEU/DCS/RR-1432, Yale University, July 2010.

Voellmy, A. and Hudak, P. *Nettle: Taking the Sting out of Programming Network Routers*. In *Proceedings of the 13th International Conference on Practical Aspects of*

Declarative Languages, PADL'11, pages 235–249. Springer-Verlag, Berlin, Heidelberg, 2011. ISBN 978-3-642-18377-5.

Voellmy, A., Kim, H., and Feamster, N. *Procera: A Language for High-level Reactive Network Control.* In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, number 12 in HotSDN, pages 43–48. ACM, New York, NY, USA, 2012. ISBN 978-1-4503-1477-0. doi:10.1145/2342441.2342451.

Wain, R., Bush, I., Guest, M., Deegan, M., Kozin, I., and Kitchen, C. *An overview of FPGAs and FPGA programming at Daresbury.* Technical Report 2.0, Computational Science and Engineering Department, CCLRC Daresbury Laboratory, Cheshire, UK, November 2006. Accessed on: 18 February 2014.
URL `http://goo.gl/zogrya`

Watson, G., McKeown, N., and Casado, M. *NetFPGA: A Tool for Network Research and Education. 2nd Workshop on Architecture Research using FPGA Platforms (WARFP)*, November 2009.
URL `http://yuba.stanford.edu/~casado/watson-stanford.pdf`

Watt, D. *Programming XC on XMOS Devices.* ISBN: 978-1-907361-03-6. XMOS Limited., September 2009. Accessed on: 1 Novemeber 2014.

Wehrle, K. *Peer-to-Peer Systems and Applications*, volume 3485. Springer Science & Business Media, 2005. doi:10.1007/11530657. ISBN: 978-3-540-29192-3, DOI: 10.1007/11530657.

Weiler, A. and Pakosta, A. *High-Speed Layout Guidelines.* Application Report SCAA082, Texas Instruments, November 2006. Accessed on: 14 April 2015.
URL `http://www.ti.com/lit/an/scaa082/scaa082.pdf`

Wens, M. and Steyaert, M. *Design and Implementation of Fully-Integrated Inductive DC-DC Converters in Standard CMOS.* Analog Circuits and Signal Processing. Springer Netherlands, 2011. ISBN: 978-94-007-1435-9.

Wilhelm, R., Seidl, H., and Hack, S. *Compiler Design, Syntactic and Semantic Analysis.* Springer Berlin Heidelberg, July 2013.

Xilinx. *ISE In-Depth Tutorial.* Tutorial UG695, Xilinx, September 2010. Accessed on: 17 February 2014.
URL `http://goo.gl/U6D09c`

Xilinx. *Spartan-6 Family Overview.* Product Specification DS160, Xilinx, October 2011.

Accessed on: 17 February 2014.
URL `http://goo.gl/VkyNqk`

Xilinx. *Constraints Guide.* Application Guide, Xilinx, April 2013. Accessed on: 18 February 2014.
URL `http://goo.gl/atZqBF`

XMOS. *XTAG-2 Hardware Manual.* Device Reference Manual Version 1.0, XMOS, October 2009. Accessed on: 3 April 2014.
URL `http://goo.gl/xwf15u`

XMOS. *XS1 Port I/O Timing Application Note.* Application Note Revision 1.0, XMOS, February 2010. Accessed on: 17 February 2014.
URL `http://goo.gl/jDdbvB`

XMOS. *XMOS SDRAM Component.* Reference Document Revision A, October 2012a. Accessed on: 14 Novemeber 2014.
URL `https://goo.gl/1Sa14x`

XMOS. *XS1-L01A-TQ128 Datasheet.* Component Reference Manual, XMOS, October 2012b. Accessed on: 16 February 2014.
URL `https://goo.gl/3lv65y`

XMOS. *sliceKIT hardware manual.* Device Hardware Manual Revision A, XMOS, November 2013a. Accessed on: 23 April 2014.
URL `https://goo.gl/IgkwDb`

XMOS. *startKIT Hardware Manual.* Device Hardware Manual Revision A, XMOS, October 2013b. Accessed on: 16 April 2014.
URL `https://goo.gl/KUYd4t`

XMOS. *xCONNECT Architecture.* Technical Information Revision A, XMOS, November 2013c. Accessed on: 18 February 2014.
URL `https://goo.gl/TSW7dU`

XMOS. *xCORE : Architecture Overview.* Technical information, XMOS, 2013d. Accessed on: 17 February 2014.
URL `http://goo.gl/XWyWnc`

XMOS. *XMOS Layer 2 Ethernet MAC Component.* Reference Document Revision A, XMOS, January 2013e. Accessed on: 20 February 2014.
URL `https://goo.gl/M2HBUA`

XMOS. *XMOS Layer 2 Ethernet MAC Component.* Applicaion Reference Manual Version: 2.2.4rc0.a, XMOS, July 2013f. Accessed on: 15 May 2015.
URL `https://goo.gl/6IyYFk`

XMOS. *XN Specification.* Configuration Document, XMOS, November 2013g. Accessed on: 15 April 2014.
URL `https://goo.gl/ae9DfF`

XMOS. *XS1-L16A-128-QF124 Datasheet.* Component Reference Manual, XMOS, December 2013h. Accessed on: 28 April 2014.
URL `https://goo.gl/QdSx9J`

XMOS. *XS1-L4A-64-TQ48 Datasheet.* Component Reference Manual, XMOS, December 2013i. Accessed on: 19 April 2014.
URL `https://goo.gl/zRcZqB`

XMOS. *XMOS Programming Guide.* Programming Manual XM004440A, XMOS, October 2014. Accessed on: 18 February 2014.
URL `https://goo.gl/750nBc`

XMOS. *Random numbers on the XS1-L1.* Application Note AN01007, XMOS, March 2015a. Accessed on: 16 March 2015.
URL `https://goo.gl/qHJvOm`

XMOS. *xCORE-200: The XMOS XS2 Architecture.* Technical Manual 1, XMOS, April 2015b. Accessed on: 8 April 2015.

XMOS. *XS1-L8A-64-TQ128 Datasheet.* Component Reference Manual, XMOS, April 2015c. Accessed on: 18 April 2015.
URL `http://www.farnell.com/datasheets/1738846.pdf`

XMOS. *sliceKIT Selector Guide.* Technical Flyer XM-002177-PC, XMOS, n.d.a. Accessed on: 14 April 2015.
URL `https://goo.gl/5a49Tn`

XMOS. *xTIMEcomposer Design Tools Suite.* Promotional article, XMOS, n.d.b. Accessed on: 18 February 2014.
URL `https://goo.gl/r8iGcf`

Zhang, H. *Basic Concepts of Linear Regulator and Switching Mode Power Supplies.* Applicaion Note AN140-1, Linear Technology, October 2013. Accessed on: 2 April 2015.

Appendices

A

URL Redirects

Throughout this document, references have been made to electronic sources either as footnotes or in the reference listing using their URL. Due to the length of some URLs, a URL shorting service was employed. This appendix records each URL shortened using the Google URL shortening service, accessible at: https://goo.gl/. The table provided in this appendix associates each shortened URL with the original full length URL for the reader.

