Master Thesis Report

For the master thesis on

Network Processor based Exchange Terminal – Implementation and evaluation

Department of Microelectronics and Information Technology,
Royal Institute of Technology (KTH)

Daniel Hedberg
e98_dhe@e.kth.se

Stockholm, Sweden,
05 December 2002

Examiner:
Prof. Gerald Q. Maguire Jr.
KTH Teleinformatics
maguire@it.kth.se

Supervisor:
Markus Magnusson
Ericsson Research
Markus.Magnusson@uab.ericsson.se

Mikael Johansson
Ericsson Research
Mikael.Johansson@uab.ericsson.se

Abstract

When communication nodes are connected to different networks, different kinds of Exchange Terminals (ETs) i.e., line card, are used. The different media we consider here have a bit rate between 1.5Mbps to 622Mbps and use protocols such as ATM or IP. In order to minimize the number of different types of ET boards, it is interesting to study the possibility of using Network Processors (NP) to build a generic ET that is able to handle several link layer and network layer protocols and operate at a wide variety of bit rates.

This report investigates the potential of implementing an ET board using a one-chip or two-chip solution using an Intel Network Processor (NP). The design is described in detail including a performance analysis of the different modules (microblocks) used. The report also provides an evaluation of the IXP2400 network processor and contrasts it to some other network processors. The detailed performance evaluation is based on a simulator of the IXP2400, which is part of Intel's Software Development Kit (SDK) version 3.0. In addition, I have investigated: the memory bus bandwidth, memory access latencies, and compared C-compiler against hand-written microcode. These tests were based on using an application for this ET board, which I have implemented.

It proved to be difficult to fit all the required functions into a single chip solution. The result is either one must wait for the next generation of this chip or one has to use a two-chip solution. In addition, the software development environment used in the project was only a pre-release, and not all services worked as promised. However, a clear result is that implementing an ET board, supporting the commonly desired functions, using a Network Processor is both feasible and straightforward.

Sammanfattning

För att koppla ihop olika noder som befinner sig på olika nätverk, använder man sig av olika Exchange Terminal-kort (ET-kort), s.k. Linjekort. De olika media vi tar i beaktning har en linjehastighet mellan 1.5 Mbps och 622 Mbps och använder protokoll som exempelvis ATM och IP. För att minimera antalet olika ET-kort är det intressant att studera möjligheten att använda sig av Nätverksprocessorer som ett allmänt ET-kort som kan hantera flera olika länklager- och nätverkslager- protokoll, och samtidigt fungera över olika hastigheter.

Den här rapporten utreder möjligheten att implementera ett ET-kort för en eller två nätverksprocessorchip tillverkad av Intel, kallad IXP2400. Designen är beskriven i detalj och inkluderar även en prestandaanalys av flera olika moduler (mikroblock) som använts. Rapporten innehåller även en utvärdering av IXP2400 där den jämförs med en liknande nätverksprocessor från en annan tillverkare. Prestandaanalysen är baserad på en simulator av IXP2400 processorn, som är en del av Intels utvecklingsmiljö kallad IXA SDK 3.0. Slutligen har jag även utvärderat minnesbussarna, minnesaccessen och ett C-kompilatortest gjord med hjälp av assemblergenererad kod och C-kod. Dessa tester gjordes på en applikation av ET-kortet som jag själv har implementerat.

Det visade sig vara svårt att få in alla krav som ställts på bara en nätverksprocessor. Resultatet är antingen att vänta tills nästa version av simuleringsmiljön kommer ut på marknaden eller att använda sig av två nätverksprocessorer. Under projektet användes bara en betaversion av utvecklingsmiljön och det har inneburit att alla funktioner inte fungerar som förväntat. Resultatet visar ändå tydligt att användning av Nätverksprocessorer är både effektiv och enkel att använda.

Acknowledgements

This report is a result of a Master's thesis project at Ericsson Research AB, in Älvsjö during the period of June to beginning of December 2002.

This project would not been successful without these persons:

- Prof. Gerald Q. Maguire Jr., for his knowledge and skills in a broad area of networking, rapid responses to e-mails, helpful suggestions, and genuine kindness.
- Markus Magnusson and Mikael Johansson, my supervisors at Ericsson Research, for their support and on helpful advising when I needed it.
- Magnus Sjöblom, Paul Girr, and Sukhbinder Takhar Singh, three contact people from Intel who supported me with help I needed to understand and program in their Network Processor simulation environment, Intel SDK 3.0

Other people that I want to mention includes Sven Stenström and Tony Rastas, two master thesis students at Ericsson Research that I worked with during my thesis.

Thank you all!

Table of Contents

1 Introduction

1.1 Background

Traditionally, when nodes are connected to different networks, different kinds of Exchange Terminal (ETs) i.e., interface boards are used. The different media we will consider here can have a bit rate between 1.5Mbps to 622Mbps and use protocols like ATM or IP. In order to minimize the number of different boards, it is interesting to use Network Processors (NP) to build a generic ET that is able to handle several protocols and bit rates.

1.2 Problem definition

In this thesis, the main task is to study, simulate, and evaluate an ET board called ET-FE4 which is used as a plugin-unit in the Cello system (see section 2.6). Figure 1 below shows an overview of blocks that are included on this ET board. The data traffic is first interfaced via a Line Interface, in this case a Packet over SONET (POS) interface, as the board is connected to two SDH STM-1 (OC-3) links. Traffic is processed, just as in a router using a Forwarding Engine (FE) in hardware to obtain wire speed routing. Error packets or special packets, called exception packets are handled in software by an on-board processor of the Device Board Module (DBM). After it has been processed, the traffic is sent on the backplane, where it is connected to a Cello-based switch fabric.

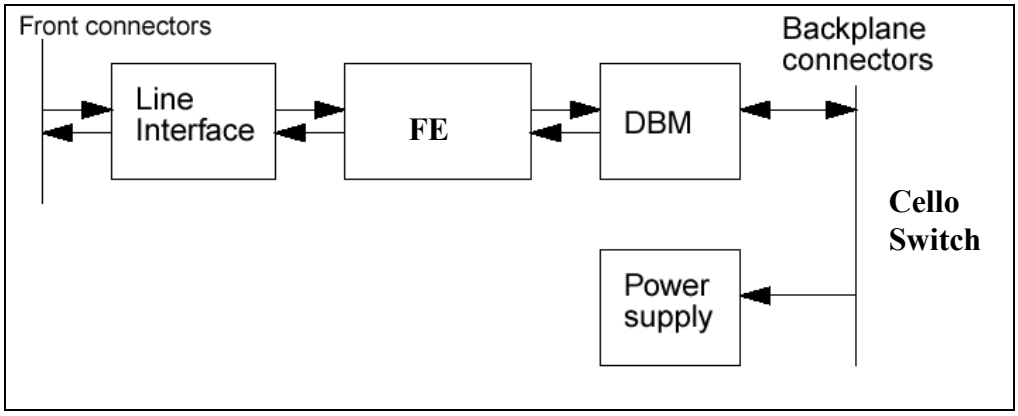

Figure 1. Block diagram of ET-FE4

To run different protocols such as IP or ATM, it is usually necessary to add or remove hardware devices on the board or to reprogram them (as in the case of Field Programmable Gate Arrays (FPGA)). Because each of these protocols has specific functionality, it is therefore generally necessary that hardware differ between these ET boards. By using a Network Processor (NP), all the needed functionality can be implemented on the **same** board. It only requires changes in the software load, to define the specific functionality.

This thesis concentrates on the implementation of the Forwarding Engine (FE) block on the ET board (see Figure 1). To implement this block, a study of the existing forwarding functionality was necessary. Then all the requirements for the FE block functionality needed to be refined to fit within the time duration of this thesis project. All the necessary requirements and functionalities are listed in Appendix A. Once the implementation phase was completed, an evaluation was performed to verify that the desired result was achieved

(i.e., wire speed forwarding). To understand better how the networking processing technology works, a comparison between Motorola's Network Processor C-5 and Intel's IXP2400 was performed. Finally, to evaluate the workbench for the Network Processor, a memory test, and a C-compiler test was performed.

1.3 Outline of the report

Chapter 2 introduces the main protocols used during the implementation of the application. Then it states how Network Processor programming works with assembly and C programming. It follows with a description of Ericsson's Cello System used in mobile 3G platforms. Finally, the chapter describes two Network Processors, Intel IXP2400 and Motorola C-5, and a comparison of both processors. Moreover, readers who are familiar with HDLC, PPP, IP, and ATM can skip the first sections up to 2.5. A reader who is familiar with Network Processor Programming, Ericsson's Cello system, Intel IXP2400, and Motorola C-5 can skip the rest of the chapter.

Chapter 3 explains the existing solutions, both with other Network Processors and with third-part companies using Intel Network Processors.

Chapter 4 provides a detailed description of how to solve the problem stated with simulation methodologies. By using existing modules (i.e. microblocks), an application can be built to achieve the goals of the project. The chapter also describes a briefly overview of methods to use in the evaluation phase of the project.

Chapter 5 analyses the application to se if it reaches wire speed forwarding. It also provides some basics test of the C-compiler, where a test on both a small and a large program of C-code and microcode are compared. In the analyses, a performance test of the application is described and a theory study on how long a packet take to travel through the application.

Chapter 6 summarises the results of this work and compares with the stated goals. It provides suggestions and Lessons Learned for the reader.

Chapter 7 indicates a suggested future work of the thesis and if application upgrades are necessary and other investigation.

2 Background

This chapter starts with an overview of all the protocols used in the applications developed for this thesis. In following, there are sections about Network Processors in general and how to program them. Section 2.6 describes an important part of the project where it describes briefly how Ericsson's Cello system works and the Exchange Terminal that is going to be implemented. Finally, the section describes two examples of popular Network Processors, Intel IXP2400 and Motorola C-5 and a comparison between them.

2.1 Data Link-layer Protocol overview

2.1.1 HDLC: an example link layer protocol

High-level data link control (HDLC) specifies a standard for sending packets over serial links. HDLC supports several modes of operation, including a simple sliding window mode (see section 7 in [4]), for reliable delivery. Since the Internet Protocol family provides retransmission via higher layer protocols, such as TCP, most Internet link layer usage of HDLC, use the unreliable delivery mode, "Unnumbered Information" (See [1]). As shown in Figure 2, the HDLC frame format has six fields. The first and the last field are the flag field, used for synchronising by the receiver so it knows when a frame starts and ends. The flag has normally "01111110" in binary and this sequence cannot appear in the rest of the frame, to enforce this requirement, the data may need to be modified by bit stuffing (described below).

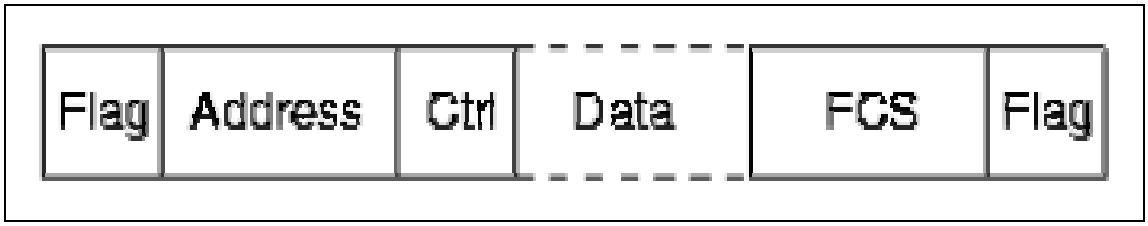

Figure 2. HDLC's frame structure

The second field is the address field, used for identifying the secondary station that sent or will receive the frame. The third field is the Control field which is used for specifying the type of message sent. The main purpose of this field is to distinguish frames used for error and flow control, when using higher-level protocols. The fourth field is the Data field, also called the HDLC information field and is the actual payload data used for the upper layering protocols. The Frame Check Sequence (FCS) field is used to verify the data integrity of the frame and to enable error detection. The FCS is a 16 bit Cyclic Redundancy Check (CRC) calculated using the polynomial $x^{16} + x^{12} + x^5+1$.

Bit stuffing

On bit-synchronous links, a binary 0 is inserted after every sequence of five 1s (*bit stuffing*). Thus, the longest sequence of 1s that may appear of the link is `0111110` - one less than the flag character. The receiver, upon seeing five 1s, examines the next bit. If zero, the bit is discarded and the frame continues. If one, then this must be the flag sequence at the start or end of the frame.

Between HDLC frames, the link idles. Most synchronous links constantly transmit data; these links transmit either all 1s during the inter-frame period (*mark idle*), or all flag characters (*flag idle*).

Use of HDLC

Many variants of HDLC have been developed. Both PPP protocol and SLIP protocol use a subnet of HDLC's functionality. ISDN's D channel uses a slightly modified version of HDLC. In addition, Cisco's routers use HDLC as a default serial link encapsulation.

Transmission techniques

When transmitting over serial lines, two principal transmissions are used. First synchronous, which enables you to send or receive a variable length of bytes. The second transmission is asynchronous, which only sends or receives one character at a time.

These two techniques are used over a several different media types (i.e., physical layers), such as:

- EIA RS-232
- RS-422
- RS-485
- V.35
- BRI S/T
- T1/E1
- OC-3

For the ET Board used in this thesis, the media type will be OC-3. OC-3 is a standard for telecommunications running at 155.52 Mbps, it makes 149.76Mbps available to the PPP protocol that will be used.

2.1.2 PPP: an example link layer protocol

Point-to-point Protocol (PPP) is a method of encapsulating various datagram protocols into a serial bit stream so that it can be transmitted over serial lines. PPP is a HDLC like frame that uses a subnet of the functionalities that HDLC provides. Some of the restrictions for the PPP frame compared to the HDLC like frame are:

- The address field is fixed to the octet 0xFF
- The control field is fixed to the octet 0x03
- The receiver must be able to accept an HDLC information field size of 1502 octets

Another thing to remember is that the HDLC information field contains both the PPP Protocol field and the PPP information field (Data field). The PPP frame format is shown in Figure 3 below.

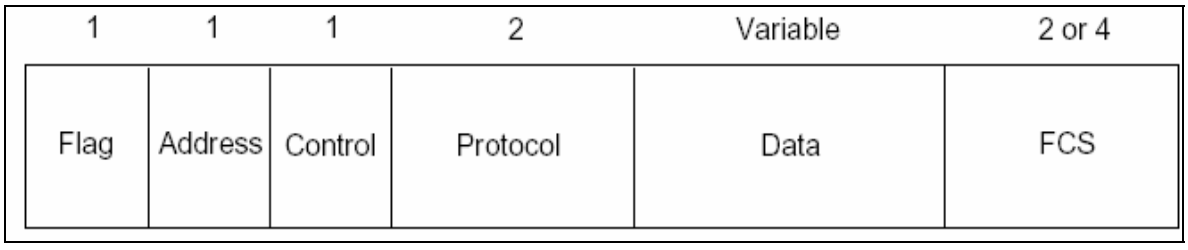

Figure 3. PPP frame format

The Protocol field identifies the type of message being carried. This could be a PPP control message such as LCP, ECP, CCP, IP-NCP (described further below) or it could be network layer datagrams such as IP or IPX. The protocol field can be 1-2 bytes depending if it is

compressed or not. The PPP information field for PPP contains the protocol packet as specified in the protocol field. At the end of the PPP frame, there is a FCS field with the same functionality as the FCS described earlier.

There are three framing techniques used for PPP. The first one is Asynchronous HDLC (ADHLC) used for asynchronous links often used for modems on ordinary PC's. The second one is Bit-synchronous HDLC mostly used for media types such as T1 or ISDN links. It has no flow control, there is no escape character used, and the framing and CRC work is done by the hardware. The last technique is Octet-synchronous HDLC, similar to ADHLC with the same framing and escape codes. This technique is also used on special media with buffer-oriented hardware interfaces. The most common buffer-oriented interfaces are SONET and SDH. In this thesis, I have concentrated on a particular interface in the SDH family called OC-3 which operates at 152.52 Mbps.

2.1.3 PPP Protocols

PPP contains several protocols such as LCP, NCP and IPCP (Described below).

Link Control Protocol (LCP)

Before a link is considered ready for use by network-layer protocols, a specific sequence of events must happen. The LCP provides a method of establishing, configuring, maintaining and terminating the connection. There are three classes of LCP packets:

- Link Configuration packets, establish and configure the link
- Link Termination packets, terminates the link
- Link Maintenance packets, manages and debugs a link

Network Control Protocol (NCP)

NCP is used to configure the protocol operating at the network layer. One example is to assign dynamic IP addresses to the connecting host.

Internet Protocol Control Protocol (IPCP)

The Internet Protocol Control Protocol is responsible for configuring, enabling, and disabling the IP protocol modules on both ends of the PPP link. PPP may not exchange IPCP packets until PPP has reached the Network Protocol Layer phase (described below). IPCP has the same functionality as the LCP protocol with the following exceptions:

- It supports exactly one IPCP packet included in the Information field. The Protocol field code is 0x8021
- Only codes 1-7 are supported in the code field. Other codes are treated as unrecognised.
- IPCP packets cannot be exchanged until PPP has reached the Network layer protocol state.

More details about IPCP are described in [13].

2.2 PPP Session

A PPP session is divided into four main phases:

- Link establishment phase
- Authentication phase
- Network-layer protocol phase
- Link termination phase

Figure 4 shows an overall view of these four phases including the link dead phase.

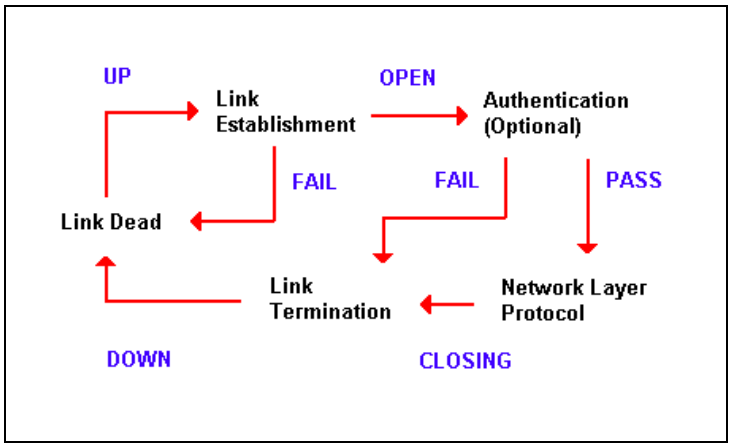

Figure 4. A link state diagram

2.2.1 Overview of a PPP session

To establish communication over a point-to-point link, each end of the PPP link must first send Link Control Protocol (LCP) packets to configure and test the data link. Then an optional authentication phase can take place. To use the network layer, PPP needs to send Network Control Protocol (NCP) packets. After each of the network layer protocols has been configured, datagrams can be sent over this link. The link remains up as long as the peer does not send an explicit LCP or NCP request to close down the link.

Link establishment phase

In this phase, each PPP device sends LCP packets to configure and test the data link. LCP packets contain a Configuration Option field which allows devices to negotiate the use of options, such as:

- Maximum Receive Unit (MRU) is the maximum size of the PPP information field that the implementation can receive.
- Protocol Field Compression (PFC) is an option used to tell the sender that it can receive compressed PPP protocol fields.
- FCS Alternatives, allows the default 16-bit CRC to be negotiated into either a 32-bit CRC or disabled entirely.
- Magic Numbering, is a random number which is used for distinguish the two peers and detect error conditions such as loop back lines and echoes. See section 3 in [1] for further explanation.

PPP uses messages to negotiate parameters between all protocols that are used. All these parameters are well described in [17]. We see that there are four of these parameters described that are used more than the others are and here is a short summary of them:

- Configure-Request, tells the peer system that it is ready to receive data with the enclosed options enabled
- Configure-Acknowledgement, the peer responds with this acknowledgement to indicate that all enclosed options are now available on this peer.
- Configure-Nack, responds with this message if some of the enclosed options were not acceptable on the peer. It contains the offending options with a suggested value of each of the parameters.
- Configure-Reject, responds with this message if it does not recognise one or more enclosed options. It contains these options to let the sender now witch options to remove from the request message.

Authentication phase

The peer may be authenticated after the link has been established, using the selected authentication protocol. If authentication is used, it must take place before starting the network-layer protocol phase. PPP supports two authentication protocols, Password Authentication Protocol (PAP) and Challenge Handshake Authentication Protocol (CHAP) [21]. PAP requires an exchange of user names and clear-text passwords between two devices and PAP passwords are sent unencrypted. Instead, CHAP uses authentication agent (typically used by a server) to send to a client program using a random number and an ID value only once.

Network-layer protocol phase

In this phase, the PPP devices send NCP packets to choose and configure one or more network layer protocols (such as IP, IPX, and AppleTalk). Once each of the chosen network-layer protocols has been configured, datagrams from this network-layer protocol can be sent over the PPP link.

Link termination phase

LCP may terminate the link at any time when a request comes from a user or a physical event.

2.3 Internet Protocol

Internet Protocol (IP) [13] is designed for use in packet switch networks. IP is responsible for providing blocks of data, called datagrams from a source to a destination. Source and destination are identified through fixed length IP addresses. IP also supports fragmentation and reassembling of large datagrams if transmission bandwidth is small on a network. Today, there exist two versions of the Internet Protocol, version 4 (IPv4) and version 6 (IPv6). IPv4 is the old protocol that now has been upgraded to a newer version, IPv6.

2.3.1 IPv4

One IPv4 datagram consists of a fixed length header of 20 bytes and a variable-length payload part. Both destination and source addresses are 32-bit numbers placed in the IP header shown in Figure 5 on next page.

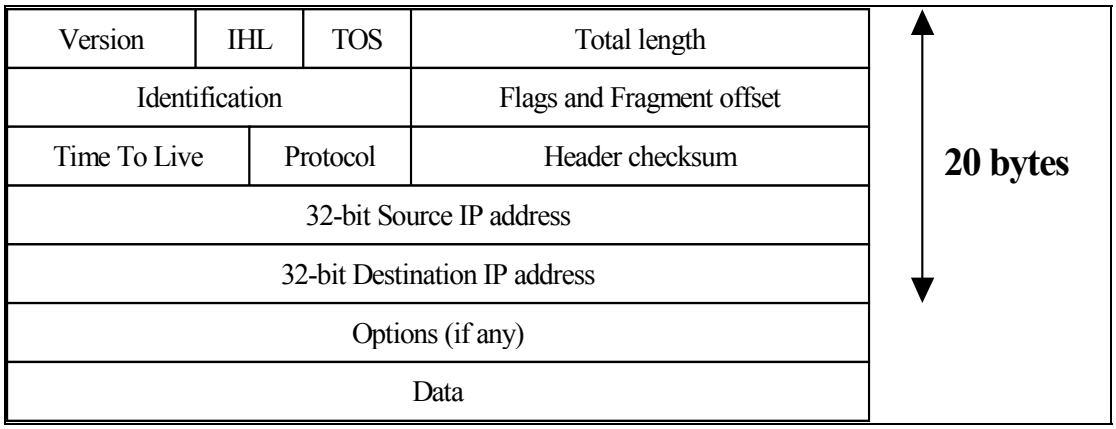

Figure 5. IP datagram

Here follows a short explanation of all the fields in the IP header:

- Version: Shows which version of the Internet Protocol a datagram belongs to
- Internet Header Length (IHL): Shows how long the header is. The minimum value is 5 bytes which is the length when no options in use
- Type of Service (TOS): Gives a priority to a datagram
- Total length: Includes both header and payload data of a datagram. The maximum value of packet size is 65 535 bytes
- Identification: Used for a destination to identify a fragment to the correct datagram
- Flags and Fragment offset: This field shows where in the datagram a certain fragment belongs to
- Time To Live: Maximum life time for a datagram in a network
- Protocol: Shows which IP User (Example TCP) is destined for
- Header checksum: Is calculated only for the IP header
- Source IP-address: Is the address where the datagram was sent from
- Destination IP-address: Shows the final destination address for the datagram
- Options: Shows different optional choices such special packet routes etc.
- Data: Actual user specific data

For more details of the IPv4 protocol, look at [2] and [3].

