Development of a Network Processor in Reconfigurable Logic (FPGA)

Ανάπτυξη Δικτυακού Επεξεργαστή σε Αναδιατασσόμενη Λογική

Technical University of Crete
Electronic and Computer Engineering Department

Author:
Constantinos Stefanatos

Supervisor:
Ioannis Papaefstathiou, Assistant Professor

Committee:
Apostolos Dollas, Professor
Dionisios Pnevmatikatos, Associate Professor

September 2009

Abstract

Network Processors play a major role in computer network infrastructure and especially in the Internet, since they are embedded in many kinds of devices critical to the correct operation of these networks, such as routers, switches, firewalls etc. By being implemented in such devices, they are responsible for much of the workload these devices have to deal with.

The purpose of this diploma thesis was to develop and implement a Network Processor in Reconfigurable Logic, supporting a specific instruction set, capable of handling some of the tasks Network Processors deal with under normal circumstances. It is capable of operating in 10, 100 and 1000 Mbit/s Ethernet speeds. The design was implemented in an advanced FPGA board.

Ελληνικά

Οι Δικτυακοί Επεξεργαστές αποτελούν ένα σημαντικό κομμάτι της υλικής υποδομής διαφόρων δικτύων υπολογιστών, με κυριότερο αυτό του Διαδικτύου. Είναι ενσωματωμένοι σε συσκευές όπως routers, switches, firewalls και συνεπώς αναλαμβάνουν ένα μεγάλο μέρος από τη δουλειά που επιτελούν οι παραπάνω συσκευές.

Στην παρούσα διπλωματική εργασία, ο σκοπός μας ήταν να σχεδιάσουμε και να υλοποιήσουμε ένα Δικτυακό Επεξεργαστή σε Αναδιατασσόμενη Λογική, ο οποίος να υποστηρίζει ένα συγκεκριμένο σετ εντολών, ικανό να εκτελέσει μέρος από τις βασικές λειτουργίες των Δικτυακών Επεξεργαστών, υποστηρίζοντας ταχύτητες λειτουργίας 10, 100 και 1000 Mbit/s Ethernet. Η υλοποίηση της αρχιτεκτονικής έγινε σε μια προηγμένη FPGA.

Contents

List of Figures

List of Tables

Acronyms

IC	Integrated Circuit
NP	Network Processor
BGA	Ball Grid Array
PBGA	Plastic Ball Grid Array
SDK	Software Development Kit
API	Application Programming Interface
MAC	Media Access Control
MII	Media Independent Interface
RMII	Reduced Media Independent Interface
SMII	Serial Media Independent Interface
GMII	Gigabit Media Independent Interface
RGMII	Reduced Gigabit Media Independent Interface
SGMII	Serial Gigabit Media Independent Interface
TBI	Ten Bit Interface
QoS	Quality of Service
TCP	Transmission Control Protocol
PCI	Peripheral Component Interconnect
USB	Universal Serial Bus

POS	Packet Over Sonet
GUI	Graphical User Interface
BSM-CCGA	Bottom Surface Metallurgy - Ceramic Column Grid Array
SAN	Storage Area Network
DSL	Digital Subscriber Line
DSLAM	Digital Subscriber Line Access Multiplexer
CMTS	Cable Modem Termination System
LAN	Local Area Network
VLAN	Virtual Local Area Network
WAN	Wide Area Network
MAN	Metropolitan Area Network
IP	Internet Protocol
IPv6	Internet Protocol version 6
VPN	Virtual Private Network
RMON	Remote Monitoring
ROM	Read Only Memory
SRAM	Static Random Access Memory
DRAM	Dynamic Random Access Memory
SDRAM	Synchronous Dynamic Random Access Memory
RDRAM	Rambus Dynamic Random Access Memory
CRC	Cyclic Redundancy Checking
ECC	Error Correction Code
IDE	Integrated Development Environment
RISC	Reduced Instruction Set Computer

IXA	Internet Exchange Architecture
DDR	Double Data Rate
QDR	Quad Data Rate
FCBGA	Flip-Chip Ball Grid Array
FLBGA	Fine-Line Ball Grid Array
VoIP	Voice over Internet Protocol
ATM	Asynchronous Transfer Mode
SAR	Segmentation and Reassembly
GTP	General Packet Radio Service Tunneling Protocol
LVDS	Low Voltage Differential Signaling
SPI	System Packet Interface
CSIX	Common Switch Interface
IPsec	Internet Protocol Security
SSL	Secure Socket Layer
TLS	Transport Layer Security
NPE	Network Processing Engine
DES	Data Encryption Standard
3DES	Triple Data Encryption Standard
AES	Advanced Encryption Standard
SHA-1	Secure Hash Algorithm 1
MD5	Message-Digest Algorithm 5
SME	Small and Medium Enterprises
IAD	Integrated Access Devices
IEEE	Institute of Electrical and Electronics Engineers

TDM	Time-Division Multiplexing
AAL	Asynchronous Transfer Mode Adaptation Layers
HDLC	High-Level Data Link Control
RFID	Radio-Frequency Identification
XP	Executive Processor
CP	Channel Processor
SDP	Serial Data Processor
SP	Services Processor
TLU	Table Lookups Unit
QMU	Queue Management Unit
BMU	Buffer Management Unit
FP	Fabric Processor
MSAP	Multiservice Access Platform
CPE	Customer Premises Equipment
PRO3	Programmable Protocol Processor
NPU	Network Processing Unit
PDU	Protocol Data Unit
CPU	Central Processing Unit
DMM	Data Memory Management
TSC	Task Scheduler
TRS	Traffic Scheduler
RPM	Reprogrammable Pipeline Module
PPE	Protocol Processing Engine
FEX	Field Exctraction Engine

FMO	Field Modification Engine
CL	Configuration Library
TCAM	Ternary Content-Addressable Memory
I/O	Input/Output
FIFO	First-In, First-Out
OSI	Open Systems Interconnection
ISO	International Organization for Standardization
LLC	Logical Link Control
SNAP	Subnetwork Access Protocol
MTU	Maximum Transmission Unit
V5	Virtex-5
TEMAC	Tri-mode Ethernet MAC
Rx	Receive
Tx	Transmit
IFG	Interframe Gap
FCS	Frame Check Sequence
DCR	Device Control Register
PCS	Physical Coding Sublayer
PMA	Physical Medium Attachment
LL	LocalLink Interface
ASM	Address Swap Module
FSM	Finite State Machine
BRAM	Block RAM

Chapter 1

Introduction

A Network Processor (NP) is an Integrated Circuit (IC), with a feature set designed to tackle the needs of the networking application domain. Typically NPs are software programmable and have generic functions, which are similar to general purpose Central Processing Units (CPUs), commonly used in many different products.

Due to the fact that in modern telecommunication systems information is transferred in a packet form instead of the analog signals used in older systems, a need has arisen to develop ICs optimized to handle such packet forms of data. These ICs are called NPs and they make use of specific features or architectures to enhance and optimize packet processing in computer networks. By evolving over time, NPs have grown to become more flexible but at the same time more complex ICs. In newer iterations, NPs are programmable, thus providing the advantage of handling many different functions using the same hardware, by only installing the appropriate software.

NPs are designed with architectures that are aimed to augment network processing needs and applications. A common set of these architectures is:

- Pipeline of processors, where each pipeline stage consists of an entire processor
- Parallel processing, making use of multiple processors
- Specialized microcoded engines

A NP supports — as we have already mentioned — a set of generic functions. Some of those are:

- Pattern matching

- Key lookup
- Computation
- Data field modification
- Queue management
- Control processing
- Recirculation of packets

NPs are employed in many different types of network equipment, including, but not limited to:

- Routers — both hardware and software
- Switches
- Firewalls
- Intrusion detection and prevention systems
- Network monitoring systems

Taking advantage of the programmability of NPs, software programs executed on the processors can be used to provide different services. Some of the most common and generally needed services performed by NPs include:

- Packet/frame discrimination and forwarding — typically needed by routers or switches
- Quality of Service (QoS) enforcement — handling and processing of packets/frames according to certain preferences
- Access Control — deciding whether a packet should be accepted on the network node
- Encryption — processor provided hardware encryption of data
- Transmission Control Protocol (TCP) offload processing

The purpose of this thesis is to develop a custom NP architecture from scratch, supporting a simple instruction set and capable of handling some of the aforementioned services and functions. The target platform of this architecture is the Virtex-5 FPGA board, since it provides a very efficient and easy to implement in user designs Ethernet MAC wrapper.

Thesis Organization

The thesis is organized into chapters, each with a distinctive theme that upon its completion, leads naturally into the following one. We are now going to present the chapters and describe in a few lines what one can expect to find in each of them.

In the second chapter, we present the theoretical background necessary for one to understand the basic concepts behind all the information presented in this thesis. We begin by describing the Open Systems Interconnection (OSI) Layers and how they affect network architectures in general and then proceed to give a short description of all the information about the Ethernet standard that is needed to get a clear idea on some of the architectural decisions. Finally, we present a short overview of all the available Physical layer interfaces in order to justify the choice we made in our design later on.

In the third chapter, we deal with related work concerning NPs in both the commercial and the academic domain. We present some companies and their respective work concerning NPs; more specifically, we focus on products by IBM/Hifn, Intel and Freescale/Motorola. Then we move on to discuss a certain NP architecture developed in academic research, that of the Programmable Protocol Processor (PRO3).

In the fourth chapter, we present the architecture designed for this thesis. We begin by examining the base of the design, which is an Ethernet wrapper embedded on the Virtex-5 board and then justify some of the choices made during its generation process. After that, we continue by describing the design that implements the NP, all its modules and their inner workings; provided in this chapter are detailed block diagrams, Finite State Machine (FSM) states and their transitions and each module's interface.

In chapter five, a short presentation of the software tools used during the design's development and verification is given. Following after that is a step by step description of the verification process. Finally, a presentation of some benchmarks along with the design's performance is given.

In the last chapter, we propose some improvements to the design in order for it to become more efficient and complete.

Chapter 2

Theoretical Background

2.1 OSI Layers

The OSI Model is an abstract description for layered communications and computer network protocol design that was developed in the late 1970s by the International Organization for Standardization (ISO). Its purpose is to provide guidelines for compatibility in newly designed computer networks, which is of utter importance, since computer networks can be developed in different architectures, yet need to be compatible with each other in order to be useful. It divides network architecture into seven layers [2, 3], as shown in table 2.1.

Table 2.1: OSI Model

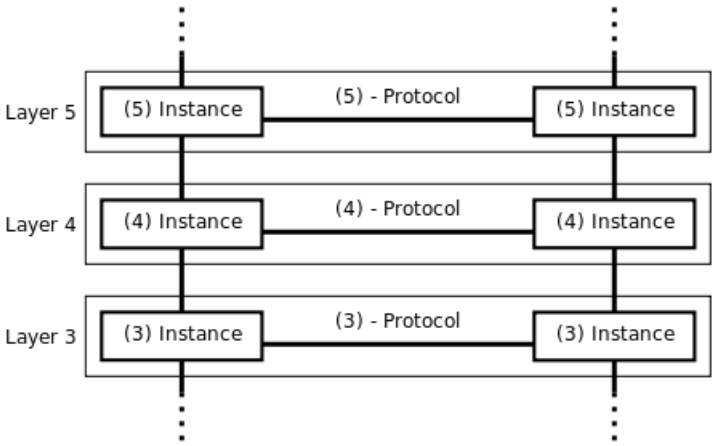
Level	Layer
7	Application
6	Presentation
5	Session
4	Transport
3	Network
2	Data Link
1	Physical

A layer is considered a collection of conceptually similar functions that provide services to the layer above and receive services by the layer below. On each layer, an instance provides services to the instances at the above layer and requests services from the layers below.

When different network nodes communicate with each other, each layer in the transmitting node passes data to the layer below while at the same time adding layer-specific data. When the Physical layer is reached, it sends the assembled data to the Physical layer of the receiving node. From there, each layer passes data to the layer above - each time removing the aforementioned layer-specific data - until the target layer is reached.

Communication in the OSI Model (see figure 2.1) is done with instances being on the same layer communicating with each other as if they exchange information directly, since through abstraction, each layer handles its own, specific part of the data.

Figure 2.1: Communication in the OSI Model



The OSI Model does not specify implementation details nor any programming interfaces; it only defines the so-called OSI Service Specifications, thus allowing for different implementations, as long as they conform to these Specifications.

2.2 Ethernet Protocol

The Ethernet [5] Protocol (standardized as IEEE 802.3 [4]) is a family of frame-based computer networking technologies. It defines wiring and signaling standards for the Physical layer (Layer 1) of the OSI Model through means of network access at the Media Access Control (MAC) — sublayer of

the Data Link layer (Layer 2) [7] — and a common addressing format. Originally developed at Xerox PARC in 1973-1975 by Robert Metcalfe, David Boggs, Chuck Thacker and Butler Lampson and in wide use from 1980 until today, it has gradually evolved into the de facto standard for wired Local Area Networks (LANs).

The main varieties of Ethernet are 10 Mbit/s (called simply Ethernet), 100 Mbit/s (Fast Ethernet), 1000 Mbit/s (Gigabit Ethernet), 10 Gbit/s (10-Gigabit Ethernet) and — still in development at the time of writing — 100 Gbit/s (100-Gigabit Ethernet). These varieties differ not only in the bandwidth they provide, but also in the physical medium they use, with the most common being twisted pair cables and more demanding applications using fibre optic cables.

2.2.1 Frame types

When data packets are transferred through the physical medium, they are referred to as frames. Ethernet frames have variations of their own: there is the Ethernet Version 2 Frame (Ethernet II Frame) or DIX Frame (DIX stands for DEC, Intel, Xerox) which is the most common today since it is used by the Internet Protocol (IP); IEEE's 802.3 Frame, Novell's non-standard frame variation of IEEE 802.3 without an IEEE 802.2 Logical Link Control (LLC) header, IEEE 802.2 LLC Frame and the IEEE 802.2 LLC\Subnetwork Access Protocol (SNAP) Frame. Optionally, all the aforementioned frame types can contain a IEEE 802.1Q tag which is used for Virtual Local Area Network (VLAN) identification and prioritization (QoS).

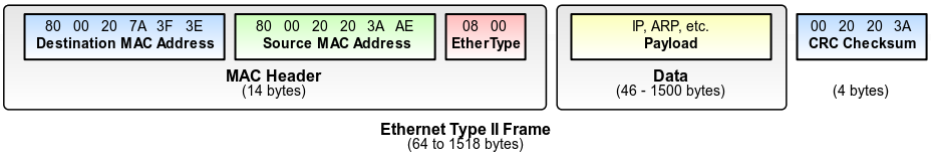
At the Physical layer, when a frame is transmitted, it is preceded by a preamble of 7 bytes — each with the hexademical value of AA — and a Start-of-Frame-Delimiter, 1 byte with the value AB (hex). After the frame follows the Interframe Gap (IFG), which consists of 12 bytes of idle characters. All of the aforementioned bytes are removed by network adapters before being passed to the Data Link layer.

2.2.2 Parts of a frame

Since it is the most common, we are going to present the Ethernet II Frame format [6]. By examining figure 2.2, we can see that the frame consists of 3 parts; a MAC Header, Data and the Frame Checksum.

The Maximum Transmission Unit (MTU) of standard Ethernet II Frames is 1500 bytes [9], which when added with the MAC Header and Frame Checksum parts, totals to a maximum frame size of 1518 bytes, while for VLAN

Figure 2.2: Ethernet II Frame Format



tagged frames, it is extended to 1522 bytes. On the other hand, the minimum frame size is 64 bytes.

The MAC Header contains the Destination MAC Address field, 6 bytes identifying in a unique way [8] the network adapter that is to receive the frame, the Source MAC Address field, 6 bytes to identify the network adapter transmitting the frame and the EtherType field, 2 bytes used to identify the type of data carried by the frame. It is worth mentioning that the EtherType field is what differentiates the Ethernet II Frame from the Institute of Electrical and Electronics Engineers (IEEE) 802.3 Frame, since the latter uses that field as a length identifier, giving the length of actual data in the frame. To circumvent such problems, a convention has been made, which states that values between 64 and 1522 indicate a Length field, while values greater than 1536 indicate an EtherType field.