http://goo.gl/nIQakV	http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344d/DDI0344D_cortex_a8_r2p1_trm.pdf
http://goo.gl/pMBtbX	http://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf
http://goo.gl/ExFzzn	http://www.digikey.com/product-detail/en/XC6SLX9-2TQG144C/122-1745-ND/2339919
http://goo.gl/XRmUF1	http://www.digikey.com/product-detail/en/XS1-L8A-64-TQ128-C5/880-1019-ND/2333460
https://goo.gl/a1BkMH	https://www.cadence.com/rl/Resources/conference_papers/tp_bestpractices_flamm.pdf
http://goo.gl/Q56Wuj	http://sjo.cn/downloads/ebook/High-Speed\%20Digital\%20System\%20Design.pdf
http://goo.gl/ApborU	http://docs-europe.electrocomponents.com/webdocs/0b80/0900766b80b80a01.pdf
http://goo.gl/MW5Q8H	http://www.sheffieldlearningcommunity.com/sites/default/files/uploads/Raspberry\%20Pi.pdf\#page=2
http://goo.gl/6QGX0y	http://www.academia.edu/9398817/INTRODUCTORY_ELECTROMAGNETICS
http://goo.gl/CMOcHa	http://www.trincoll.edu/Academics/MajorsAndMinors/Engineering/Documents/IEEE\%20Standard\%20for\%20Ethernet.pdf
http://goo.gl/zogrya	http://www.mecatronica.eesc.usp.br/wiki/upload/b/bd/An_overview_of_FPGAs_and_FPGA_programming.pdf
http://goo.gl/atZqBF	http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/cgd.pdf
http://goo.gl/VkyNqk	http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf
http://goo.gl/U6D09c	http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_4/ise_tutorial_ug695.pdf
https://goo.gl/r8iGcf	https://www.xmos.com/en/download/public/xTIMEcomposer-Flyer(1.0)pdf
https://goo.gl/750nBc	https://www.xmos.com/download/public/XMOS-Programming-Guide-(documentation)(E).pdf

https://goo.gl/IgkwDb	https://www.xmos.com/download/public/sliceKIT-Hardware-Manual(1.0).pdf
https://goo.gl/KUYd4t	https://www.xmos.com/en/download/public/startKIT-Hardware-Manual(1.0).pdf
https://goo.gl/TSW7dU	https://www.xmos.com/en/download/public/xCONNECT-Architecture(1.0).pdf
http://goo.gl/XWyWnc	https://www.xmos.com/download/public/xCORE-Architecture(X9650A).pdf
https://goo.gl/M2HBUA	https://www.xmos.com/download/public/XMOS-Layer-2-Ethernet-MAC-Component-(documentation)(2.2.2rc0.a).pdf
https://goo.gl/QdSx9J	https://www.xmos.com/download/public/XS1-L16A-128-QF124-Datasheet(X8006A).pdf
https://goo.gl/zRcZqB	https://www.xmos.com/download/public/XS1-L4A-64-TQ48-Datasheet(X2612E).pdf
https://goo.gl/1Sa14x	https://www.xmos.com/download/public/XMOS-SDRAM-Component-(documentation)(1.0.1rc0.a).pdf
https://goo.gl/3lv65y	https://www.xmos.com/download/public/XS1-L01A-TQ128-Datasheet(X1154A).pdf
http://goo.gl/jDdbvB	http://cdn.sparkfun.com/datasheets/Components/General\%20IC/XS1-Port-I-O-TimingX5821A.pdf
http://goo.gl/xwf15u	http://www.xmos.com/download/public/XTAG-2-Hardware-Manual(1.0).pdf
http://goo.gl/kWG3p2	http://www.avtic.com/attachments/files/a088afad33f6fdab9c884b0c16925987/df40-full-documentation.pdf
https://goo.gl/NMNsbJ	https://www.silabs.com/Support\%20Documents/TechnicalDocs/AN203.pdf
https://goo.gl/5a49Tn	https://www.xmos.com/download/private/sliceKIT-Selector-Sheet(4.0).pdf
https://goo.gl/qHJvOm	https://www.xmos.com/download/public/AN01007\%3A-Random-numbers-on-the-XS1-L1\%281.0.0\%29.pdf
https://goo.gl/6IyYFk	https://www.xmos.com/support/boards?version=latest\&product=15830\&component=15949\&page=0

http://goo.gl/Jw3CxY	http://www.microsemi.com/document-portal/doc_view/130050-ac160-ieee-standard-1149-1-jtag-in-the-sx-rtsx-sx-a-ex-rt54sx-s-families-app-note
http://goo.gl/nmjkGD	http://www.ecma-international.org/activities/Communications/TG11/s020269e.pdf
http://goo.gl/vHqvYP	http://www.jnltech.com/CST415Web/Project/802.3-2002.pdf
http://goo.gl/094GgN	http://www.analog.com/media/en/training-seminars/tutorials/MT-101.pdf
http://goo.gl/22mwd0	http://standards.ieee.org/findstds/standard/802.1Qav-2009.html
http://goo.gl/zC1A74	http://www1.futureelectronics.com/doc/RICHTEK/RT8250GSP.pdf
http://goo.gl/xf1bEl	http://wolfpaulus.com/wp-content/uploads/2012/08/LAN9512.pdf
http://goo.gl/22Bie5	https://www.xmos.com/support/tools/documentation?subcategory=&component=14787
http://goo.gl/WVV8aF	https://www.xmos.com/support/tools/documentation?subcategory=&component=14801
http://goo.gl/gU1TP2	https://download.xmos.com/XM-002177-PC-4.pdf?auth=WzAsIjEwNS4yMTAuxNzctUEMtNC5wZGYiXQ==,
http://goo.gl/4wCH9O	http://www.researchgate.net/publication/255960327_Dont_configure_the_network_program_it!_Domain-specific_programming_languages_for_network_systems

B
Code Listings

Throughout this document, code listings are used describe application functionality and to highlight features and errors identified during development. The majority of these listings are suitably small to be presented inline with the relevant text without disrupting document flow. Some listings which are required to address areas to the research were considered too large to place inline and are instead presented in this appended and are referred to from the relevant bodies of text.

B.1 Listing A

```
#include <timer.h>
#include <platform.h>
#include <print.h>
#define timeout 1000000 //An arbitrary duration

int main(void)
{
  chan p[4];            //An array of four channels
  par
  {    //All blocks of code in a par block are executed in parallel
    {  //Code within this block is executed sequentially
      p[0] <: 'a';
      p[1] <: 'b';
      p[2] <: 'c';
      p[3] <: 'd';
    }
    { //Code within this block is executed sequentially
      int time, no_timeout = 1;
      char msg;
      timer t;
      t :> time;
      while(no_timeout) select {
        case p[0] :> msg : printcharln(msg);
        break;
        case p[1] :> msg : printcharln(msg);
        break;
        case p[2] :> msg : printcharln(msg);
        break;
        case p[3] :> msg : printcharln(msg);
        break;
        case t when timerafter ( time + timeout ) :> void :
        printstrln("No event occured before timeout.");
        no_timeout = 0;
        break;
      }
    }
  }
  return 0;
}
```

Listing B.1: XC Parallel Execution Without Replicator

B.2 Listing B

```
#include <timer.h>
#include <platform.h>
#include <print.h>
#define timeout 1000000 //An arbitrary duration

int main(void)
{
  chan p[4];            //An array of four channels
  par
  {    //All blocks of code in a par block are executed in parallel
    {  //Code within this block is executed sequentially
      p[0] <: 'a';
      p[1] <: 'b';
      p[2] <: 'c';
      p[3] <: 'd';
    }
    { //Code within this block is executed sequentially
      int time, no_timeout = 1;
      char msg;
      timer t;
      t :> time;
      while(no_timeout) select {
        case (int i = 0; i < 4; i++) p[i] :> msg : printcharln(msg);
        break;
        case t when timerafter ( time + timeout ) :> void :
        printstrln("No event occured before timeout.");
        no_timeout = 0;
        break;
      }
    }

  }
  return 0;
}
```

Listing B.2: XC Parallel Execution With Replicator

B.3 Listing C

```
import socket

server_ip = '192.168.137.100'
server_port = 5001
ss = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ss.bind((server_ip,server_port))
ss.listen(5)

print 'Server [Address,Port]:','['+server_ip+', '+str(Port)+']'
while True:
  con, addr = ss.accept()
  buf = con.recv(64)
  if len(buf) > 0:
    print 'Client [Address,Port]:','['+addr[0],+', '+str(addr[1])+']'
    print "==================Payload===================="
    print buf
    break
```

Listing B.3: Basic NAT Example: Server Application

```
import socket

target_ip = '192.168.137.100'
source_ip = '192.168.137.102' #This is tested and confirmed before testing
source_port = 6828
target_port = 5001

cs = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
cs.bind((source_ip, 6828))
cs.connect((target_ip, target_port))
cs.send('Client [Address,Port]:','['+source_ip+', '+str(source_port)+']')
```

Listing B.4: Basic NAT Example: Client Application

During test of the completed system, a simple client server application was required to demonstrate network communication. Listing B.3 contains the source code for the server component of the application while listingB.4 conatins the source code for the client component of the application.