2.3.2 IPv6

Internet Protocol version 6 (IPv6) [20] is known as the new version of the Internet Protocol, which is designed to be an evolutionary step from IPv4. It is a natural increment to IPv4 and one of the big advantages is the address space available. IPv4 had 32-bit address while IPv6 now uses 128-bit address. It can be installed as a normal software upgrade in Internet devices and is interoperable with the current IPv4. Its deployment strategy is designed to not have any flag days or other dependencies. A flag day means a software change that is neither forward- nor backward-compatible, and which is costly to make and costly to reverse. IPv6 is designed to run well on high performance networks (e.g. Gigabit Ethernet, OC-12, ATM, etc.) and at the same time still be efficient for low bandwidth networks (e.g. wireless). In addition, it provides a platform for new Internet functionality that will be required in the near future.

The feature of IPv6 includes:

- *Expanded Routing and Addressing Capabilities*
  IPv6 increases the IP address size from 32 bits to 128 bits, to support more levels of addressing hierarchy and a much greater number of addressable nodes, and simpler auto-configuration of addresses. Multicast and anycast have been built in IPv6 as well. Benefiting from big address space and well-designed routing mechanism, for example, Mobile IP, it make it possible to connect anyone everywhere at any time.
- *Simplified but Flexible IP Header*
  IPv6 has a simplified IP header, some IPv4 header fields have been dropped or made optional, to reduce the common-case processing cost of packet handling and to keep the bandwidth cost of the IPv6 header as low as possible despite the increased size of the addresses. Even though the Ipnv6 addresses are four times longer than the IPv4 addresses, the IPv6 header is only twice the size of the IPv4 header. To make it flexible enough to support new service in future, header options are introduced.
- *Plug and Play Auto-configuration Supported*
  A significant improvement of IPv6 is that it supports auto-configuration in host. Every device can plug and play.
- *Quality-of-Service Capabilities*
  IPv6 also designed for support QoS. Although there are no clear idea on how to implement QoS in IPv6, IPv6 reserve the possibility to implement QoS in future.
- *Security Capabilities*
  IPv6 includes the definition of extensions, which provide support for authentication, data integrity, and confidentiality. This is included as a basic element of IPv6 and will be included in all implementations.

2.4 ATM

Asynchronous Transfer Mode, ATM is a proposed telecommunications standard for Broadband ISDN. The basic idea is to use small fixed packets (cells) and switch these over a high-speed network on a hardware level.

ATM is a cell-switching and multiplexing technology that combines the benefits of circuit switching and packet switching such as constant transmission delay, guaranteed capacity, flexibility and efficiency for intermittent traffic. ATM cells are delivered in order, but it is no guarantee for delivery. Line rate for ATM cells are 155 Mbps, 622 Mbps or more. This section describes briefly how ATM cells look like and which layers are used.

2.4.1 ATM Cell format

An ATM cell is a short fixed-length packet of 53 bytes. It consists of a 5-byte header containing address information and a fixed 48 bytes information field (See Figure 6 on next page). The ATM standards groups (ATM Forum) [52] have defined two header formats: The UNI header format (defined by the UNI specification) and the Network-Node Interface (NNI) header format (defined by NNI specification). The only difference between the two headers is the GFC field. This field is not included in the NNI header. Instead, the VPI field is increased to 12 bits.

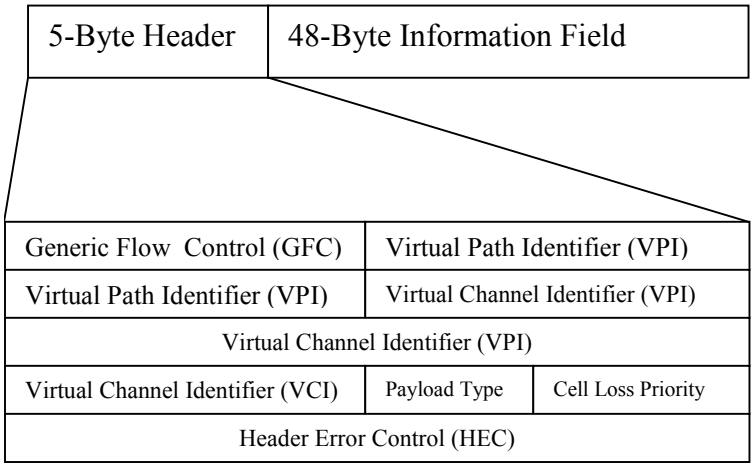

Figure 6. ATM Cell

The ATM Cell header fields include following:

- *Generic Flow Control (GFC):* First 4 bits of the cell header contain the GFC, used for control traffic flow onto the ATM network by UNI.
- *Virtual Path Identifier (VPI):* Next 8 bits contain the VPI used to specify a virtual path on the physical ATM link.
- *Virtual Channel Identifier (VCI):* Next 16 bits contain the VCI used to specify a virtual channel within a virtual path on the physical ATM link.
- *Payload Type (PT):* Next 3 bits contain the PT used to identify the type of information the cell is carrying (For example, user data or management information).
- *Cell Loss Priority (CLP):* Last 4 bits indicate the CLP used to identify the priority of the cell and whether the network can discard it under heavy traffic conditions.
- *Header Error Control (HEC):* Last byte of the ATM header contains HEC used to guard against misdelivery of cells due to header or single bit errors.

All 48-bytes of payload (Information field) can be data or it can also be optionally 4 byte ATM adaptation layer and 44-bytes of actual data depending if a bit in the control field is set. This enables fragmentation and reassembly of cells into larger packets at the source and destination. The control field have also a bit to specify whether the ATM cell is a flow control cell or an ordinary cell.

The path of an ATM cell passing through the network is defined by its virtual path identifier (VPI) and virtual channel identifier (VCI), used in the ATM cell header above. Together, these fields specify a connection between two end-points in an ATM network.

2.4.2 ATM Reference Model

In the reference model, ATM consists of four layers: Physical layer, ATM layer, ATM adaptation layer, and higher layers. First is the physical layer which controls the transmission and reception of bits on the physical medium. It also keeps track of ATM cell boundaries and it package cells into the appropriate type of frame for the physical medium being used.

Second layer is the ATM layer, defines how two nodes transmit information between them and is responsible for establishing connections and passing cells through the ATM network. Third layer is ATM adaptation layer (AAL) used to translate between larger Service Data Units (SDU) of upper layer processes and ATM cells.

The AAL layer is divided into two sub layers: Convergence Sublayer (CS), Segmentation and Reassembly (SAR) Sub layer. These two layers convert variable-length data into 48-byte segments. ITU-T has defined different types of AALs, AAL3, AAL3/4, AAL4, and AAL5. These handle different kinds of traffic needed for applications to works with packets larger than a cell. Some other AAL services are flow control, timing control and handling of lost and bad inserted cell conditions. The most common AAL is AAL5, mostly used for UDP. Next section below describes AAL5 more briefly.

AAL5

AAL5 is the adaptation layer used to transfer data, such as IP over ATM and local-area network (See Figure 7). Packets to be transmitted can vary from 1 to 65,535 bytes. The Convergence Sublayer (CS) of AAL5 appends a variable-length pad and an 8-byte trailer to form a frame, creating a CS Protocol Data Unit (PDU). The pad is used to fill in if the data is not big enough to fit in a 48-byte payload of the ATM cell. The trailer includes the length of the frame and a 32-bit CRC computed across the entire PDU. The SAR layer segments the CS PDU into 48-byte blocks and the ATM layer places each block into the payload field of an ATM cell. For all cells except the last one of a data stream, a bit in the PT field is set to be zero to indicate that the cell is not the last cell in a frame. For the last cell, the bit is set to one. When the cell arrives to its destination, the ATM layer extracts the payload field from the cell, the SAR layer reassembles the CS PDU and uses the CRC and the length field to verify that the frame has been transmitted and reassembled correctly.

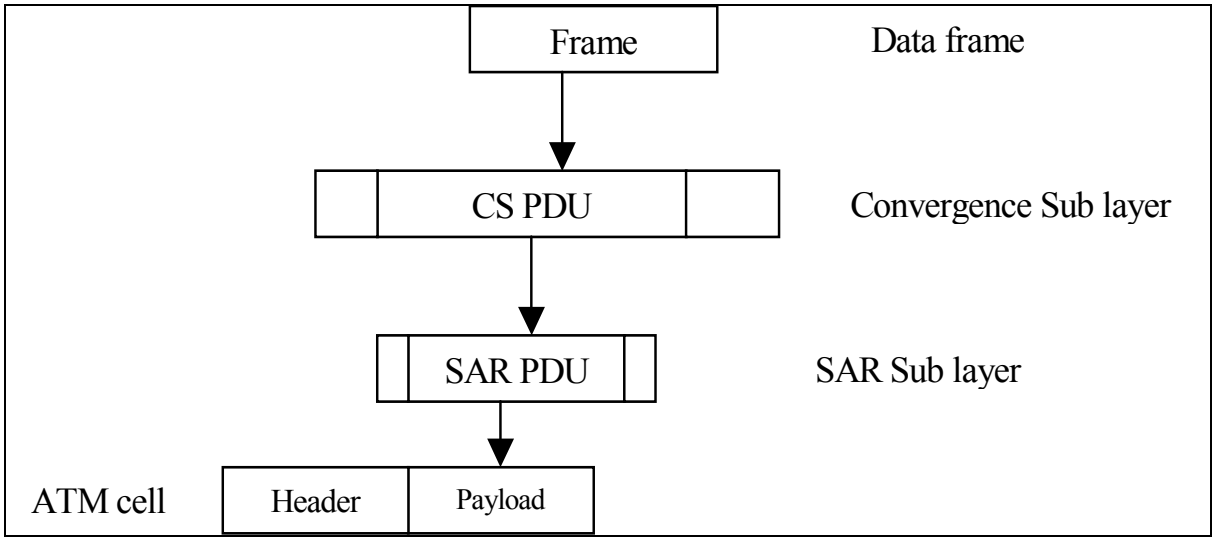

Figure 7. ATM Adaptation Layer 5

2.5 Queuing Model

Queuing is a function used in routers, line cards etc. The queuing lends itself to innovation due to it is design to allow a broad range of possible implementations using common structures and parameters [22].
Queuing systems perform three distinct functions:

- It store packets using queues
- Modulates the departure of packets belonging to various traffic streams using scheduler
- Selectively discards packets using algorithmic droppers

2.5.1 Queues

Queuing elements modulate the transmission of packets belonging to different traffic streams and determine ordering of packets, store them temporarily or discard them. Packets are usually stored either because there is a resource constraint such as available bandwidth, which prevents immediate forwarding, or because the queuing block is being used to alter the temporal properties of a traffic stream (i.e., shaping).
Packets are discarded for one of the following reasons:

- Buffering limitations
- A buffer threshold has exceeded (including shaping)
- A feedback control signal used to reactive control protocols such as TCP
- A meter exceeds a configured profile (i.e., policing).

FIFO

First in First out (FIFO) queue is the simplest queuing algorithm and is widely used over Internet. It leaves all the congestion control to the edger (i.e. TCP). When the queue gets full, packets are dropped.

2.5.2 Scheduler

A scheduler is a queuing element, which gates the departure of each packet arriving on one of its inputs. It has one or more inputs and exactly one output. Each input has an upstream element to which it is connected, and a set of parameters which affects the scheduling of packets received at that input.
The scheduling algorithm might take any of the following as its input(s):

- Static parameters such as relative priority associated with each input of the scheduler
- Absolute token bucket parameters for maximum or minimum rates associated with each input of the scheduler
- Parameters, such as packet length or Differentiated Services Code Point (DSCP) associated with the packet currently present at input.
- Absolute time and/or local state

Here follows a short summary of common scheduling algorithms:

- Rate Limiting, packets from a certain traffic class are assigned a maximum transmission rate. The packets are dropped if a certain threshold is reached.
- Round Robin, All runnable processes are kept in a circular queue. The CPU scheduler goes around this queue, allocating the CPU each process for a time-interval.
- Weighted Round Robin (WRR), Works in same manner as Round Robin, where packets from different streams are queued and scheduled for transmission in an assigned priority order.
- Weighted Fair Queuing (WFQ) and Class Based Queuing (CBQ), when packets are routed to a particular output line-card interface, each flow receives an assigned amount of bandwidth.
- Weighted Random Early Detection (WRED), Packets from different classes are queued and scheduled for transmission. When packets from a low priority use too much bandwidth, a certain percentage of its packets are randomly dropped.
- First Come First Serve (FCFS)

Some scheduler uses Traffic Load Balancing, which is not really a scheduling algorithm. Traffic Load Balancing issues equal-size tasks to multiple devices. This involves queuing and fair scheduling of packets to devices such as database and web servers.

2.5.3 Algorithmic droppers

The algorithmic dropper is a queuing element responsible for selectively discard packets that arrive at its input, based on some discarding algorithm. The basic parameters used in the algorithmic droppers are:

- Dynamic parameters, using average or current queue length
- Static parameters, using threshold on queue length
- Packet-associated parameters, such as DSCP values

2.6 Ericsson's Cello system

The Cello system is a product platform for developing switching network nodes such as simple ATM switches, Radio Base Stations (RBS), or Radio Network Controllers (RNC). The Cello system has a robust real time distributed telecom control system which supports ATM, TDM [4], or IP transport. The Cello system is designed for interfaces that run at 1.5 Mbit/s – 155 Mbit/s. In the backbone, the limit is even higher (622 Mb/s). Therefore, there should not be a problem to upgrade card such as ET boards to run at 622 Mb/s.

To build a switching network node, we need both the Cello platform and a development environment. The platform consist of both hardware and software modules. To transport cells from one device to another, it uses a Space Switching System (SPAS). The SPAS switch is an ATM based switch which connects to internal interfaces, external interfaces, or both. Internal interfaces can be Switch Control Interfaces (SCIs), interfaces providing node topology, or interfaces to administer the protection switching of the internal system clock. External interfaces can be Switch Access Configuration Interfaces (SACI) or a hardware interface, Switch Access Interface (SAI) [37], which is used as an access point for data transfer through a switch.

2.6.1 Cello Node

A Cello node is simply a switching network node which can be scaled in both size and capacity. The Cello node scales in size depending on how many subracks it consists of. At least one subrack (see Figure 8) must be connected. A subrack has several Plugin-Units such as Main Processor Boards (MPBs), Switch Core Boards (SCBs), different ET boards, and device boards. All of these units are attached to a backplane (SPAS Switch) and a Cello node needs at least one processor board depending on the processing power needed and the level of redundancy desired. A bigger Cello node consists of several subracks that are connected together through SCB links.

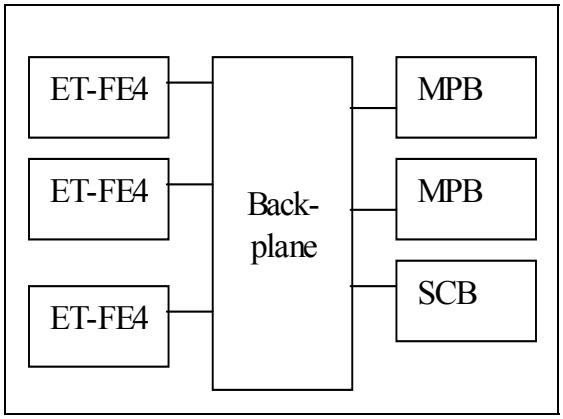

Figure 8. A Single subrack configuration [5]

2.6.2 Exchange Terminal (ET)

Traditionally Ericsson produced several Exchange Terminal boards which handle both ATM and IP traffic. Different ET boards are necessary for implementing adaptations to different physical media and different link layer and network layer standards. Some of them are listed below:

- ET-M1, ATM board supports link speeds over 1.5 Mbit/s and interfaces to T1/E1 links, supports 8 ports
- ET-M4, ATM board supports link speeds over 155 Mbit/s and interfaces to STM-1/OC-3 optical or electrical links and supports 2 ports
- ET-FE1, IP forwarding board supports link speeds over 1.5 Mbit/s and interfaces to T1/E1 links
- ET-FE4 IP forwarding board supports link speeds over 155 Mbit/s and interfaces to 2 optical STM-1/OC-3 links

This thesis concentrates on the existing ET-FE4 board and specifically the forwarding engine block (see Figure 1) on it. As described in the figure, the ET-board consists of three main modules: Line Interface, Forwarding Engine, and the Device Board Module. Here follows a short description of these modules.

Line interface

The line interface performs clock recovery and data extraction. It consists of two optical modules and PMC-Sierra 5351chip [29], which processes duplex 155.52 Mbit data streams (OC-3). The PMC-Sierra chip is a STM 1 payload extractor sending out extracted data on a POS-PHY Lev 2 link connected to the forwarding engine.

Forwarding Engine

The forwarding engine contains two Field Programmable Gate Arrays (FPGAs) [36]. One FPGA is used for manage IP forwarding and some QoS. For the ingress part, the FPGA handles IP forwarding using forwarding table lookup. On the egress part, the FPGA is used for some QoS functionality such as Diffserv queuing of packets. The second FPGA contains both a HDLC Protocol unit and a PPP protocol unit used for processing PPP packets and transmitting packets over serial links. It also has a Multilink Protocol unit for fragmenting packets and transmitting them over serial links.

Device Board Module (DBM)

The Device Board Module (DBM) is a processor platform for the device boards used in Cello. It contains interfaces for test and debugging as well as a connector to the backplane. The DBM has one FPGA, used for segmentation and reassembly of AAL5 packets and AAL0 cells. It has also a main processor, PowerPC 403GCX [28], that handles all the instructions needed to handle the traffic from the ET board to the backplane.

2.7 Network Processors (NPs)

2.7.1 Definition of a Network Processor

A Network Processor (NP) is a programmable (processor) integrated as a single semiconductor device which is optimised to primarily handle network processing tasks. These processing tasks include: receiving data packets, processing them, and forwarding them.

2.7.2 Why use a Network Processor?

Today, the networking communication area is constantly changing. The bandwidth grows exponentially and will continue for many years ahead. The growing bandwidth of optical fibre results to even grow faster than the speed of silicon. For example, the CPU clock speed grows with a factor of 12 and the network speed increases with a factor of 240. Higher bandwidth results in more bandwidth-hungry services on Internet, such as Voice over IO (VoIP), streaming audio and video, Peer-to-Peer (P2P) applications, and many others which we have not yet thought of. For networks to effectively handle these new applications, new protocols need to be supported to fulfil new requirements including differentiated services, security, and various network management functions.

To implement all these changes in hardware would be both inefficient and costly for both developer and customer. For example, when developing a new protocol, hardware needs to be developed to handle this protocol and the hardware development cycle is often much longer than the software development cycle. Therefore, a programmable configuration would be preferred, as it only needs to be modified or reprogrammed and then restarted. This saves both time and money for both developer and customers. This software implementation can be done for a Network Processor and are specially designed to handle networking tasks and algorithms such as packet processing.

A network processor is often used as a development tool but it can also be used for debugging and testing. Most of the NPs focus on processing headers, processing the packet contents is an issue for the future.

Some of the Network Processor vendors such as Intel, Motorola, and IBM provide a Workbench for a simulator of their Network Processors. A Network Processor simulator is always released before the actual hardware is shipped out. A good benefit is then to start developing software on the simulator, where it easily to debug and optimise using cycle-accurate simulation. If the application works on the simulator, there is compatible to be used in the hardware.

2.7.3 Existing hardware solutions

Today, most of the hardware implementations of switches are based on Field Programmable Gate Arrays (FPGAs) for low level processing and General Purpose Processors (GPPs) for higher level processing. Here are some of the existing system implementations:

- *General Purpose Processor (GPP)*, used for general purpose processing such as protocol processing on desktop and laptop computers. They are inefficiently due to the control overhead for each instruction since it must be fetched and decoded, although some of the processors may use very large caches.
- *Fixed Function ASIC (Application Specific Integrated Circuit)*, designed for one protocol only. They work at speeds round OC-12 and OC-48. Their major problem is their lack of flexibility, for example with longer time and cost to implement a change. ASICs are widely used for MAC protocols such as Ethernet. ASICs are expensive to develop therefore they are low cost only for very large sales volume.

- *Reduced Instruction Set Computer (RISC) with Optimised Instruction Set [9]*, is a microprocessor architecture similar to an ASIP except that it is based on adding some instructions to the RISC core instruction set. The program memory is separated from the data memory allowing fetch and executes to occur in the same clock cycle with on stage pipelining. The RISC design generally incorporates a large number of registers to prevent in large amounts of interactions with memory.
- *Field Programmable Gate Array (FPGA)* [36], is a large array of cells containing configurable logic, memory elements and flip flops. Compared to an ASIC, the FPGA can be reprogrammed at the gate level, where the user can configure interconnection between the logical elements, or configure functions on each element. Therefore, the FPGA has a better flexibility with shorter time-to-market and less design complexity than an ordinary ASIC. However it has still lower performance than an ASIC and higher performance compared to a GPP.
- *Application Specific Instruction Processor (ASIP)*, has instructions that map well to an application. If some pairs of operations appear often, it may be useful to cluster these operations into a single operation. It is specialised for a particular application domain. Normally, it has better flexibility than a FPGA but lower performance than a hardwired ASIC.

In September 2001, Niraj Shah at University of California in Berkeley compared the different system implementations above, using metrics such as flexibility, performance, power consumption, and cost to develop [39]. The results showed a clearly, that using an ASIP would be the best approach for most network system implementations. It provides the right balance of hardware and software to meet all the necessary requirements.

This thesis uses a Network Processor which is basically a reprogrammable hardware architecture concept using the ASIP technology. To gain further information about the different hardware solutions, see [6]. To gain knowledge about flexibility and performance differences between the solutions above, see [39].

2.7.4 Network Processors in general

A Network Processor's main purpose is to receive data, operate on it, and then send out the data on a network at wire speeds (i.e., only limited by the link's speed). They aim to perform most network specific tasks, in order to replace custom ASICs in any networking device.

A NP plays the same role in a network node as the CPU does in a computer. The fundamental operations for packet processing consist of following operations:

- *Classification*, parsing of (bit) fields in the incoming packet and table lookup to identify the incoming packets, followed by a decision based regarding the destination port of the packet.
- *Modification of the packet*, data fields in the header are modified/updated. Headers may be added or removed and this usually entails recalculation of CRC or checksum.
- *Queuing and buffering*, packets are placed in an appropriate queue for the outgoing port and temporary buffered for later transmission. The packet may be discarded if the capacity would be exceeded.
- *Other operations,* such as security consideration, policing compression, traffic metrics.

Network Processor Composition

A typical architecture of a Network Processor is shown in Figure 9. One central theme, when creating a Network Processor is employing multiple processors than instead of a large processor. A Network Processors contains of many Processing Elements (PEs), which perform most of the functions such as classification, forwarding, computation and modification, etc.

A Network Processor contains a Management processor, which handles: off-loaded packet processing, loading object code to the Processing Elements, and communicates with host-CPU. A Network Processor can also contain a Control processor, which are specialised for a specific task such as pattern matching, traffic management, and security encryption.

Network Processors interfaces host-CPU through PCI or similar bus interface. They also interfaces SRAM/DRAM/SDRAM memory units to implement lookup tables, and PDU buffer pools.