The Data part of the frame contains the actual data, which can range from 46 to 1500 bytes. Since data could be less than 46 bytes, padding is used in order to make the frame reach the minimum required size of 64 bytes.

Finally, the Frame Checksum part, uses Cyclic Redundancy Checking (CRC) to calculate 4 bytes used to check the correctness of the frame. The Checksum is generated by the network adapter before the transmission of the frame begins and is calculated again upon receival of the frame; if the calculated code matches the one in the Checksum part, the frame was succesfully transmitted and so it is passed on the Data Link layer. Otherwise, the frame is dropped, thus considered to be erroneous and not passed on to the Data Link layer.

2.3 Physical Layer Interfaces

There is a selection of interfaces available to achieve data transfer between the Physical and the Data Link layers, and these are:

- Media Independent Interface (MII)
- Reduced Media Independent Interface (RMII)
- Serial Media Independent Interface (SMII)
- Gigabit Media Independent Interface (GMII)
- Reduced Gigabit Media Independent Interface (RGMII)
- Serial Gigabit Media Independent Interface (SGMII)

2.3.1 MII

MII [10, 36] is a standard interface used to connect Ethernet and Fast Ethernet MAC sublayers to the Physical layer. The media independent part means that the same interface can be used to connect to different types of Physical layer devices without the need to redesign or replace the MAC hardware. It is a parallel interface, transferring 4-bit words, operating at 2.5 MHz for Ethernet and 25 MHz for Fast Ethernet, that uses 16 pins to connect the Physical layer with the MAC sublayer.

2.3.2 RMII

The RMII [11] also supports Ethernet and Fast Ethernet MAC sublayers to Physical layer interfacing, differentiating itself from MII by reducing the pin count from 16 to a variable number of 6 to 10 (hence the reduced part of the title), by operating at a constant clock rate of 50 MHz and by transferring 2-bit words.

2.3.3 SMII

SMII is a serial implementation of MII.

2.3.4 GMII

GMII [12, 36] was developed in order to support Gigabit Ethernet, uses 8-bit data words, can operate at a maximum clock rate of 125 MHz and increases the pin count to 24. It is backwards compatible with MII, since it can operate at its offered speeds of 2.5 MHz and 25 MHz, thus making it able to connect to Ethernet and Fast Ethernet MAC sublayers as well. This of course renders the MII an actual subset of the GMII, making the latter a better choice, since

by a small increase in pins, a ten-fold increase in the supported Ethernet operating speed is achieved.

2.3.5 RGMII

RGMII [13, 36] is once again a reduced — concerning pin count, which is only half (12) compared to that of GMII — version of the GMII, which was introduced by Hewlett Packard. Performance is on par though and this is achieved by transferring data on both positive and negative clock edges (Double Data Rate (DDR)). This fact makes RGMII a much better alternative to GMII.

2.3.6 SGMII

The SGMII [14, 36] Physical layer interface was defined by Cisco Systems [15], a market leader in the networking domain. It converts the parallel nature of the GMII into a serial format using 2 data and 2 clock signals for each data direction (receive and transmit), thus exchanging frame data between the Physical layer and the MAC sublayer with a total pin count of 8. The data signals operate at a rate of 1.25 Gbps, while the clock signals operate at a 625 MHz rate (DDR). Due to these speeds, differential pairs are used in order to provide signal integrity and at the same time minimize system noise. Because of the increased performance and the low pin count, SGMII is usually preferred by manufacturers [36].

Chapter 3

Related Work

NPs have been an important subject in both commercial and academic research. In the following section we are going to present some companies and their respective work concerning NPs and in the next section we are going to present the work done in an academic research about a NP architecture.

3.1 Commercial Solutions

In the commercial domain, the companies developing NPs are many and some of them are listed in table 3.1.

Table 3.1: Network Processor Manufacturers

Agere	Altera	AMD
Analog Devices	Applied Micro Circuits Corporation	Bay Microsystems
Broadcom	Cavium Networks	Conexant
EZchip	Freescale	Hifn
Infineon	LSI Corporation	Mindspeed
Motorola	Netronome	Raza Microelectronics Inc
SiberCore	Wintegra	Xelerated
Greenfiled	Ubicom	Xilinx
IBM	Intel	

By examining table 3.1, we can see that there is a large number of companies researching, developing and manufacturing NPs.

In the rest of this section, we are going to focus on three manufacturers, IBM/Hifn, Intel and Motorola/Freescale.

3.1.1 IBM/Hifn

IBM is a company with major presence in the computer industry, that ventures both in software and in hardware. It has a NP line, dubbed PowerNP that has been acquired in 2004 by Hifn [16, 17], a company which specialises in integrated circuits and software for network infrastructure developers. IBM is still producing the NPs, though they are being distributed by Hifn as part of their product line. Nevertheless, the architecture remains an intellectual property of IBM.

The PowerNP line is currently represented by a single product according to the website of Hifn [18], the 5NP4G NP, which uses the NP4GS3 [19, 20]. The NP4GS3 is an advanced, robust, programmable, high-performance NP optimized for packet processing which integrates a switching engine, a search engine, frame processors and multiplexed MACs. It is designed to satisfy enterprise, core and edge networking and Internet requirements at wire speed. Being scalable, the NP4GS3 can meet increasing bandwidth and functional demands. Thus, it can offer a wide selection of services, including but not limited to QoS, scheduling and flow control.

It features an embedded IBM PowerPC microprocessor, 16 programmable picoprocessors and multiple hardware accelerators. With the above hardware characteristics it can process up to 32 frames in parallel. The hardware accelerators can perform tree searches, frame forwarding, frame filtering, frame alteration and other functions, while the embedded PowerPC allows manufacturers to support their own custom functions, such as enhanced frame processing and higher-layer protocols.

Continuing on the features list, the NP4GS3 offers advanced flow control in order to prevent TCP collapse, hardware support for port mirroring and multi-threads support to improve performance.

The available interfaces are an integrated Peripheral Component Interconnect (PCI) interface that allows peripheral devices to be attached to it as well as OC-3, OC-12 and OC-48 Packet Over Sonet (POS) interfaces.

Its integrated MACs can support up to 4 Gigabit Ethernet or 40 Fast Ethernet ports, that can be accessed through SMII, GMII and Ten Bit Interface (TBI).

The NP4GS3 does not lack in programmability, since it is supported by software tools available for Windows, Solaris and Linux environments. These software tools include an assembler, a full-function simulator, a Graphical

User Interface (GUI) debugger, a test case generator, demonstration picocode and test case scripts. Its simulation environment supports a distributed software model, which enables flexible testing configurations from software unit test to system test, even before having the actual hardware at hand. These tools enable using the NP4GS3 for rapid development and deployment of new services which can be applied as software upgrades, without altering the underlying hardware.

The packaging of the NP4GS3 is a 1088-pin Bottom Surface Metallurgy - Ceramic Column Grid Array (BSM-CCGA).

Some of the NP4GS3's applications, include:

- Multi-layer Chassis Switch-Router
- Server Load Balancer
- Network Edge QoS Traffic Manager
- Storage Area Network (SAN) Switch
- Core/Edge Router
- Digital Subscriber Line Access Multiplexer (DSLAM)
- Web Caching Server
- Wide Area Network (WAN) IP Switch
- Firewall and Virtual Private Network (VPN) Appliances
- IP Service Blade

3.1.2 Intel

Intel is a renowned processor manufacturer and has had a long line of NPs in its product briefcase. Their IXP NP range is discontinued, yet it is worth mentioning, as it has evolved greatly since its incubation.

3.1.2.1 IXP12xx

The first NP family developed by Intel was the IXP12xx [21, 22]. It was meant to be used in web switches, broadband access platforms and network appliances and could offer Layer 2 and Layer 3 forwarding, protocol conversion, QoS, filtering, firewalling, handling of VPNs, load balancing, Remote Monitoring (RMON) and intrusion detection.

They featured an integrated Intel StrongARM processor core, compatible with the ARM architecture, 6 integrated programmable multi-threaded microengines, an open IX bus architecture, an integrated PCI interface and integrated high-performance memory controllers for Static Random Access Memory (SRAM) and Synchronous Dynamic Random Access Memory (SDRAM). Also in the list of features are CRC and Error Correction Code (ECC) memory.

Lastly, the processor family delivered extensive programmability in software through the provided Intel Internet Exchange Architecture (IXA) Software Development Kit (SDK) 2.0, which could be used to extend their range of applications. Also provided were the Intel IXDP1200 Advanced Developer Platform, a Windows NT Integrated Development Environment (IDE) for Embedded Linux and example designs for an ATM/OC-3 to Fast Ethernet IP router and a WAN/LAN access switch.

The NPs in the IXP12xx family were available with core speeds of 166, 200 and 232 MHz, with a low power consumption of a typical value of 5 W or less. The package form of this family was a 432-pin HL-Ball Grid Array (BGA).

3.1.2.2 IXP24xx

The architecture of the IXP24xx [23] NP family consists of an integrated Intel XScale Core (32-bit Reduced Instruction Set Computer (RISC)), 8 integrated fully programmable multi-threaded microengines (second generation), integrated PCI interface, receive and transmit interfaces supporting Utopia, SPI-3 or CSIX, integrated high-performance memory controllers for DDR Dynamic Random Access Memory (DRAM) and Quad Data Rate (QDR) SRAM and hardware support for memory access queuing.

This NP family was accompanied by the Intel IXA SDK 3.0 complemented by a hardware development platform with supporting software and tools, which account for its flexibility and extension of its capabilities. Included are Intel's own Microengine C compiler and Microengine C Networking library. Also included was the very important capability to retarget code developed for the 12xx family in order to make it compatible with the 24xx family.

The IXP24xx family was available at core speeds of 400 and 600 MHz, with a typical power consumption of 10 W. The packaging is a 1356-pin Flip-Chip Ball Grid Array (FCBGA).

Since it supports OC-48 line rates, these NPs are used in a wide selection of applications such as WAN multi-service switches, DSLAMs, Cable Modem Termination System (CMTS) equipment, 2.5G and 3G wireless infrastruc-

ture base station controllers and gateways and Layer 4 to Layer 7 switches with content-based load balancing and firewalls. Their programmability also allows them to be used in Voice over Internet Protocol (VoIP) gateways, multi-service access platforms, high-end routers, remote access concetrators and VPN gateways. Their usage models include:

- Aggregation, Asynchronous Transfer Mode (ATM) Segmentation and Reassembly (SAR), traffic shaping, policing, forwarding and protocol conversion in DSLAM equipment
- Aggregation, forwarding and protocol conversion in CMTS equipment
- ATM SAR, encryption and forwarding in base station controllers/radio network controllers
- General Packet Radio Service Tunneling Protocol (GTP) and Internet Protocol version 6 (IPv6) forwarding in wireless infrastructure
- ATM SAR, traffic shaping, policing, protocol conversion and aggregation for multi-service switches
- Content-aware load balancing, forwarding and policing

3.1.2.3 IXP28xx

The IXP28xx [24] NP family's architecture is comprised of an integrated Intel XScale Core, 16 integrated programmable multi-threaded microengines (second generation), a PCI interface, two unidirectional 16-bit Low Voltage Differential Signaling (LVDS) data interfaces programmable to be System Packet Interface (SPI)-4.2 or Common Switch Interface (CSIX), an 8-bit asynchronous control interface, 5 industry-standard high-performance memory controllers for Rambus Dynamic Random Access Memory (RDRAM) and QDR SRAM memory and hardware support for memory access queuing.

The family's life is extended through the provided IXA SDK 3.0 and the hardware development platform, since it allows for great programmability and thus extension of the NPs' functionality.

The XScale core is clocked at 700 MHz, while the microengines are clocked at 1.0 and 1.4 GHz, while the family's power performance is at a typical 14 W.

This family supports OC-192 line rates, which makes it ideal for high-performance applications such as Metropolitan Area Network (MAN) switches and routers, Internet edge and core switches and routers, multi-service

switches, 10 Gbps enterprise switches and routers for advanced data centers, SAN and content-aware server off-load/web switches. Its programmability allows it to be used to provide Internet Protocol Security (IPsec) and VPN solutions. It is also great in wireless infrastructure equipment. Its functionality includes:

- Ethernet/POS/ATM Layer 4 forwarding in core, MAN and edge applications
- Protocol conversion, forwarding and aggregation for multi-service switches, cable headends and DSLAM aggregation
- ATM SAR and forwarding with advanced traffic shaping
- Content-aware load balancing, forwarding and policing
- Encryption for VPNs and IPsec applications
- GTP and IPv6 in wireless infrastructure applications
- TCP/IP termination for enterprise data center and SANs
- Secure Socket Layer (SSL)/Transport Layer Security (TLS) acceleration

3.1.2.4 IXP42x

Architecturally, the IXP42x [25] NP family changes quite a bit when compared to the previously mentioned IXP families. This one, features an Intel XScale processor along with up to 3 Network Processing Engines (NPEs) which are basically processors of their own. Also integrated are a PCI interface, a Universal Serial Bus (USB) 1.1 interface, a Utopia interface, 2 integrated 10/100 Ethernet MACs (MII), a high-performance SDRAM memory controller and hardware support for Data Encryption Standard (DES), Triple Data Encryption Standard (3DES), Advanced Encryption Standard (AES), Secure Hash Algorithm 1 (SHA-1) and Message-Digest Algorithm 5 (MD5) encryption and hashing algorithms. The latter hardware support is provided through the aforementioned NPEs.

Once again, a SDK supporting Windows and Linux as well as a hardware development platform are provided, thus gracing this NP family with great extensibility and longevity.

This family has outstanding energy efficiency, since it has a typical system power consumption of 1-1.5 W while operating at a clock speed of up to 533 MHz. The package is a 492-pin Plastic Ball Grid Array (PBGA).

Its applications include, but are not limited to, high-performance Digital Subscriber Line (DSL) modems, high-performance cable modems, residential gateways, Small and Medium Enterprises (SME) routers, Integrated Access Devices (IAD), set-top boxes, DSLAMs, wireless access points in accordance with the IEEE 802.11a/b/g protocols, industrial controllers, network printers and the control plane. The capabilities of this family, in more detail, include:

- Layer 2, Layer 3 forwarding
- ATM, Time-Division Multiplexing (TDM), Ethernet MAC filtering
- Asynchronous Transfer Mode Adaptation Layers (AAL) SAR, TDM framing, High-Level Data Link Control (HDLC) processing
- Hardware supported DES, 3DES data encryption
- Hardware supported SHA-1, MD5 hashing algorithms

3.1.2.5 IXP43x

The IXP43x [26] NP family differentiates itself from the IXP42x by providing up to 2 NPEs that support double the amount of integrated instruction and data memory when compared to the NPEs in the IXP42x line, while at the same time adding support for DDR1 and DDR2 memory with ECC and USB 2.0.

The packaging is now a 460-pin PBGA and the available operating clock speeds are 400, 533 and 667 MHz, with a typical power consumption of 3.44 W.

3.1.2.6 IXP45x

The IXP45x [27] NP family offers support only for DDR1 memory and integrates up to 3 10/100 Ethernet MACs (MII).

The packaging once again changes to a 544-pin PBGA and the available clock speeds are 266, 400 and 533 MHz.

3.1.2.7 IXP46x

The IXP46x [28] NP family differentiates itself from the other IXP4xx families by integrating up to 3 10/100 Ethernet MACs (MII or SMII) and by offering integrated hardware support for IEEE 1588 protocol.

The core this time operates at a selection of 266, 400, 533 and 667 MHz while being packaged in 544-pin PBGA.

3.1.3 Motorola/Freescale

Motorola is a company specialised in communications, manufacturing products ranging from mobile phones to Radio-Frequency Identification (RFID) solutions. Among their products of course, lies a NP family, the C-Port family, consisting of processors C-3e, C-5 and C-5e. These NPs are manufactured through its spinoff company, Freescale Semiconductor.