B.4 Supporting Functions

The final contribution to this appendix is a listing of functions designed to support application development associated with this research. Only function names are described here in an attempt to minimise unnecessary information and to allow the user to quickly grasp the functions currently supported by this research. Details relating to these functions are further discussed in sections 4.9.3 and 6.2.4.

```

/**************************************************************************
Functions
**************************************************************************/
//Frame realated characteristics
has_vlan(Frame &f);
is_arp(Frame &f);
is_ip(Frame &f);
is_tcp(Frame &f);
is_udp(Frame &f);
is_icmp(Frame &f);
get_frame_ethertype(Frame &f);
set_frame_ethertype(Frame &f, ethertype);
get_source_mac(Frame &f);
get_destination_mac(Frame &f);
set_destination_mac(Frame &f, Mac m);
set_source_mac(Frame &f, Mac m);

//ARP related characteristics
arp_get_hardware_type(Frame &f);
arp_get_protocol_type(Frame &f);
arp_get_operation(Frame &f);
arp_get_target_mac(Frame &f);
arp_get_target_ip(Frame &f);
arp_get_source_mac(Frame &f);
arp_get_source_ip(Frame &f);
arp_set_hardware_type(Frame &f, unsigned int);
arp_set_protocol_type(Frame &f, unsigned int);
arp_set_operation(Frame &f, unsigned int);
arp_set_target_mac(Frame &f, Mac mac);
arp_set_target_ip(Frame &f, unsigned int);
arp_set_source_mac(Frame &f, Mac m);
arp_set_source_ip(Frame &f, unsigned int);
```

```

//IPv4 related characteristics
ip_get_source(Frame &f);
ip_get_destination(Frame &f);
ip_get_ttl(Frame &f);
ip_get_version(Frame &f);
ip_get_header_length(Frame &f);
ip_get_DSCP(Frame &f);
ip_get_ECN(Frame &f);
ip_get_length(Frame &f);
ip_get_ID(Frame &f);
ip_get_flags(Frame &f);
ip_get_offset(Frame &f);
ip_get_protocol(Frame &f);
ip_get_checksum(Frame &f);
ip_set_source(Frame &f, ip);
ip_set_destination(Frame &f, ip);
ip_set_ttl(Frame &f, ttl);
ip_set_version(Frame &f, version);
ip_set_header_length(Frame &f, hl);
ip_set_DSCP(Frame &f, dscp);
ip_set_ECN(Frame &f, ecn);
ip_set_length(Frame &f, length);
ip_set_ID(Frame &f, id);
ip_set_flags(Frame &f, flags);
ip_set_offset(Frame &f, offset);
ip_set_protocol(Frame &f, prototcol);
ip_set_checksum(Frame &f, checksum);

//TCP related characteristics
tcp_get_source_port(Frame &f);
tcp_get_destination_port(Frame &f);
tcp_get_sequence_number(Frame &f);
tcp_get_acknowledgment_number(Frame &f);
tcp_get_data_offset(Frame &f);
tcp_get_reserved(Frame &f);
tcp_get_flag_NS(Frame &f);
tcp_get_flag_CWR(Frame &f);
tcp_get_flag_ECE(Frame &f);
tcp_get_flag_URG(Frame &f);
tcp_get_flag_ACK(Frame &f);
tcp_get_flag_PSH(Frame &f);
tcp_get_flag_RST(Frame &f);
tcp_get_flag_SYN(Frame &f);
```

```
tcp_get_flag_FIN(Frame &f);
tcp_get_flags(Frame &f);
tcp_get_window_size(Frame &f);
tcp_get_checksum(Frame &f);
tcp_get_urgent(Frame &f);
tcp_set_source_port(Frame &f, source_port);
tcp_set_destination_port(Frame &f, destination_port);
tcp_set_sequence_number(Frame &f, sequence_number);
tcp_set_acknowledgment_number(Frame &f, acknowledgment_number);
tcp_set_data_offset(Frame &f, offset);
tcp_set_reserved(Frame &f, reserved);
tcp_set_flag_NS(Frame &f, flag);
tcp_set_flag_CWR(Frame &f, flag);
tcp_set_flag_ECE(Frame &f, flag);
tcp_set_flag_URG(Frame &f, flag);
tcp_set_flag_ACK(Frame &f, flag);
tcp_set_flag_PSH(Frame &f, flag);
tcp_set_flag_RST(Frame &f, flag);
tcp_set_flag_SYN(Frame &f, flag);
tcp_set_flag_FIN(Frame &f, flag);
tcp_set_flags(Frame &f, flags); //This does not include the NS flag!
tcp_set_window_size(Frame &f, window_size);
tcp_set_checksum(Frame &f, checksum);
tcp_set_urgent(Frame &f, urgent);

//Axillary functions
is_src_in_range(Frame &f, start, end);
is_src_in_subnet(Frame &f, address, prefix);
is_dst_in_range(Frame &f, start, end);
is_dst_in_subnet(Frame &f, address, prefix);
ip_recalculate_checksum(Frame &f);
ip_update_checksum(Frame &f, old_val, new_val);
tcp_update_checksum(Frame &f, old_val, new_val);
update_checksum_32(old_checksum, old_val, new_val);
ip_swap_addresses(Frame &f);
tcp_swap_ports(Frame &f);
```

Listing B.5: Supporting Function Library

C
User Interface

This appendix attempts to provide the reader with screen captures relating to the control component of the user interface described as part of this research. This component is responsible for providing the user with statistical summaries of the network traffic received by each of the currently active ports associated with the implemented platform. Due to the size of the screen captures, it was decided to separate the figures from the text where they could be referred to rather than attempting to provide the figures inline.

In addition to statistical summaries of network traffic, this component also supports a tab dedicated to the enable feature associated with active applications. This feature allows the user to enable or disable custom modules or spores during application execution providing greater flexibility of applications.

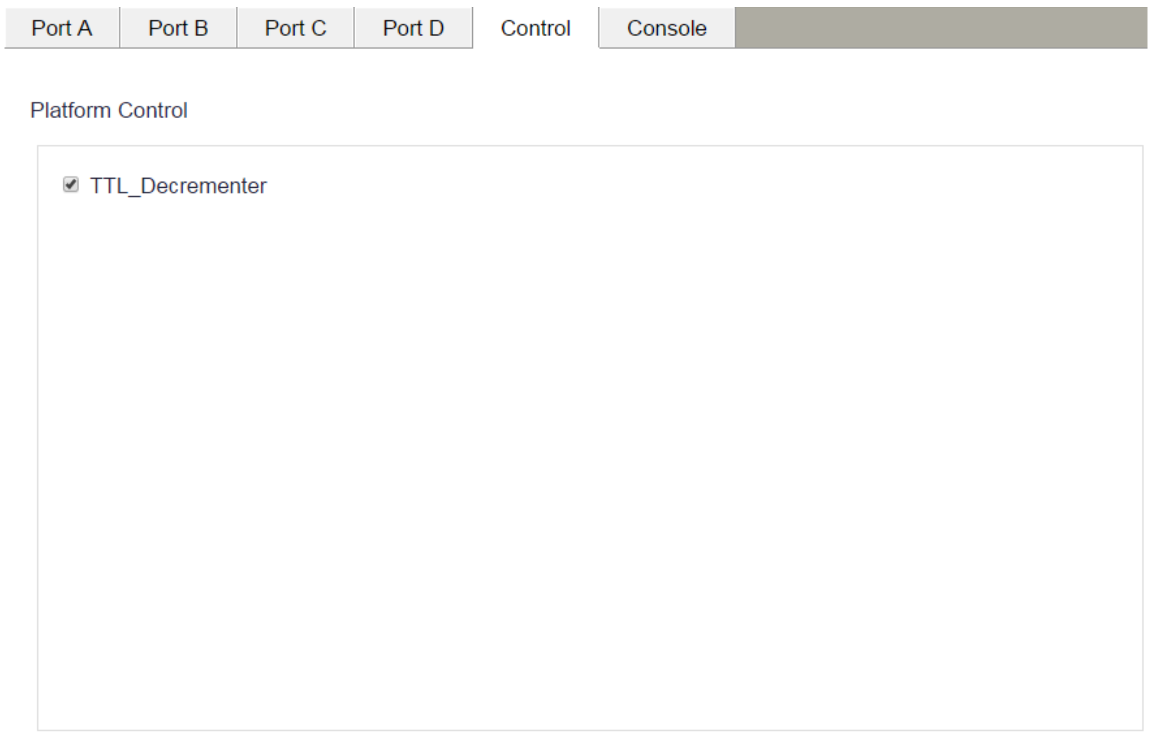


Figure C.1: Screen Capture of Enable Table

C.1 Enable Tab

Figure C.1 presents the functionality that the enabler tab provides as part of the user interface, this screen capture of the tab was recorded during the TTL modification test described in section 6.2.3. Though minimal, this capture shows the intended functionality of the enable feature. All user defined modules associated with the current application are presented on this page for the user to select or deselect which in turn enables or disable the respective module during execution.