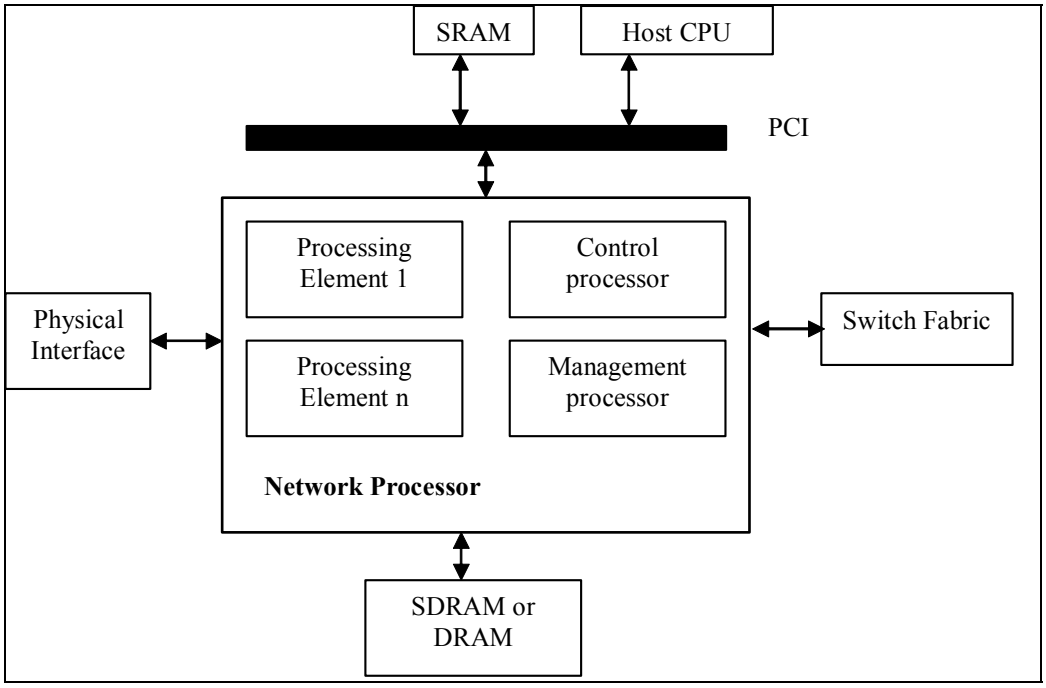

Figure 9. Typical Network Processor Architecture

Data plane vs. Control Plane

The network processing tasks are divided into two kinds of tasks: Data plane and Control Plane tasks. Data plane tasks handle time-critical duties in the core design. Less time critical tasks that fall outside the core processing or forwarding requirements of a network device are called Control Plane tasks. Another way to distinguish between these to types of tasks is to look at each packet's path. Packets handled by the data plane usually travel through the device, and the packets that are handled by the control plane usually originate or terminate at the device.

2.7.5 Fast path and slow path

The data plane and the control plane are processed over a fast path or a slow path depending on the packet. As a packet enters a networking device, it is first examined and processed further on either the fast path or slow path. The fast path (most data plane tasks) is used for minimal or normal processing of packets and the slow path are used for the unusual packets and control plane tasks that needs more complex processing. After processing, packets from both slow and fast path may leave via the same network interface.

2.7.6 Improvements to be done

Today a Network Processor moves packets surprising well, but still the processors can be improved to achieve better performance. An important thing to remember is that all the control of the traffic flowing through a NP should be implemented in software. Otherwise the flexibility is no better than a common ASIC [41]. According to a white paper written by O'Neill [40], today there are three main issues to improve the performances for a NP:

- Deeper pipelines, the relatively infrequent branches and their high degree of predictability can be exploited.
- Higher clock rate, can be reached if more effective using of caching on application is done and this improves the traditional path allowing it to be more effective.
- A multi-issue out-of-order architecture, with larger basic blocks loaded into the system improves the performance.

2.8 Network Processor Programming

Today, many network processors only have capacity for a few kilobytes of code. Intel still recommends writing in assembly code until their C-compiler has been further developed. Some NPs use functional languages to produce smaller programs with fewer lines of code. These languages are more complex, but programming effort can be saved.

2.8.1 Assembly & Microcode

Assembly, or microcode, is the native language for a NP. Although microcode for different NPs may look the same, there are huge differences. Each network processor has its own architecture and instruction set. Thus programs for the same purpose are quite different between different NPs. Therefore, the NP industry is heading for a serious problem for the future, how to standardize coding, so programs can be reused in another NP.

2.8.2 High-level languages

Most vendors supply code libraries and C-compilers to use for their NP. A code library usually covers basic packet processing code needed for IPv4 forwarding or ATM reassembling. There are significant advantages to use high-level languages such as C instead of microcode:

- C is the most common choice for embedded system and network application developers.
- A high-level language is much more effective at abstracting and hiding details of used instructions.
- It is easier and faster to write modular code and maintain it in high-level language with support for data types, such as type checking.

One of the upcoming programming techniques is functional programming where the languages describe the protocol rather than a specific series of operations. For example, Agere Systems NPs (see [33]) are supported with functional languages used for classification.

To read more about Assembly and high-level languages, see [7].

2.8.3 Network Processing Forum (NPF)

There have been steps towards standardized code for general interfaces. In February 2001, almost all Network Processor manufacture companies gathered together to found an organization, called Network Processing Forum (NPF) [50]. NPF establishes common specifications for programmable network elements to reduce time-to-market and instead increase the time-in-market. The desired norm should be rapid product cycles and in-place upgrades to extend the life of existing equipment. This also reduces the manufacturers' design burden, while still providing the flexibility enabled by using their own components to meet the requirements. Since 2001, NPF has grown to almost 100 members around the world.

2.9 Intel IXP2400

2.9.1 Overview

The Intel IXP 2400 chip has eight independent multithreaded 32-bit RISC data engines (microengines). These microengines are used for packet forwarding and traffic management on chip. IXP 2400 consists of these functional units:

- 32-bit XScale processor, used to initialise and manage the chip and for higher layer network processing tasks, and for general purpose processing. It runs at 600 MHz
- 8 Microengines, used for processing data packets on the data plane
- 1 DRAM Controller, used for data buffers
- 2 SRAM Controllers, used for fetching and storing instructions
- Scratchpad Memory, general purpose storage
- Media Switch Fabric Interface (MSF), used by the NP to interface POS-PHY chips, CSIX Switch Fabrics, and other IXP 2400 processors.
- Hash unit, XScale and microengines can use this when hashing is necessary
- PCI Controller, can be used to connect to host processors or PCI devices
- Performance Monitor, counters that count internal hardware events, which can be used to analyse performance

All these functional units are shown in Figure 10.

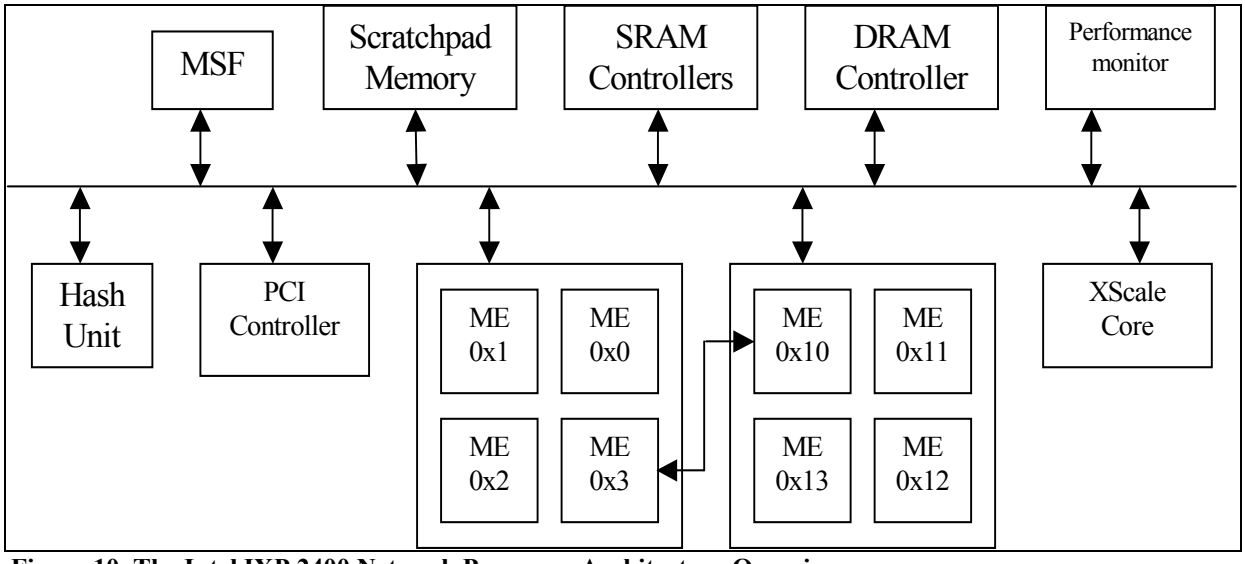

Figure 10. The Intel IXP 2400 Network Processor Architecture Overview

2.9.2 History

On April 1999 Intel Corporation announced that they would release their first Network Processor called Intel IXP 1200. It consisted of one StrongARM processor (predecessor of the XScale), six microengines, and interfaces to SRAM/SDRAM memory, FIFO Bus Interface (FBI), and PCI bus. The StrongARM processor is used for slow path processing, and the six microengines with four threads each handle fast processing. The IXP 1200 was intended for layers 2-4 processing and it supports data rates up to 2.5 Mbps. Today Intel is working on two Network Processors (Intel IXP 2400 and Intel IXP 2800) and a development toolkit called IXA 3.0. These are all still under development, therefore Intel has only a pre-release of the development toolkit, which is available for testing. In this thesis, I am currently using the pre-release 4 of the toolkit. The final release of the toolkit is planned for the first quarter of 2003. Both Network Processors are expected to be shipped sometime late in 2003.

2.9.3 Microengine (ME)

In the IXP 2400, there are eight Microengines (sixteen in IXP 2800) in one Network Processor. Each ME has eight threads each providing an execution context. It contains following features:

- 256 32 bits General Purpose Registers
- 512 Transfer Registers
- 128 Next Neighbour Registers
- 640 32-bit words of Local Memory
- 4 K instructions in the Control Store
- 8 Hardware Threads
- Arithmetic Logic Unit
- Event signals

General Purpose Registers (GPRs)

These registers are used for general programming purposes. They are read and written exclusively under program control. When a GPR are used as source operand in a instruction, it supplies operands to the execution datapath.

Transfer Registers

Transfer registers are used for transferring data to/from a Microengine, and to locations external (for example, SRAMs, DRAMs, etc.) to the Microengine.

Next Neighbour Registers

Next Neighbour (NN) registers are used as a source register in an instruction. They are written either by an adjacent Microengine or by the same Microengine. This register can rapidly pass data between two neighbour Microengines using NN ring structure (Same as dispatch loop, see 2.11.2), and when a Microengine write to its own neighbour register, it must wait 5 cycles (or instructions) before it can write new data. The NN registers can also be configured to act as a circular ring instead of addressable registers. The source operands are now popped from the head of the ring and the destination results are pushed to the tail of the ring.

Local Memory (LM)

The Local Memory is an addressable local storage in the Microengine used for read and write exclusively under program control and it can be used as source operand or destination operand for an ALU operation. Each thread on a Microengine has two LM address registers, which are written by special instructions. There is a 3 cycles latency between local memory address allocation and its de-allocate of the same address.

Hardware Threads (contexts)

Each context has its own register set, program counter and controller specific local registers. Using fast context swapping allows another context to do computation while the first context waits for an I/O operation. Each thread (context) can be in one of four different states:

- Inactive, used if applications don't want to use all threads
- Ready, this thread is ready to execute
- Execute, this is the executing state, a thread stays in this state until a instruction causes it to go to next state (Sleep) or a context swap is made
- Sleep, this state the thread waits for external events to occur

When one context is in the executing state, all others must be in another state, since only one context can be in the executing state (as it is a single processor).

Event signals

The Microengines supports event signalling. These signals can be used to indicate occurrence of some external events such as, when a previous thread goes to a state of "sleeping". Typical use of event signals includes completion of an I/O operation (such as DRAM) and signals from other threads. Each thread has 15 event signals to use, and each signal can be allocated and scheduled by the compiler in the same manner as a register and allows a large number of outstanding events. For example, a thread can start an I/O to read packet data from a receive buffer, start another I/O to allocate buffer from a free list, and start a third I/O to read next task from a scratch ring. These three I/O operations can be executed in parallel using threads with signalling.
Many microprocessors schedules multiple outstanding I/Os, normally handled by the hardware. By using event signals, the Microengine places much of the burden on the compiler instead of hardware. This simplifies the hardware architecture of a processor.

2.9.4 DRAM

The IXP2400 have one channel of industry standard DDR DRAM running at 100/150 Mhz providing 19.2 Gb/s of peak DRAM bandwidth. It supports up to 2 Gb of DRAM and is primary used to incoming buffer packets. All DRAM memory is spread out on four memory banks, where the DRAM addresses are interleaved and different operations on DRAM can be performed concurrently. There is no DRAM used in IXP1200 network processor, instead it uses SDRAM.

2.9.5 SRAM

The IXP 2400 provides two channels of industry standard QDR SRAM running at 100-250 MHz providing 12.8 Gb/s of read/write bandwidth and a peak bandwidth of 2.0 Gbytes/sec per channel. These two channels can use up to 64 MB of SRAM memory per channel. The SRAM is primary used for packet descriptors, queue descriptors, counters, and other data structures. In the SRAM controller, access ordering is guaranteed only for read coming after write.

2.9.6 CAM

Many of the network designers are discovering that the fastest and easiest way to process a packet is to offload the packet classification function to a co-processor. One of the best co-processors today is Content Addressable Memory (CAM) [10] [45]. CAM is a memory device to accelerate applications that requires fast searches of database, list, or pattern in communication networks. It improves a usage of multiple threads on same data and the result can be used to dispatch to the proper code. It performs a parallel look-up on 16 entries of 32-bit value. This allows a source operand to be compared against 16 values in a single instruction. All entries are compared in parallel giving a result of the loopback written into the destination register. It reports one of two outcomes: a hit or a miss. A hit indicates that the lookup was found in CAM. The result also contains the entry number that holds the lookup value. A miss indicates that the lookup value was not found in CAM. The result also contains the entry of the Least Recently Used (LRU) entry, which holds can be suggested to use as a replace entry.

2.9.7 Media Switch Fabric (MSF)

The MSF is used to connect an IXP 2400 processor to a physical layer device and/or to a switch fabric. It contains of separate receive and transmit interfaces. Each of these interfaces can be configured for UTOPIA (Level 1, 2, and 3), POS-PHY (Level 2 and 3) or CSIX protocols. UTOPIA [37] is standardized data path between the physical layer and the ATM layer. The ATM Forum defines three different levels of UTOPIA. Common Switch Interface for Fabric Independence and Scalable Switching (CSIX) [38] is a detailed interface specification between port/processing element logic and interconnect fabric logic. The IXP 2400 Microengines communicated with the MSF with the Receive Buffer (RBUF) or the Transmit Buffer (TBUF). RBUF is a RAM memory used to store received data from the MSF in sub-blocks referred as elements. The RBUF contains a total of 8 KB data and it is possible to divide it into 64, 128, or 256 byte elements. For each RBUF element there exist a 64-bit receive status word to describe the contents and status of the contents of the receive element. Content status such as a byte counts for a packet, or a flag to indicate if the received packet is the beginning or end of a packet. TBUF acts the same way as RBUF, except that it stores data to be transmitted instead of receiving data and it is divided in TBUF elements. A TBUF element is associated with a 64-bit control word used to store: packet information such as, payload length, flag indication if it is the beginning or end of a packet.

Looking at IXP1200 network processor, there is no MSF used, instead it uses a FIFO Bus Interface (FBI) unit. The FBI contains receive and transmit buffers (RFIFO and TFIFO), scratchpad RAM, and a hash unit.

2.9.8 StrongARM Core Microprocessor

The StrongARM core is a general-purpose 32-bit RISC processor. XScale and StrongARM are compatible with the ARM instruction set, but only implement the ARM integer instruction set, thus do not provide floating-point instruction support.

The XScale core supports VxWorks (v.5.4), and embedded Linux (kernel v.2.4) as an operating system to control the Microengine threads. Each microengine contains a set of control and status registers. These registers are used by the StrongARM core to program, control, and debug the Microengines. The XScale has uniform access to all system resources, so it can effectively communicate with the Microengine through data structures in shared memory.

2.10 Intel's Developer Workbench (IXA SDK 3.0)

To program the Intel Network Processor IXP2400, Intel has developed a workbench/transactor called Intel IXA SDK 3.0 (see Figure 11) used for assembling, compiling, linking, and debugging microcode that runs on the NPs Microengines [31]. The workbench is graphical user interface tool running on Windows NT and Windows 2000 platforms. The workbench can be run either from the development environment or as a command line application. The microengine development environment has some important tools such as:

- *Assembler*, used to assemble source files
- *Intel Microengine C Compiler*, generates microcode images
- *Linker*, links microcode images generated by the compiler or assembler to produce an object file
- *Debugger*, used for debug microcode in simulation mode or in hardware mode. (Hardware mode is not supported in the pre-release versions)
- *Transactor*, when debugging, the transactor provides debugging support for the Developers workbench. The transactor executes the object files built by the linker to show the functionality, statistics for Microengines, behaviour and performance characteristics of a system design based on the IXP2400.

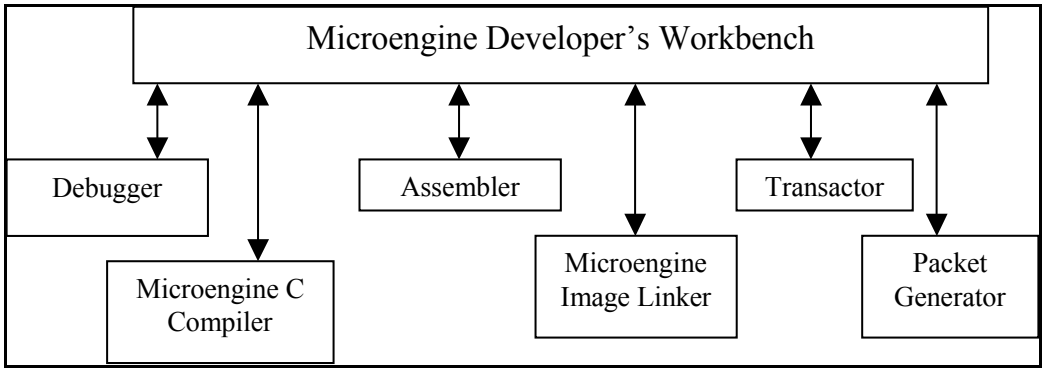

Figure 11. Overview of Intel IXA SDK 3.0 workbench

In this development toolkit, three data plane libraries are available. First, it has a Hardware Abstraction Library (HAL). This library provides operating system-like abstraction of hardware assistant functions such as memory and buffer management, and critical section management. The second library contains Utilities to provide range of data structures and algorithm support, such as: generic table lookups, byte field handling and endian swaps. The third library is a Protocol Library, used to provide an interface supporting link layer and network layer protocols through combinations of structures and functions.

IXA SDK 3.0 also includes other functionalities such as:

- *Execution History*, this show execution coverage on all thread in each used Microengine.
- *Statistics*, this shows statistics data from threads, Microengines, SRAM controllers, DRAM controller, and more. For example it can show how much time a certain Microengine has been executing or being idle.
- Media bus device and network traffic simulator

- Command line interface for the network processor simulators, this enables a user to specify options for commands to execute in a certain way.

2.10.1 Assembler

The Assembler is a fully compliant superset of a processor manufacturer's recommended Assembly Language. The Assembler recognizes conditional assembly directives which can be used to efficiently tail or code to multiple execution environments. One way to implement assembly language code is to use Macros. Macros (#macro, #endm, etc.) are a series of directives and instructions grouped together as a single command. Optional parameters can be passed to the macro for processing. To write macros is useful when writing in modular and readable code.

The assembler has a built-in facility implementing parameter substitution by a variable number of arguments and, as an extension to the language, allows the omission of any argument. Macros and repeat blocks may be nested. Macro constructs may contain local labels and the scope of these labels is selected through a command-line option.

The assembler has functionalities included such as:

- Processes directives flow can be seen in Figure 12 below
- Performs macro inline expansion
- Processes loop, conditional expressions
- Low-level syntax check
- Assign symbolic variables to GPRs, Xfer Register, signals
- Branch optimisation

Before an assembly process begins, a source file (.uc) needs to be created. This file contains three types of elements:

- Instructions, consists of opcode and arguments generate microwords in the ".list"- file
- Directives, it passes information either to the pre-processor, assembler, or to the downstream components (such as linker).
- Comments, used for have a clean code to understand

The pre-processor is automatically invoked by the assembler to transform to a program before the actual assembling. The pre-processor provides these operations:

- *Inclusion of files*, these files can be substituted into the main code.
- *Macro expansion*, the pre-processor replaces instants of macros with their definitions.
- *Conditional compilation*, it enables including or excluding code based on various conditions.
- *Line control*, used to inform the assembler of where each source file came from.
- *Structured assembly*, it organises the control flow of ME instructions into structured blocks.
- *Token replacement*, it causes instances of an identifier to be replaced with a token string.

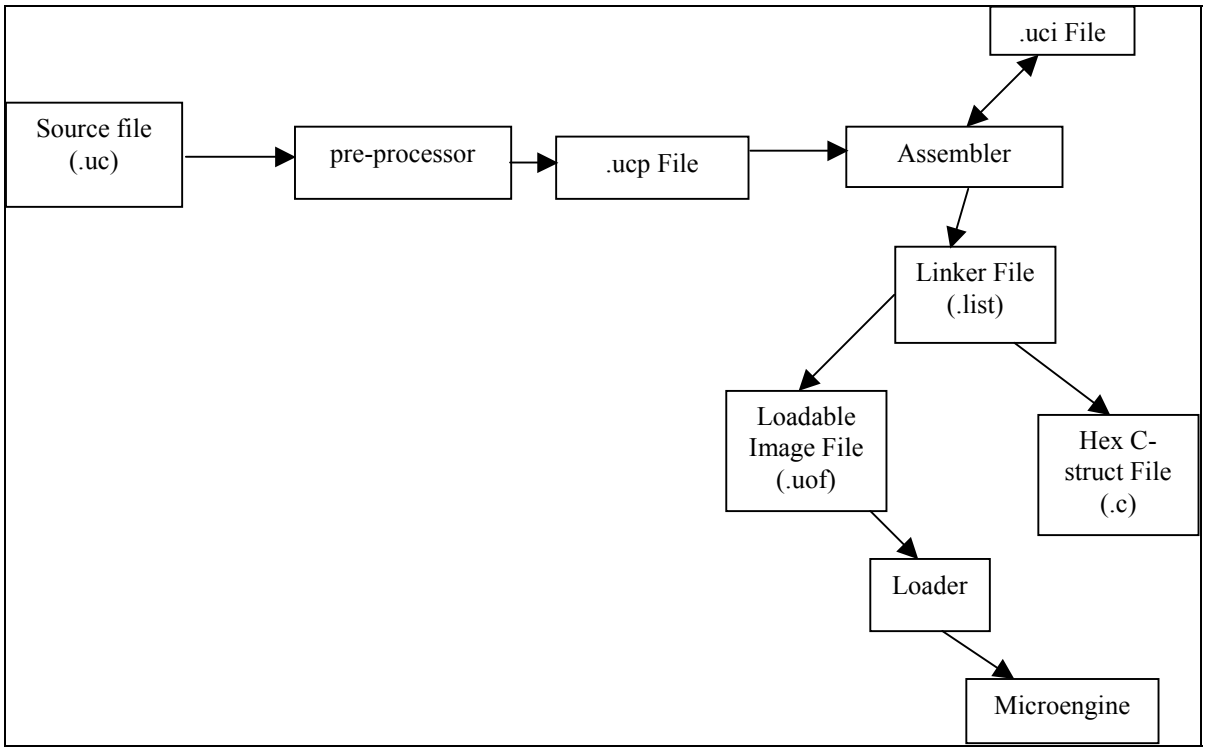

Figure 12. Assembly flow

The pre-processor takes the source file and creates an ".ucp"-file for the assembler. Now, the assembler takes this file and creates intermediate file with the filename extension of ".uci". The .uci file is used by the assembler to create the ".list"- file and provides error information, used for debugging. To convert an ".uc"-file to a ".list"-file, it process following functions:

- Checks instruction restriction
- Resolves symbol names to physical locations
- Optimises the code, by inserting defer[ ] optional tokens
- Resolves label addresses

2.10.2 Microengine C compiler

Intel's Microengine C compiler provides high-level language through C and it is specially optimised for the network processors IXP2400 and IXP2800. Some of the special features for these NPs are:

- The compiler allows the programmer to specify which variables must be stored in registers and which may be stored in memory.
- The compiler allows the programmer to specify which type of memory (SRAM, DRAM) to be used to allocate a specific variable.
- The compiler supports intrinsic and inline assembly for handling specific hardware features.
- The compiler has a packet format for bitfield structures. Unlike standard C, there are no restrictions on these bitfields. This is highly suitable for defining and accessing fields of protocol headers.