3.1.3.1 C-3e

The first member of the C-Port line of NPs is the C-3e [29, 30]. Its architecture consists of 17 programmable RISC cores, where one is used as the Executive Processor (XP) and the other 16 as Channel Processors (CPs). The CPs have 2 Serial Data Processors (SDPs) each, and 8 of them implement external programmable interfaces, while the other 8 are used internally, as Services Processors (SPs).

Also part of the architecture are an integrated Table Lookups Unit (TLU) coprocessor, a Queue Management Unit (QMU) coprocessor, a Buffer Management Unit (BMU) coprocessor, and a Utopia interface Fabric Processor (FP).

Integrated in the architecture are interfaces such as PCI, serial, 10/100 Ethernet MACs (RMII), 1000 Ethernet MAC (GMII, TBI), FibreChannel MACs (TBI) and Utopia. It supports OC-3 and OC-12 line rates.

The C-3e NP also provides complete programmability using a standard Application Programming Interface (API) and C programming language, which we need to state that are fully software compatible with the rest of the C-Port NP family.

Its operating frequency is up to 180 MHz with a typical power consumption of 5.5 W, and it comes in a highly-integrated 728-pin BGA package.

3.1.3.2 C-5

The C-5 [31, 32] is the next NP of the C-Port family. This NP's architecture is comprised of 16 CPs and 5 coprocessors responsible for supervisory tasks (XP), high-speed fabric interface management (FP), networking lookups (TLU), queue control (QMU) and payload storage (BMU). Each of the CPs contains a RISC programmable core and 2 SDPs, the one to receive, the other to transfer.

The integrated interfaces are PCI, serial, Utopia, 10/100 Ethernet MACs (RMII), 1000 Ethernet MAC (GMII, TBI) and FibreChannel MAC. The line rates supported by the C-5 are OC-3, OC-12 and OC-48.

This NP supports extensive programmability, through the provided API and using C/C++ as programming languages.

The C-5's operating frequencies are 166, 180, 200 and 233 MHz with a respective typical power consumption of 15, 16, 17.5 and 20 W. The packaging of this NP is an 838-pin BGA.

A small list of C-5's breadth of applications follows:

- Multiservice Access Platforms (MSAPs)
- DSLAM
- Cable and wireless head-end systems
- Ethernet/IP/Frame Relay/ATM interworking
- Internet access switch/routers
- Load balancing web server switches
- Optical edge switch/routers and add/drop multiplexers
- IP Gigabit/Terabit routers
- WAN Customer Premises Equipment (CPE)
- MAN CPE and head-end equipment

3.1.3.3 C-5e

The last member of the C-Port NP family we are presenting is the C-5e [33, 34]. The architecture is quite similar to its aforementioned siblings, since it also consists of 16 CPs (with a programmable RISC core and 2 SDPs each), an XP responsible for supervisory tasks and management of host processing, a FP for high-speed fabric interface management, a TLU for networking lookups and classification, a QMU for queue control and traffic management and a BMU for payload storage.

The interfaces integrated into C-5e's architecture are 10/100/1000 Ethernet MACs (RMII for 10/100, GMII and TBI for 1000), FibreChannel, PCI, serial and Utopia while the line rates supported are OC-3, OC-12 and OC-48.

Programming the C-5e is accomplished through the use of the C programming language and the provided standard API.

The C-5e NP comes in an 840-pin BGA package, with the available operating frequencies being 266 and 300 MHz, with a typical power consumption of 9.2 and 10.6 W respectively.

3.2 Academic Research

In the academic domain, a lot of research has been conducted concerning NPs and their applications in security and routing and of course their processing capabilities. We are going to focus on the PRO3[35] architecture.

3.2.1 The PRO3 Architecture - Overview

The PRO3 is a novel hybrid Network Processing Unit (NPU) architecture, which uses sophisticated interface hardware modules specifically designed from the ground up, special-purpose programmable processors with a low hardware complexity, high-performance general-purpose processors optimized for fast context switching and an on or off-chip control processor. As an overview, the aforementioned hardware modules and special-purpose processors are responsible for handling most of the computation-intensive and real-time protocol functions, while the general-purpose processors handle all the remaining functions, including the higher layer protocols. The on or off-chip control processor is responsible for all the computations that are not on the fast path, such as control-plane or exception processing. The main benefit of this architecture is that it combines wire-speed processing up to the Network layer with best-effort processing for higher layer protocols.

The PRO3 targets systems with requirements such as the following:

- Traffic concetrators supporting enhanced per-flow services, such as security systems performing packet filtering and protocol-aware connection tracking
- Signaling controllers
- Traffic-policing
- Traffic-metering
- Statistics-collecting

It has been designed taking into consideration that a number of functions frequently used in common protocols cannot be executed in an efficient manner using generic RISC processors. These functions are classified as follows:

- Bit and byte-level operations for header parsing and modification
- Efficient memory management

- Complex task and traffic-scheduling algorithms supporting QoS
- Context switching that occurs more frequently than in desktop processors
- Interconnecting a number of different processing modules avoiding the introduction of bottlenecks

The PRO3 resolves these issues by using its special-purpose processors, which are responsible for the header field's exctraction and modification, a sophisticated memory management hardware block and 2 very efficient hardware scheduling modules responsible for fair and balanced packet processing and for controlling data streams generated by the internal modules. In addition, the 2 general-purpose processors are used to complement the special-purpose processors and the hardware modules in the protocol processing parts that they cannot handle in an efficient manner. Communication between the aforementioned hardware parts of the architecture occurs using a 12.8 Gbps internal bus, which is coordinated by a central arbiter.

The PRO3 is implemented in 0.18 μm technology, operates at 200 MHz and is packaged in a 1,096-pin Fine-Line Ball Grid Array (FLBGA).

3.2.2 The PRO3 Architecture - In depth look

Now we are going to take a closer look at the building blocks in PRO3's architecture. These are:

1. The packet preprocessor,
2. the Data Memory Management (DMM) unit,
3. the Task Scheduler (TSC),
4. the Traffic Scheduler (TRS) and
5. the Reprogrammable Pipeline Module (RPM).

The packet preprocessor, the DMM unit and both of the schedulers are optimized for handling a very large number of independent queues while the RPM is very efficient at executing the majority of the required network protocol processing. Concerning the modules' bandwidth, the packet preprocessor, the DMM unit and both of the schedulers can all support a constant rate of 2.5 Gbps of traffic processing under any circumstances, while the RPM supported network rate is application-dependent.

At the following sections we are going to examine each of the aforementioned building blocks.

3.2.2.1 Packet Preprocessor

The Packet Preprocessor block is responsible for looking up of address or other fields used by most network applications. It takes advantage of the features of the programmable field-processing engine which in fact are identical to the ones of the Protocol Processing Engine (PPE) although they are a different entity; it also makes extensive use of an external Ternary Content-Addressable Memory (TCAM) which is used for classification of incoming Protocol Data Units (PDUs).

The tasks performed by this block are header parsing, checksum calculation, classification and generation of a unique flow ID for the incoming PDUs. That flow ID specifies in which queue the packet belongs to — information that is handled by the DMM unit — as well as the internal processing unit needed to handle it and the corresponding software needed by that unit to accomplish the requested task.

3.2.2.2 Data Memory Management unit

The DMM unit implements per-flow queuing for up to 512,000 flows and provides efficient queue handling, variable-length packet storage and access to specific packet segments by the processing engines to the PRO3 architecture. Its main functions are storing incoming traffic, receiving packet data and forwarding that data either to the processing modules or the output interface. It is able to handle both fixed and variable length packets.

Its interface consists of 4 ports, 2 incoming and 2 outgoing with a bandwidth of 2.5 Gbps each. One port is used to receive data from the network, one to transmit data to the network, and a bidirectional port for exchanging data from and to the internal bus.

In order to achieve the total aggregate througput of 10 Gbps, the DMM unit employs an optimized free list organization and memory access reordering. By using 2 separate free lists the memory accesses are reduced during buffer releasing by 70% during writes and 46% during reads. Also, by reordering the read and write commands — and subsequently their corresponding memory accesses — a 30% reduction in mean access latency is achieved.

3.2.2.3 Scheduler modules

A need for scheduling arises since the RPM engines sometimes process packets at a lower rate than that of the packet arrival rate. To face this issue, the

packets need to be stored and scheduled for processing once the target processing module is available. Concerning incoming traffic, the TSC maintains priority queues in order to schedule the forwarding of packets for processing according to a configurable priority per flow, which is set by the administrator. On the other hand, handling the scheduling of outgoing traffic is the TRS, which shapes transmitted traffic according to traffic management specifications. The overall scheduling is controlled by command and status messages issued by the DMM, the CPU and the Packet Preprocessor. These commands are routed to the appropriate scheduler through the Service and Redirect block.

Multiple flows are multiplexed round-robin in one scheduling queue. The TSC supports a total of 32 hierarchical scheduling queues, where the first queue is treated with highest priority than the others. The rest of the queues can be treated equally or by dividing them into 2 groups with different priorities. The TSC uses a connection table to store per-flow state information and connection-specific parameters such as the flow's priority weight. When flows map to the same scheduling queue, they are stored in a linked list. The TRS supports 32 scheduling queues as well and uses the same data structures as the TSC.

3.2.2.4 Reprogrammable Pipeline Module

This is the block that is mainly responsible for protocol processing in the PRO3 architecture. It is comprised of 3 units: the PPE, the Field Exctraction Engine (FEX) and the Field Modification Engine (FMO). These units form a 3-stage pipeline that constitutes the PRO3's software-processing heart.

The PPE lies in the core of the RPM and contains a RISC-CPU core and external control logic. The RISC-CPU core is a Hyperstone E1-32XS which is modified by partitioning its registers into two sets, where the first one is accesible by the CPU while the other by the PPE control logic. The processor switches register sets and informs the external logic — via a special instruction — once it has finished processing a packet. Direct Input/Output (I/O) is feasible through the processor's dual-port data memory. Network processing is substantially accelerated since I/O operations and packet processing can be performed concurrently, thus these actions' latency is partly hidden.

The FEX is responsible for parsing packet headers and subsequently loading the required protocol fields to the PPE in order to be processed. Its instruction set consists of 9 basic instructions and 4 commands (instructions with no arguments) which operate on a First-In, First-Out (FIFO) buffer of 32-bit words. Supported by the instruction set are variable-length field

extraction, backward and forward movement in the data FIFO, conditional branches — which are based on extracted fields and header parsing — and addition.

The FMO is tasked with packet construction or reconstruction and header modification. Its own instruction set includes 16 instructions and 6 commands, which support field extraction and insertion/modification.

The FEX and FMO use only 4 and 5 generic registers respectively, and additionaly a special-purpose register and a Data Pointer. Their instructions can be combined and executed in parallel with any of the commands, thus reducing code size and increasing performance. It is also worth mentioning that the FEX and FMO are optimized for bit and byte processing.

3.2.3 The PRO3 Architecture - Development Tools

On the software development side, the PRO3 is accompanied by a development suite used for configuring registers, initializing memory locations and loading software modules into the PRO3. The user is provided with a configuration GUI, which contains a separate page for each of the PRO3's hardware modules. On these pages, the user is able to configure or develop code for that specific module.

In its core, the development suite contains a Configuration Library (CL) which maintains an internal register map of the whole chip. The aforementioned GUI utilizes a set of functions implemented and exported by the CL in order to read and write the values of configuration registers and internal memory locations. The CL is also responsible for reading, parsing and compiling the configuration files and software programs for all of PRO3's processing units. Lastly, it maintains a TCP-based session with a configuration server which is running either on the on-chip or on-board control processor.

The development suite can be used to configure either the actual PRO3 chip, when connected with a development board, or a hardware description language model or netlist.

Chapter 4

Architecture

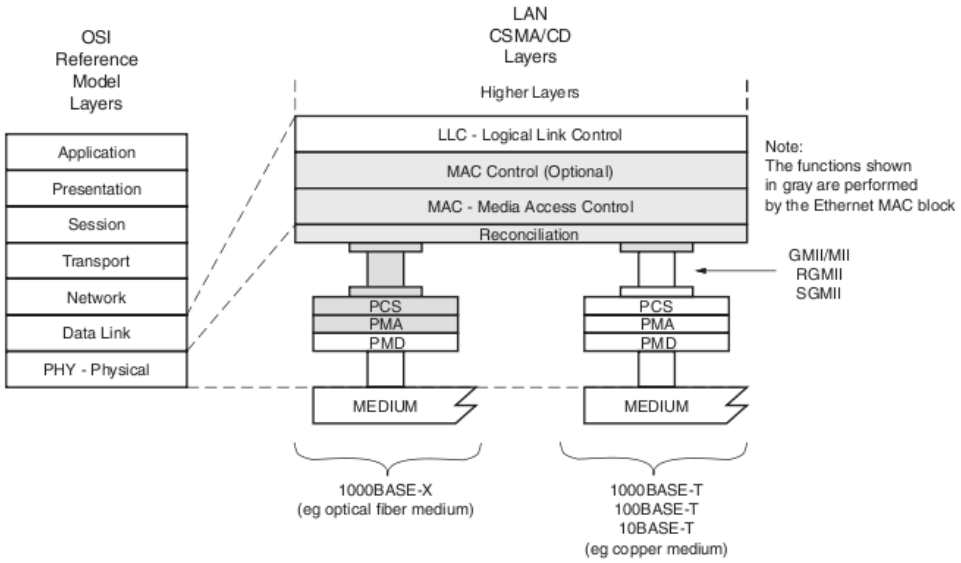
4.1 Virtex-5 FPGA Embedded Tri-mode Ethernet MAC

The Xilinx Virtex-5 (V5) FPGA board has an embedded Tri-mode Ethernet MAC (TEMAC) Wrapper [36, 37, 38], which is the base of our design and provides us with fully-fledged Physical and Data Link layers (see figure 4.1).

Its key features [36] are:

- Fully integrated 10/100/1000 Mbps Ethernet MACs
- Designed to the IEEE standard 802.3-2002 specification
- Configurable full-duplex operation in 10/100/1000 Mbps
- Configurable half-duplex operation in 10/100 Mbps
- Management Data Input/Output (MDIO) interface to manage objects in the Physical layer
- User-accesible raw statistic vector outputs
- Support for VLAN frames
- Configurable IFG adjustment in full-duplex operation
- Configurable in-band Frame Check Sequence (FCS) field passing on both transmit and receive paths
- Auto padding on transmit and stripping on receive paths

Figure 4.1: Virtex-5 Tri-mode Ethernet MAC-supplied OSI Layers



- Configured and monitored through a host interface
- Hardware-selectable Device Control Register (DCR) bus or generic host bus interface
- Configurable flow control through Ethernet MAC PAUSE frames; symmetrically or assymetrically enabled
- Configurable support for jumbo frames of any length
- Configurable receive address filter for unicast, general and broadcast addresses
- Media Independent Interface (MII), Gigabit Media Independent Interface (GMII) and Reduced Gigabit Media Independent Interface (RGMII)
- 1000BASE-X Physical Coding Sublayer (PCS) and Physical Medium Attachment (PMA) sublayer included for use with the Virtex-5 RocketIO serial transceivers to provide a complete on-chip 1000BASE-X implementation

- Serial Gigabit Media Independent Interface (SGMII) supported through the RocketIO serial transceivers' interfaces to external copper Physical layer for full-duplex operation

It is able to provide 2 Ethernet MACs, which can be configured independently. A comprehensive list of some of the available configuration options along with all the possible choices for each of them is given in table 4.1.