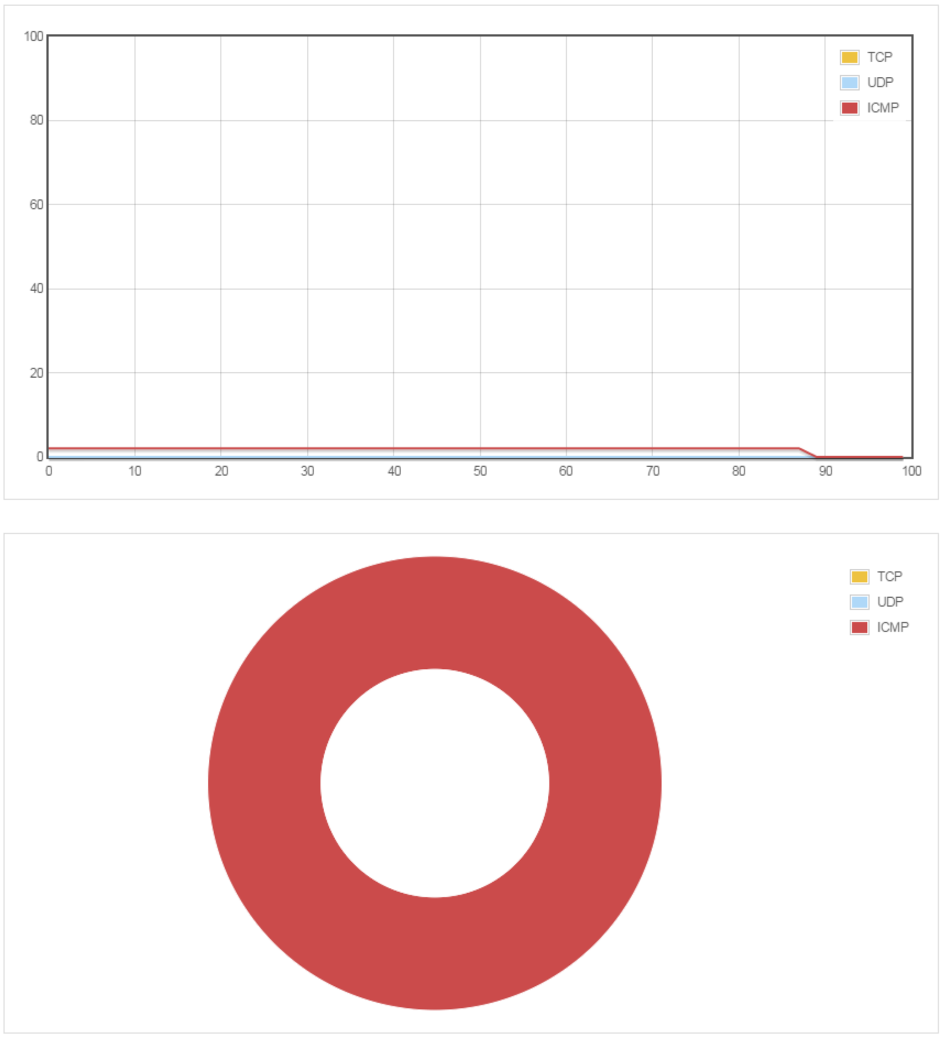


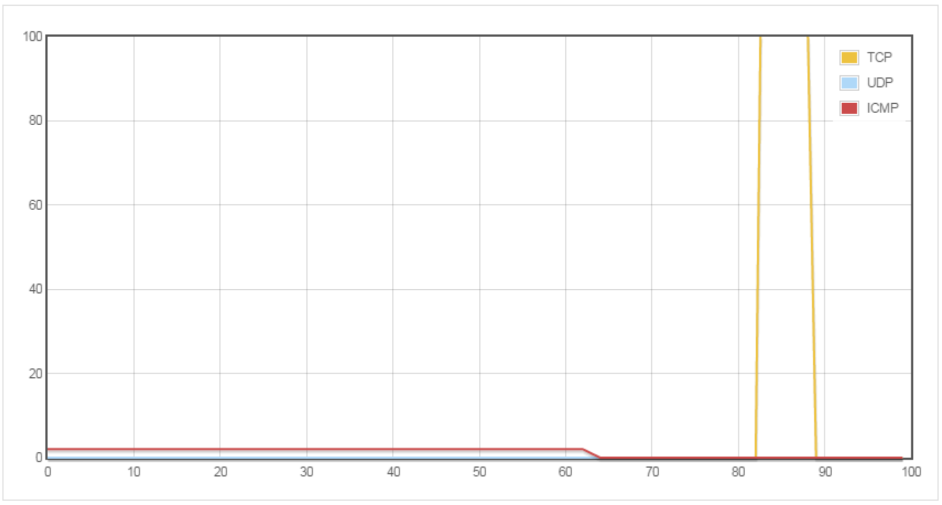
Figure C.2: Screen Capture After Continuous ICMP Message Transfer

C.2 Traffic Summaries

During the testing phase of the completed system, the platform was to pass traffic unmodified between two hosts. During the procedure, screen captures of the statistical summary page which forms part of the user interface were recorded and are presented in this appendix.

The intent of the statistical summary page is to provide the user with a brief overview of the network traffic currently being received by the platform. The page consists of a series of tabs, one associated with each network port with periodically submits a summary of traffic received to the base platform as discussed in section 4.10.

Figure C.2 presents a rendered summary of the data received by port A during a ping test. The only traffic seen during this evaluation was ICMP messages wich is reflected in



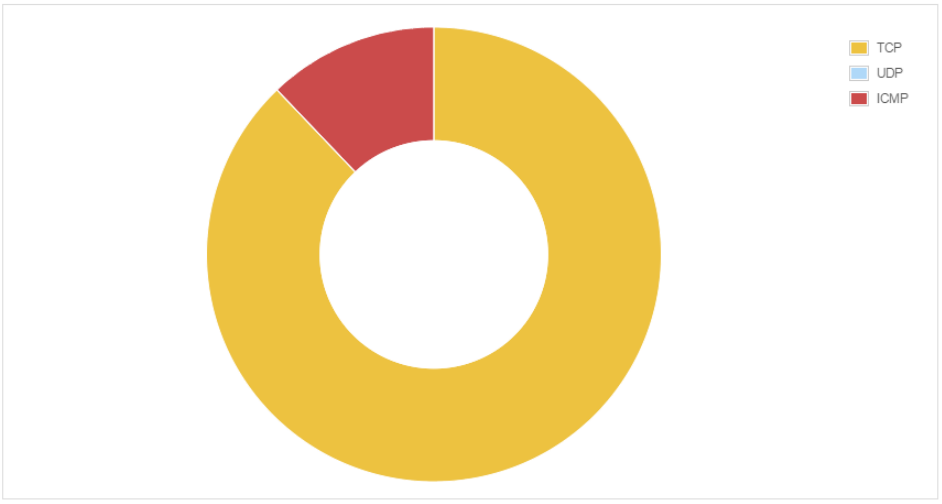
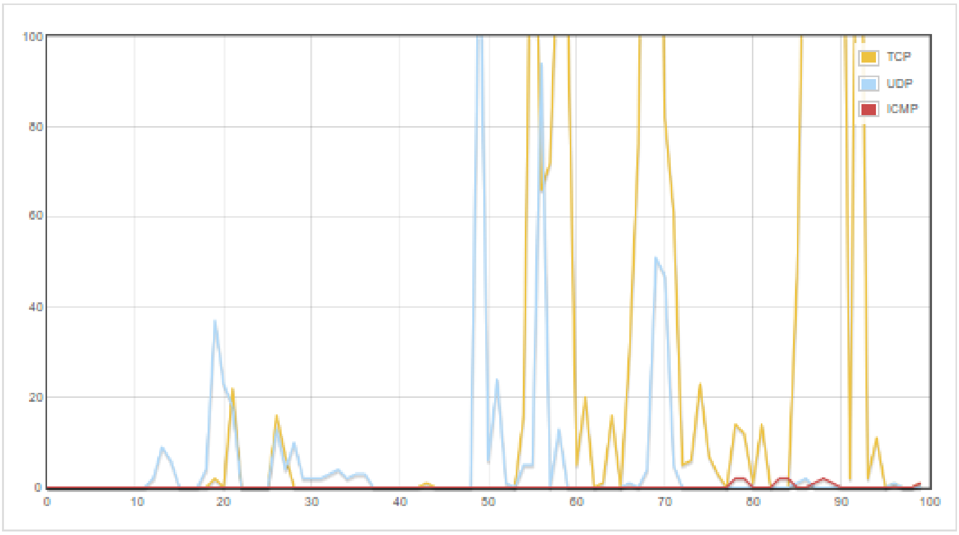


Figure C.3: Screen Capture Of Bulk TCP Transfer Following Continuous ICMP Message Transfer

the doughnut graph. The line plot in figure C.2 shows a constant low frequency of ICMP based traffic which is expected as the ping test would only perform in ICMP request every second.