The C-compiler supports two C code compilations methods. First, the regular compilation of C source files (*.c, *.i) into object files (*.obj). The other method is to compile and link a Microengine program, see Figure 13. The C-compiler provides data types such as: 8-bit char,

16-bit short, 32-bit Int, 32-bit Long, and 32-bit pointers typed by memory type. The C-compiler accesses machine specific features through intrinsic functions and is supported for inline assembly. There is a subset of the standard C library with suitable extensions/modifications for network applications.

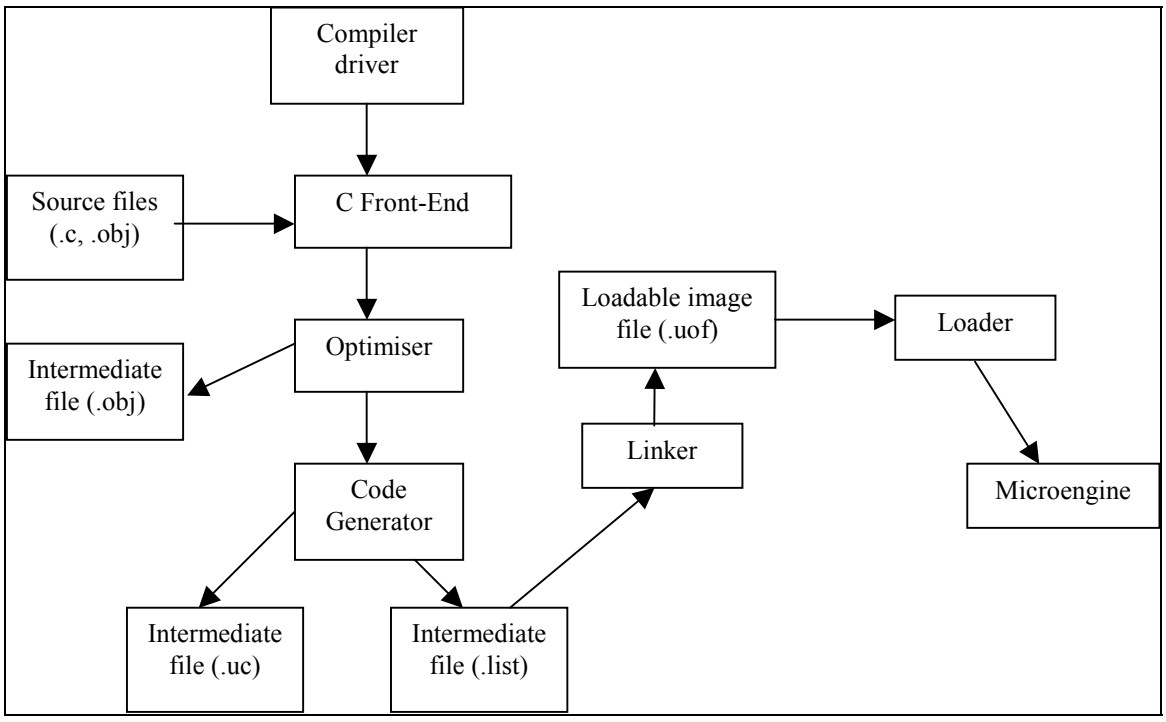

Figure 13. C compiling flow

Some missing features of the compiler are:

- Float and double data types
- Recursion
- Pointers to functions
- Variable length function argument lists (printf)

2.10.3 Linker

The linker is used to link microcode images, generated by the microcode compiler or Assembler. The linker carries out the following functions:

- It resolves Inter-Microengines symbolic names for Next Neighbour Registers, transfer registers, and signals.
- Creates internal tables for example: import variables, export functions, and image names.
- Choice of output format on either a loadable image file or a Hex image in “C” struct format.

2.10.4 Debugger

Using the workbench, the microcode can be debugged in both simulate and hardware mode. The debug menu in the workbench provides following capabilities:

- Set breakpoints and control execution of the microcode
- View source code on a per-thread basis
- Display the status and history of Microengines, threads, and queues
- View and set breakpoints on data, registers, and pins

When debugging in simulation mode, the Transactor and it is hardware model must be initialised before it can run microcode. This is done with script files, which is loaded under the Simulation menu. The workbench can execute one or more script files in a row when debugging starts.

The run control menu in simulation mode lets you govern the execution of Microengines. Different operations are available from the Workbench such as running Microengines indefinitely or only single step one Microengine cycle at a time.

Packet simulation is available from the Simulation menu. The workbench is able to simulate devices and network traffic such as:

- Configure devices on the media bus
- Create one or more data streams (Ethernet frames, ATM cells, POS)
- Assign one or more data streams or a network traffic DLL to each device port connected to the network traffic

2.10.5 Logging traffic

The packet simulator supports logging of received and transmitted packets for all ports on used devices. Logging is done on a per-port basis where received and transmitted logs being written to separate files. Only complete packets are logged, meaning that logging on a port starts when the next Start of Packet (SOP) bit is set on the arriving packet from the MSF. The logged data consists of the used data stream and can optionally show both frame numbers and media bus cycles for SOP and End of Packet (EOP). If logging uses both frame numbers and cycle times, the logged data looks like:

25 4387 4395 01010101010202020202…

Here we have 25 showing the frame number, 4387 shows the media bus cycle for the SOP, 4395 shows the media bus cycle for EOP, and finally the rest data in the row shows the actual packet.

2.10.6 Creating a project

A project consists of one or more IXP2400 processor chips, micro source code files, debug script files, and Assembler, Compiler, and Linker settings used to build the microcode image files. Each project has a system configuration defined, where a programmer can change settings such as clock frequencies on SRAM and DRAM memories, and PCI unit frequencies.

All these configurations can be accessed under the simulation menu in the workbench.

The executable image for a Microengine is generated by a single invocation of the Assembler that produces an output ".list" file. This output file can be designated to be loaded into more than one Microengine. In order for the Workbench to build list and image files, it must assign a ".list" file to at least one Microengine.

2.11 Programming an Intel IXP2400

The IXP2400 provides two different programming languages: Microcode and Microengine C. Microcode are analogous to assembly on a general-purpose processor allowing fine-grained access to register allocation, which SRAM and DRAM transfer registers can be explicated used. It has no notion of pointers or functions, but does allow modularisation of code using inline-macros. Microengine C is very similar to classic C language. It offers type safety, pointers to memory, and functions.

The IXP2400 network application structures are divided in three logical planes shown in Figure 14 below:

- Data plane, it processes and forwards packet at wire speed. It consists of: a fast path, which handles most of the packets (For example, forwarding IPv4 packets), a slow path used for handling exception packets (For example, handling fragmented packets).
- Control plane, it handles protocol messages and is responsible for setup, configure, and update tables used by the data plane. For example, the control plane processes RIP, OSPF packets containing routing information to update IPv4 forwarding tables.
- Management plane, it is responsible for system configuration, gathering and reporting statistics, stopping or starting application. It typically implements a GUI for displaying and getting information from a user.

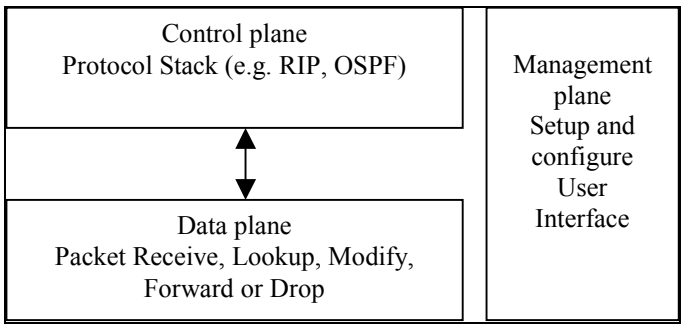

Figure 14. Logical planes used in IXP2400

2.11.1 Microblocks

The data plane processing on the Microengines is divided into logical network functions called microblocks. Microblocks can be written in either microcode or C code. Several microblocks can be combined into a microblock group where it has a dispatch loop which defines the dataflow for packets between different microengines in the microblock group. A microblock group can be instantiated on one or more microengines, but two microblock groups cannot share the same microengine.

Microblocks can communicate with the XScale core by using a dispatch loop (See section 2.11.2 below) to handle packets that come from the XScale Core component and steers it into the right microblock. Typical examples of microblocks are: IPv4 forwarding, PPP header termination, Ethernet Layer 2 filtering, etc. All the microblocks have the intent to be written independently from each other. By providing clean code, it enables to modify, add, or remove more microblocks without affecting the behaviour of the other blocks.

Microblock types

There are three different kind of microblocks used in the IXA SDK. These are:

- *Source Microblock*, runs at the beginning of a dispatch loop and is responsible for a packets processing in the rest of the pipeline. Source blocks are used for either reading data from media interfaces or schedules packets for transmission from a set of queues.
- *Transform Microblock*, this microblock processes a packet and passes it to the next microblock. It can modify buffer data, gather statistics on the buffer contents, or steer the buffer between multiple of blocks. It obtains buffer handles and relevant state information from the dispatch loop global state.
- *Sink Microblock*, this microblock is responsible for disposing off a packet within a current Microengine. It can include queuing packets to another microblock, or transmitting packets out of a media interface. A sink block is the last block executed in a microblock group.

A microblock written in micro code consists of at least two macros. First, an initialisation macro only called by the dispatch loop (See section 2.11.2) in the startup sequence. The second macro is a processing macro called for every packet received to the microblock. For a microblock written in C, there exist two functions instead of macros: An initialisation function and a processing function.

Each microblock can have one or more logical outputs to indicate where a buffer should follow next. These logical outputs are passed along by setting a dispatch loop global variable to a specific value.

Configuring a microblock

There are a couple of ways of how to configure a microblock. For example, each microblock has an SRAM area used for communication with its associated XScale component. This area stores parameters which may change at run-time. There is also some imported variables used to be determined during load-time of the microcode and do not change subsequently. There are tables and other data structures in SRAM, DRAM, or Scratch memory that are shared between the microengine code and the XScale core. One example of shared data is the IPv4 forwarding table used in IPv4 microblock.

2.11.2 Dispatch Loop

A dispatch loop combines microblocks on a microengine and implements the data flow between them. It also caches common variables used in registers or local memory. Examples of variables can be:

- Buffer handle for containing the start of a packet
- Packet size, to show the total length of a packet across multiple buffers
- Input port, shows that port a packet was received on

These variables can be accessed by a microblock by calling macros or C-functions. Such macros or C-functions can be: Get cell count from the buffer handle, allocate a buffer, setting an input port etc. The dispatch loop also provides communication between the XScale core and the sink/source microblocks to send or receive packets to XScale. For example, an exception packet is detected by a source microblock and needs to be sent to the XScale core. The source microblock sets a specific variable for exception packets called exception id. This id is then identified by the sink microblock and then being forward to the XScale core.

2.11.3 Pipeline stage models

For the IXP 2400 NP, the programming model provides two different software pipeline models:

- A context pipeline, different pipeline stages are mapped to different Microengines. A packet is passed from Microengine to Microengine where each stage operates on the packet and the available compute budget per context pipeline stage is the same as the budget available per Microengine
- A functional pipeline remains a packet context within a Microengine while different functions are performed on the packet as the time progresses. The Microengine execution time is divided into pipestages and each pipestage performs different function

These two different pipeline models have their own limitations. To get the best possible performance, a solution of combining both pipelines is necessary. The mixed pipeline has some stages running as a context pipeline and some stages running as functional pipeline. To choose which pipeline model to use, the best pipeline solution is based on the characteristics of each of the pipeline stages. The characteristics can be based on the total compute and total IO operations required for a given microblock.

Both ingress and egress application have a low line rate and therefore it can run one microblock on each microengine. Therefore, a context pipeline is the best solution to tie together all the microblock with their microengines.

2.12 Motorola C-5 DCP Network Processor

The Motorola C-5 DCP is a multi-processor Network Processor [24] and contains the following functional units:

- 16 Channel Processors (CPs)
- Executive processor (XP)
- Fabric processor (FP)
- Buffer Management Unit (BMU)
- Table Lookup Unit (TLU)
- Queue Management Unit (QMU)
- Three different buses which provide 60Gb of aggregate bandwidth, (Ring bus, Global bus, and Payload bus)

The C-5 processor is designed for use at layers 2-7 and it processes data at 2.5Gbps. A good overview of all the functional units is shown in Figure 15 on next page.

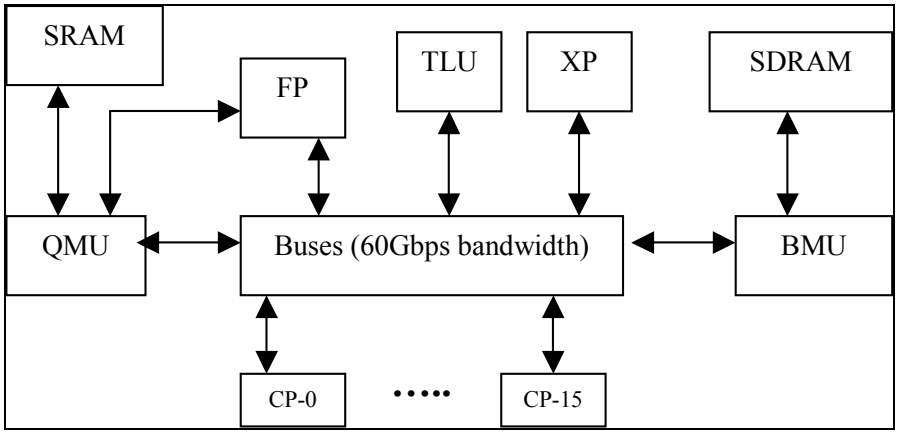

Figure 15. Basic architecture overview of a C-5 Network Processor [8]

2.12.1 Channel processors (CPs)

A C-5 NP [25] contains 16 programmable Channel Processors (CPs) that receive, process, and transmit cells and packets. Each CP has four threads each providing an execution context. Each context has its own register set, program counter and controller specific local registers.

Each Channel Processor (see Figure 16) contains four important components:

- Serial Data Processor (SDP), it has both receive and a transmit processor responsible for selecting the fields to be modified from a stream of data. The SDPs can handle common, time-consuming tasks such as: Programmable field parsing, extraction, insertion, and deletion. Other tasks performed by the SDPs are: CRC validation/calculation, framing and encoding/decoding.
- Channel Processor RISC Core (CPRC), processes the data which the SDP has chosen to modify. The RISC core specifically manages: Characterising cells/packets, collecting table lookup results, classifying cells/packets based on header data, and traffic scheduling.
- Instruction memory (IMEM), each CP has 6kB of instruction memory to store RISC instructions.
- Data memory (DMEM), each CP has 12kB local non-cached data memory to store data.

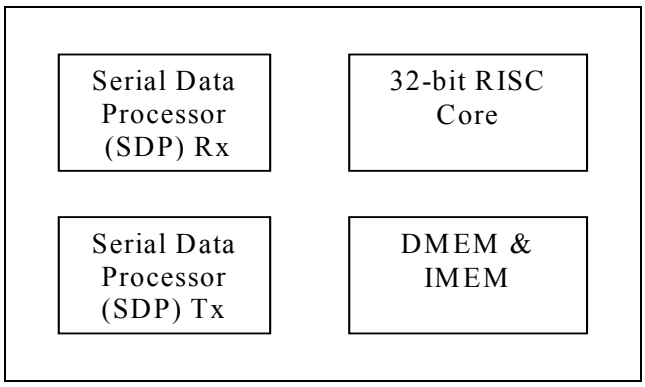

Figure 16. Channel Processor organization

The Channel Processors can be combined in several ways to increase processing power, throughput, and bandwidth. Typically, one CP is assigned to each port from medium bandwidth applications to provide full duplex wire-speed processing. To scale serial bandwidth capabilities, the CPs can be aggregated together for wider data streams and still

providing a simple software model implementation. Both these models can be applied simultaneously, see Figure 17, for minimising complexity of software development.

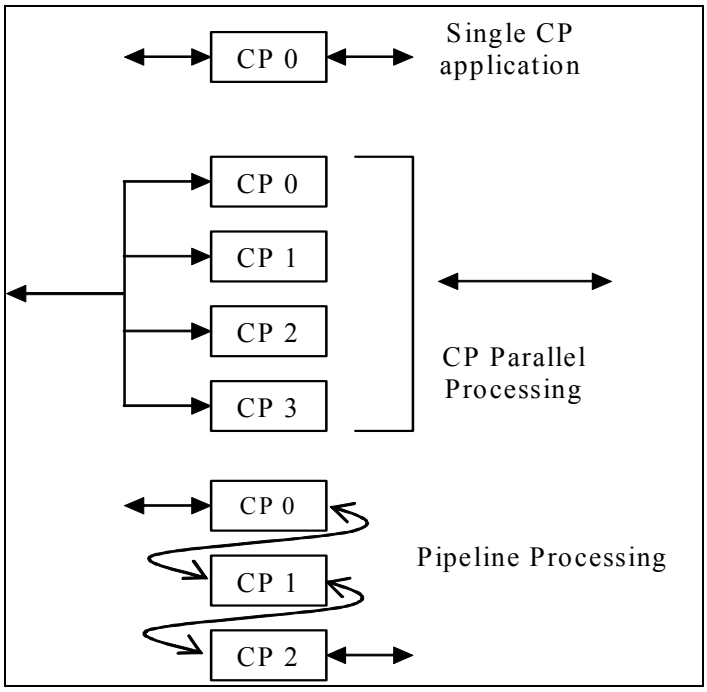

Figure 17. Parallel and pipelined processing

Context switching

Using internal registers, context switching on a Channel Processor is accomplished in two ways:

- Control Processor instructions (Software)
- Hardware Interrupts, where all interrupts are disabled until a Restore from Exception instruction has occurred.

Actual processing can begin on a different context in 2 cycles.

2.12.2 Executive processor (XP)

The Executive processor (XP) is the central processor unit of the NP and it can be used to connect several C-5 processors. It provides network control and management functions in user application and handles the system interface, for example a PCI bus which can be used to connect to a host.

Three main tasks for the XP are:

- Manage the statistics from the CP, DMEM, and TLU.
- Detect failure
- Routes or switches traffic

2.12.3 System Interfaces

The XP has these system interfaces:

- PCI Interface an industry standard 32 bit 33/66 MHz PCI channel interface. This is typically connected to a host.
- Serial Bus Interface contains three internal buses with an aggregate bandwidth of 60Gbps. It allows the C-5 NP to control external logic via either the MDIO (high-speed) protocol or low-speed protocol.
- PROM Interface, allows the XP to boot from non-volatile, flash memory.

2.12.4 Fabric Processor (FP)

The Fabric Processor works as a high-rate network interface. It can be configured to connect several C-5 processors to each other or to other interfaces such as Utopia level 1, 2 or 3. It also supports the emerging CSIX standard. This processor can be compared to Intel's Media Switch Fabric (MSF).

2.12.5 Buffer Management Unit (BMU)

The BMU Manages centralized payload storage during the forwarding process. It is a an independent high-bandwidth memory interface connected to external memory such as SDRAM memory for the actual storage of payload data. It is used by both the XP and the FP.

2.12.6 Buffer Management Engine (BME)

The BME handles the data buffers to/from SDRAM and it executes BMU commands.

2.12.7 Table Lookup Unit (TLU)

The Table Lookup Unit performs table lookups in external SRAM to the CPs, XP and FP. It supports multiple application-defined tables and multiple search strategies can be used for routing, circuit switching, and QoS lookup tasks.

2.12.8 Queue Management Unit (QMU)

The Queue Management Unit handles inter-CP and inter C-5 NP descriptor flows by providing switching and buffering. Each descriptor contains information about the fabric id, control data, and control commands used to setup a queue. The BMU can also perform descriptor replication for multicast applications. The QMU provides queuing using SRAM as an external storage for the descriptors. It supports up to 512 queues and 16384 descriptors.

2.12.9 Data buses

There are three independent buses used in the C-5 NP. First, a Payload bus used to carry payload data and payload descriptors. Second bus is a Global bus used to support an interprocessor communication via a conventional flat memory-mapped addressing scheme. Third bus is a Ring bus, used to provide bounded latency transactions between the processors and the TLU.

2.12.10 Programming a C-5 NP

To program a C-5, Motorola has developed a toolkit called C-Ware Software Toolset v.2.0 (For datasheets, see [24]). It is possible to write up to 16 different C/C++ programs for each of the 16 Channel Processors, as well as writing microcode for the serial data processors. System level code is required to tie both C code and microcode together. The C-ports Core development tools are based on the GNU gcc compiler and gdb debugger, modified to work

with Motorola's RISC cores. The C-port also contains a traffic generator and a traffic analyser.

C-port provides application library routines, called the C-Ware Application library, used for compatibility with future generations of Motorola's Network Processors. These routines cover features of both RISC cores and their co-processors, including tables, protocols, switch fabric, kernel devices, and diagnostics.

2.13 Comparison of Intel IXP 2400 versus Motorola C-5

A C-5 NP has enough processing power to implement both data and control operations itself or it can communicate with a host CPU across a PCI bus interface. Motorola's C-5 NP has 16 Channel Processors which have the same functionality as Intel's Microengines. These two NPs use parallel processing to increase the throughput of their device. Both Motorola and Intel run multiple processors independently on each processing element (MEs or CPs). Intel and Motorola have two different approaches as to how to parallel process traffic. Intel uses pipelined processing, where each processor (Intel's ME) is designed for a particular packet-processing task. Once a Microengine finishes a packets processing, it sends it to the next downstream element (ME).

Motorola uses parallel processing where each processor element (CP) performs similar functions. This is commonly used when using co-processors on specific computations. In Table 1 on next page, there are listed some comparisons between Motorola's NP and Intel's NP.

Intel and Motorola have different ways on how to distinguish traffic from the same network device. If traffic arrives on more than one device, it gets difficult for the Intel NP to know which of the traffic to process. The Channel Processor divides its 4 context into 2 contexts handling receiving tasks and 2 contexts handling transmitting tasks. Intel solves the problem by software programming all threads to listen to a creation port/device.