Table 4.1: Selection of configuration options for Virtex-5 FPGA Tri-mode Ethernet MAC

Option	**Choices**
Ethernet Speed	Tri-Speed 1000 Mbps 10/100 Mbps
Physical Layer Interface	MII GMII RGMII SGMII 1000BASE-X PCS/PMA (fibre)
Transmit (Tx) Flow Control Enable	True/False
Receive (Rx) Flow Control Enable	True/False
Jumbo Frame Enable	True/False
In-band FCS Enable	True/False
VLAN Enable	True/False
IFG Adjust Enable (Tx only)	True/False
Rx Disable Length (Rx only)	True/False
Address Filter Enable	True/False; if True, specify MAC Address

Using Xilinx's CORE Generator software, the V5 TEMAC is configured and the VHDL files that are needed in order to use it are generated; apart from the VHDL files, an example design is also generated, so one can use the TEMAC immediately. This example design connects the TEMAC with a client interface consisting of a Rx and a Tx FIFO, which are connected through a LocalLink Interface (LL) with an Address Swap Module (ASM). The example design functions as a loopback that receives frames from the TEMAC, stores them in the Rx FIFO, sends them through the LL to the ASM, which after exchanging the Destination MAC Address with the Source MAC

Address, sends the modified frame to the Tx FIFO, which in turn sends the frame to the TEMAC for transmission back through the board's network adapter.

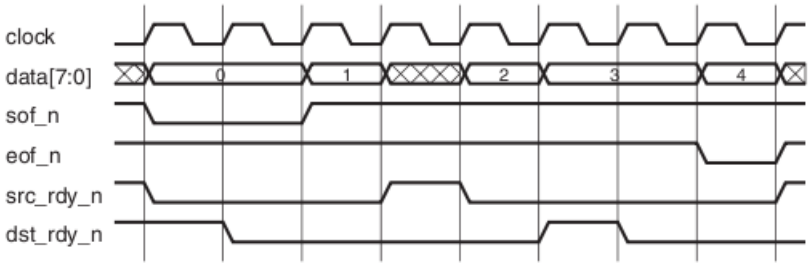
The LL is especially useful, since it renders communication with the TEMAC quite easier when compared to directly interfacing with the TEMAC component. The LL is described in table 4.2, where signals ending with "_n" are active low.

Table 4.2: LocalLink Interface

Path	Port Name	Type	Size (bits)
Receive (Rx)	rx_ll_clock	in	1
	rx_ll_reset	in	1
	rx_ll_data	out	8
	rx_ll_sof_n	out	1
	rx_ll_eof_n	out	1
	rx_ll_src_rdy_n	out	1
	rx_ll_dst_rdy_n	in	1
	rx_ll_fifo_status	out	4
Transmit (Tx)	tx_ll_clock	in	1
	tx_ll_reset	in	1
	tx_ll_data	in	8
	tx_ll_sof_n	in	1
	tx_ll_eof_n	in	1
	tx_ll_src_rdy_n	in	1
	tx_ll_dst_rdy_n	out	1

To get a better understanding of the way the LL operates, we are going to present a simple example (see figure 4.2). In order to exchange data through the LL, both src_rdy_n and dst_rdy_n have to be asserted at the same time; only then is data transferred across the LL. Should one of these two signals be deasserted at any time, transfer is paused, thus achieving flow control. The sof_n signal is asserted only at the beginning of the frame and respectively, the eof_n signal only at the end of the frame.

Figure 4.2: Frame Transfer with Flow Control across LocalLink Interface



4.2 Our Design

4.2.1 Configuration of the Tri-mode Ethernet MAC

The TEMAC used in our design is configured to have a single MAC, Tri-speed setting for the Ethernet speed, a SGMII Physical Layer interface, flow control for both Rx and Tx and an Address Filter set to a MAC address of "AA:BB:CC:DD:EE:FF".

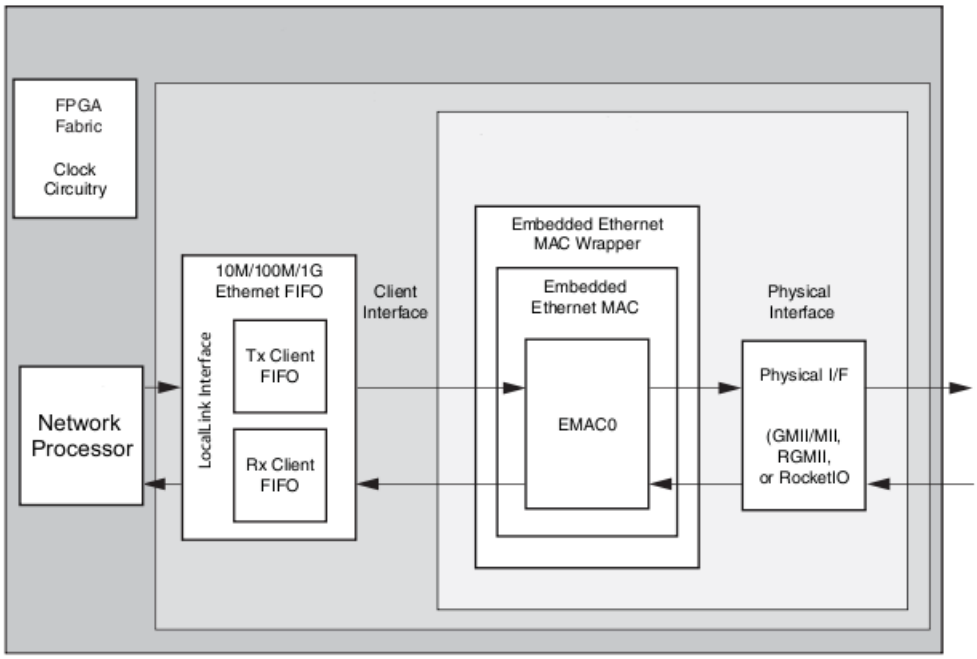
The reason we selected a Tri-speed setting, is mainly to be compatible with any network adapter we connect the design to, without the need to re-generate the TEMAC for a different speed setting. The SGMII Physical layer interface was selected since it radically reduces I/O count when compared to the other choices and of course because regular copper Ethernet cables are widely available when compared to fibre optic cables (thus ruling-out the 1000BASE-X PCS/PMA interface). Finally, the Address Filter is used so that our design will only process frames containing the specified MAC Destination address.

By selecting not to enable In-band FCS passing, we let the TEMAC check the CRC field of the frame and if it is erroneous, it will not pass it along to

our design. Moreover, this makes the TEMAC calculate the CRC field and add it to the frame before transmission. By not disabling the Rx length checking, the TEMAC is responsible for determining whether we have a length or type field and if it is a length field, to check if the value found in that field is consistent with the actual length of data in the frame. If it is not, then the frame is erroneous and so it is not passed from the TEMAC to our design. The latter choices take care of the necessary frame checks.

Our own design is implemented on the top level of the example design, as shown in figure 4.3, since we take advantage of the LocalLink Interface (see table 4.2) in order to communicate in a more high-level and understandable manner with the TEMAC.

Figure 4.3: Design Overview Block Diagram



4.2.2 Architecture - Overview

The Network Processor that we designed is implemented in a modular architecture with the datapath flow being as follows: connected with the Rx

Client FIFO of the example design is the Rx2Mem module (an abbreviated version of Receive-to-Memory), whose main objective is to pass along frame data as it is received from the FIFO to the Control module, to provide the frame's length, once it has finished transmission and to provide the number of the currently received frame. The Control module is essentially the module where we store data and information about the frames we have received and processed, before forwarding them to either the Process module or the Tx client FIFO. The Process module is the heart of our Network Processor, since it is the module where frame processing occurs, according to the software program loaded in its instruction memory. After processing has finished, frame data is sent from the Process module back to the Control module, where it is stored either to be processed furthermore or to be transmitted to the Tx FIFO.

The commands supported by our NP are listed in table 4.3, showcasing their syntax and their corresponding opcode.

Table 4.3: Command Syntax and Opcodes

Instruction	**Syntax**	**Opcode**
Add	FrameNumber, Address1, Data	000
Change	FrameNumber, Address1, Data	001
Compare	FrameNumber, Data	100
Exchange	FrameNumber, Address1, Address2	101
Remove	FrameNumber, Address1	010
Report	FrameNumber, Address1	011
Transfer	FrameNumber	110

The functions performed by each command are:

Add adds Data in position Address1 in the frame FrameNumber

Change replaces data in position Address1 of frame FrameNumber with Data

Compare compares (searches) frame FrameNumber for Data and returns byte number, if found

Exchange exchanges data from position Address1 with data from position Address2 in frame FrameNumber

Remove removes data in position Address1 from frame FrameNumber

Report reports data in position Address1 from frame FrameNumber

Transfer commences the transmission of frame FrameNumber

Commands refer to frames in a sequential manner and a Non-Transfer policy is followed. In order to make things a little bit clearer, the sequential manner of commands refers to the fact that commands are structured in a way that FrameNumbers are sequential: if a batch of commands refer to frame five, all subsequent commands must refer to either frame five or greater. The Non-Transfer policy means that a frame is transferred exclusively with the use of the Transfer command. So, if a frame does not have a Transfer command associated with it, no transmission of this frame's data is going to occur, regardless of whether other commands have processed it or not.

4.2.3 Architecture - In-depth

Now we are going to provide an in-depth look at each of the modules of the architecture, their interfaces, their inside structure and their operation.

4.2.3.1 Rx2Mem

The Rx2Mem module is the first module of our design, whose purpose is to forward frame data received through the LL to the Control module, to calculate the length of each frame and to keep a count of all the frames it has received. Presented in table 4.4 is the module's interface, while in figure 4.4 a block diagram is given.

The FSM in figure 4.4 has a total of 5 states and is responsible for controlling the byte and frame counters, generating the start_addr, addr_inc and frame_done outputs of the Rx2Mem module and finally detecting whether we have a back-to-back frame transmission — this occurs when immediately after the eof_n signal has been asserted, the sof_n signal is asserted. The byte counter is capable of properly keeping count of the number of bytes a frame has, even in back-to-back transmission of frames, thus being capable of providing valid information about each frame. The frame counter increases its output each time a new frame begins transmission. To get a detailed view of the way the FSM in the Rx2Mem module operates, consult figure 4.5, where states and transitions are presented.

Table 4.4: Rx2Mem Interface

Name	**Type**	**Length**	
clk	in	1	
reset	in	1	**Module**
sof_n_i	in	1	Rx FIFO
eof_n_i	in	1	
data_i	in	8	
src_rdy_n_i	in	1	
dst_rdy_n_o	out	1	
dst_rdy_n_i	in	1	Control
start_addr	out	1	
data_o	out	8	
frame_num	out	10	
frame_len	out	11	
frame_done	out	1	
addr_inc	out	1	

Figure 4.4: Rx2Mem Block Diagram

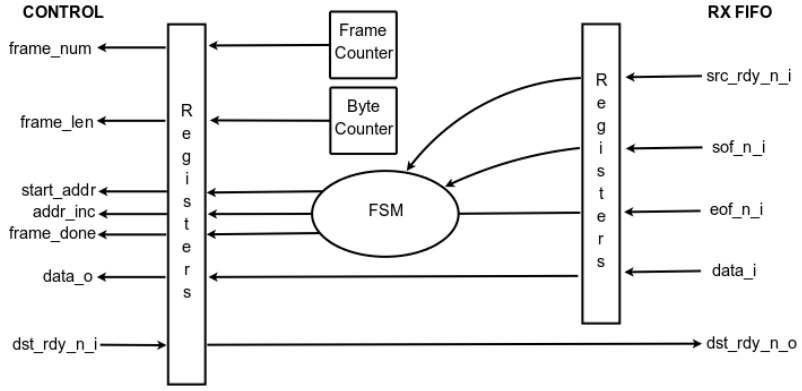
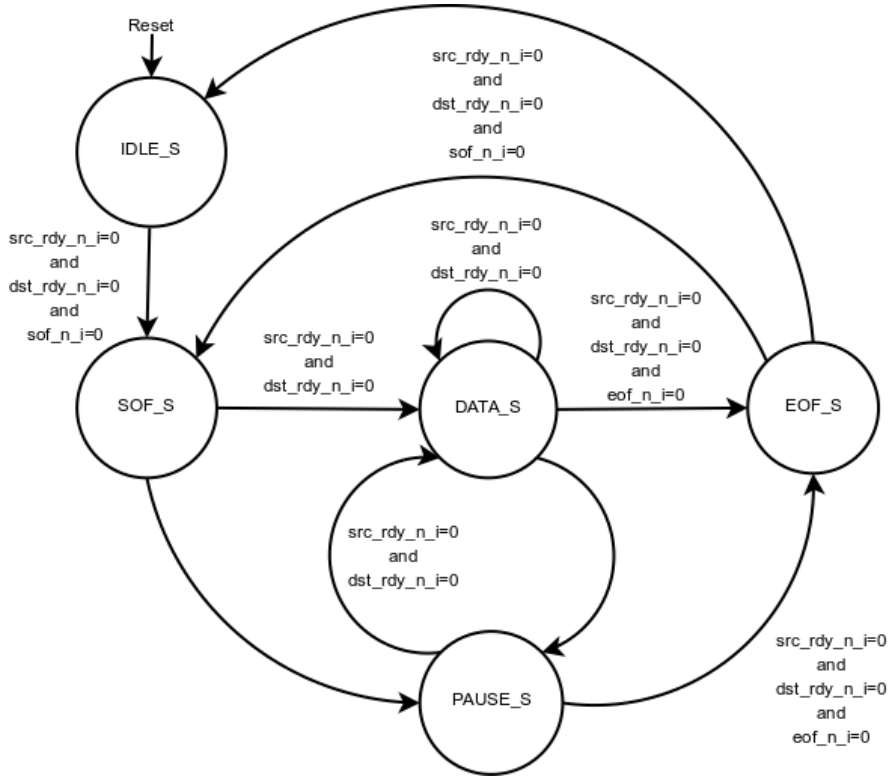


Figure 4.5: Rx2Mem FSM



4.2.3.2 Control

This module is the heart of our architecture. It is heavily dependent on FSMs to accomplish its task, which is the management of all the frame data, whether they be incoming from the Rx FIFO, outgoing to process, incoming from process or finally outgoing to Tx FIFO. It can be abstractly divided into 4 parts: the R2M part (Rx-to-Memory), the M2P part (Memory-to-Process), the P2M part (Process-to-Memory) and the M2T part (Memory-to-Tx). In between the aforementioned parts, lies the data core of our NP. That is, three Block RAMs (BRAMs) of 64 kB each that hold frame data, two BRAMs of 27 kB each that hold information about the frames we have received and the frames we have finished processing and finally a FIFO used to queue outgoing frame information.

In the first data BRAM we store incoming frame data, and it is called data_ram_Rx. The second (called data_ram_Tx) and third (data_ram_processed) BRAMs actually contain mirrored data, as they are both used to store processed data. The reason behind this data mirroring is the fact that we are using simple dual port BRAMs, where the first port is used exclusively for writing while the second exclusively for reading. Subsequently, in order to avoid delays if a frame is to be transferred and another to be processed at the same time — and since we have no space issue on the V5 development board — we chose to instantiate 2 BRAMs for storing processed data and mirror their content. In that way, the first can be used to transfer frame data to the output, while the second can be used to send data to be processed. All the aforementioned BRAMs have a word size of 8 bits.

We have two BRAMs used to store information about frames. These BRAMs can store information for up to 1024 frames in total. With a word size of 27 bits, the information stored about each frame is the address where the frame's data starts in the data BRAM and the length of this frame's data.