Following the ICMP test, a bulk TCP transfer was done over the same configured network using the iperf application. Figure C.3 presents the results recorded by port A after after the iperf TCP transfer was complete. Both the doughnut and line graphs reflect the TCP transfer with TCP traffic quickly becoming the majority of traffic seen by the platform in the past 100 seconds. The line graphs indicates the traffic as a sudden spike lasting for approximately ten seconds.



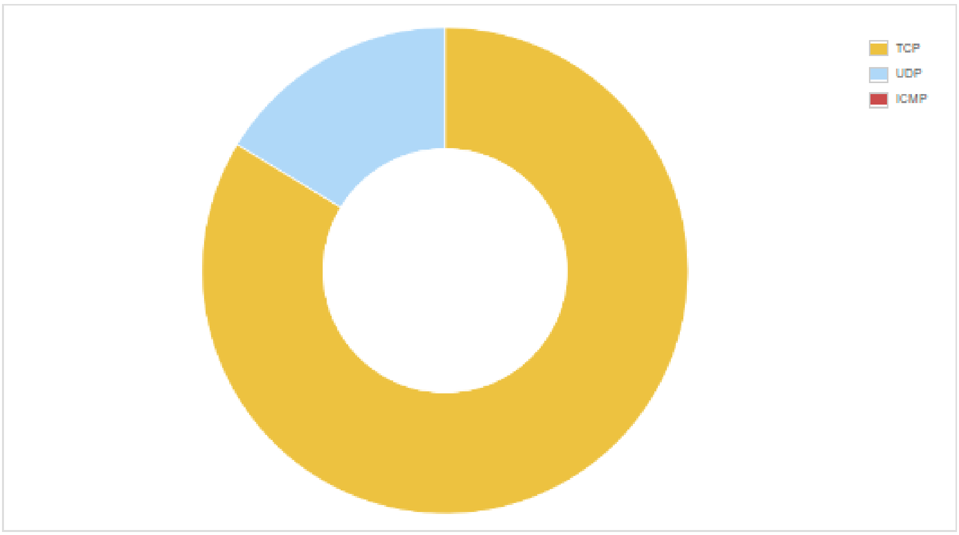


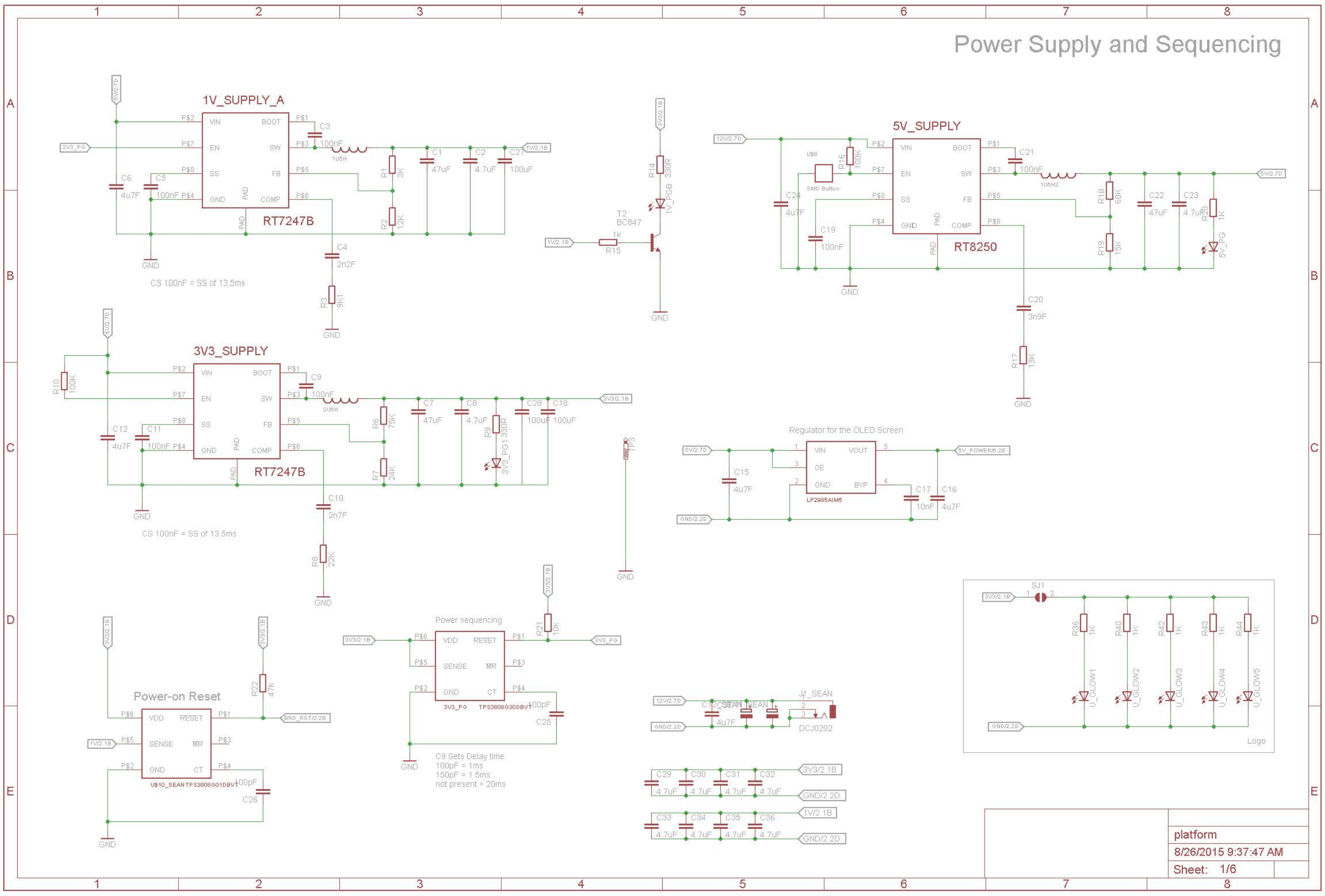
Figure C.4: Screen Capture During Standard Network Use