Table 1. Comparison between Intel & Motorola network processors

	Intel IXP 2400	**Motorola C-5**
Central Control Processor	32-bit XScale Core 400/600 MHz	32-bit Executive Processor (XP) 66MHz
Interfaces	33/66 MHz PCI bus (64 bit) UTOPIA (Level 1-3) SPI-3 (POS-PHY 2/3) CSIX-L1B	33/66 MHz PCI bus (64 bit) UTOPIA (Lev. 2 &3) CSIX-L1B Power X Prizma
Processing Elements (PEs)	8 Microengines (ME) with 8 context each (Supports up to OC-48)	16 Channel Processors (CPs) with 4 context each (Supports OC-12)
Compilers	C Compiler & Assembler	C & C++ Compiler
Memory in Core Processor	Instruction: 32Kbyte Data: 32Kbyte 2Kbyte mini cache	Instruction: 48Kbyte Data: 32Kbyte
Memory Processor Element	Instruction: 4Kbyte	Instruction: 6Kbyte (24Kbyte in a cluster of 4 CPs) Data: 12Kbyte
SRAM	2 channel x 64 MB (QDR) Runs at 100-250 Mhz	8 MB TLU SRAM (143 Mhz) 512 KB QMU SRAM (100 Mhz)
SRAM Bandwidth	12.8 Gbps total 6.4 Gbps per channel	1.04 Gbps
SDRAM	-----	128 MB
DRAM	1 64-bit channel 2 Gb (ECC - DDR)	128 MB ECC DRAM
DRAM Bandwidth	19.2Gbps (peak)	1.6Gbps
Consuming	10 W (Typical for 600MHz) 7 W (Typical for 400MHz)	15 W (Typical)
Media Interface Bandwidth **(Line rate)**	2.5Gbps (full duplex) 4.0 Gbps (Maximum)	3.2 Gbps (full duplex)
Instructions/cycle	8	16
Package layout	1356 pin FCBGA	838 pin BGA
MIPS	4800 (1000 in XScale)	3200
Operating - temperature range	-40° to +85°C	-40° to +85°C
IC process	0,18µm	0,18µm
Expected prize	$360 (600 MHz) $230 (400 MHz)	$400 (in quantities of 1000 devices)

3 Existing solutions

This thesis has concentrated on packet forwarding. Today, there exist many solutions for different types of forwarding tasks. However, this thesis is not simply about how to forward packets, a network switch must also handles incoming packets, reassemble incoming ATM cells, and split frames into ATM cells for transmitting over a Utopia interface.

The main goal of my thesis is to cover most of the requirements stated in Appendix A, necessary to create a highly flexible and functional Packet Over SONET Line Card, to replace the Ericsson Exchange Terminal (ET-FE4) while showing this is feasible to implement with a network processor.

Many of the NP vendors have their own modules for forwarding implemented for their own NP. For example, Alcatel and Intel have each developed their own implementations, and in the Intel's case, some third party companies have developed their own applications running on Intel's IXP1200 NP.

3.1 Alcatel solution

Alcatel has developed a forwarding engine module (FEM) [41]. The module uses a Network Processor, called *Alcatel 7420 ESR* (See [35]). FEM is responsible for forwarding, filtering, classification, queuing, protocol encapsulation, policing, and statistics generation. FEM uses four NPs, two NPs are for the incoming packets and two for the outgoing side, see Figure 18. Packets are received by a physical interface and first delivered to the Inbound Data NP on the ingress side. This processor determines the protocol encapsulation and forwards the header information to the Inbound Control NP for processing. The Inbound Control NP performs a Content Addressable Memory (CAM) lookup, classification, longest match lookup, and creates a Frame Notification (FN) message to be used by the Outbound NPs. The Outbound Control NP receives the FN and queues it depending on the message. The Outbound Control NP sends a message to the Inbound Data NP when it is ready to transmit. The Inbound Data NP sends the packet to the Outbound Data NP that transmits the packet out the physical port.

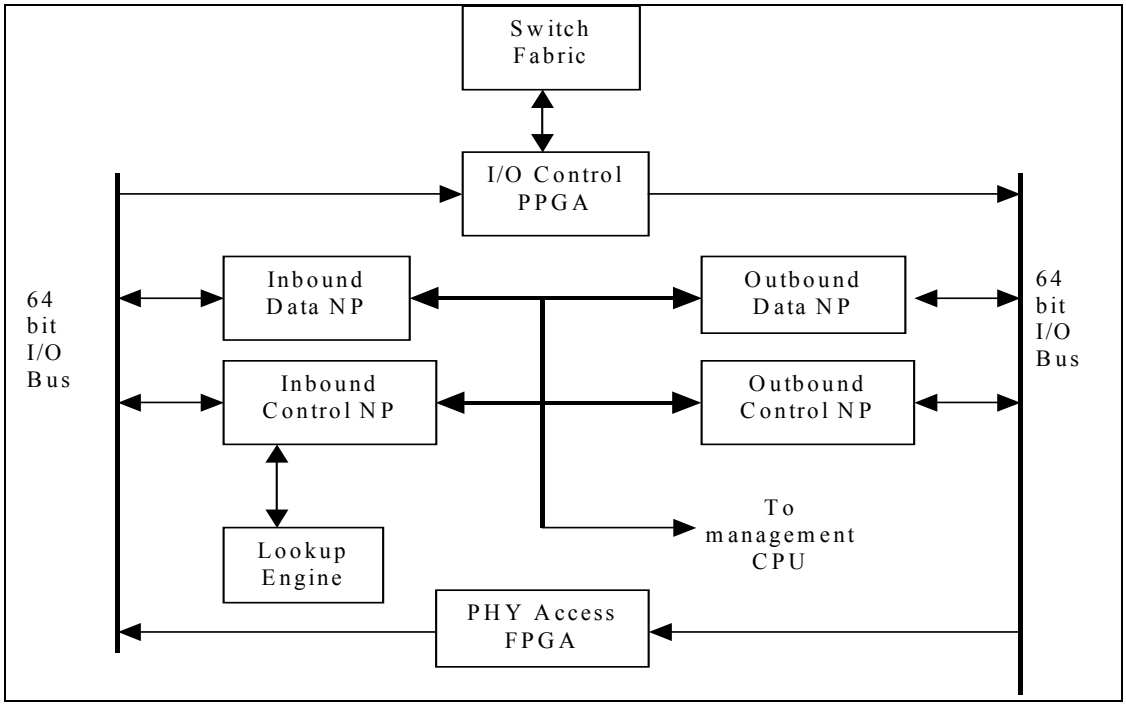

Figure 18. Basic description of Alcatel's Forwarding Engine [41]

3.2 Motorola C-5 Solution

Motorola has developed an application for POS traffic forwarding [26]. The application is a Packet-over-SONET application running on Motorola's C-5 Network Processor (C-5 NP) on OC-48 links. The C-5 NP is described in section 2.12. The section below describes briefly the application and the tasks for the Channel Processors (CPs) and the Executive Processor (XP). Finally the ingress and egress data flow are explained in detail.

3.2.1 Overview

The C-5 NP supports the following features:

- Layer 3 forwarding, processing and a forwarding IP frames at layer 3 based upon the IP destination address.
- Diffserv QoS (Currently not supported)
- 16-way aggregation is supported on each Channel Processors. It relies on sequence numbers between physical interface and the QMU to maintain traffic sequencing allowing aggregation across larger group of CPs.
- IP Flow routing, the application implements a multi-field classification scheme based on an IP flow concept where the flow is defined by fields from layer 3 and 4. A zero value in the TOS field will result in Layer 3 IP routing of the packet.
- ICMP support: However, only Time Exceeded, No Route, and Destination Unreachable messages are fully supported. It only needs these messages, though these are the only messages used in the data plane.
- Fabric port support (Back to back)
- PPP Statistics
- Multi-field classification

Even though the application is a Packet over SONET application, the SONET framing and overhead processing is performed by a SONET framer on the other side of the physical interface.

XP

For this application, the XP processor is used for boot and a two-phase initialisation of the network processor chip. The first initialisation phase allocates queues, buffer pools, and configures the mode registers used by each CP. The second phase initialises the services functions, configures the QMU, and initialises the host processor interface and TLU with static table data. It also configures the Fabric port for back-to-back operations, loads all CPs.

CP

All CPs used for this application performs the same functions. These includes following:

- Initiating receive and transmit programs for the DSPs
- Supports 16-way aggregation
- Processing lookup from the TLU
- Constructing descriptors for forwarding frames via the QMU to the Fabric Port
- Processes descriptors from the QMU for forwarding frames from the Fabric Port to the Physical Interface

FP

The application configures the Fabric Port (FP) to operate in “back-to-back” mode for connection to another C-5 Network Processor through a switch fabric. The application uses the FP to forward descriptors and data to another C-5 NP. In the buffer handles, there are some bits used for the FP to recognise where in the buffer memory a certain frame is stored, the length of a frame, and the target of a queue to which the frame should be sent

3.2.2 Ingress data flow

First, the Channel Processor (CP) checks that no errors were detected during header parsing. If errors are detected, the packet is discarded and statistics are updated based on frame status reported by the Serial Data Processor (SDP). If no errors are detected, it checks if the routing protocol is identified by the SDP as either IP or IP flow, otherwise the packet is dropped as an invalid protocol supported. If the protocol is valid, the lookup results for the lookup launched by the SDP are retrieved. IF the route was not found, the packet is handled as an ICMP destination unreachable message. In addition, if the route was found, but not through the fabric, the packet is handled as an ICMP redirect. If the route was valid, it performs a header length check. If the length is too small, the packet is discarded, and if the length is sufficiently large, a speculative enqueue is performed to provide the QMU with knowledge about the packet sequence number.

Now, the processing waits for the payload reception to complete. When it is completed, a final check of frame status from the SDP is done to detect CRC errors or oversized frames. If it detects a frame error, the frame is then discarded and statistics updated based on the frame status. If no errors are detected, either a normal enqueue, or a commit with valid status is performed to forward the frame to the fabric. Context swaps to the egress processing thread are performed whenever the processing is stalled waiting for an event in another component of the system to occur. The most demanding tasks for the ingress flow are:

- Waiting for an extract scope from the SDP
- Waiting for the lookup results from the TLU
- Waiting for the payload reception to complete
- Waiting to allocate a buffer either for initialising conditions for next reception or to prepare an ICMP response

3.2.3 Egress data flow

First, the egress processing waits on a dequeue token in the egress Channel Processor cluster. After the token is available, then it waits for a non-empty transition on the output queue. When traffic is present in the queue for transmission, a dequeue action with sequence number is initiated. The processing then waits for the dequeue action to complete. If the dequeue was not successful then the associated buffer is de-allocated. If the dequeue was successful, and the frame is sufficiently small, and the dequeue token is passed to the next CP in the cluster, otherwise token passing is delayed until later. If token passing was delayed, the CP now begins monitoring the number of bytes remaining to be transmitted to the physical interface. Once the bytes drop below the required threshold, the dequeue token is passed to the next CP in the cluster. The delay is necessary to prevent overflowing FIFOs used, in the case of two large frames being transmitted.

3.3 Third parties solution using Intel IXP1200

Two third party companies or laboratories have developed applications using Intel's IXP1200. One laboratory, *the IXP Lab* at Computer of Science Department at Hebrew University of Jerusalem [34] has tested and implemented code for packet forwarding.

Teja Technologies has developed a software platform which is an integrated network platform for forwarding and the control plane. One of the applications developed by Teja, called G2RLFT is a complete RFC 1812-complaint IP forwarding application which uses two full-duplex Gigabit Ethernet ports. It has two pipelines consisting three stages:

- Stage 1, two Microengines used for receiving packets from the IX-bus and also perform layer 2 filtering.
- Stage 2, single Microengine used for IPv4 forwarding. An incoming packet causes an IPv4 lookup and the routing decision is based on the longest prefix match in a routing table.
- Stage 3, two threads on a Microengine sends packets over two separate gigabit Ethernet ports.

The platform also includes a graphical development environment for system design, code generation, testing, and debugging. With this software, Teja has developed an application building block for IPv4-forwarding [32]. To accommodate Classless Internet Domain Routing (CIDR), it uses a Longest Prefix Match (LPM) process and uses Forwarding Routing Tables and Forwarding Tables to make the best forwarding decision. This implementation has been successful, therefore Teja and Intel have decided to continue to work together with the next future network processors. The application supports a range of 10/100Gbps Ethernet routing and forwarding at wire-speed.

4 Simulation methodology for this thesis

In this thesis, simulation of the network processor will be done using an Intel development toolkit (IXA SDK 3.0). The toolkit/workbench is a simulator for the upcoming network processor Intel will ship in 2003. The simulator was used to simulate the Network Processor code which later will be used in the actual hardware. Unfortunately, this toolkit is still being developed. The final release will only be available sometime in the beginning of 2003. This release is only a pre-release of the final version, and that makes it a little more difficult to know how reliable the simulator is. If it supports all the necessary functionalities and do not have too many bugs of importance. However, Intel will continue to add more functionality until the final release is shipped. Currently the fourth version is available for customers. My implementation activity is divided in four different phases:

- Studying existing modules which are already implemented, i.e. software and hardware (for the existing Ericsson ET and library modules for the processor)
- Design and implementation of a simple forwarding module
- Modify it into a more complex module, to support full duplex forwarding
- Analyse this module's performance

Only some of the modules which should be included in the final release of the toolkit are available, therefore it is a good idea to first analyse and study the existing modules that come along with the toolkit. There are also some examples from the older toolkit for the IXP 1200. Fortunately, the new toolkit is source compatible with the old processor. When more modules are released, they can be analysed as well and perhaps used if they are suitable.

The first phase of my work was to gain knowledge and understanding of which modules could be reused and which modules should be modified or removed.

The next phase was to design and implement a basic IP forwarding module. The functionality should be as simple as possible. First, it receives a Packet over SONET (POS), strips of the PPP header, forwards the enclosed IP packet into an output queue, and emits the frame as ATM cells.

The third phase was to develop a more complex forwarding module which supports both frame ingress and egress. This module should evolve into a final implementation. For example, the packets should be able to be forwarded depending on different parameters. One difficulty is handling exception packets. The processing of these packets is a task for the control plane (that consists of the XScale core). A programmer can only access the data plane by programming the microengines of the NP. Unfortunately, the XScale core components are not currently supported by the development environment, so therefore this initial implementation will only mark the exception packets and then drop them. The XScale emulator is going to be supported in a later release, so if time is left during the project this part can also be implemented.

The final phase involves evaluation of my implementation, documenting how it works, and indicating future work, which should be done in the next project.

4.1 Existing modules of code for the IXA 2400

Using existing modules can save a lot of time during the implementation phase. The more reuse, the more time I will have for other things, such as performance analysis or perhaps implementing more functionality.

Intel has already developed a group of microblocks, called applications. Two of these applications are POS Ingress and POS Egress, used for OC-48 links. The first application covers the ingress side, which receives packets from a POS media interface, processes it, manages it, frames it into CSIX frames (see [38]) and sends it out on a Media Switch Fabric (MSF) interface. The other application (egress side) receives CSIX frames from the MSF interface, reassembles them into IPv4 packets, adds a PPP header and transmits the frame over a POS interface.

The existing applications run on one NP for the ingress side and one for the egress side. This can be both expensive and inefficiency with respect to the goal of this thesis project. For example, the requirements for this thesis project are to handle line rates of OC-3 links (155Mbit/second). All existing applications runs over OC-48 links with a line rate of 2.5 Gbit/second, thus for OC-3 the NP has much more time to process the packets. Therefore, I am going to implement an application where I use existing modules form the ingress and egress application discussed above, see Figure 19, except for the QoS block, which is not yet implemented by Intel. The QoS microblock is a microblock group providing DiffServ functionality. It consists of three microblocks: WRED, Metering, and Classifier. On the existing applications (ingress and egress) uses both a queue manager and scheduler. Here, we use lower line rate, which makes it possible to skip these blocks on the ingress side. Therefore, all the necessary microblocks can now fit within eight microengines with one microblock for each microengine.

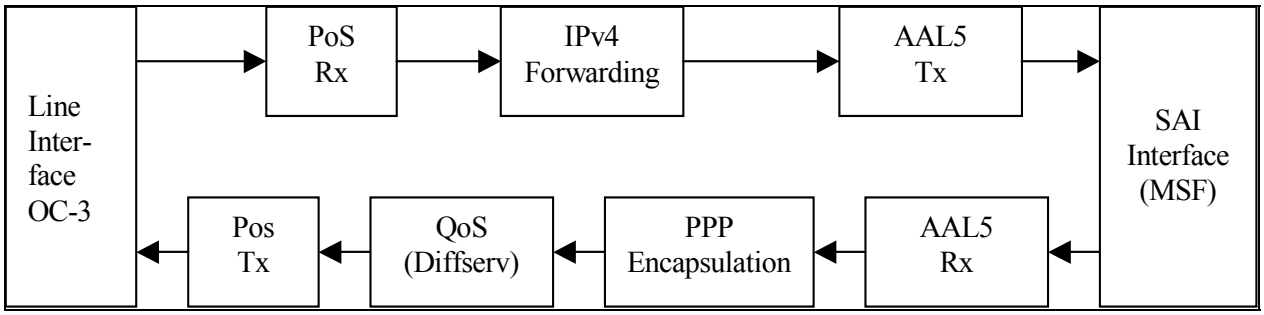

Figure 19. Single chip solution

As we see in Figure 19, two main parts are needed, a single ingress side which sends traffic to the right and a single egress side with the opposite flow. The ingress side consists of several pipelines using microblocks. These pipelines are connected to each other through a Scratch ring (also called dispatch loop, see section 2.11.2).

If this implementation is unable to implement the microblocks as specified, then there is a second solution, see Figure 20. In this alternate implementation, the ingress part is augmented with a Cell Queue Manager and a Cell Scheduler. This solution is only used to simplify the programming. The Queue Manager modifies a cell count variable in a complex way. In the first solution, there is not done yet, and if it is not possible to fix a correct cell count, then the second solution is the best way. On the egress side, the QoS microblock group have been replaced with a Queue Manager and a packet scheduler. This affects the proposed goal to have a DiffServ functionality to classify incoming cells. However, the main responsibility for the egress side is to receive cells, reassembles it into IP packets, encapsulate PPP header and send it out. Therefore, the exactly functionality of using a Diffserv based Queue is not the most important functionality for this application. The only reasons to not implement the QoS microblock group is that Intel are still developing these blocks, and they have not released it for customer use. Thorough, this can be a work for the future, where the Packet Scheduler and Packet Queue Manager are modified and added together with the QoS microblock group.

Therefore, due to limited of time and limitations of the workbench, there is a high risk that the second solution will be the only solution implemented as part of this thesis project.

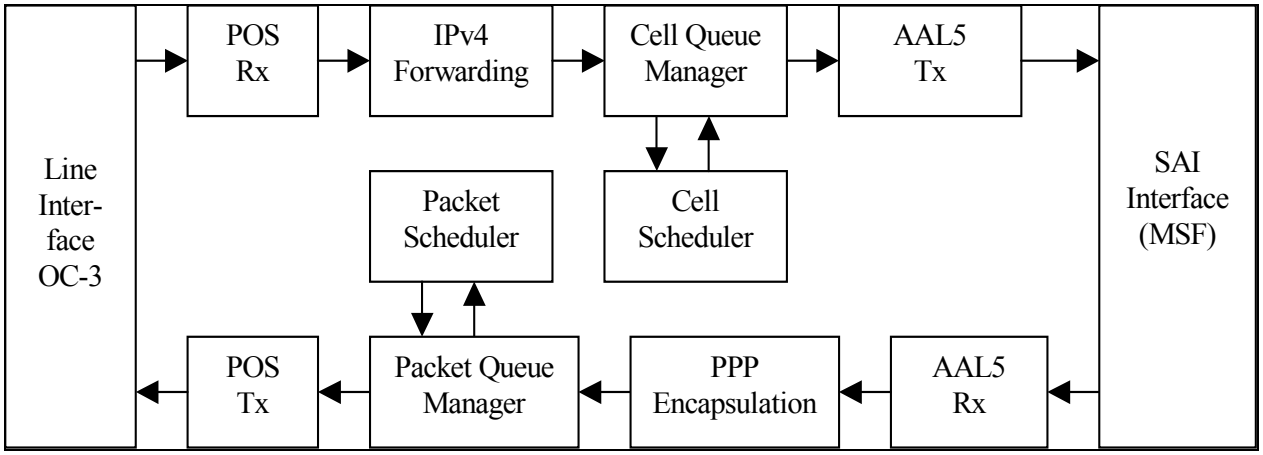

Figure 20. Dual chip solution

4.2 Existing microblocks to use

The existing microblocks implemented by Intel were designed for OC-48 line rates for both IPv4 forwarding and ATM reassembly. These microblocks can be used on both the ingress and egress side. The section below describes these microblocks, which can be reused.

4.2.1 Ingress side microblocks

POS Rx

This block runs on a single microengine with eight threads and it performs frame reassembly on the incoming mpackets from the POS media interface. An mpacket is a packet with a specified size from MSF. Each of the packets is checked by the POS Rx block to see if it is a PPP control packet (LCP or IPCP) or not. If they are, then the packets are sent to the XScale core for further processing. All other packets (i.e., IPv4) are queued in a scratch ring for processing by the next stage of the pipeline. These packets are marked either to be dropped or to be sent to the XScale core marked as exception. POS Rx uses checks these tags, and then sends the packets that been dropped to a drop scratch ring and the exception packets to the XScale core. Until the core components are fully supported, all exception packets will be dropped.

IPv4 Forwarding

This microblock runs on a single microengine. Actually, it consists of two microblocks, PPP decapsulate and IPv4 forwarder. The two blocks are integrated together in a microblock group on the same Microengine. The PPP decapsulate microblock is small enough to work together on the same Microengine as IPv4 forwarder. From now, we will only mention IPv4 forwarding microblock as a single microblock, just for ease of understanding.

The IPv4 forwarding Microblock dequeue packets from the scratch ring which the POS Rx used for storing packets. Then it validates and classifies the PPP header of the packet. If the header contains the correct PPP protocol (see Appendix A) it decapsulates the PPP header and validates the IPv4 header. If the validity check fails, the packet will be dropped. Otherwise, it performs a Longest Prefix Match (LPM) on the IPv4 destination address. If no match is found, the packet is sent to the XScale core for further processing. When a packet needs to be fragmented, it is also sent to the XScale Core. The IPv4 microblock checks if the

packet should be dropped or sent to the XScale core. If not, it sends the packet as an enqueue request to the Queue manager over a scratch ring. This needs to be modified so instead of sending the packets to a queue, it sends to the AAL5 Tx block.

Header validation

The microblock performs a specified header validation including both MUST and SHOULD requirements stated in [19] and are summarized in Table 2 below.

Table 2. RFC1812 MUST & SHOULD Header checks [19]

Serial No.	RFC 1812 Check	Action
1	Packet size reported is less than 20 bytes	Drop
2	Packet with version !=4	Drop
3	Packet with header length<5	Drop
4	Packet with header length>5	Exception
5	Packet with total length field<20 bytes	Drop
6	Packet with invalid checksum	Drop
7	Packet with destination address equal to 255.255.255.255	Exception
8	Packet with expired TTL (Is checked after the forwarding decision is made)	Exception
9	Packet length< total length field	Exception
10	Packet with source address equal to 255.255.255.255	Drop
11	Packet with source address 0	Drop
12	Packet with source address of form {127, <any>}	Drop
13	Packet with source address in Class E domain	Drop
14	Packet with source address in class D (multicast domain)	Drop
15	Packet with destination address 0	Drop
16	Packet with destination address of form {127,<any>}	Drop
17	Packet with destination address in Class E domain	Drop
18	Packet with destination address in class D (multicast domain)	Drop

The IPv4 microblock also has a directed broadcast table used to list addresses, which are invalid. The following addresses belong to this list:

- {<Network-prefix>, -1}
- {<Network-prefix>, 0}

This directed broadcast addresses are populated by the XScale core and the IPv4 microblock uses macros or C function calls to perform a lookup on this table. Only the source address is subjected to a lookup in this table, the destination address check is a by-product of the lookup in the route table.

Another function implemented are Next Hop Information. A result of a route lookup is a next hop index, which is used to obtain the next hop information. It uses a Next hop database to store next hop information corresponding to a route.

Longest Prefix Match Lookup (LPM)

The longest prefix match (LPM) [15] is a key component of the IPv4 microblock. A single instance of the routing table occupies a minimum of 256 KB. The LPM algorithm consists of trie blocks. A trie is an array of entries indexed by a portion of destination IP address. The trie entries can obtain index to next hop information, an index to another trie, or both. This implementation requires 16 bits for the next trie, which implies an address space of 64 K tries. Each trie contains sixteen entries, except the starting block, which is the root trie. The algorithm takes four bit at a time from the destination address and the number of tries traversed in a lookup is based on how long the prefixes are in the routes installed of the route table. This algorithm is called dual lookup algorithm as two tables are used to perform a lookup. The purpose to have two tables is to have at most five dependent SRAM reads for a route lookup as well to reduce the number of entries. When a route is added/deleted, there is a maximum of eight trie entries and a minimum of one trie entry.