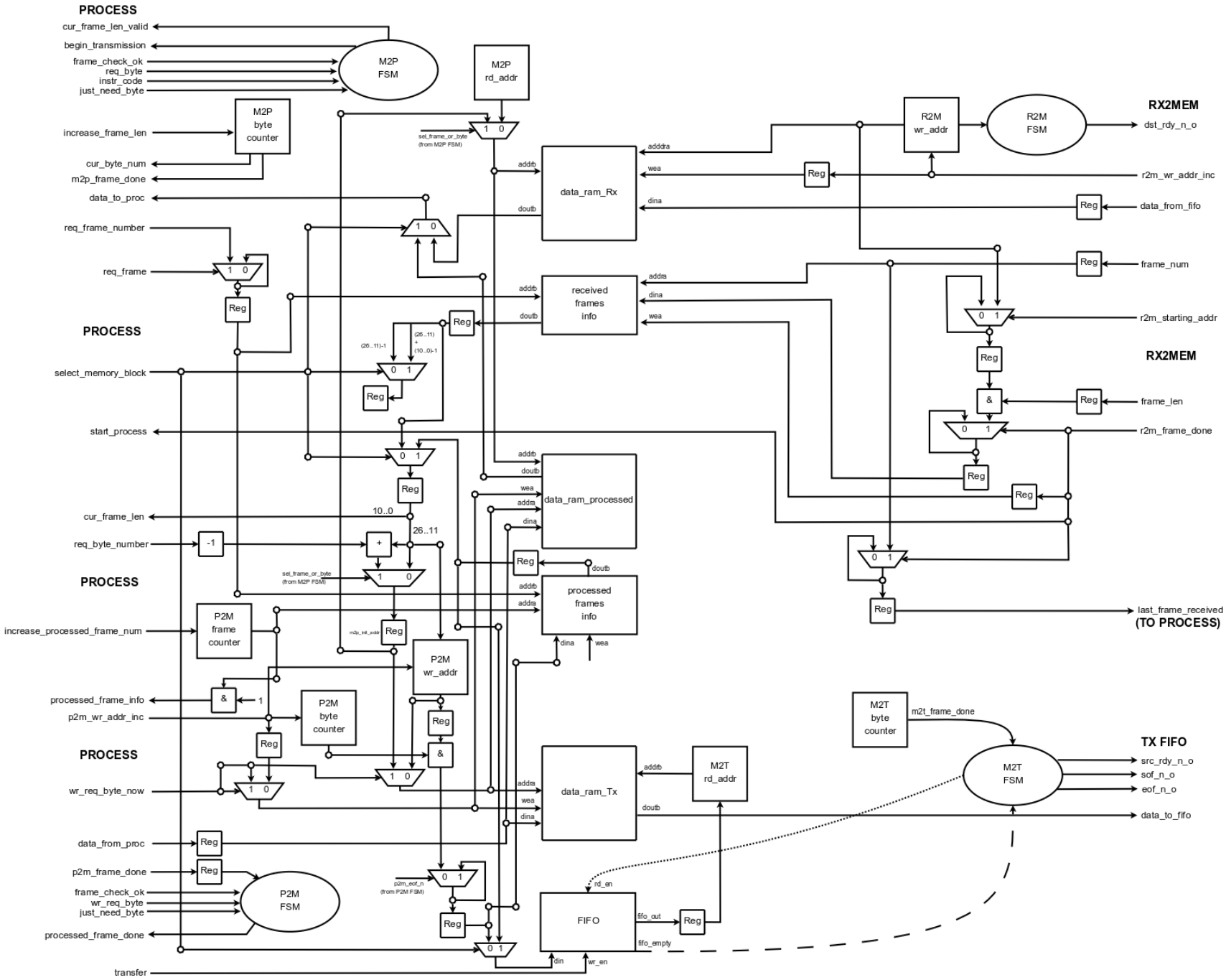
Finally, the FIFO is used as a queue of the frames that are waiting to be transmitted to the MAC. In it we store information regarding those frames and not actual frame data, since the information provided is the frame's first byte address in data_ram_Tx and its length in bytes. Using these pieces of information, the M2T FSM takes care of the transfer of data from the BRAM to the output.

In table 4.5 we give the interface of this module, while in figure 4.6 we present its complete block diagram. In that block diagram the 4 FSMs of the module are visible, yet their full presentation is done using some detailed figures. Specifically, figure 4.7 presents the R2M FSM, figure 4.8 the M2P FSM, figure 4.9 the P2M FSM and finally figure 4.10 presents the M2T FSM.

Table 4.5: Control Interface

Name	Type	Length	
clk	in	1	
reset	in	1	**Module**
dst_rdy_n_o	out	1	
r2m_starting_addr	in	1	
data_from_fifo	in	8	
frame_num	in	10	Rx2Mem
frame_len	in	11	
r2m_frame_done	in	1	
r2m_wr_addr_inc	in	1	
m2p_frame_done	out	1	
p2m_frame_done	in	1	
req_frame	in	1	
req_frame_number	in	10	
req_byte	in	1	
req_byte_number	in	11	
wr_req_byte	in	1	
wr_req_byte_now	in	1	
frame_check_ok	in	1	
select_memory_block	in	1	
just_need_byte	in	1	
instr_code	in	3	
transfer	in	1	Process
increase_frame_len	in	1	
cur_frame_len	out	11	
cur_frame_len_valid	out	1	
begin_transmission	out	1	
cur_byte_num	out	11	
last_frame_received	out	10	
start_process	out	1	
processed_frame_info	out	11	
processed_frame_done	out	1	
increase_processed_frame_num	in	1	
p2m_wr_addr_inc	in	1	
data_to_proc	out	8	
data_from_proc	in	8	
data_to_fifo	out	8	
src_rdy_n_o	out	1	Tx FIFO
sof_n_o	out	1	
eof_n_o	out	1	

Figure 4.6: Control Block Diagram



4.2.3.3 R2M

In the following segments we are going to give an in depth look of each of the 4 abstract parts that make up the Control module, starting with the R2M part.

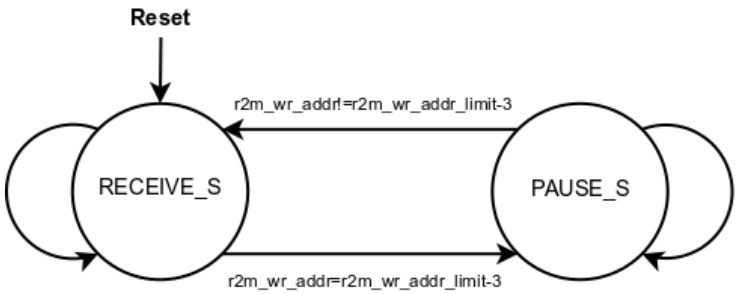
In the R2M part, data is received from the Rx2Mem module, along with signals indicating that the current memory address is the frame's starting address, the frame's number (according to the frame counter in Rx2Mem), the frame's length in bytes along with a signal indicating that the frame is finished. Finally a last signal is used to increment the memory address as each new byte is transferred from Rx2Mem. All the aforementioned signals originate from the Rx2Mem module and are used to control the proper memory allocation of data, along with correct handling of frame information.

The write address of data_ram_Rx is incremented using the r2m_wr_addr_inc signal, which is also used as the write enable signal of data_ram_Rx, after passed through a register in order to synchronize it with the actual data of the frame. Using the r2m_starting_addr signal which indicates that this is the frame's first byte, we store the frame's starting address in a register in order to use it later on in the assembly of the frame's information, concatenating it with the frame's length — provided by the frame_len signal — upon the completion of its transfer, indicated by the r2m_frame_done signal. These two pieces of information concatenated form the frame's information which is registered and then input to the corresponding BRAM. Addressing this BRAM is the frame's number, provided by the frame_num signal, while enabling the write port is accomplished by using the r2m_frame_done signal. The aforementioned signal is passed along to the Process module through the interface's start_process port, to indicate that the first frame has been accepted into the BRAM and thus the processing can commence. Also passed to the Process module is the frame_num in order for it to be aware of the last frame number we have received. This is done using the last_frame_processed output port of the interface, which is updated for each frame we receive.

Its FSM (see figure 4.7) is quite simple, since it only needs 2 states, one to receive data and one to pause the transferring of data from the Rx2Mem module. The latter state is used in order to protect the overwriting of data in the data_ram_Rx memory, since frames could be received at a very fast rate.

Taking into consideration the fact that we process frames sequentially, memory management is essential. This is achieved using a register that holds the so called r2m_wr_addr_limit, that is the limit the r2m_wr_addr pointer is allowed to reach when writing new data. To make it clearer,

Figure 4.7: R2M FSM



the address pointer is incremented with each new byte of data and since it is allowed to wrap — if it reaches all ones and incremented once more, it turns into all zeroes thus wrapping and starting over — it can overwrite pre-existing bytes. These bytes can sometimes be needed, since for example, we could be processing frame number five and while at it, new frames have been written in memory and have caused wrapping of the address pointer. In such a case, we could continue writing new data and then reach the starting address of frame number five, which is still under process. We cannot allow the Rx2Mem module to continue increasing the address pointer and overwrite data belonging to frame five. In order to achieve this we use the limit register by checking the current address pointer's value against the one in the aforementioned register. Should it be within 3 bytes of that register's value, we pause the transfer of data from the Rx2Mem module (using dst_rdy_n_o output port of Control), by transitioning to the pause state displayed in figure 4.7. In that way, we cause the Rx FIFO in the example design to store data without passing it directly onto our design, in that way gaining some time to finish processing frame five and change the limit register to the starting address of frame six.

4.2.3.4 M2P

After that, the M2P part begins. Here we have the most complex function of the module, which is to pass the requested frame data to the Process module after determining from which BRAM we shall take the data from, along with setting up the address pointers to the correct place, according to the requested frame's corresponding start address. Furthermore, if a frame has already been processed, we can set the address pointer to a specific

byte's address, in order to minimize latency of commands such as Change, Exchange and Report. To do that, a register is initialized every time we want to process a frame to the value of either the frame's starting address or the frame's starting address plus the requested byte's offset. The requested frame's number along with the requested byte number is provided by the Process module, along with signals indicating that we do want a frame (req_frame) or byte (req_byte). At the cases where we have a Change, Exchange or Report commands and the frame has already been processed, we intend to alter/request only a specific byte; this is indicated through the just_need_byte signal, also originating from the Process module. Another useful signal is the wr_req_byte signal, which is used to determine whether we need to write a byte to the memory or not.

Once we have initialized the address pointers using the start address in the received or processed frames info (depending on whether the requested frame has already been processed or not, using signal select_memory_block), we begin transmission of the frame's data (all of them if the frame is being processed for the first time or if the command executed is one of Add, Compare or Report; one of them if the frame has already been processed and the executed command is Change, Exchange or Report; a special case exists concerning the Transfer command, where if the frame has already been processed, no data is transmitted to the Process module). The aforementioned address pointer is used in both the data_ram_Rx and data_ram_processed BRAMs, so we receive the proper data using the same pointer.

Transmission keeps going until we reach the final byte of the frame; this is accomplished using a byte counter, counting each of the bytes we transfer up until this number reaches the frame's length minus one. When we reach the penultimate byte, we signal to Process that the frame is finished using the m2p_frame_done signal, so it knows that it is about to receive the last byte of the requested frame.

In case we are executing an Add command, we need to increase the frame's length, since the length used in the aforementioned check would cause us to stop transmitting one byte earlier. This is done using the signal increase_frame_len. The counter outputs the current byte number to the Process module, so the latter will be able to execute its command at the proper byte.

The FSM employed in this part 4.8, is quite complex compared to the previous one we presented (R2M - figure 4.7). Needed to make some of the transitions is the opcode of the in execution command, so it is passed by the Process module through the instr_code signal. Also necessary is the signal indicating that the frame has passed all the necessary checks in the Process

module; this signal is the frame_check_ok signal.

The multitude of states concerning the requesting of the frame is explained by the fact that as we have already mentioned, we need to load information concerning the requested frame from the proper BRAM and then we need to initialize the address pointer to the correct position.

After that we send the frame's length to Process, in order for it to check whether the bytes requested by the currently loaded command are in range (for example the command can concern byte number 200 while the frame only has 100 bytes). For that reason we wait for a confirmation from Process that these checks have been passed; if not, we return to an Idle state, waiting for the next request by the Process module.

If succesful, we either begin transmitting data to Process, or proceed to sending a specific byte; alternatively, we can remain in a state where we allow data to be written to the proper target memory.

Concerning the signal sel_frame_or_byte shown in figure 4.6, which is generated by the FSM and used for controlling two multiplexers, it is asserted (active high) during states Req_Byte, Pre_Send_Byte, Send_Byte and Write_Byte, thus changing the output of the multiplexers according to the command's need. The top multiplexer, when the aforementioned signal is asserted, sets the read address of data_ram_Rx and data_ram_processed to be equal to the starting address of the frame plus a byte offset, thus providing the requested byte's data in the next clock cycle. The bottom multiplexer sets the initialization address equal to the aforementioned sum. When deasserted, the value is equal to just the starting address of the frame.

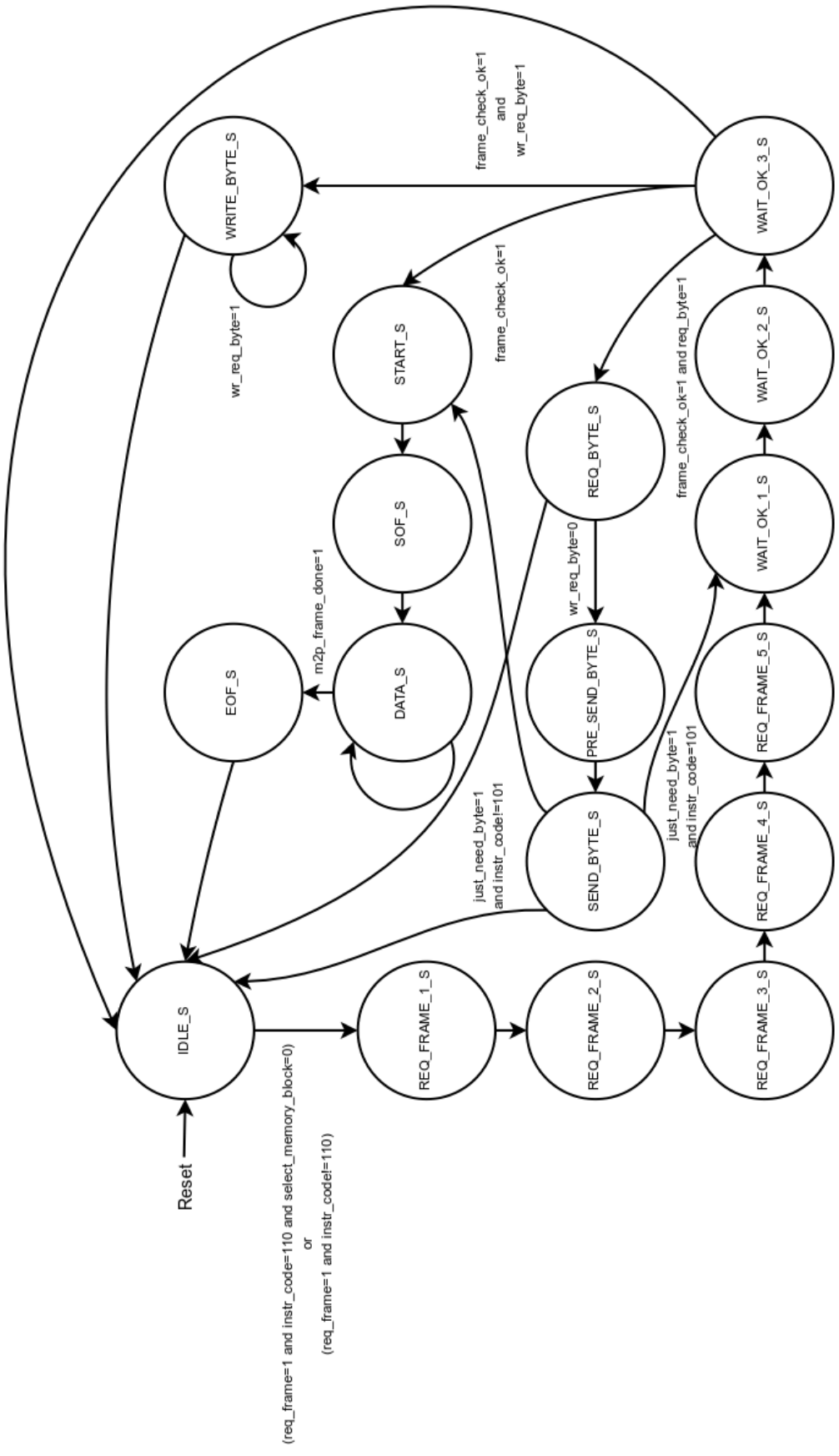


Figure 4.8: M2P FSM

4.2.3.5 P2M

P2M is the third part comprising our Control module. It is responsible for receiving the processed frame data from the Process module and storing them properly in the two BRAMs it controls, data_ram_processed and data_ram_Tx, which as we have already mentioned contain mirrored data. It also assembles and stores each processed frame's information — its starting address and its length. Finally, it is responsible for queuing information about the frames to be transferred to the LL in the FIFO it maintains, in order for them to be processed later on by the fourth and final part of Control, the M2T part.