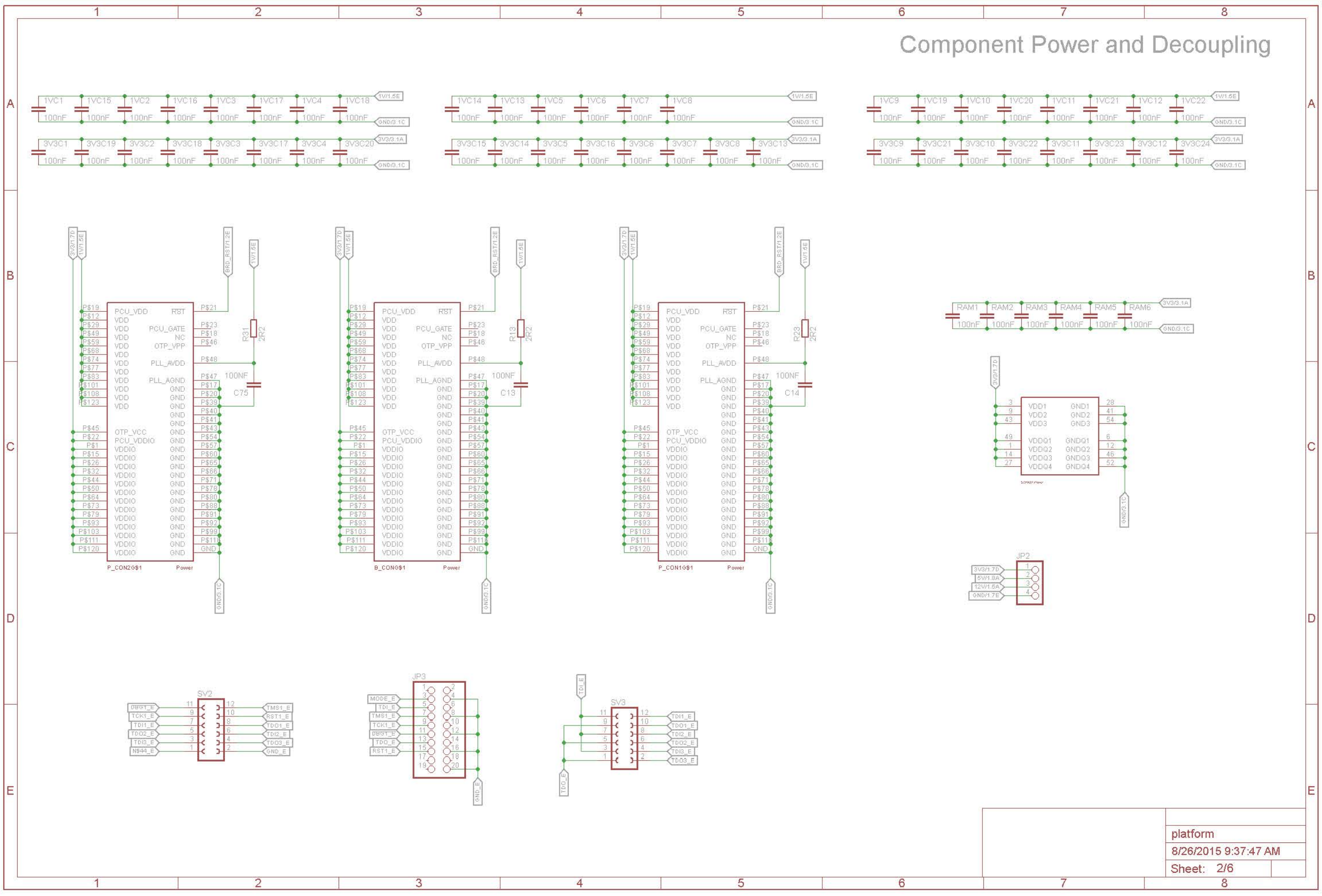
Finally, the network configuration was used to transfer more generic network traffic was as small TCP downloads and general web browsing. As indicated in figure C.4, such traffic is largely composed of a mix between UDP and TCP communication. It is important to note that the current platform configuration does not record traffic relating to the IPv6 protocol and only the UDP, TCP, and ICMP protocols are currently supported.