Cell Queue manager

The cell-based Queue manager (QM) is a context-pipe-stage microblock implemented on a single Microengine. It is responsible for performing enqueue and dequeue operations on the transmit queues. The QM supports up to 16 dequeue descriptors cached in the Q-array on SRAM memory and it uses CAM to maintain this cache of queue descriptors. The queue descriptor contains following: a buffer handle for the head queue, buffer handle for the tail queue, packet/cell count variable, and user defined bits.

When the QM receives an enqueue operation, the QM checks the CAM to see if the queue descriptor for the certain queue is cached in the local memory. If it is, it performs the enqueue operation or otherwise a LRU (Leas Recently Used) queue descriptor is evicted from the Q-array and written back to SRAM. A similar operation is implemented on a dequeue operation.

The QM runs on all eight threads on the Microengine with each thread handling:

- An enqueue request from the pipeline (scratch ring), these requests are on per-packet basis
- A dequeue request from the transmit scheduler Microengine
- An enqueue transition
- A dequeue transition (or invalid dequeue)

Cell Scheduler

The cell-based scheduler is context-pipe-stage microblock implemented on a single Microengine. The scheduler schedules packets (ATM cells) to be transmitted to the Media Switch Fabric (MSF), one at a time. The scheduler supports up to 64 ports to the fabric and 16 QoS classes per port with one queue per class. The scheduling algorithm is Round Robin among the ports on the fabric and optionally Weighted Round Robin among queues (i.e. classes) on a port (See section 2.5.2).

The scheduler handles:

- Flow Control messages from the fabric. These messages are only useful in full duplex mode, i.e. a two-chip model. The messages are used to send to the other chip, if the fabric wants to stop scheduling packets.
- Queue transition messages from the Queue manager to the Scheduler via a Next Neighbour Ring. A queue is scheduled only if there is data in the queue
- MSF transmit State Machine, the scheduler monitors how many cells are in the pipeline and if it exceeds a certain threshold, then it stops scheduling
- Checks if the Tx pipeline is within the threshold for sending cells
- Dequeue messages to the Queue manager, used to signal the Queue manager to dequeue a certain packet from the queue and send it further on the pipeline to the transmit Microblock

The scheduler contains four threads each of which is assigned different tasks shown below:

- *Scheduler Thread*, this thread is responsible for scheduling a queue and send a dequeue request to the QM Microengine. Every time the scheduler thread runs, it schedules a queue and writes a dequeue request to the QM scratch ring before it swaps out to let other threads run
- *QM Message Handler Thread*, this thread handles the messages that come back from the QM. The thread updates the bit vectors indicating which queues have data based on the messages received.
- *Flow Control Handler Thread*, this thread reads the Flow Control FIFO on the MSF and updates the flow control bit vector accordingly. Each flow control frame read from the fabric has n messages of one word each. It contains a 16-bit header, which contains the payload length and a 16-bit trailer, which contains the vertical parity, see [38]. Every time the thread runs, it handles a flow control message and then swaps out to let other threads run.
- *Packets in Flight Handler Thread*, this thread runs in an infinite loop reading how many packets that are transmitted to the MSF. This is used to throttle the scheduler if the number of packets exceeds a certain threshold.

AAL5 Tx

This microblock runs on a single microengine and it performs AAL5-CPCS, AAL5-SAR and ATM layer functions. The AAL5-SSCS layer is not used.
It receives transmit messages from the queue manager. For each transmit request, the microblock processes a CPCS-SDU (i.e. user data frame) between 1-65535 bytes. The last transmit request will process between 0-48 bytes. When the AAL5 Tx microblock receives a transmit request, it checks if the request refers to a new packet or not. If it is a new packet, it tries to read at most 48 bytes of data for the packet. If the packet is smaller than 48 bytes, it

then pads the rest of the packet to fill up to 48 bytes. After one 48-byte cell has been created, a CRC calculation is done and it stores the result in a transmit context. Finally, it copies the 48-byte cell along with a 4-byte header into a TBUF element for the MSF to transmit. If the end of the packet is reached, the packet length and stored CRC are used to create the 8-byte CPCS trailer. It pads the SDU, such the SDU plus an 8-byte trailer becomes a multiple of 48 bytes.

4.2.2 Egress side microblocks

AAL5 Rx

The AAL5 Rx microblock runs on a single microengine and runs on only 4 threads, therefore no inter-Microengine signalling is required. It supports up to 64 K Virtual Circuits (VCs). For each of the 64 K VCs, there is an associated Reassembly Context (RXC) of nine long words stored in SRAM. This helps the frame reassembly task to spread out on multiple threads. Since AAL5 frames may be up to 64 KB, some large packets may be stored in multiple buffers chained together in a link list. The buffer handles for the first and last buffer of the packet, are passed to the next microblock in the functional pipeline.

For each ATM cell received by the MSF, the MSF autopushes a Receive Status Word (RSW) to the transfer registers of a Microengine thread. The thread makes a hash lookup based on the VPI, VCI and input port of the received cell to get the location in SRAM. The AAL5 Rx microblock then receives the RXC from SRAM and copies the entire ATM cell into the Microengines registers. It strips of the ATM header, recalculates the CRC over the remaining 48-bytes. The payload of the cell is then written to DRAM. If the cell is an EOP, the CRC is validated against the CRC in the trailer of the AAL5 frame. The packet length is also validated against the number of bytes received for the packet. Finally, if it is a valid cell, it passes the required data to the next Microblock in the pipeline via a scratch ring.

Packet Queue manager

The packet-based Queue manager (QM) is almost identical to the ingress application microblock called Cell Queue manager. One main difference is that the ingress QM works on a cell mode while this egress QM works on a per-packet basis. This means that the egress QM dequeues an entire packet and sends it to the transmit block, the ingress QM dequeues cell in a packet and sends it to the transmit block. Another difference is that the egress QM also sends out a response message on every dequeue request, instead of only sending out in the case of invalid dequeue requests or if a queue goes from empty to non-empty or vice-versa.

Packet Scheduler

The packet scheduler is a frame-based scheduler, i.e. it schedules a complete packet at a time. This differs from the ingress scheduler, which is a cell-based scheduler that schedules an ATM cell at a time.

It supports up to 16 virtual ports implementing Weighted Round Robin (WRR) scheduling on the ports. For each port, the scheduler supports up to 16 QoS classes and 1 queue per class to a total of 256 queues. It implements Deficit Round Robin (DRR) scheduling on the queues within a port. The scheduler runs on one Microengine using only two threads:

- Scheduler Thread, is a thread with the same functionality as the Cell Scheduler
- QM Message Handler Thread, is a thread that receives dequeue requests from the scheduler, and for each request, it sends a transmit message to the transmit Microengine and a dequeue response to the Queue manager

PPP Encapsulation

This microblock runs on a single Microengine. It adds the PPP header to the packet, based on the header type field in meta-data stored in SRAM. If the header type field is IPv4, then the PPP header is set to 0x0021.

If the next-hop ID is invalid, then the microblock assumes that the PPP header has already been added to the packet by a previous stage.

POS Tx

This microblock is used for transmitting packets over the POS interface. It reassembles IP packets, segments it into frames, adds the layer-2 (PPP) header, and moves all the frames into a buffer for the POS interface to transmit. In this implementation we will use a multi-physical-port POS transmit microblock (MPH-16). In the first pre-release, I used pre-release 3 where it only existed one microblock for transmitting POS packets. From pre-release 4 there now exists two different microblocks, Packet Tx (SPHY) and Packet Tx (MPHY-4 or MPHY 16). The microblock I choose to call POS Tx is Packet Tx (MPHY-16) and handles up to 16 virtual ports. The POS Tx receives packet-based transmit requests from the Queue manager through a scratch ring. The POS Tx then, segments the packet into mpackets, copy them into TBUFs and send them over the fabric interface.

4.3 Evaluating the implementation

A thorough evaluation is compulsory. To get a true view of the functionality it is necessary to develop a good test suite. Recently, the Network Processor Forum (NPF) has defined a standardized specification on how to benchmark a Network Processor [46]. The document defines methodologies for creating standard-based network processors benchmarks suites. The benchmark suits tests both control and data plane functionality. General-purpose processor benchmarking tests such as SPEC [51] uses C or high-level language code for benchmarking. Instead, this document is a functional specification used to define all test configuration variables and metrics for benchmark suits and is now available as open standards.

In this thesis, I used a pre-release of the Intel development tools and this lead to problems when trying to evaluate the performance for real traffic. For example, while Intel promises a certain performance for their C-compiler, this may or may not be achieved with the pre-release compiler. As my evaluation is based on using the pre-release tools, this analysis can not be seen as full evaluation which would satisfy both Intel and their customers, rather the results point out weaknesses and limits of the Intel IXP2400 and Intel's development tools for it.

This thesis focuses on:

- Evaluating the C-compiler against microcode
- How memory configuration effects the processing
- Both practical and theory study of the microblocks used in the ingress and egress application I have implemented
- Test if loopback is possible for this Network Processor. (Although not yet tested, one could generate packets on the ingress side, send them through the pipeline, and loop them back to the output side of the OC-3 input. This means that the simulator must handle full duplex traffic, this would provide a more realistic simulation before construction of the actual ET board.) For the moment, this is not yet done, but one could generate packets on the ingress side and send them through the pipeline and loop them back to the output side of the OC-3 input. This means that the simulator must handle full duplex traffic, thus giving a more realistic evaluation before construction of the actual ET board.

To benchmark the application, it is interesting to look at metrics such as: Instructions per cycle, latencies in accessing memory, branch prediction, and average number of instructions in flight. One test example, which I have used, is to use 3 different packet sizes, 31, 42, 54, and 92 bytes. Each test run forwarded 200 packets.

5 Performance Analysis

Today several benchmark programs such as SPEC CPU2000 [51], TPC-C [47], TPC-H [48], and TPC-W [49] are in use to evaluate performance for traditional CPUs However, such benchmarks are just beginning to emerge in the Network Processor field. Network Processors needs to be evaluated at hardware and software level, therefore the benchmarking applications poses more demand than regular ones. One key ingredient for performance analysis methodology is a detailed data movement model of the target application. This model should describe various operations performed by the network processor on every received packet. Depending on the application, the operations can include: error checks on the protocol header, route lookup, filtering, queuing. Next, we use the data movement model to estimate the number of compute cycles and total I/O references required for these operations on a per-packet basis. Total computed cycles is the number of instructions which needs to be executed to complete a certain task. The I/O references include operational to external memories to read and write information required during a processing stage.

Another important aspect of performance analysis is the estimation of the total available budget for packet processing. This budget is allocated on a per-microengine basis and it determines how much processing a NP can perform on each packet. It is mainly based on packet-arrival time, NP frequency, receiving packet data rate, and how small the received packet size is.

5.1 Following a packet through the application

There has been a study on how to evaluate the Intel IXP2400 Network [42]. Inspired by this, I will examine how many instruction cycles and how much I/O latency each microblock has. This gives an overview on how long a packet takes (in time) to travel from the receiving interface to the transmit interface.

At the Packet RX microblock, the smallest IP packet which can arrive is 31 bytes. This includes a 2 Byte PPP header, a 20 Byte IP Header, a 4 Byte UDP header, a 4 Byte PPP trailer, and a payload of 1 byte. This packet size is the minimum POS packet which the application should handle. The packet size can be even smaller than 31 bytes, using compressed headers. Thorough, in the line card by Ericsson (ET-FE4), compressed headers are not supported. Two reasons to not use compressed headers are: It takes more power to compress and uncompress packets, and with the low line rate used in Cello system, it is not necessary to use compressed headers, it still has time to process larger headers.

The used line rate is OC-3, 155.520 Mbps (payload rate of 150.336 Mbps) [53]. In this case, there are two OC-3 links connected to the POS framer. The POS framer supports dual HDLC extraction, therefore the line rate to the POS interface should have a line rate of 2x OC-3 line rate. To get a clear overview of how often a packet arrives, a calculation of arrival time is needed. The formula for this arrival time is stated below:

Arrival Time for one packet = Packet Size / Data Rate

The packet size is in bytes, if we multiply it with eight, the packet size is then counted in bits instead of bytes. Data Rate shows how many data bits (Mbps) arriving to a certain device. The arrival time is the time (μs) to receive one packet on the line interface, or the time between to packets arriving to line interface. Assume a packet size of 31 bytes. Multiply it with 8, and then divide it with the actual payload line rate 2*150.336 Mbps. Now we have a packet arrival time of 0.825μs. This time can be converted into processor clocks using following formula:

Microengine clock cycles to arrive one packet = Arrival Time for one packet (ns) / microengine cycle time (ns)

The Intel IXP2400 processor with eight microengines runs at a frequency of 600 MHz (i.e. one processor cycle time is 1.67 ns). The formula above enables us to calculate the available number of Microengine cycles between arriving packets. The arrival time (ns) is divided with the clock cycle time (ns). This gives us a minimum arrival cycle time of 495 cycles between two minimal POS packets. To keep up with the line rate, every processing stage of the pipeline must complete all its required processing for a given packet within this cycle budget.

The MSF runs at a different data rate, specifically at an OC-12 line rate of 622.08 Mbps (this corresponds to a payload rate of 601.344 Mbps) both on the receive and transmitting sides of the interface. We assume one ATM cell has a fixed size of 52 Bytes. This analysis does not include the HEC byte in the ATM header shown earlier, in section 2.4.1, as we assume that the ATM framer in MSF adds this byte, rather than requiring us to compute it in our receiving/transmitting ATM microblock. Using 52-byte cells, we get an arrival time of one ATM cell every 692 ns or 415 microengine cycles.

Reducing data line rate or increasing the packet size increases the available compute cycles per Microengine. In other words, increasing the data rate reduces the available cycles per Microengine. Figure 21 shows how different data rates and packet sizes effects the available Microengine cycles per Microengine.

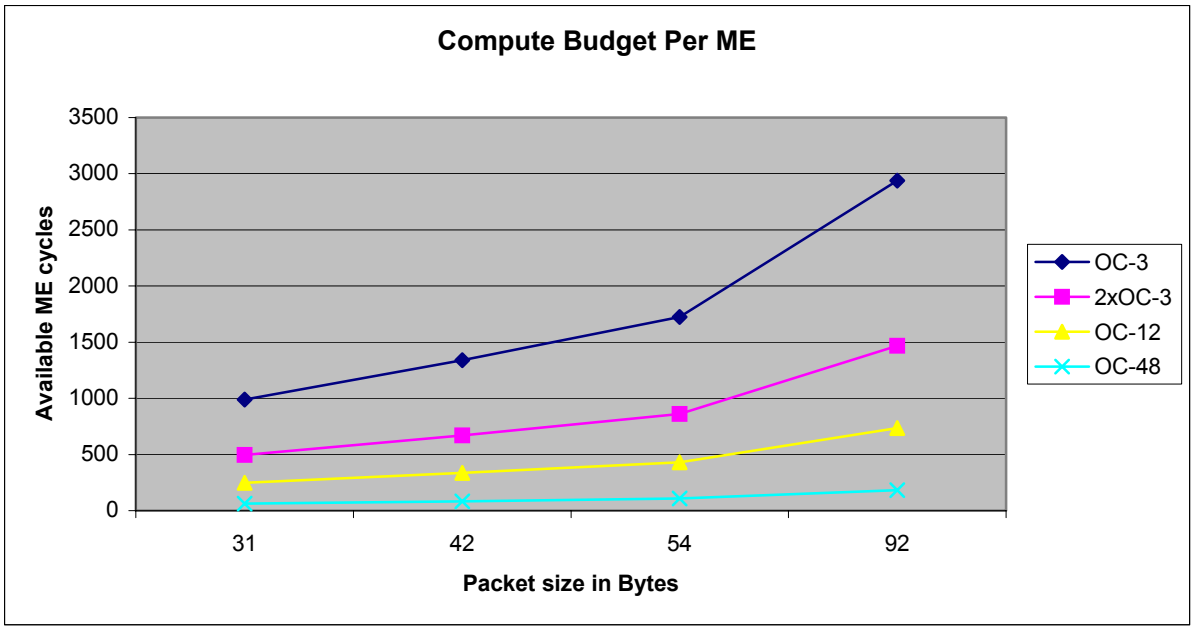

Figure 21. Compute Budget per Microengine

As IXP 2400 has eight Microengines, the total available compute cycles for the entire application is eight times the total available budget per Microengine. In other words, the total available compute cycles (budget) for handling minimal POS packets on the ingress side is 8*495 or 3960 cycles. On the egress side, where it interfaces with AAL5 cells of 52 bytes, the total available compute cycles budget are 415*8 or 3320 cycles.

Packet processing involves accesses to internal and external memories such as scratch memory, SRAM, DRAM, etc. Typically, the latency for accessing SRAM memory is 125-150 cycles, while DRAM latency is about 250-300 cycles.

The software-controlled multi-threaded features on the Microengine provide the mechanism to hide this high latency. Our ingress application with a per stage budget of 3960 cycles,

would allow up to 23 SRAM operations or around 14 DRAM operations in one single pipe stage while still keeping up with minimal packets at the specified line rate.

5.1.1 Performance budget for microblocks

POS Rx

The first block in the ingress flow is POS Rx block. This microblock runs only on one microengine and it is responsible for reassembly of POS packets in variable length packet sizes. The packets are divided in fixed size packets called mpackets for MSF to generate and transfer it into to a receive buffer, RBUF. The mpackets can have a size of 64, 128, and 256 bytes. The POS Rx block uses all eight treads on the Microengine, and each tread reads one mpacket from RBUF at a time. Each thread of the microengine checks the Start Of Packet (SOP) and End Of Packet (EOP) bits of the mpacket, identifies the port where the packet came from, allocates a DRAM buffer, moves the data from RBUF to the DRAM buffer, signals the next stage when it handles the mpacket which has the EOP set. This procedure is the same for all incoming packets.

This pipestage requires four I/O operations:

- 1 SRAM read to allocate a new buffer to store buffer handles
- 1 SRAM write to update the packet descriptor information
- 1 DRAM write to move the packet from receive buffer to DRAM
- 1 Scratch memory write, used to signal next pipestage (microblock) when the entire packet has been reassembled

With a pseudo-code analyse, it requires approximately 70 instruction cycles and four I/O operations to reassemble one minimum POS packet. Since each Microengine has a cycle budget of 495 cycles for handling a POS minimum packet, the reassembly function meets the cycle count budget for one Microengine. If we look at the total I/O latency for the reassembly block, and uses eight threads in a context pipestage. Then, the total available I/O budget is 3960 cycles, where we have multiplied eight threads with the budget of one Microengine (495). Now, assume the latency of one SRAM operation or one scratch operation is 125 cycles each, and one-DRAM operation is 250 cycles. This gives a total I/O latency of: 2*125+1*125+1*250=625 cycles. We see that the Microengine executes this Microblock fits within the 3960-cycle available budget.

IPv4 forwarding

This is the second microblock in the ingress application. It is a microblock group consists of the IPv4 forwarder microblock and PPP decapsulate microblock running on a single microengine. The IPv4 forwarding microblock is responsible to first receive the incoming POS packets and decapsulate/classify the PPP header and then validate/update the IPv4 header so it can be sent further on the pipeline.

Via a pseudo-code analysis, the worst-case estimated instruction cycle count for the IPv4 forwarding Microblock is 250 cycles where we have assumed 110 cycles for the Longest Prefix Match (LPM) algorithm. This fits well within the cycle budget of 495 cycles. The pipestage requires these I/O operations:

- 3 to 5 SRAM accesses for IP header validation and route lookup
- 1 DRAM read, used to for reading packet header from DRAM
- 1 DRAM write, used to write packet header to DRAM

- 1 DRAM read to read next hop info from DRAM
- 1 Scratch memory write, used to signal next pipestage (microblock) when the entire packet has been reassembled
- 1 Scratch memory read, used to read from earlier pipestage

This analysis assumes a worst-case scenario, i.e. it takes five SRAM access for one route lookup and one IP header validation. Assuming an SRAM/Scratch memory latency of 125 cycles and a DRAM latency of 250 cycles, the total I/O latency cycle count for one IPv4 forwarding Microengine is: 5*125+3*250+2*125 = 1625 cycles.

The IPv4 forwarding microblock fits well within the 3960-cycle available budget.

Queue Manager (Cell and Packet based)

This Queue Manager (QM) is responsible to perform enqueue and dequeue operations on the transmit queues for all packets. The functionality is therefore identical on both ingress and egress application. It processes enqueue and dequeue requests from other pipestages (Microblocks), and performs necessary operations on the queue array structures. Worst-case estimated instruction cycle count for the Queue manager is 74 cycles for the cell based and 81 cycles for the Packet Queue Manager. The Cell Queue Manager with 74 cycles fits well within the ingress application budget of 495 cycles. The same is with the Packet Queue Manager, where it fits with 81 cycles compared to the cycle budget of 415 cycles on the egress application. Both Cell and Packet based Queue Manager requires the same I/O operations shown below:

- 2 SRAM accesses for Enqueue/Dequeue operation
- 2 SRAM accesses for read/write queue descriptor used between the Q-array and SRAM memory

Assuming an SRAM/Scratch memory latency of 125 cycles and a DRAM latency of 250 cycles, the total I/O latency cycle count for one Queue manager Microengine is: 4*125 =500 cycles. This fits within the 3960-cycle available budget.

Scheduler (Cell and Packet based)

The scheduler schedules packets to be transmitted to the MSF fabric. The scheduling algorithm implemented is Round Robin among the ports used. It can optionally use Weighted Round Robin among the queues on a port. The scheduling algorithms are presented earlier in section 2.5.2.

A pseudo-code analysis of the scheduler shows that this microblock is compute-intensive. The worst case for the cell based scheduler is when the scheduler has to process fabric flow control messages, process QM queue transition messages, check for MSF transmit count, and schedule an ATM cell at each minimum packet arrival slot (495 cycles).

The worst case for the packet based scheduler is when the scheduler must schedule a minimum POS packet, handle 1 enqueue transition message, and handle a dequeue response at each minimum packet arrival slot (415 cycles).

The total estimated instructions are for the worst case 107 cycles for cell based and 87 cycles for the packet based scheduler.

The I/O operations are negligible for the scheduler and therefore this microblock is implemented as a context pipe-stage executing on only one microengine with only four threads. First thread is used for scheduling algorithm and the other tree handles the fabric flow control messages, QM transition messages, and the transmit MSF counter.

AAL5 Tx

The AAL5 Tx is implemented as a two-stage functional pipeline. Each stage has one critical section. The two stages interact on the same Microengine. The existing code is optimised to run on more than one Microengine. Therefore, this code should be modified for better performances if it needs one Microengine. This pipestage can be analysed in either a worst case or an average case. The average case appears at least in 97% of all cases. Here the average case is when a cell lies completely in one buffer. The total estimated instruction cycles in the average case are 287 cycles and for the worst case is 340 cycles. Both cases fit well in the cycle budget of 495 cycles for one Microengine.

This pipestage requires these I/O operations:

- 11 SRAM accesses
- 2 DRAM accesses
- 3 Scratch memory accesses

Assuming an SRAM/Scratch memory latency of 125 cycles and a DRAM latency of 250 cycles, the total I/O latency cycle count for one AAL5 Tx Microengine is:
11*125+3*125+2*250=2250 cycles

This fits within the 3960-cycle available budget.