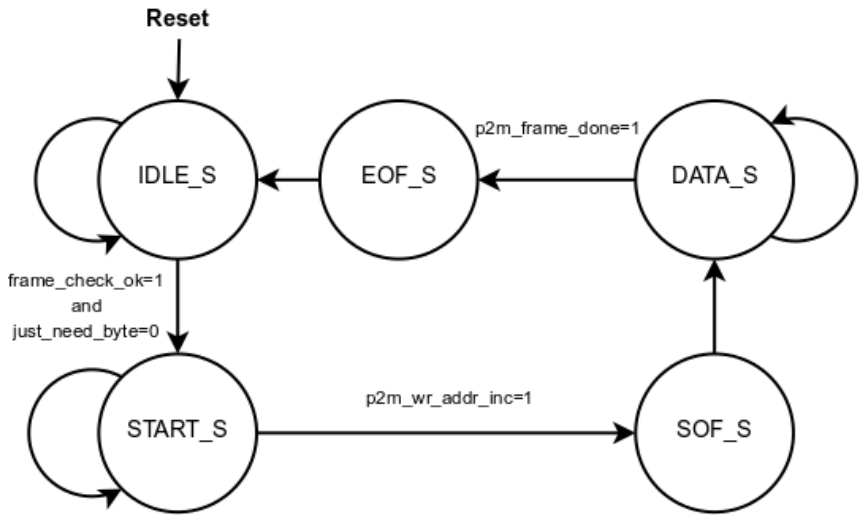
P2M employs another frame counter, which keeps track of the number of frames processed so far. This counter's output is used for two purposes: first, it is fed back to the Process module after concatenation with a single 1 as processed_frame_info, which is used in Process to maintain a frame substitution BRAM (more on that when we provide a more detailed description of the Process module). Second, it is used to address the processed frames information BRAM, in order to store the information about this frame in its according position.

Its write address can be initialized to the starting address of the frame (if it has already been processed that is) in order to overwrite processed frame data — if we execute another command referring to the same (already processed) frame — or to the frame's starting address incremented by an offset if we only want to write a specific byte. Controlling the write address pointer's incrementation is the signal p2m_wr_addr_inc which is generated by the Process module. This signal is also used at the write enable port of data_ram_processed and data_ram_Tx after it has been passed through a register, in order to synchronize it with the bytes of data, since it precedes them by a single clock cycle. Alternatively, the aforementioned BRAMs' write enable port is connected to the signal wr_req_byte_now — also coming from Process — in order to enable writing to the BRAMs when a single write is required. The write address value selection is accomplished by the multiplexer connected to the write address port (port A) of the data BRAMs, controlled by the signal wr_req_byte_now, while enabling write operations on these BRAMs is achieved by another multiplexer connected to their write enable ports.

In this part we have another counter used for counting bytes as we receive them, in order to produce the processed frame's length and recalculate it everytime it is processed again, since its length can be altered by commands such as Add and Remove.

The starting address of the currently processed frame is stored in a register and after concatenating it with the length of that frame, we store it in the processed frames' information BRAM. This information can also be passed on to the FIFO used to queue frames for transmission, if a transfer command is executed.

Figure 4.9: P2M FSM



This part's FSM (shown in figure 4.9) is quite simple; it will wait for the Process module's frame check we have already mentioned in the M2P part to finish and if the command under execution is not a single byte operation, it will commence its operation by transitioning from the Idle to the Start state. After that, it waits for the assertion of the p2m_wr_addr_inc signal — which is an indicator that in the next clock cycle we will have incoming data from the Process module — to transition to the Sof state and then to the Data state, where it will remain until it receives the p2m_frame_done signal, meaning that the last byte of data is about to be received. After that is done, it returns to an Idle state. A Pause state is not necessary, since control of the address pointer of the target memory is done by the Process module, through its p2m_wr_addr_inc signal.

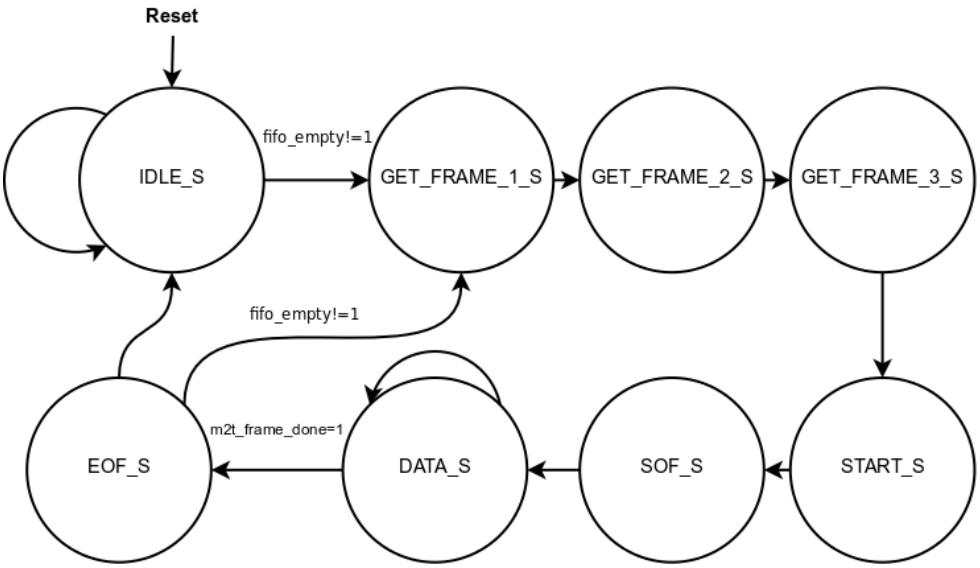
The FSM generates the output signal processed_frame_done which is asserted high only during Eof state and only if the frame is being processed for the first time. During the Eof state, another signal used for controlling a multiplexer is asserted low, the p2m_eof_n signal. As we can see in fig-

ure 4.6, that multiplexer is located at the bottom part, next to the FIFO and is used to control whether the frame information stored in a register is to be renewed or not.

4.2.3.6 M2T

This is the final part that completes the Control module and its functionality. It is the only part not directly depending on any other of the modules of our design, since its actions are purely the initialization of an address pointer on the read port of the data_memory_Tx BRAM and the proper transmission of a frame's data, along with signalling its beginning and its end. The end of the frame is recognized through the use of another counter and the frame's length; when the counter reaches the penultimate byte number, we know we are about to transfer the last byte of the frame, thus asserting the eof_n signal (active low).

Figure 4.10: M2T FSM



M2T's FSM (depicted in figure 4.10) is quite similar to P2M's FSM, only it contains three more states, used to get the information concerning the frame to be transferred and initialize the read address pointer. Transition from its Idle state occurs when the FIFO is no longer empty or when a frame is in

queue to be transferred. After that, we get that frame's information, initialize the address pointer and then proceed with transferring of the frame's data, which is done until the m2t_frame_done signal is asserted high, thus indicating that the last byte is to be transmitted. After that, we have two paths: either proceed to fetch the next frame's information and begin its transmission, or return to the Idle state.

4.2.3.7 Process

The Process module is the final component of our design and essentially the module where all the frame processing occurs. Contained within this module's architecture is an instruction Read Only Memory (ROM) holding all the instructions/commands of the program we want to execute, a frame substitution BRAM and a FSM that takes care of all the necessary actions needed to perform the task at hand. Its interface is presented in table 4.7, while its complete block diagram is given in figure 4.11. The FSM, all its states and their respective transitions are presented in figure 4.12.

The command ROM has a capacity of 1024 commands, each command 35 bits long, making it a total size of 35 kB.

Besides the command ROM, the Process module also contains a frame substitution BRAM capable of storing a total of 1024 10 bit words (total size 10 kB) used to substitute frame numbers. Let us justify this BRAM's existence. Since we already mentioned that once a frame has been processed it is no longer acquired from data_memory_Rx BRAM but from data_memory_processed BRAM, the frame number provided with the command executed is valid only for the first time a command on the specified frame is executed, since the processed frame, if processed again, is in a new position in data_memory_processed, no longer corresponding to the information found in received frames' information BRAM. An example might help to make things clearer: let's suppose that the first command we execute is an Add on frame number five. Data_memory_processed is going to store its first frame, since it is the first frame we are processing, thus making subsequent commands on frame five actually referring to frame one in data_memory_processed. To counter that, we introduce the aforementioned frame substitution BRAM, which is used by the Process module to check if the loaded command's frame has already been processed and if so, substitutes it with its new FrameNumber. This BRAM's operation is accomplished by using the FrameNumber provided with each command as an address pointer and the data corresponding to this address being indication whether the frame has already been processed or not and if so, its substitute FrameNumber. The tenth bit of each word

is used as the frame_already_processed indication while the remaining nine bits give the substitute_frame_number. This information is provided by the Control module upon each frame's first completed command through the processed_frame_info and processed_frame_done inputs, used at the BRAM's data input and write enable ports respectively. To sum up, each time a new command is fetched by the command BRAM, the FrameNumber provided is set as the read address in the frame substitution BRAM and the latter's output is checked; if that frame has already been processed, the requested frame number (provided by the req_frame_number output port) is set as the subtitute frame number provided. If not, req_frame_number is set as the FrameNumber provided by the command. This functionality is achieved by a multiplexer controlled by the frame_already_processed signal mentioned above.

We are now going to give a detailed presentation of the FSM and its transitions. The Process module is initially in Idle state and remains there until start_process input port is asserted high from Control. This signals that the first frame has been written in memory and so we can begin the processing.

Processing begins by initially fetching the next command from the instruction BRAM (FSM states Fetch_IR 1,2 and 3) and then decoding (Decode state) its different parts according to table 4.6.

Table 4.6: Command bit decomposition

Signal	**Bits**
Opcode	34 . . . 32
FrameNumber	31 . . . 22
Address1	21 . . . 11
Address2	10 . . . 0
Data	7 . . . 0

In the same state (Decode) it checks if the frame to be processed has been received or not and if not, it remains in that state until it has. Moreover, checks are conducted concerning the command's opcode and only if the frame is in memory and the opcode is one of the supported ones are we allowed to continue on to the next state. Should a command contain an opcode unknown to Process, we transition to the Error_Opcode state where we store the command's address in a register and fetch the next one.

If all checks are passed, the actual frame is requested from the Control

module by using the req_frame (active high) and req_frame_number outputs in Req_Frame_1 state. After that we move to Req_Frame_2 state, where we wait while Control sends back the frame's length. This is necessary in order to check if the byte address(es) in the command are actually in range for the requested frame (for example a command could mean to Add to byte number 100 in frame 5, while frame 5 only has 64 bytes; this would be erroneous).

A single situation is an exception to the norm when we are in Req_Frame_2 state: if the command decoded is a Transfer command and the frame has already been processed, this means that Control can begin the frame's transmission to the LL without the need to check the frame's length, so we transit to the Transfer state where we activate the transfer output (active high) to Control and then fetch the next instruction.

Back to the norm now, when the frame length is provided through input port frame_len (and its validity indicated by frame_len_valid input port), we transit to the Check_Frame_Length state, in order to perform the aforementioned check. If the byte requested by the command is out of range, this is of course an error, so we transit to the Error_Byte state reporting once again the erroneous command's address and fetching the next.

If the check is passed however, we have a selection of states to transit to: if we have any of the commands Add, Change, Remove or Report and the frame has not been processed before or if the command is a Compare or a Transfer, we transit to Send_OK state, where we acknowledge to Control that all checks have passed and we can proceed with data transmission after M2P initializes all its memory address pointers as we have already described in the respective part of Control. After that we move on to the Start state and await for regular frame transmission to commence from Control; this is indicated through input port begin_transmission. After that we move on to Sof state, afterwards Data state and when receiving_done input port is asserted high and the command under execution does not refer to the last byte of the frame (indicated by internal active high signal last_byte_action), we move to Eof state. After the Eof state, we fetch the next command to be processed, and so return to the beginning of this description, or if a Transfer command for a non processed frame was executed, the FSM's current state transitions to Pre_Transfer state and then to Transfer state, in order to commence the frame's tranmission after we have succesfully written all the frame's data in data_memory_processed and data_memory_Tx BRAMs.

Back to Check_Frame_Length state, if the frame has already been processed and the requested command is a Change or a Report, or if we have an Exchange command, regardless of the frame having already been pro-

cessed or not, we move to Change_Byte_1, Req_Min_1 or Req_Max_1 states respectively.

When a Change command is executed and the frame has been processed before, transition to the Change_Byte_1 state occurs and then to Change_Byte_2 and 3 states where we directly write the specific byte of data provided with the command to the BRAM in Control. After that execution of the command has finished and we fetch the next one.

When a Report command is in progress on an already processed frame, transition to Req_Min_1 state occurs, followed by subsequent transitions to Req_Min_2, 3, 4 and 5 states, to end up in state Report, where after that execution is completed and Process fetches the next command.

When an Exchange command is loaded, the FSM moves from the Check_Frame_Length state to Req_Max_1 state and subsequently to Req_Max_2, 3, 4, Get_Max and Store_Max states, in order to get the data residing in byte number Address2 of the currently processed frame from Control. If the frame is being processed for the first time (frame_already_processed=0) then we go to Start state. If it has been processed before, we move to Req_Min_1 (and then 2, 3, 4), Get_Min and Store_Min states, to get the data from byte number Address2 of the frame. After that we reuse the Change_Byte states (1 to 3) also used in Change commands to alter the data of byte number Address1 in memory and after that we continue on to respective Change_Byte_4, 5 and 6 states to alter the data of byte number Address2. After that the command has finished execution and we proceed to fetch the next instruction.

Finishing up with the normal state transition description, a careful look at figure 4.12 will show that once in Start state transitions can occur to command-specific states depending on a couple of parameters; these parameters being detection of whether the command is referring to the first byte of data in the frame (indicated by the first_byte_action internal signal) and the command's opcode. If in Sof or Data states, similar transitions to command-specific states can occur when the requested byte address is reached; by checking the command's opcode yet again, we determine the target transition state.

When an Add command is in execution, we will transit to the Add state; Change, Remove, Report and Exchange_1 are the target transition states for each of the respective commands. While in these states, if the receiving_done input port is equal to 1, transition to Eof state is dictated; if last_byte_action signal is equal to 1, this means the command was referring to the last byte of the frame and thus has finished processing that frame, so we need to fetch the next command by transitioning to Fetch_IR_1 state; finally, if none of

these conditions apply, we return to Data state. When in Exchange_1 state, if the second byte to be exchanged is right after the first, we transition to Exchange_2 state and then the same conditions as mentioned above apply. Finally, when in Report state, if the frame has already been processed, we have finished this command's execution and proceed to fetch the next command.

That wraps up the description of all the frame transitions of the FSM in the Process module.

Next we are going to explain some of the inner workings of the module concerning actual execution of the commands, besides of the state transitions we have just described.

Inbound data from the Control module (M2P part) is output back to Control (to the P2M part) as it is received. When a command alters frame data — the commands that do that are Add, Change, Exchange and Remove — the required functionality of outputting proper data is accomplished by the use of the small chain of multiplexers depicted in the lower left part of the module's block diagram (see figure 4.11). To have a better understanding of the way each command operates on frame data and the way these multiplexers come into play, we shall present a detailed look at each of the commands.