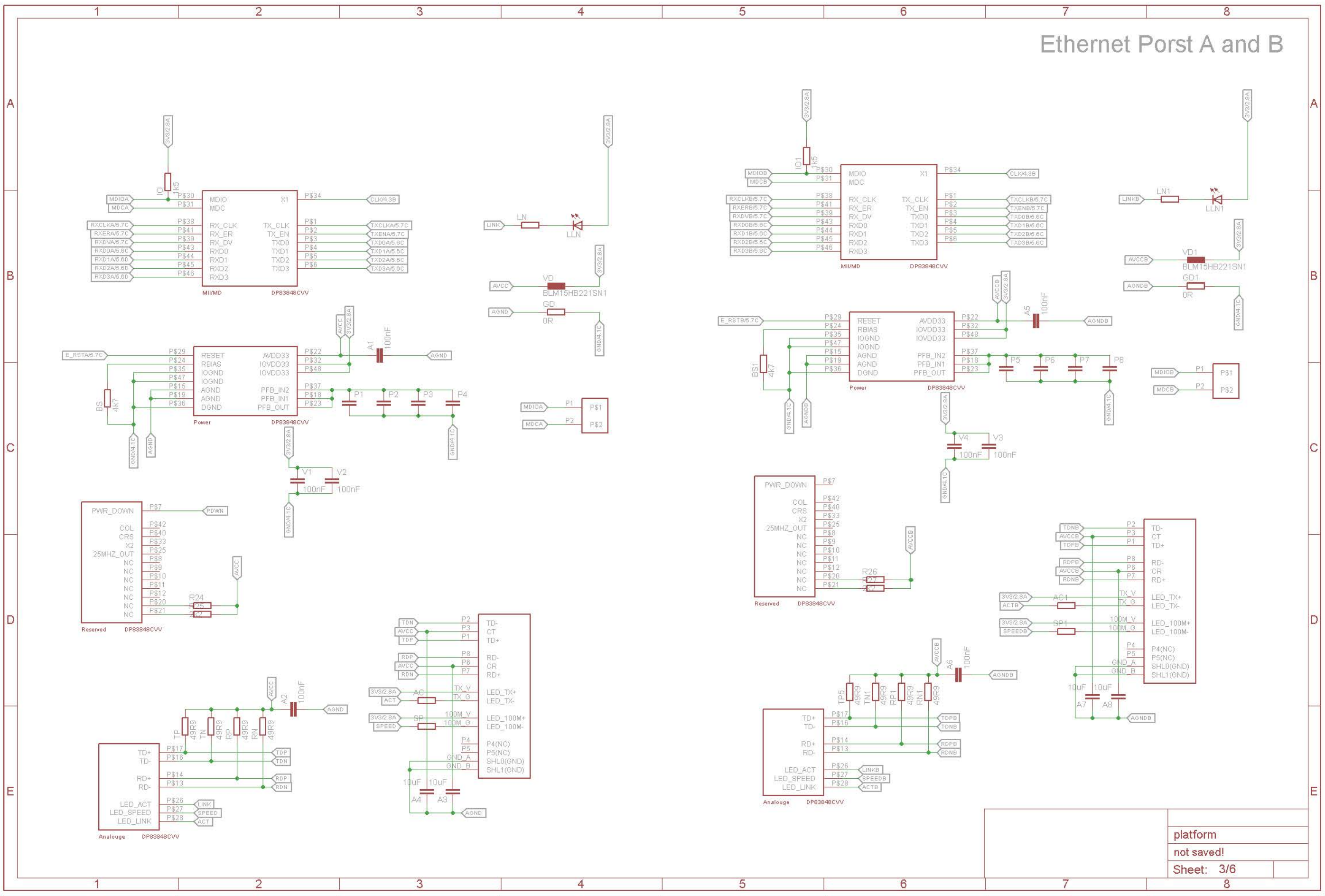
D
Board Design

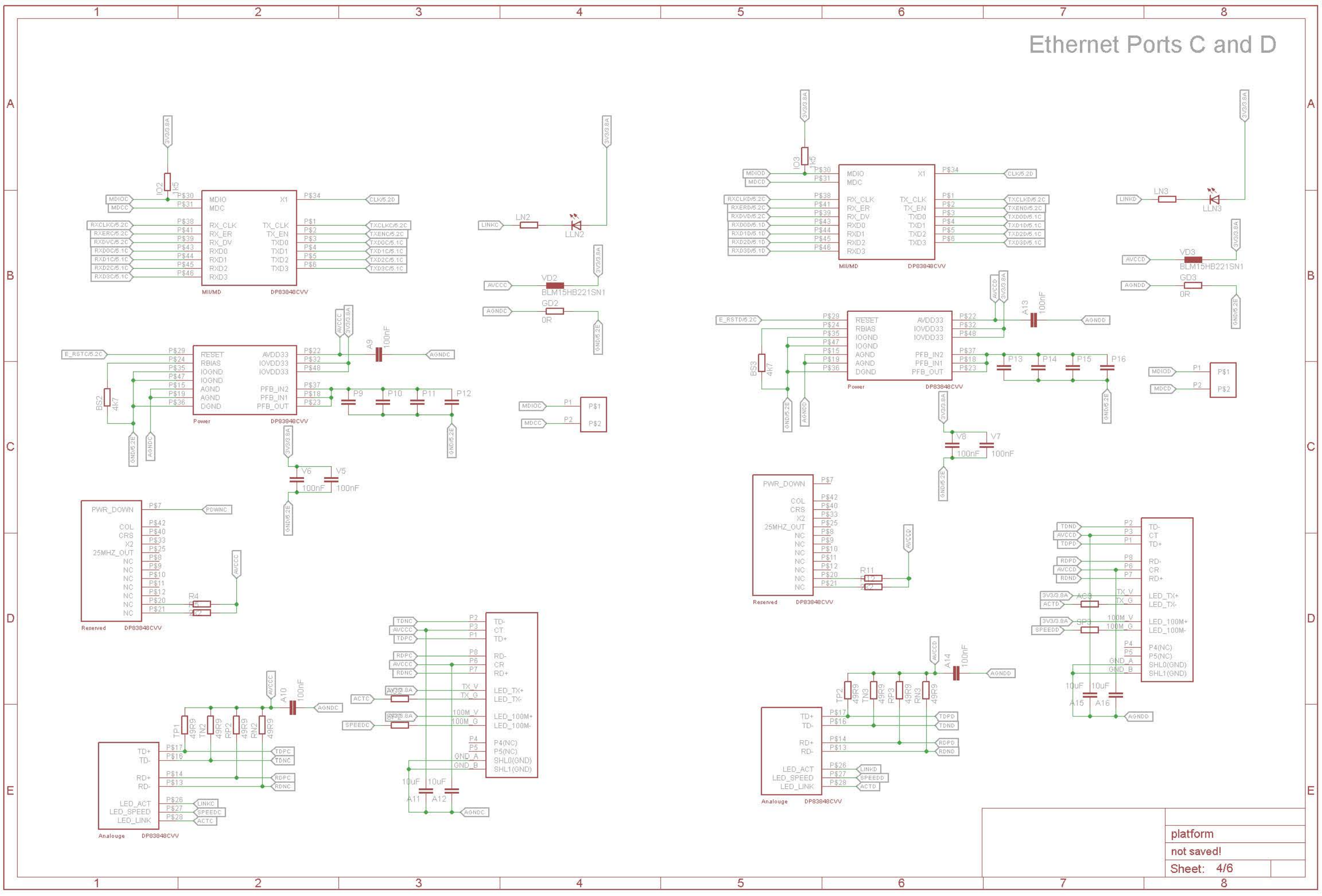
D.1 FRAME Board and Schematics

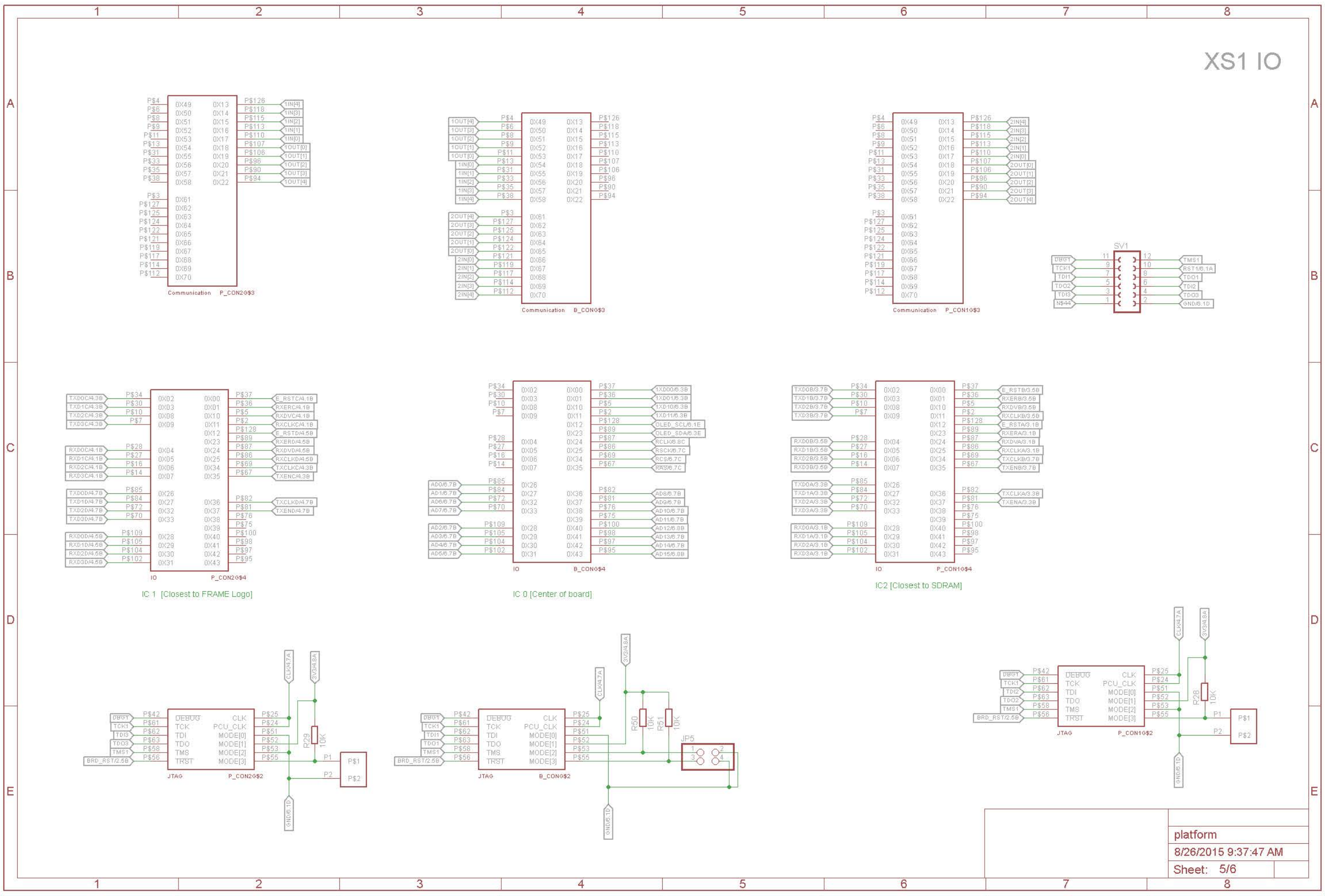
As part of the research described in this document, a hardware platform labelled the FRAME board was designed a fabricated. This appendix serves to provide the reader with the schematic designs relating to this board fabrication. It is important to note that these schematics include the proposed board modifications to account for design flaws described in section 5.4.2.