AAL5 Rx

The AAL5 Rx block runs on one microengine. A pseudo-code analysis of the AAL5 Rx microblock shows that the microblock is divided in 4 different phases. This phases executes have different estimated instruction cycles depending on which type of mpacket being processed. These mpackets are one four types: SOP, MOP, EOP, or EOP&SOP in same mpacket. The total estimated instructions vary between 154-203 cycles depending which of these four mpackets it is. The worst case is then when we have an EOP with 203 instruction cycles. Since each ME has a cycle budget of 415 cycles for handling an ATM cell of 52 bytes, the AAL5 frame reassembly function meets the cycle count budget for one Microengine.

This pipestage requires these I/O operations:

- 1-2 SRAM accesses for VC pointer lookup
- 2 SRAM accesses for get/evict RXC from SRAM
- 2 SRAM accesses for prefetch/drop buffer handle
- 1 SRAM write used for writing meta data to SRAM
- 1 DRAM write used to write from RBUF to DRAM
- 1 Scratch memory write, used to signal next pipestage (microblock) when the entire packet has been reassembled

If we look at the total I/O latency for the AAL5 frame reassembly block, we first assume that eight threads are used in the Microengine. The total available I/O budget is then 8*415, 3320 cycles.

Assume an SRAM/Scratch memory latency of 125 cycles and a DRAM latency of 250 cycles. Then if we assume the worst case with long lookup of VPI we have a total I/O operation latency of: 7*125+1*125+1*250=1250 cycles

This fits well within the 3320-cycle available budget.

PPP encapsulation

This microblock code is divided in 4 phases where the total worst case estimated instructions are 83 cycles. Here is the header unaligned, and if we assume that the header is at a known compile time offset, the microblock can be substantially simplified. Worst case of 83 cycles fits well within the ME budget of 415 cycles.

The PPP encapsulation microblock pipestage requires six I/O operations:

- 1 SRAM read to read SOP meta data
- 1 SRAM write to update SOP meta data
- 1 DRAM read of SOP payload offset
- 1 DRAM write to put PPP header before payload in DRAM
- 1 Scratch memory read for reading from earlier microblock
- 1 Scratch memory write, used to signal next pipestage (microblock) when the entire packet has been reassembled

Assume an SRAM/Scratch memory latency of 125 cycles and a DRAM latency of 250 cycles. The total I/O operation cycle count for the PPP encapsulation Microengine is: 2*125+2*125+2*250=1000 cycles and this fits well within the 3320-cycle available budget.

POS Tx

The microblock is predefined to use the interface with 16 multi physical ports (MPHY-16). The microcode is divided in three phases where the total worst case estimated instructions of 163 cycles. Since each Microengine has a cycle budget of 415 cycles for handling an ATM cell of 52 bytes, the Packet Tx function meets the cycle count budget for one Microengine. The Packet Tx pipestage requires six I/O operations:

- 2 SRAM accesses, reads SOP meta data and next buffer meta data
- 2 DRAM read
- 2 Scratch memory read, reading for transmit request and next available TBUF element

Assume an SRAM/Scratch memory latency of 125 cycles and a DRAM latency of 250 cycles. The total I/O operation latency for the POS Tx Microengine is:

2*125+2*125+2*250=1000 cycles

This fits within the total 3320-cycle available budget.

5.1.2 Performance Budget summary

As we have studied both compute budget and I/O budget on all the microblocks used in both ingress and egress application, we can see that there is no problem with lack of budget. This shows that both applications have no problem to execute the necessary code on each microengine without have increased queues or buffers.

5.2 Performance of Ingress and Egress application

The following section describes the measurement made of the ingress and egress applications using statistics gathered by the development workbench.

5.2.1 Ingress application

This test was run on the ingress application implemented on a single Intel IXP2400 chip (See Appendix E for configuration). In this test, the packet generator generates 200 POS packets using one of five different packet sizes. A packet of each size is generated sequentially starting with the lowest packet size first and when the largest packet has been generated, the packet generator starts all over again with the smallest frame size. The frame sizes are 31 bytes (which includes a 2 byte PPP header, 4 byte PPP trailer, 25 byte IP packet), 42 bytes, 54 bytes, and 92 bytes.

The ingress application reassembles these frames, decapsulate the PPP header, classifies and modifies, segments the IP packets into ATM cells, and finally the transmit microblock send these cells out on the media interface. If we receive out 200 POS packets, 50 packets of each sizes stated above, then the transmit block should send out 300 cells. This estimated count for cells are assumed from what the different packet sizes generates in cells. For example, a 31-byte packet is included in one cell and a 54-byte packet is included in two cells.

The ingress application runs on 2*OC-3 line rate, where the payload line rate is 301 Mbps [53]. In the test, the achieved line rate reached 309 Mbps, and the throughput where 438 Mbps on the output port. The line rate decreased as the time exceeded, and therefore it should reach 301 Mbps later on. The receiving buffer (RBUF) used 20-25% of the capacity while the transmit buffer (TBUF) only used 0-5 %. The higher throughput can be explained with the overhead problematic. If we have a small packet (31 bytes), we still need to create an ATM cell of 48 bytes, which gives a large overhead. The line rate is based on how many bits of data that comes to an interface. If the AAL5 Tx creates larger packets than initiated, then the line rate should be higher. In Figure 22, we can see the low execution of the first Microengine, only executing 12% of the time. This can be improved by optimisation, such as only executing some threads, or by putting it together with another microblock with low execution requirements. Note that in Microengine 4, the scheduler is executing with only four threads.

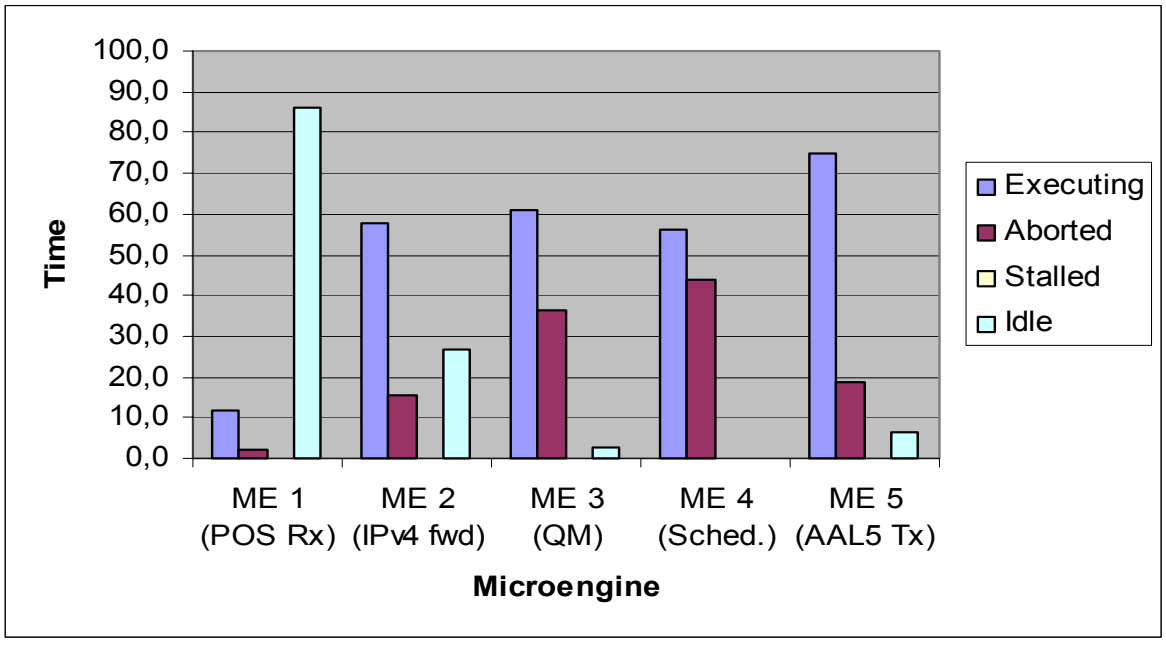

Figure 22. Execution statistics on the Ingress application

It took 182469 cycles to send out 300 ATM cells. To fetch one 31-byte POS packet from RBUF, process it, and send it out to the TBUF on MSF takes around 7252 cycles. In Table 3, we see the worst case of consumed resources per thread on each Microengine. The scheduler Microengine acts different from the others, where it executes different codes on different

threads. Still the worst case is showed. We can se that there are no bottlenecks on consuming resources.

Table 3. Consumed resources for the ingress application

Item	ME 1 (POS Rx)	ME 2 (IPv4)	ME 3 (QM)	ME 4 (Scheduler)	ME 5 (AAL5 Tx)
GPR (Allowed: 32)	24	21	16	10	29
SRAM (Allowed: 16)	5	8	6	4	16
DRAM (Allowed: 16)	0	10	0	0	16
Signals (Allowed: 15)	7	5	8	4	13
Instructions (Allowed: 4096)	769	632	272	546	1157
Headroom (Instructions in Control Store)	3327 (81.2%)	3464 (81.2%)	3824 (93.4%)	3550 (86.7%)	2939 (71.8%)

5.2.2 Egress application

This test was run on the egress application implemented on a single Intel IXP2400 chip (See Appendix F for configuration). In this test, the packet generator generates AAL5 frames using one of three different PDU sizes. A packet of each size is generated sequentially starting with the lowest frame size first and when the largest frame has been generated, the packet generator its starts all over again with the smallest frame size. The frame sizes are 48 bytes (which includes a 31-byte POS packet), 48 (a 42-POS packet), 96 bytes (a 54-byte POS packet), and 144 bytes (a 96-byte POS packet). These frames are reassembled into IP packets in the receiving microblock. It then encapsulates a PPP header over the IP packet and sends out these packets (POS packets) on the media bus interface. When the egress application receives a frame of cells, one POS packet is included in this frame, which can be more than one ATM cell, if the packet size is larger than 40 bytes.

The egress application runs on OC-12 line rate, where the payload line rate should be 601 Mbps. In the test, the achieved line rate reached 601 Mbps, and the throughput where 301 Mbps on the output port. The receiving buffer used only 15-20% of the memory while the transmit buffer where only used 45-50% of its capacity.

Figure 23 shows how much each Microengine executes in the application. We can see that the first Microengine, that executes the AAL5 Rx microblock, is only executing 32% of the time. This can be improved by optimisation, such as only executing some threads, or by putting it together with another microblock with low execution requirements. Note that in Microengine 3 the scheduler is executing with only two threads.

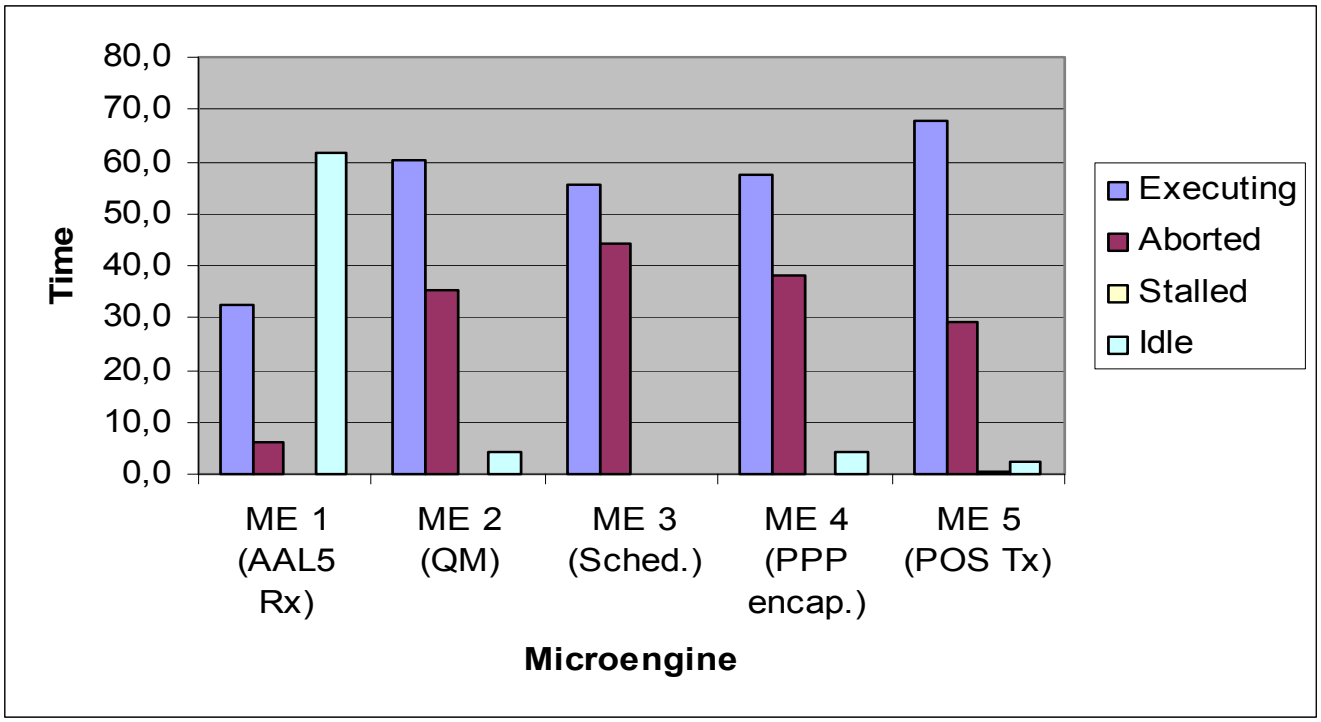

Figure 23. Execution statistics on the Egress application

It took 159069 cycles to send out 200 POS packets. To fetch one AAL5 frame (include only one 48-byte ATM cell) from RBUF, process it, and send it out to the TBUF on MSF takes around 6815 cycles. In Table 4, we see the resource located for the egress application. The scheduler acts different from the others, where it executes different codes on different threads. Still the worst case is showed. We can se that there are no bottlenecks on consuming resources.

Table 4. Consumed resources for the egress application

Item	ME 1 (AAL5 Rx)	ME 2 (QM)	ME 3 (Scheduler)	ME 4 (PPP Enc.)	ME 5 (POS Tx)
GPR (Allowed: 32)	23	15	9	20	28
SRAM (Allowed: 16)	10	7	4	8	7
DRAM (Allowed: 16)	13	0	0	6	4
Signals (Allowed: 15)	8	8	5	10	12
Instructions (Allowed: 4096)	1036	293	439	248	1145
Headroom (Instructions in Control Store)	3060 (74.7%)	3803 (92.8%)	3657 (89.3%)	3848 (93.9%)	2951 (72.0%)

5.2.3 SRAM and DRAM bus

In this section, we look at the SRAM and DRAM data bus for both the ingress and egress application. To connect to these memories, IXP2400 uses a single DRAM bus and two SRAM buses. The DRAM bus read/write operations are generated by Microengines, XScale Core, and PCI bus. The DRAM bus runs on 150 MHz and is 128-bit wide divided into a 64-bit push and pull bus. The DRAM bus runs at 150 MHz and is 128-bits wide divided into a 64-bit push and 64-bit pull bus. The two SRAM buses have read/write operations generated by Microengines, XScale Core, and PCI bus. The two SRAM buses runs at 250 MHz and are each 64-bits wide -- divided into a 32-bit push and 32-bit pull bus.

As we can see in Table 5 and Table 6 below, both the push and pull bus on the DRAM bus were idle most of the time (>94%). In the remaining time, data was transferred from/to the receive/transmit FIFO buffers. For the two SRAM buses, there are almost the same results as for DRAM, except for one SRAM push bus executing in 21 percentage of time. In summary, neither the SRAM nor the DRAM bus seems to be a bottleneck for the IXP2400.

Table 5. Memory utilization for ingress

Bus	In use (%)	Idle (%)
DRAM Push Bus	6.0	94.0
DRAM Pull Bus	3.9	96.1
SRAM Push Bus I	21.1	78.9
SRAM Push Bus II	3.3	96.7
SRAM Pull Bus I	6.0	94.0
SRAM Pull Bus II	6.0	94.0

Table 6. Memory utilization for egress

Bus	In use (%)	Idle (%)
DRAM Push Bus	2.5	97.5
DRAM Pull Bus	4.1	95.9
SRAM Push Bus I	8.9	91.1
SRAM Push Bus II	23.2	76.8
SRAM Pull Bus I	1.4	98.6
SRAM Pull Bus II	3.3	96.7

5.2.4 Summary of the Performance on Ingress and Egress application

For the ingress application, there is no bottleneck on the resources and it executes the code easily on the Microengines. To optimise the ingress application, it can be possible to put together the POS Rx microblock and Cell Scheduler microblock to run on only one Microengine. The POS Rx microblock only executes in 12 % of the time, and the Scheduler only uses four threads. Therefore, the POS Rx microblock could run on 4 threads and still have time process a packet in time.
The egress application did not have any bottleneck on resources and it executed all the code easily on the Microengines. Here, the AAL5 Rx microblock only executed in 32% of the time, and it would be possible to integrate with the packet scheduler on the same

Microengine. The scheduler only executes on two threads, and therefore the AAL5 Rx could then execute on six threads and still reach the wire speed requirements.

5.3 C-code against microcode

This section compares C-code and hand-written microcode. To compile C-code, Intel has developed a C-compiler called, Microengine C-compiler. The purpose of these tests is to investigate how effectively the compiler generates code as compared to hand-written microcode. Intel has promised a C-compiler, which generates code with a size penalty only 10% greater than hand-written (and optimised) microcode.

There are three tests, one small program, which tests a scratch ring. The second program is a microblock used in ingress application earlier, Cell based scheduler. The final test is on an application provided by Intel on the pre-release 5, which is the latest pre-release before this project ends. The application differs from the other two tests, where it is running on a higher line rate, OC-48 instead of OC-3.

5.3.1 Compiler test on a Scratch ring

The application uses two Microengines executing only on one thread each. It utilizes a scratch ring to store data between the Microengines, see Figure 24. The scratch ring has a size of 128 bytes (allocated from scratch memory). The first Microengine is responsible for producing data and placing it into the scratch ring, and the second Microengine is responsible for consuming the data. Both Microengines are first initialised and then each executes a while loop. In this loop the producing Microengine checks if the ring is full, if so it signals the consuming Microengine to start consuming the data and waits for a signal from the consuming Microengine when it is finished consuming. If the ring is not full, it puts 4-bytes of data to the scratch ring and repeats the loop. The consuming Microengine initially waits for a signal from the producing Microengine, when the signal comes, it checks if the scratch ring is empty or not. If it is empty, it signals the consuming Microengine to say that it is finished (consuming). Otherwise, it consumes 4-bytes of data and repeats the loop.

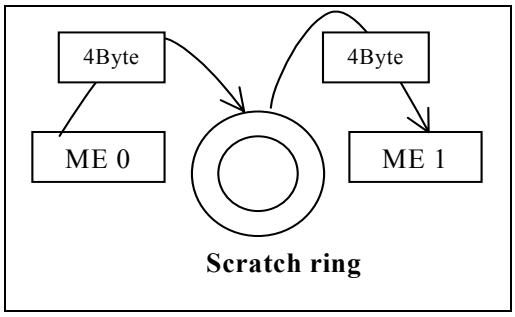

Figure 24. Consumer and producer

There are four tests, one using Microcode, and three using C-code. The C-code differs from each other in optimisation. The three different C-code optimisations are: Non-optimised, Size-optimised, and speed-optimised. All the results from the four tests are shown in Table 15, Table 16, Table 17, and Table 18 in Appendix B.
In Figure 25, we summarise all statistics gathered, to compare the C-compiled code with the handwritten code. For the producer, the C-code executes in both instructions and cycles around 175-220 percentages more than the handwritten code. In addition, we see a strange behaviour for the consumer, where the C-code executes faster than the microcode. This behaviour is explained with a not-optimised microcode. The critical loop of the microcode consists of two branch instructions and one NOP instruction. The C-code only consist of one

branch instruction, the microcode version should then be fairly optimised so it then can be compared with the C-code.

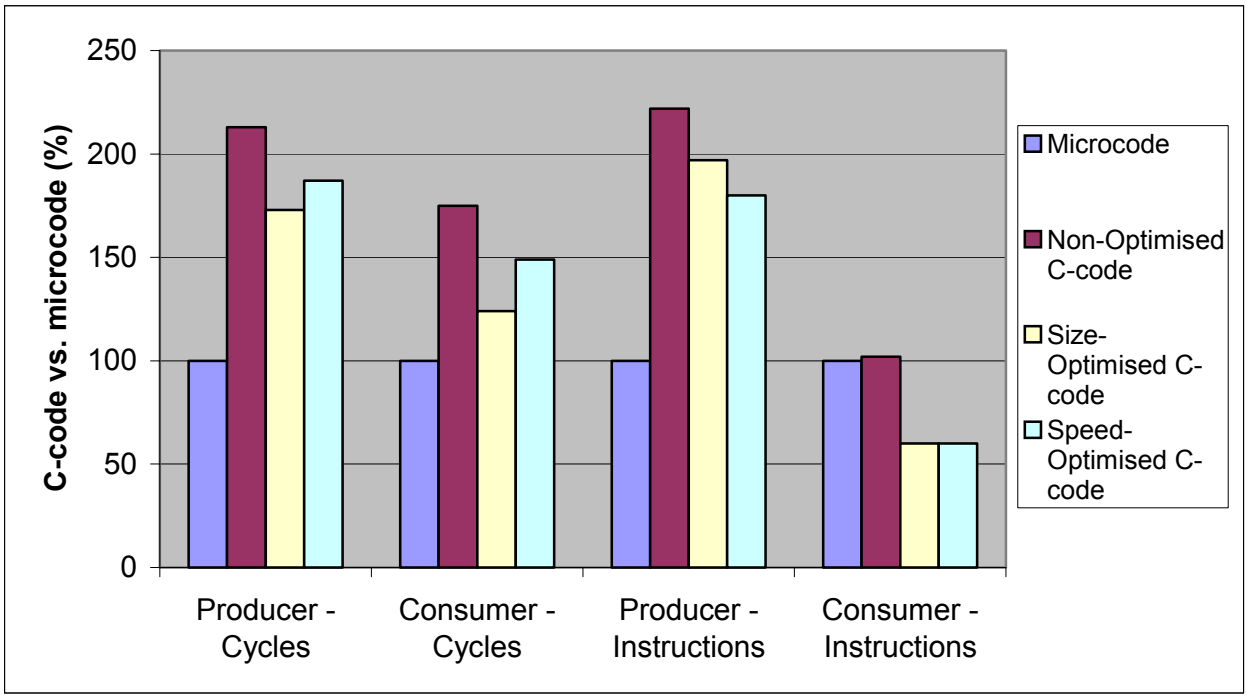

Figure 25. Comparison between microcode and C-code on Scratch ring example

5.3.2 Compiler test on the cell based Scheduler

This test is based on the microblock described earlier in 4.2.1.

The scheduler consists of four threads: A scheduler thread, a QM message handler thread, a flow control thread, and packets in flight handler thread. These threads are tested one at a time, both looking at the initialisation phase, and the main critical phase. All critical phases are mainly built up by one while-loop executing instruction specific on each thread.

The first test is in microcode, where we can use the whole ingress application to test on. Under testing, the microblock was integrated with the rest of the ingress application. This gives a more realistic test than the first test above. The c-code written microblock was successfully integrated with the micro-code handwritten ingress application.

There are four tests, one using Microcode, and three using C-code. The C-code differs from each other in optimisation. The three different C-code optimisations are: Non-optimised, Size-optimised, and speed-optimised. All the results from the four tests are shown in Table 20, Table 21, Table 22, and Table 23 in Appendix C.

In Figure 26 below, we summarise all statistics gathered, to compare the C-compiled code with the handwritten code. Here we see that there is no comparison on the Flow Control thread. The microcode had no flow control implemented yet, and therefore it was no use to compare between the existing C-code. In addition, the “packets in flight thread” shows good results, it executes 114 % (cycles) over the microcode, and equals instructions compared to the microcode. However, this thread execute to few instructions, to really say something about the compilers performance.