Starting with the Add command, we constantly check the current byte number (provided by input port byte_number), in order to identify when the position in which we are going to add data to the frame comes. Then, we output the Data provided by the command and after that we keep transmitting frame data as it is received from Control. To achieve that, we use the two multiplexers controlled by signals delay_sel and change_mux_sel. The first multiplexer selects whether we want the actual data or the data delayed by one clock cycle, using a simple byte register. The second multiplexer is used to select the incoming frame data or the data provided by the command. Therefore, when the position to add data to the frame has been reached, change_mux_sel is set to 1, thus selecting to output the Data part of the command and in the next clock cycle, it is set to 0, outputting the frame data. However, should we just output frame data, we would actually just replace data in position Address1 and not Add to that position, since by using this multiplexer alone we would miss the byte located at position Address1. To counter that, the second multiplexer is used, the one controlled by delay_sel. This signal is set to 0 during an Add command, thus outputting regular frame data as it is received. Once Address1 has passed though, it is set to 1, thus outputting delayed (by one clock cycle) frame data; if of course Address1 is number 1 — the first byte —, then delay_sel is set to 1 from the

beginning of the execution. This allows for proper data insertion — as an Add command dictates — without missing any byte of data. During an Add command, the last multiplexer in the data flow following the aforementioned two multiplexers, has its select signal (perform_exchange) constantly set to 0.

In a Change command, data is output in a regular fashion, until byte number equal to Address1 arrives. To change that byte's data to the Data part of the command, we use the same chain of multiplexers yet again. This time, delay_sel and perform_exchange are constantly set to 0, while change_mux_sel is set to 1 only when we receive byte from Address1, in order to accomplish the desired change. After that, change_mux_sel is once again set to 0 and transmission resumes in a regular fashion.

In a Compare command, frame data is transmitted back to Control as it is received, so all the multiplexer select signals we have mentioned above remain set to 0, while a simple process constantly checks incoming data against the Data part of the Compare command. If a match is found, the byte number is stored in a register for later use.

When executing an Exchange command, the most complex command of our instruction set, additional steps are required. To begin with, data from positions Address1 and Address2 of the frame being processed are stored in a respective register each, conveniently named address1_data and address2_data. After that is done, a multiplexer controlled by internal signal exchange_mux_sel controls which register to read from (0 for the first, 1 for the second). This multiplexer's output is sent to the last multiplexer mentioned above, the one controlled by perform_exchange. The latter signal is set to 0 while the byte number is equal to neither Address1 or Address2. When it is equal to either of the aforementioned values, it is set to 1 thus outputting the data contained in the registers we have mentioned just above. The other two multiplexers' select signals are constantly set to 0 during an Exchange command.

During a Remove command, these multiplexers are not used at all, so all their select signals are constantly 0; this is because the removal of a particular byte of data is accomplished through the deactivation of the p2m_wr_addr_inc output port when that byte number is reached; this output port controls the address pointer incrementation in the P2M part of control, along with the write enabling of the memories controlled by P2M, so when it is deactivated no writing occurs, effectively ignoring the byte of data we want to remove.

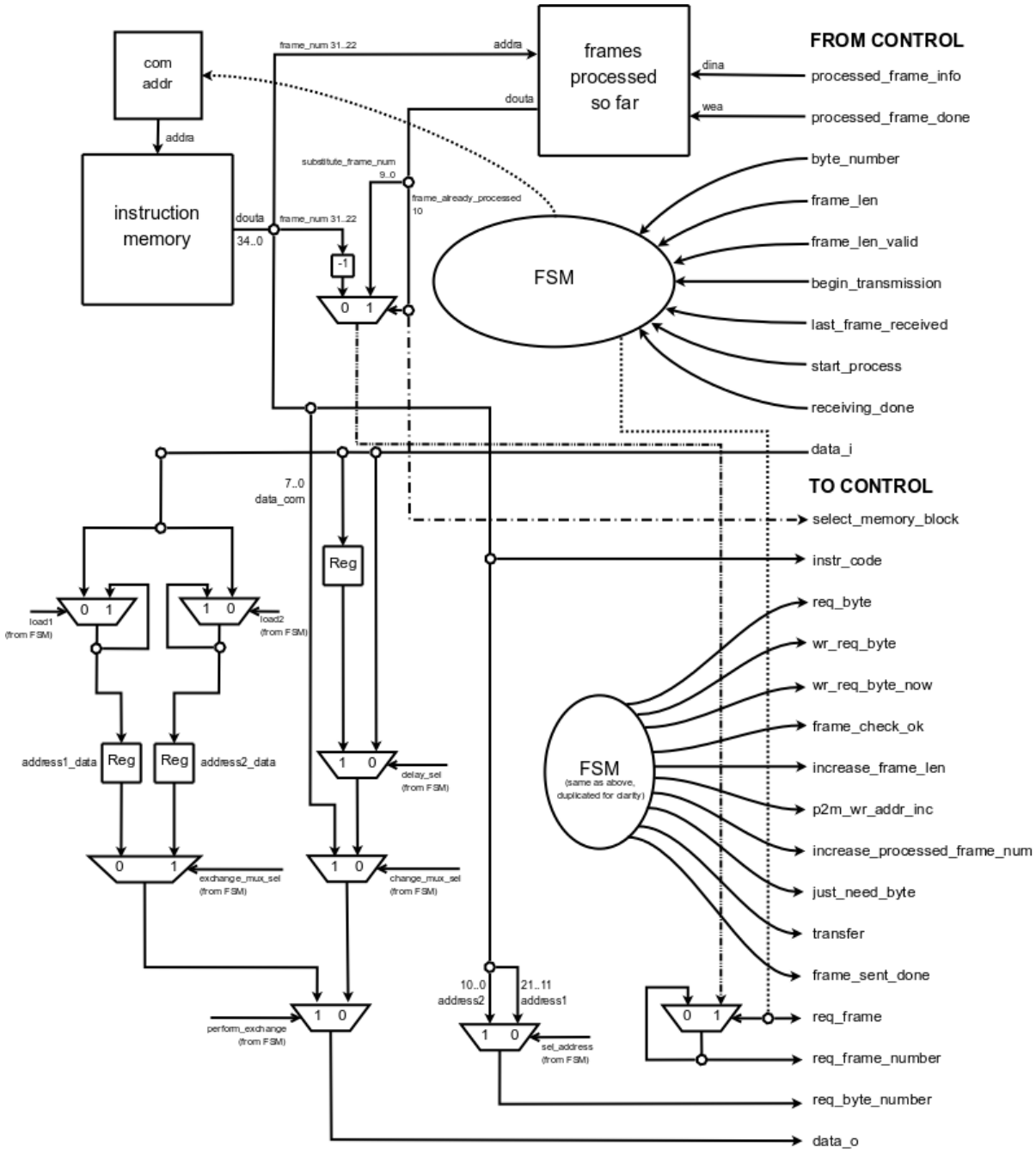
While executing a Report or a Transfer command, no alteration of data occurs and frame data is transmitted back to Control as is, so once again,

all multiplexer select signals are set to 0.

Table 4.7: Process Unit Interface

Name	Type	Length	
clk	in	1	
reset	in	1	**Module**
data_i	in	8	Control
byte_number	in	11	
frame_len	in	11	
frame_len_valid	in	1	
begin_transmission	in	1	
last_frame_received	in	10	
start_process	in	1	
receiving_done	in	1	
req_frame	out	1	
req_frame_number	out	10	
req_byte	out	1	
req_byte_number	out	11	
wr_req_byte	out	1	
wr_req_byte_now	out	1	
frame_check_ok	out	1	
increase_frame_len	out	1	
p2m_wr_addr_inc	out	1	
increase_processed_frame_num	out	1	
processed_frame_info	in	11	
processed_frame_done	in	1	
select_memory_block	out	1	
just_need_byte	out	1	
instr_code	out	3	
transfer	out	1	
data_o	out	8	
frame_sent_done	out	1	

Figure 4.11: Process Block Diagram



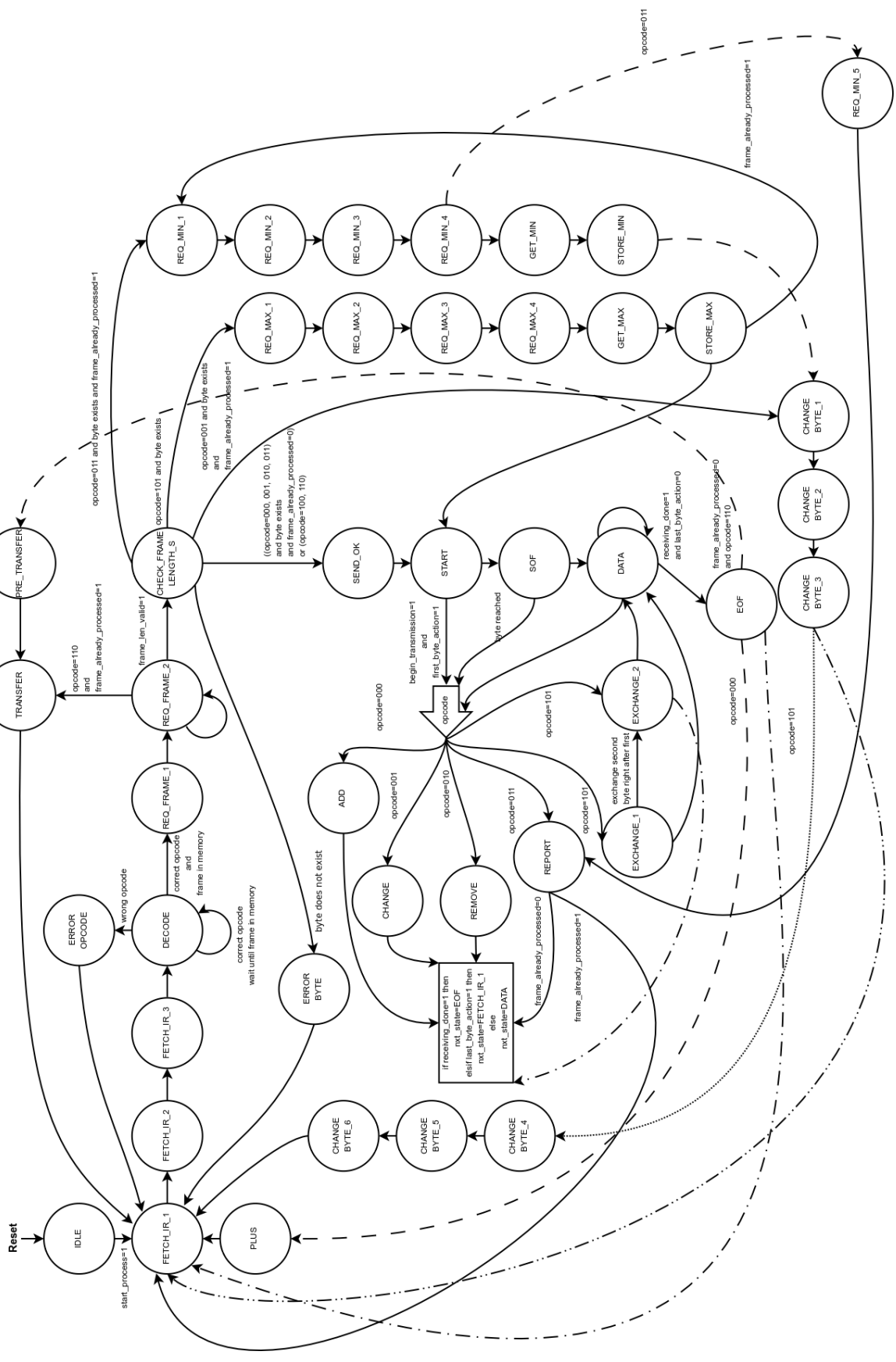
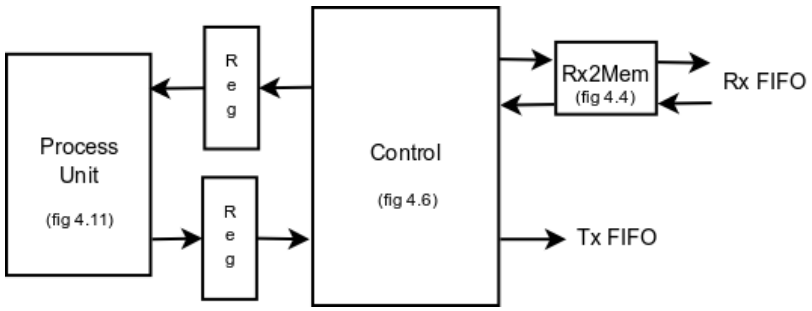


Figure 4.12: Process FSM

4.2.3.8 Top Level

The Top Level is the module that encapsulates our design (see figure 4.13), connecting all the different modules of the Network Processor into a single component, allowing for easy integration in other designs.

Figure 4.13: Top Level Block Diagram



The way the Top Level operates is pretty straightforward; it simply acts as a container for the different modules we have already mentioned, instantiating them and providing connections between them. Its interface is given in table 4.8.

Table 4.8: Top Level Interface

Name	Type	Length	
clk	in	1	
reset	in	1	**Module**
dst_rdy_n_o	out	1	
src_rdy_n_i	in	1	
sof_n_i	in	1	Rx FIFO
eof_n_i	in	1	
data_i	in	8	
src_rdy_n_o	out	1	
sof_n_o	out	1	
eof_n_o	out	1	Tx FIFO
data_o	out	8	

4.2.4 Device Utilization

The complete design, when downloaded on the target V5 FPGA board, has the device utilization presented in table 4.9.

Table 4.9: Device Utilization Summary

Resource	Number used	Percent used
BRAMs	71 out of 148	47
DSPs	3 out of 64	4
Slice LUTs	2248 out of 69120	4

The clock speed at which our design operates is at a minimum 125 MHz, thus allowing for proper Gigabit Ethernet operation.

Chapter 5

Verification & Performance

5.1 Hardware used

5.1.1 Virtex-5 Board

The FPGA board we used to implement our design in, was the Xilinx V5LX110T Evaluation Platform. It employs a Xilinx Virtex-5 XC5VLX110T FPGA chip, of the Virtex-5 LX family.

Among the features of this board are:

- Memory support for Flash, Compact Flash, SRAM and DDR
- Tri-speed Ethernet Physical layer interface supporting MII, GMII, RGMII and SGMII
- USB host and peripheral controllers
- Programmable system clock generator
- Stereo AC97 codec
- RS-232 port
- 16x2 character LCD screen

5.1.2 Ethernet Category 5e crossover cable

In order to connect to the FPGA board, we used an Ethernet Category 5e crossover cable, capable of supporting Gigabit Ethernet. A crossover cable was used instead of a normal patch cable, since we are connected directly to the FPGA's network adapter, without the interference of a switch or a hub.

5.1.3 PC Workstation

The PC we used during the design's development and verification was a standard PC, since no special requirements are needed by our design. The only requirement was a network adapter and our workstation was equipped with a Gigabit Ethernet one, thus allowing us to operate at 1000 Mbit/s Ethernet speed, the maximum supported by our design.

5.1.4 Connectivity

The connectivity needed to download the design to the FPGA board was to power on the board, connect one end of the Xilinx programmer cable provided with the board to a free USB port on our workstation and the other end on the board itself.

In order to verify the design's operation we connected the Ethernet cable to our workstation's network adapter and to the board's Ethernet port.

5.2 Software used during Development

5.2.1 Xilinx ISE

Xilinx's ISE is the base software used during the development of our architecture. It is a full featured front-to-back FPGA design solution and offers tools for HDL synthesis and simulation, implementation and device fitting.

Using it we created all the code, simulated our design in behavioral and post-place and route simulation modes and performed miscellaneous adjustments and optimizations in order to improve the functionality of our design.

5.2.2 Xilinx CoreGenerator

Xilinx's CoreGenerator tool is a tool used for instantiating different components. It offers an easy to use GUI, providing an efficient way to generate the desired components and customize them in a variety of ways. We used it to generate all the BRAMs, the ROM, the FIFO and of course the V5 TEMAC we used in our design.