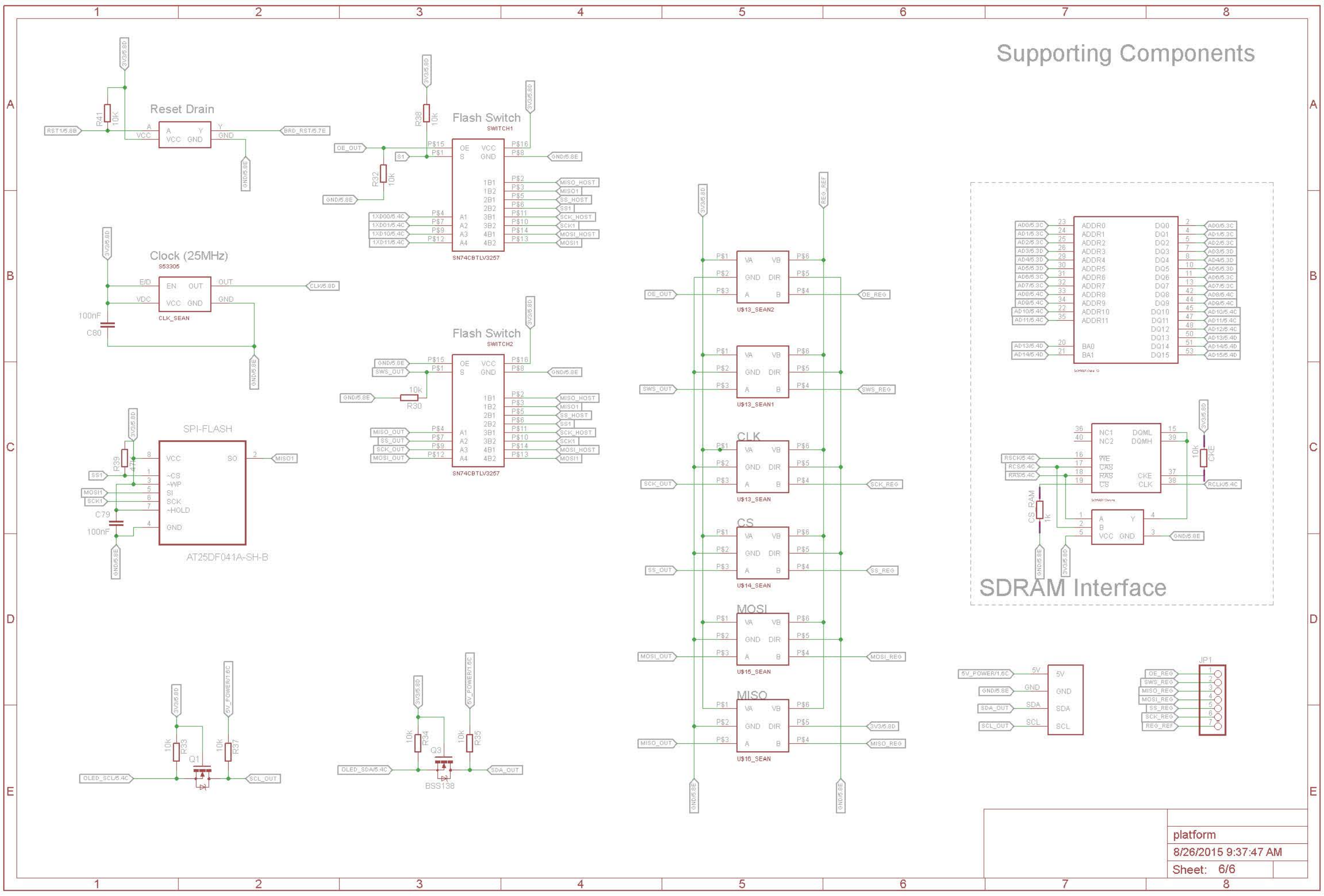

Power Supply and Sequencing
1V_SUPPLY_A
RT7247B
CS 100nF = SS of 13.5ms
3V3_SUPPLY
RT7247B
CS 100nF = SS of 13.5ms
5V_SUPPLY
RT8250
Regulator for the OLED Screen
LP2985AIM5
Power-on Reset
Power sequencing
C8 Sets Delay time
100pF = 1ms
150pF = 1.5ms
not present = 20ms
DCJ0202
Logo
platform
8/26/2015 9:37:47 AM
Sheet: 1/6


Component Power and Decoupling
P_CON20$1
B_CON0$1
P_CON10$1
SV2
JP3
SV3
JP2
platform
8/26/2015 9:37:47 AM
Sheet: 2/6


Ethernet Porst A and B
DP83848CVV
BLM15HB221SN1
platform
not saved!
Sheet: 3/6


Ethernet Ports C and D
DP83848CVV
BLM15HB221SN1
platform
not saved!
Sheet: 4/6


XS1 IO
Communication P_CON20$3
Communication B_CON0$3
Communication P_CON10$3
SV1
IO P_CON20$4
IO B_CON0$4
IO P_CON10$4
IC 1 [Closest to FRAME Logo]
IC 0 [Center of board]
IC2 [Closest to SDRAM]
JTAG P_CON20$2
JTAG B_CON0$2
JTAG P_CON10$2
JP5
platform
8/26/2015 9:37:47 AM
Sheet: 5/6


Supporting Components
Reset Drain
Flash Switch
SWITCH1
SN74CBTLV3257
Clock (25MHz)
CLK_SEAN
Flash Switch
SWITCH2
SN74CBTLV3257
SPI-FLASH
AT25DF041A-SH-B
U$13_SEAN2
U$13_SEAN1
CLK
U$13_SEAN
CS
U$14_SEAN
MOSI
U$15_SEAN
MISO
U$16_SEAN
BSS138
SDRAM Interface
JP1
platform
8/26/2015 9:37:47 AM
Sheet: 6/6

E

Link DSL Grammer

During research, a DSL termed Link was designed and developed to assist in the generation of network applications compatible with the FRAME board. This appendix presents the grammar used to parse all applications written in Link which are presented as part of this dissertation. This grammer was written using the Extended BNF notation, compatible with the compiler generation, Coco/R. The grammar was extended using supporting functions and structures written in C#, responsible for maintaining application state and detection of errors specific to the FRAME board such as exceeding logical processor count.

The presented grammar does not include any annotations used to produce a functional code generator for translating the source language or any of the supporting functions written in C#. Furthermore, due to length considerations of this dissertation, elements of the `Printable` and `Operable` sets are not presented. The intent of this appendix is to provide the reader with a overview of the DSL functional layout and as such, excluded components were considered unnecessary to achieve this.

```
CHARACTERS
 lf = CHR(10) .
 backslash= CHR(92) .
 control= CHR(0) .. CHR(31) .
 bin= "01" .
 digit= "0123456789" .
 hexDig = "ABCDEFabcdef" + digit.
 letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
 stringCh = ANY - '"' - control - backslash .
 charCh = ANY - "'" - control - backslash .
 printable= ANY - control .

TOKENS
 identifier = letter { letter | digit | "_" } .
 number = digit { digit } .
 stringLit= '"' { stringCh | backslash printable } '"' .
 charLit= "'" ( charCh | backslash printable ) "'" .
 hex= "0x" hexDig {hexDig}.
 binary = "0b" bin {bin} .

COMMENTS FROM "//" TO lf
COMMENTS FROM "/*" TO "*/"

IGNORE CHR(9) .. CHR(13)

PRODUCTIONS

DSL = { LinkSetup } Chain { Chain } .

 LinkSetup        = ( SwitchLink | ModLink ).
 SwitchLink       = "Switch" identifier "{"  SwitchStates  "}" .
 ModLink          = "Modifier" identifier "{" ModStates "}" .

 SwitchStates     = SwitchState{ SwitchState } .
 ModStates        = ModState{ModState } .
 SwitchState      = (("on" StateType | "Default" ) ":"
                    ("PASS" | "FAIL" | Respond |Log )
                    |Log )
                    ";".
 ModState         = "on" StateType
                     ( "{"  ModStates "}"
                       | ":"( Respond | Log ) ";")
                     | Assignment
```

```
                          | ("ip_update_checksum"
                          | "tcp_update_checksum"
                          | "ip_recalculate_checksum"
                          | Respond
                          | Log )";" .
Assignment       = Operable "=" Expression";"    .
StateType        =   Conditional
                        | Random
                        | "src_in_subnet" "(" Expression")"
                        | "dst_in_subnet" "(" Expression")"
                        | "src_in_range""(" Expression "," Expression")"
                        | "dst_in_range""(" Expression "," Expression")".
Conditional      = "Condition" "(" Condition ")" .
Random           = "Random" "(" Expression "%" ")".
Respond          = "Respond" "(" FrameIdentifier ")" .
Log              = "Log" "("   Printable {","Printable}")" .
FrameIdentifier  = "ICMP_Expired" .
/* Additional printables have been removed due to length */
Printable        = "length" | stringLit /*....*/ .
Chain            = Port"—>" { Intermediate"—>" } Port ";" .
Intermediate     = identifier [ "<" "on" ("PASS" | "FAIL") ">" ]
                   | Delay .
Delay            = ("Delay1"| "Delay2"| "Delay3"| "Delay4") "(" number ("s"
    | "ms"| "us" ) ")".
Port             =  "PortA" | "PortB" | "PortC" | "PortD" .
Condition        = Expression [("=="
                 | "!="
                 | ">"
                 | "<"
                 | ">="
                 | "<="
                 )Expression ] .
Expression       = AddOp<out expl, register_offset> {("+"  |"−"  ) AddOp }.
AddOp            = MulOp{("*" |"/" ) MulOp } .
MulOp            = number  | Operable | Operation | "(" Expression ")" .
/*Additional operables have been removed due to length. Every fields
    relating to the IP, ICMP, ARP, and TCP header frames have an associated
    operable here*/
Operable         = "frame_data" "[" Expression "]" [".as_char" ] /*....*/ ")
    " .
END DSL .
```

Listing E.1: Overview of the DSL Grammar Used in the Development of Link

F
Electronic Sources

Source code and electronic schematics relating to research presented in this document have been made available to the reader using the Bitbucket project hosting platform. The source material is divided into three repositories available at:

Resource	Address
Thesis Document	`https://bitbucket.org/CountBadger/msc_thesis`
DSL Implementation	`https://bitbucket.org/CountBadger/Flow`
Board Schematics	`https://bitbucket.org/CountBadger/Frame`