5.2.3 Xilinx EDK

Xilinx's EDK is another tool we used during development. It offers an easy way to customize a design and download it to the specified FPGA

board, taking care of the design's final synthesis, optimization, mapping and placing for the target device specified. After that, it generates the bitstream describing the design and uses it to configure the FPGA board.

5.3 Software used during Verification

Initially the design's verification was accomplished by detailed simulation in both behavioral and post-place and route modes. For further verification, we used the following tools:

5.3.1 Xilinx ChipScope Pro

ChipScope Pro was used after downloading the design to the FPGA board, in order to have a detailed look of the way it operated under real world conditions and not just on simulation. It offered an outstanding way to detect possible errors and debug our design, since by using it to view user-specific signals, we managed to detect some errors and correct them, thus making the design operate flawlessly.

5.3.2 Wireshark

Wireshark is an open source network protocol analyzer application with a GUI. It can be used for network troubleshooting, analysis, software and communications protocol development. It is more widely known with its original name, Ethereal, but was renamed due to trademark issues. It has a rich feature set which includes the following:

- Deep inspection of hundreds of protocols
- Live capture and offline analysis
- Standard three-pane packet browser
- Multi-platform: supports Linux, Windows, Mac OS X, Solaris, FreeBSD etc.
- Captured data can be browsed via GUI or via terminal (through tshark utility)
- Powerful display filters
- Rich VoIP analysis

- Read/write many different capture file formats
- Capture files compressed with gzip can be decompressed on the fly
- Live data can be read from a selection of network types, including but not limited to, Ethernet, IEEE 802.11, Bluetooth, loopback etc.
- Decryption support for many protocols, including IPsec, Kerberos, SSL/TLS etc.
- Colouring rules can be applied to the packet list to provide quick analysis
- Exporting of captured data to XML, PostScript, CSV or plain text

We used Wireshark to verify that the commands we had loaded into the instruction memory of our design had correctly altered the specified frames.

5.3.3 packEth

packEth is an open source packet generator with a GUI which allows the user to send randomly generated or detailed custom packets of data over a selection of network interfaces. It was originally developed for Linux and later ported to Windows through use of the GTK+ library. Its feature list includes:

- Selection of network interface to use
- Creation and sending of any single Ethernet packet; supported protocols are:
  - Ethernet II, Ethernet 802.3, Ethernet 802.1q, QinQ
  - ARP, IPv4, user defined network layer payload
  - UDP, TCP, ICMP, IGMP, user defined transport layer payload
  - RTP (payload with options to send sin wave of any frequency for G.711)
- Sending a sequence of packets, allowing for customization of:
  - delay between packets
  - number of packets to send
  - speed at which to send — option for maximum speed, thus approaching the theoretical boundary

  - parameters can be altered while sending packets

- User can save configuration to a file

packEth was used to construct the frames we sent to the design, once it was downloaded on the FPGA board.

5.4 Verification Process

After downloading the design on the Virtex-5 board and connecting the Ethernet cable from our workstation to the board, we use Wireshark and packEth to verify the correct operation of our design.

Specifically, we start Wireshark and begin capturing frames on our workstation's Gigabit Ethernet network adapter. After that, packEth is initialized to generate the frame(s) that we want to send to our design. Once the frame(s) are configured and the interface is set to our workstation's network adapter, we begin transferring frames to the board. Back in Wireshark, we can see the outgoing frames as we send them along to the board and according to the commands loaded in the instruction memory we can notice the incoming processed frames. In that way we can verify whether the program loaded in our instruction memory performs correctly or not.

5.4.1 Commands loaded in Memory

In the following section we are going to provide a list with all the commands we have loaded in the instruction memory during the final verification process.
Change 1, 1, 0A
Change 1, 3, 0B
Change 1, 15, 0C
Report 1, 1
Report 1, 3
Report 1, 14
Add 1, 1, 09
Add 1, 3, 09
Add 1, 15, 09
Add 1, 16, 09
Compare 1, 99
Compare 1, 09
Remove 1, 1

Remove 1, 3
Remove 1, 15
Exchange 1, 1, 15
Exchange 1, 3, 8
Transfer 1

Add 2, 31, 0A
Add 2, 31, 0B
Add 2, 31, 0C
Add 2, 31, 0D
Add 2, 31, 0E
Add 2, 31, 0F
Add 2, 31, 10
Add 2, 31, 11
Add 2, 31, 12
Add 2, 31, 13
Change 2, 1, FF
Transfer 2

Change 3, 7, 0A
Change 3, 8, 0C
Change 3, 9, 0D
Change 3, 10, 0E
Change 3, 11, 0F
Change 3, 12, 0B
Transfer 3

Change 6, 1, 0F
Exchange 6, 13, 14
Transfer 6

Add 10, 15, AA
Add 10, 16, BB
Add 10, 17, CC
Remove 10, 15
Remove 10, 16
Remove 10, 15
Transfer 10

Exchange 16, 13, 14

Add 16, 15, FF
Add 16, 16, EE
Exchange 16, 15, 16
Change 16, 15, 00
Change 16, 16, 11
Transfer 16

5.5 Performance

5.5.1 Loopback mode benchmarks

As we have already mentioned, the example design generated by Xilinx's CoreGenerator tool during the generation of the V5 TEMAC implements a simple loopback function, where data is received from the Rx LL FIFO, passed through an ASM module and then sent through the Tx LL FIFO back to the board's MAC.

We have conducted a simple benchmark with our design, in loopback mode, by not executing any commands in the Process module and simply sending frame data from data_memory_rx to data_memory_tx and then through the ASM module to the Tx LL FIFO. This was done in order to get a better estimate on the maximum bandwidth our design can deliver, since when executing commands we cannot easily get such a reading. In table 5.1 we show the numbers we have measured from both the example design generated by CoreGenerator and our own design. During the aforementioned benchmark, we sent frames of maximum allowed size (1518 bytes, including all the Ethernet frame parts) for a transmission time equal to 15, 20 and 30 seconds, using packEth and monitored the results using Wireshark. The numbers we have gathered were provided using Wireshark's statistics utility.

The total bandwidth provided by each design includes the download and the upload rates of the board (in Mbit/s), while for a clearer view of the board's actual data throughput capability, we also provide the upload rate alone.

5.5.2 Design Latency and Throughput

5.5.2.1 Latency

In table 5.2 we present a comprehensive look at the latency of each of the commands supported by our design's instruction set. These latencies are dependent on a number of factors, these being firstly the command itself,

Table 5.1: Loopback mode benchmarks (T is for total, U for upload)

Time (secs)	**15**	**20**	**30**
Example Design	394.077 T 197.473 U	337.998 T 170.783 U	257.837 T 139.207 U
Our Design	430.154 T 215.391 U	399.802 T 200.220 U	418.614 T 209.379 U

since each command follows a different path of execution, secondly whether the frame has already been processed or not and finally the number of bytes in the frame. However, there is a common latency shared among the instructions, that being the instruction fetching (3 clock cycles), its decoding (1 clock cycle) and the memory address pointer initialization in Control (7 clock cycles). Besides that common latency, each of the commands, according to the aforementioned three factors, has a different special latency.

Table 5.2: Instruction Set Latency (N is the number of bytes in a frame)

Instruction	**Latency (clock cycles)**	
	Frame not processed	Frame processed
Add	16+(N+1)	16+(N+1)
Change	16+N	15
Compare	16+N	16+N
Exchange	21+N	32
Remove	16+N	16+N
Report	16+N	18
Transfer	18+N	12

After examining table 5.2, we can notice that Add and Remove have the same latency regardless of whether the frame has already been processed or not. That is because they both follow the same execution in either case, since when adding or removing a byte to/from the frame we need to shift the remaining bytes of the frame in memory, in order to keep a correct representation of the frame. The same applies to the Compare command, since we need to compare every byte in the frame with the one in the command, regardless of the frame having being processed already or not.

The overall worst case latency of our design is the one of the Add com-

mand, which is equal to 16+(N+1) clock cyles. Assuming the maximum frame size one Add command takes a total of 12.248 μs, since we have a clock cycle equal to 8 ns. An indication of our design's performance is the VoIP application: in such an application, which is quite intolerant of lagging performances, a latency up to 150 ms is considered high quality; so our design is capable of performing a total of approximately 12247 Adds while still delivering a high quality VoIP performance.

5.5.2.2 Throughput

The maximum throughput our design can provide is equal to the number of frames it can transfer in a second. Assuming a frame of maximum size (1518 bytes), the time required for our design to transfer a non-processed frame is equal to (18+1514 clock cycles) $\cdot$ 8 ns=1532 clock cycles $\cdot$ 8 ns=12256 ns. So our design has a throughput of approximately 81.592 KPPS (thousand packets/frames per second) when dealing with non-processed frames. In the aforementioned equation, we have used 1514 instead of 1518 bytes, because we do not deal with the 4 bytes in the FCS field of the frame, since it is stripped by the V5 TEMAC wrapper before being passed to our design. It is also automatically calculated and added to the frame outside our design, once again by the wrapper.

Chapter 6

Future Work

The design described in this thesis is complete and performs quite efficiently. Yet there is always room for improvements.

Memory management could be optimized, since modifications could allow for better memory allocation.

The instruction set currently implemented by this design could be extended to support even more commands/instructions, in order to make the NP more complete and allowing it to offer more services and functions.

The Process module could be separated into different engines, each specifically tasked with the execution of a certain set of operations, instead of an all-in-one implementation as the one provided in this design. The control module could also be divided into separate parts, thus creating more pipeline stages and possibly providing improvements in speed and handling of all the operations.

A way for commands to be inserted or updated in a more dynamic manner would really offer a great improvement, since now, everytime the program loaded in the instruction memory is to be changed, the instruction memory's .coe file has to be altered, the memory has to be regenerated using the CoreGenerator tool and the whole design needs to be updated before downloaded again to the board. Such an improvement would offer a major gain in deployment time. This dynamic manner could very well be through the RS232 port of the board. This could lead to better error handling, since it could allow for error correction by on the fly altering the erroneous command if the user can input a correct byte number or opcode. Furthermore, by adding RS232 support, we could use the serial port to immediately view the results of commands such as Compare and Report, along with all the erroneous command addresses, through a terminal program.

Finally, software could be developed to take advantage of the board's MicroBlaze processor; by doing so, parts of the design could be executed in software providing parallel execution with the hardware modules and thus could provide better efficiency and performance.

Bibliography

[1] Wikipedia, the free encyclopedia, *Network processor article*, http://tinyurl.com/krmquu [Web site]

[2] Wikipedia, the free encyclopedia, *OSI model article*, http://tinyurl.com/5bvu8 [Web site]

[3] Jean Walrand, *Communication Networks*, Greek Language, Second Edition, University of Athens, 2003 [Book]

[4] IEEE Working Group, *IEEE 802.3 Ethernet*, http://tinyurl.com/cjkrb [Web site]

[5] Wikipedia, the free encyclopedia, *Ethernet article*, http://tinyurl.com/3ht64 [Web site]

[6] Wikipedia, the free encyclopedia, *Ethernet II framing article*, http://tinyurl.com/2mub5x [Web site]

[7] Wikipedia, the free encyclopedia, *Media Access Control article*, http://tinyurl.com/y5o2o8 [Web site]

[8] Wikipedia, the free encyclopedia, *MAC address article*, http://tinyurl.com/powre [Web site]

[9] Wikipedia, the free encyclopedia, *Maximum transmission unit article*, http://tinyurl.com/95rdc [Web site]

[10] Wikipedia, the free encyclopedia, *Media Independent Interface article*, http://tinyurl.com/2eo5gr [Web site]

[11] Wikipedia, the free encyclopedia, *Reduced Media Independent Interface article*, http://tinyurl.com/nkjgcy [Web site]

[12] Wikipedia, the free encyclopedia, *Gigabit Media Independent Interface article*,
http://tinyurl.com/mtj45c [Web site]

[13] Wikipedia, the free encyclopedia, *Reduced Gigabit Media Independent Interface article*,
http://tinyurl.com/lusbs5 [Web site]

[14] Wikipedia, the free encyclopedia, *Serial Gigabit Media Independent Interface article*,
http://tinyurl.com/npqrca [Web site]

[15] Cisco Systems, *Serial-GMII Specification*,
http://tinyurl.com/l3qf7j [PDF]

[16] Hifn, *Hifn Acquires Picoprocessor PowerNP Products from IBM Press Release*,
http://tinyurl.com/neloeb [PDF]

[17] IBM, *End-of-life and acquisitions - IBM Microelectronics, IBM Picoprocessor PowerNP*,
http://tinyurl.com/ncf4h4 [Web site]

[18] Hifn, *Hifn - Network Processors - 5NP4G Overview*,
http://tinyurl.com/msloqz [Web site]

[19] Hifn, *Hign 5NP4G Network Processor Product Brief*,
http://tinyurl.com/nr6ryn [PDF]

[20] IBM, *PowerNP NP4GS3 Network Processor Data Sheet*,
http://tinyurl.com/kpeds7 [PDF]

[21] Intel, *Intel IXP1200 Network Processor Datasheet*,
http://tinyurl.com/lmqjjx [PDF]

[22] Intel, *Intel IXP1200 Network Processor Family Product Brief*,
http://tinyurl.com/nexvv8 [PDF]

[23] Intel, *Intel IXP2400 Network Processor Product Brief*,
http://tinyurl.com/klxk95 [PDF]

[24] Intel, *Intel IXP2800 Network Processor Product Brief*,
http://tinyurl.com/kj7qfk [PDF]

[25] Intel, *Intel IXP425 Network Processor - Overview*,
http://tinyurl.com/34qg98 [PDF]

[26] Intel, *Intel IXP43X Product Line of Network Processors - Overview*,
http://tinyurl.com/kkf24e [PDF]

[27] Intel, *Intel IXP455 Network Processor - Overview*,
http://tinyurl.com/nkjmxn [PDF]

[28] Intel, *Intel IXP465 Network Processor - Overview*,
http://tinyurl.com/mk9uj7 [PDF]

[29] Freescale Semiconductor, *C-3e Network Processor Silicon Revision B0 Data Sheet*,
http://tinyurl.com/lmjhcg [PDF]

[30] Freescale Semiconductor, *C-3e Network Processor Product Brief*,
http://tinyurl.com/m9jmhz [PDF]

[31] Freescale Semiconductor, *C-5 Network Processor Silicon Revision D0 Data Sheet*,
http://tinyurl.com/ml7zdq [PDF]

[32] Freescale Semiconductor, *C-5 Network Processor Product Brief*,
http://tinyurl.com/lcr6zn [PDF]

[33] Freescale Semiconductor, *C-5e Network Processor Silicon Revision B0 Data Sheet*,
http://tinyurl.com/no9uto [PDF]

[34] Freescale Semiconductor, *C-5e Network Processor Product Brief*,
http://tinyurl.com/nt3jzf [PDF]

[35] Ioannis Papaefstathiou, Stylianos Perissakis, Theofanis G. Orphanoudakis, Nikos A. Nikolaou, George Kornaros, Nicholas A. Zervos, George Konstantoulakis, Dionisios N. Pnevmatikatos and Kyriakos Vlachos, *PRO3: A Hybrid NPU Architecture*,
Micro IEEE, Volume 24, Issue 5, September-October 2004 [Paper]

[36] Xilinx, Inc., *Xilinx UG194 Virtex-5 FPGA Embedded Tri-Mode Ethernet MAC, User Guide*,
http://tinyurl.com/mfas88 [PDF]

[37] Xilinx, Inc., *Xilinx DS550 Virtex-5 Embedded Tri-Mode Ethernet Wrapper v1.5, Data Sheet*,
http://tinyurl.com/kselx8 [PDF]

[38] Xilinx, Inc., *Xilinx UG340 Virtex-5 Embedded Tri-Mode Ethernet MAC Wrapper v1.5, Getting Started Guide*,
http://tinyurl.com/lbxtp5 [PDF]