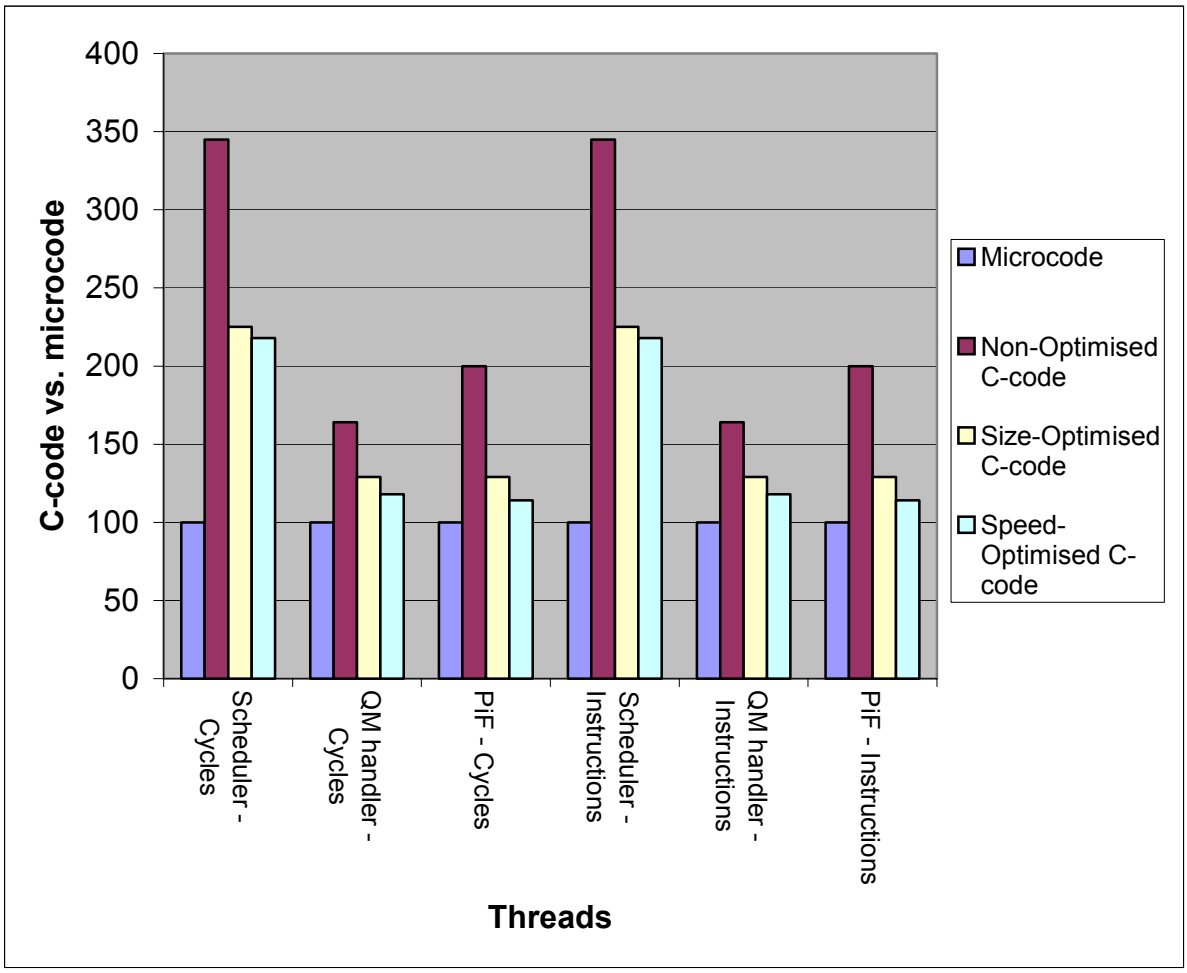

Figure 26. Comparison between microcode and C-code on Cell Scheduler microblock

5.3.3 Compiler test on the OC-48 POS ingress application

This is a test between an application written in both handwritten microcode and C-code. These two applications run on OC-48 (2.5 Gbps) line rate and both applications are included in the pre-release 5 of IXA 3.0. The application forwards IP packets from a POS interface to a CSIX interface. Both applications are similar to the implemented ingress application described earlier in 4.2.1. There are only two main differences between the applications. It is a higher line rate (OC-48 instead of OC-3), and the transmit microblock has been changed from AAL5 Tx to CSIX TX microblock.

Since CSIX has a fixed-length header larger than the POS packets, it produces an overhead. If we send out 500 46-byte POS packets, looking at the log files, the applications send out CSIX frames with a size of 68 bytes. To achieve wire speed without increasing the receive buffers, the Media bus needs to have a throughput of 68/46*2.5Gbps = 3,70Gbps

There are three tests, one using Microcode, and three using C-code. The C-code differs from each other in optimisation. The two different C-code optimisations are: Size-optimised, and speed-optimised. All the results from the four tests are shown in Table 25, Table 26 and Table 27 in Appendix D.

In Table 7 below, we summarise all statistics gathered, to compare the C-compiled code with the handwritten code. Here we see that the throughput is lower than expected, we can also se this by examining the receive buffers and se that they are full most of the time. The microcode has about 20-25 % less of executed cycles. This is a result better than the other Compiler tests described earlier. Still it needs more optimising to lower the C-code to achieve

to have only 10 % over the microcodes execution in cycles. We can also see that the throughput is not as high as expected. In microcode, the throughput was 3241 Mbps, where it should reach 3700 Mbps without affecting the buffers to be overloaded. In the microcode test, it did not exceed the buffer limit; however it did exceed in the other two tests. Note from the other tests, where we also tested on non-optimised C-code. In this test, the non-optimised code had errors in the compilation, and therefore the test was skipped.

Table 7. Comparison between microcode & C-code for the OC-48 POS ingress application

Phase	Microcode	Size-Optimised C-code	Speed-Optimised C-code
Process one packet (Cycles)	3983	4873	4902
Process one packet (%)	100	122	123
Send out 500 CSIX frames (Cycles)	60797	72658	75034
Send out 500 CSIX frames (%)	100	120	123
Throughput (Mbps)	3241	2676	2577

5.4 Memory configuration test

This section tests how a different speed of DRAM and SRAM memory affects the applications performance. The SRAM memory can run at either at 100, 150, or 200 MHz. DRAM runs at either 100 or 150 MHz. The Intel IXP 2400 processor has only a single DRAM-channel, which can control four banks. By interleaving the DRAM banks, we can improve bandwidth utilization and increase concurrency. The application we use here is the ingress application used in section 5.2.1.

5.4.1 DRAM test

Here we changed the clock rate from 150 MHz (default) to 100 MHz. The application now sends out 300 cells in 182698 cycles. If we compare it with the earlier test (See section 5.2.1) where we used 150 MHz DRAM bus, the results are the same in execution time in cycles. The result shows that the DRAM memory is not a bottleneck for this implementation.

5.4.2 SRAM test

Here we change the clock rate on both SRAM controllers from 250 MHz (default) to 100 MHz. The application now sends out 300 cells in 183520 cycles. It takes a little more execution time to send all cells, but the overall performance is not affected.

5.5 Functionality test on IPv4 forwarding microblock

This section describes a test being made on the IPv4 microblock. This test shows if the IPv4 microblock functionality works as expected, a test has been made stated in Appendix H. The test first looks at the IP header validity checks based on RFC 1812 MUST&SHOULD statements. The purpose of the test is to determine if the microblock drops, forwards, or sets exceptions on packets correctly. The second test is of the PPP header classifier. This test

determines if the microblock drops, forwards, or set exception on packets correctly based on the protocol.

The result of the test was successfully, all the tests acted as expected. This means that the IPv4 microblock have the functionality as proposed. The functionality includes classifying incoming packets according to RFC 1812 [19] and classifies packets based on PPP header stated in Appendix A.

5.6 Loop back: Connecting ingress and egress

In the existing release of SDK, there is no hardware support for loop back. When I tried to connect two Network Processor chips together, it did not work at all. It did not copy data from the first chip's TBUF to the second chip's RBUF. Intel knows the problem, and it should be fixed to the final release.

This causes some problems in evaluating both transmit and receive applications on the same chip. As described earlier, one of the main goals of this project was to try to implement all the specified functionality using only one network processor chip. To test this without loop back functionality seems difficult. In order to implement this in software I needed to modify the receiving blocks on the ingress and egress side, i.e., AAL5 Rx and POS Rx. Both of these blocks have eight contexts (threads) to handle cells or packets from one device. This could be changed to have four threads listening to one device and the other four threads listening to another device. Now the MSF can receive different kinds of packets from different devices. For example, the MSF is able to receive AAL5 cells on one port and POS packets on another port. Each of the receiving microblocks listens to a specific port in order to receive the right traffic, i.e., AAL5 Rx listens to the port of the MSF that receives ALL5 cell traffic.

Notice, the above changes are only made when we want to loop back traffic from ingress to egress on one chip. In this project, I was not successful in meeting the functional requirements while using only a single chip, i.e., eight microblocks. Thus, an ET implementation would need to run on two chips instead of one, with each application, ingress and egress, running on a single chip. When each is running in a separate chip they can easily be connected to each other, by connecting the ingress transmit side to the egress receive side, and thus loop back is straightforward. The MSF simply copies data from the ingress chip transmit buffer to the receive buffer of the egress chip.

5.7 Assumptions, dependencies, and changes

5.7.1 POS Rx

The POS Rx block needs to maintain the order of the arriving packets. The buffer handle for packets enqueue in order on the scratch ring, this requires that threads in this microblock execute in order. The POS Rx block will send all non-PPP control block to the XScale core.

The RBUF element size is selected to be either 128 or 256 bytes.

For Start of Packet (SOP) buffers, the POS Rx block will start to write on DRAM with an offset of 128 bytes to allow blocks later in the pipeline to add and remove headers before the existing buffer. Non-SOP buffers are written with an offset of 0.

The POS Rx block will compute and setup the cell count for every buffer. The cell-size is set to be 48 bytes (default).

5.7.2 IPv4 forwarding block

This IPv4 forwarding block consists of two microblocks (IPv4 forwarder and PPP decapsulate) running on one single Microengine. Normally, running on higher line rate such as OC-48, this Microblock needs faster processing. All the applications provided by Intel runs this microblock on four Microengines.

The threads executing the microblock do not handle packets in strictly in order. The packets of different sizes may be processed at different times, and it is up to the dispatch loop to maintain strict ordering of the packets. This problem also occurs for the exception packets, when they are sent to XScale processor.

When a packet is dropped, packet content will not be logged as recommended by [RFC1812]. This is not practical is such a high rates.

The IPv4 forwarding Microblock does not support multicasting, and the TOS field in the IPv4 header is not used and is therefore left unmodified.

Change of code

The PPP classify macro has been changed, so it sets LCP, IPCP, IPv6, and IPv6CP packets to be exception packets.

5.7.3 Cell Queue Manager

Some of scheduling algorithms used by the scheduler microengine might need the Queue Manager to send the packet size to the scheduler along with the queue transition message. This microblock does not support it.

Change of code

I have changed the number of queues per queue group from 32 to 4 and the number of queue groups from 32 to 4. This is made only for faster compilation/debugging.

5.7.4 Cell Scheduler

The Queue Manager sends queue transition messages to the scheduler via a Next Neighbour (NN) ring. The design currently assumes that the queue and port data structures used for scheduling are cached in local memory in the scheduler. It can supports up to 64 ports and 16 classes per port.

In this release, the flow control messages are not supported. This implies that the flow control handler thread is not running, as it should.

Change of code

I have changed the number of queues per queue group from 32 to 4 and the number of queue groups from 32 to 4. This is made only for faster compilation/debugging.

5.7.5 AAL5 Tx

The AAL5 Tx assumes a cell-transmit request for packets on the same queue are not interleaved, i.e. all the cell transmit requests for a packet appear contiguously before a request is received to transmit a cell from a different packet.

If a transmit request for the last cell of a packet has more than 40 bytes of payload to be transmitted, it creates an extra cell to with the necessary CPCS trailer. The AAL5 Tx sends out 52-byte cells via a TBUF-element. The ATM framer (hardware) adds the HEC byte in the cell header.

5.7.6 AAL5 Rx

The AAL5 Rx assumes that received cells belong to an AAL5 frame in order from the MSF and that it receives a 52-byte framer, i.e. it assumes that the framer strips off the HEC header byte. This implies that no byte-alignment is required to process the ATM cell.

The RBUF element is implemented using 64 bytes, but it supports up to 128 bytes. The first buffer of every packet, the AAL5 Rx allocates some headroom to allow microblocks later in the pipeline to add or remove headers. The AAL5 Rx block will compute and set the cell count for every buffer depending on the transmit media.

5.7.7 Packet Queue Manager

The Packet QM sends a response message to the scheduler for every dequeue request.

Change of code

I have changed the number of queues per queue group from 32 to 4 and the number of queue groups from 32 to 4. This is made only for faster compilation/debugging.

5.7.8 Packet Scheduler

The design currently assumes that the queue and port data structures used for scheduling are cached in local memory in the scheduler.

Change of code

I have changed the number of queues per queue group from 32 to 4 and the number of queue groups from 32 to 4. This is made only for faster compilation/debugging.

6 Conclusions

6.1 Meeting our goals

The main goal for this project was to implement and evaluate an application covering functionality based on an already existing ET board used in Ericsson's Cello system. This goal was unfortunately not fully achieved. One of the goals was to achieve all the ET board functionalities using only a single Network Processor (Intel IXP2400) chip.

During the project, it was difficult to fit in all the microblocks used into only a single chip, therefore two solutions were proposed, a single-chip solution and a two-chip solution. The single-chip solution I proposed was more difficult to implement than I expected. Implementing an application without queues on the ingress side was almost impossible, unless major changes were to be made to existing modules. Moreover, on the egress side, it was difficult to get the necessary (and promised) microblocks from Intel (For example: WRED, Metering, etc.). These blocks are needed to implement the desired QoS functionality. Near the end of this project, I finally got these microblocks (written in C-code) from Intel. However, they involved much more code than was expected, and this made it impossible to fit them into my one single-chip solution. Therefore, I used the second solution, in which I implemented two applications, with separate ingress and egress applications.

The evaluation of the ingress and egress applications showed no major bottlenecks. Both applications performed wire-speed processing on the incoming packets/cells. However, some optimisation could be done on both ingress and egress side. For example, the receiving microblock at ingress side (POS Rx) only executed at 12% of the time. This is not efficient, we could instead execute the same code on only 1-2 threads instead of all 8, and use the other threads for executing another microblock.

Distributing the microcode is very important, as it can both increase the performance and at the same time save space, thus hopefully reduce the number of Microengines which are necessary.

6.2 How to choose a Network Processor

When network equipment vendors select a network processor, they make a significant commitment to use it for years to come. It is important to ensure that the network processor environment selected is flexible and can be scaled to protect the vendor's investment. Software reusability and tools included for developing should be a key consideration when selecting a network processor.

Network processors do not yet offer any simple formula for determining whether using them will lower costs and increase flexibility or not [11].

Another key concern is product availability; routing equipment can have long lifetimes and router vendors sometimes need assurance that a product will still be available for 4 years or more. In addition to benchmarking performance, there is a wide variety of other NPU evaluation criteria.

- For example, does a network processor support coprocessors and switch fabric interfaces?
- Does it comply with emerging or current network bus interfaces or memory interfaces?

- Does it support a well-known high-level language such as C++?
- How much power does it consume, and how much does it cost in volume?

In conclusion, network processors have yet to deliver on their promised hype. Despite the wide variety of NPU evaluation criteria, benchmark results can still emerge as the best way to evaluate and choose a network processor.

6.3 Suggestions & Lessons Learned

During this project, I have discovered several disadvantages with the workbench, which made a realistic and fast implementation almost impossible. My hope was to be able to add/remove/modify microblocks as I wanted, in order to fulfil my requirements as specified. The microblocks would be integrated into a full-duplex application on a single-chip solution. This failed for the reasons mentioned earlier, and this made it more difficult to test my applications in a real environment. The existing workbench was only a pre-release, and therefore not all the promised functionality was included. For example, the loopback interface functionality is not yet supported. The loopback should connect two chips together for copying data from one chip's transmit side (TBUF) to another chip's receive side (RBUF).

To start working with a project on Intel's IXP2400, it is a good idea to first sit down with the workbench as early as possible. Try using the existing applications that comes along with the workbench to gain intuition about both the network processor and the workbench.

Another approach to think about is to determine all the blocks needed, what new microblocks need to be implemented, cycle budget for the whole application, incoming and outgoing line rates, memory consumption, etc. This will give a better overall view of what resources are needed and if it is possible to fit the requirements into a single-chip solution or not.

Good things about the workbench of Intel IXA SDK 3.0 pre-release 4:

- It is pretty straight forward to organise code for each Microengine
- The project organisation is simple and good. You can organise all your project files in specific folders, just as you want.
- The simulation environment has a lot of functionality to look at. For example, you can see statistics about how much each Microengine executes and you can see how long a specific instruction executes or how much occurs during a specified number of cycles. This provides good opportunities for the programmer to test and evaluate their code.

Bad things about the workbench of Intel IXA SDK 3.0 pre-release 4:

- It is difficult using only a single packet to simulate how long it will take a packet to get through the whole application. The solution is to create a stream of packets, where we only use one packet in the stream. Even so, it is hard to see how packets travel through the microengines. Either you need to set breakpoints in the code to follow the packet through the application, or you need to stop the simulation when a certain device port receives a cell/cells or once a device sends out a certain amount of cells.
- Currently you can only generate POS IP, Ethernet, PPP, and AAL5 ATM cells. However, Intel has promised to support AAL2 cells in their final release.
- Poor flexibility, you have to trust that the existing implementation of the function library provided in the SDK is optimal. For example, some of the existing microblock written in microcode are not optimised (Look at scheduler).

- The debug window is hard to work in. It can be hard to follow the cursor between all context swaps. For example, it would have been better if it could always follow the cursor through all thread on a Microengine, instead of only showing one thread's cursor.
- The pre-release 4 had a menu for loop back. When I tried, it did not work at all. It did not copy data from one chip's TBUF to the other chip's RBUF. Intel knows the problem, and it should be fixed to the final release.
- There are some problems with comments in script files. When using "/*" and "*/" it reports errors (This is now fixed in pre-release 5).

7 Future Work

This project did not fulfil all the requirements necessary to cover all the ET-FE4 functionalities. However, some modifications can easily be done with a later release of the Software Development Toolkit. I believe that it is best to wait for Intel's final release of the toolkit, so it has all the necessary support included.

Two major changes of the existing solution should be considered. First, is to change the egress microblocks used for queuing and scheduling. These microblocks should be changed to a microblock group that covers all the QoS functionality such as Diffserv. Intel has promised these microblocks would be included in the pre-release 5. Second, is to remove the queuing and scheduling microblocks on the egress application. They are not necessary where we are using so low line rate for processing packets. This saves two Microengines that could be useful to other functionalities.

This project has mainly used the pre-release (i.e., version 4). Intel plans to release the final workbench sometimes in the beginning of 2003. Hopefully, this final release should solve some of the problems which occurred in this thesis, such as providing a loopback interface and the missing microblocks necessary for QoS.

A very interesting and important issue is to investigate how to distribute code over the Microengines. Today it is up to the programmer to determine which microblock to assign to which Microengine. This can be inefficient, for example in both application used on this thesis, the receive blocks only executes 25% of the time. An overall application would be much simpler to program if the programmer need not bother with how to distribute the code over the Microengines. Intel plans in the future to release some programming tools for more efficient programming, where it allows the programmer to only program one code, where a processor then distribute it efficiently over the Microengines.

It might also be a good idea to look at the generic problem of how to benchmarking Network Processors. Today it is hard to compare two Network Processors from different vendors, as they each have their own specific test equipment to gather statistics on their processor. Many processors do not actually measure exactly the same things even if they claim to. Furthermore, processing packets such as in the case of IPv4 forwarding can be done in several ways, and therefore it can be measured in different ways. This require careful consideration to be made when comparing two (or more) Network Processors.

Glossary

AAL	ATM Adaptation Layer
AAL5	ATM Adaptation Layer type 5
API	Application Program Interface
ASIC	Application Specific Integrated Circuit
ATM	Asynchronous Transfer Mode
BOOTP	Bootstrap Protocol
CAM	Content Addressable Memory
CBQ	Class Based Queuing
CLP	Cell Loss Priority
CPU	Central Processing Unit
CS	Convergence Sublayer
CSIX	Common Switch Interface for Fabric Independence and Scalable Switching
DBM	Device Board Module
DSCP	Differentiated Services Code Point
DRAM	Dynamic Random Access Memory
FCFS	First Come First Serve
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GFC	Generic Flow Control
GPP	General Purpose Processor
HDLC	High-Level Data Link Control
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPCP	Internet Protocol Control Protocol
LCP	Link Control Protocol
LPM	Longest Prefix Match
ME	Microengine
MSF	Media Switch Fabric
OC	Optical Carrier
PB	Processor Board
PCI	Peripheral Component Interface
PDU	Protocol Data Unit
POS	Packet over SONET
PPP	Point-to-Point Protocol
QOS	Quality of Service
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
SAI	Switch Access Interface

SACI	Switch Access Configuration Interface
SAR	Segmentation and Reassembly
SDH	Synchronous Digital Hierarchy
SDRAM	Synchronous Dynamic Random Access Memory
SONET	Synchronous Optical Network
SOP	Start of Packet (Start Of Payload)
SRAM	Static Random Access Memory
STM	Synchronous Transfer Mode
TOS	Type of Service
VCI	Virtual Channel Identifier
VPI	Virtual Path Identifier
WFQ	Weighted Fair Queuing
WRED	Weighted Random Early Detection
WRR	Weighted Round Robin

References

Books

[1]. James Carlsson, *PPP Design and Debugging*, Addison-Wesley, Massachusetts, 1999, ISBN 0-201-18539-3

[2]. W. Richard Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley, Massachusetts, 1994, ISBN 0-201-63346-9

[3]. W. Richard Stevens, *TCP/IP Illustrated, Volume 2: The Implementations*, Addison-Wesley, Massachusetts, 1995, ISBN 0-201-63354-X

[4]. William Stallings, *Data &Computer Communications*, Prentice Hall, Upper Saddle River; New Jersey, 2000, 6th edition, ISBN 0-13-084370-9

White papers & Reports

[5]. Johan Börje and Hans-Åke Lund, ”Real-time routers for wireless networks”, Ericsson Review no.4, 1999

[6]. Tomas Henriksson, "Hardware Architecture for Protocol Processing", Licentiate Degree Thesis No. 911, Computer Engineering, University of Linköping, December 2001, ISBN: 91-7373-209-5,

[7]. Björn Liljeqvist, “Visions and Fact – A survey of Network Processors”, Electronic Engineering, Chalmers University of Technology, 2002

[8]. David Husak and Robert Gohn, “Network Processor Programming Models: The Key to Achieving Faster Time-to-Market and Extending Product Life”, White Paper, Motorola Corp. May 4, 2000.

[9]. Paul Chow, “Reduced Instruction Set Computers”, IEEE Potentials, pages 28-31, October 1991

[10]. Louis E. Frenzel, “Network Processors Evolve To Meet Future Line Speeds”, Report in Electronic Design, September 17, 2001

[11]. R. Munoz, “ASICs vs. Net Processors: Understanding the True Costs”, Report in Communication Systems Design, CMP Media LLC, April 30, 2002.

[12]. Mel Tsai, “Network Processor Benchmarks”, Version 1.01, U.C Berkeley University, California, May 2002

RFC

All RFC documents can be found at the IETF website: http://www.ietf.org/rfc

[13]. J. Postel, RFC 0791 - *Internet Protocol*, September 1981

[14]. G. McGregor, RFC 1332 - *The PPP Internet Protocol Control Protocol (IPCP)*, 1992

[15]. V. Fuller, T. Li, J. Yu, and K. Varadhan, RFC 1519 - *Classless Inter-Domain Routing (CIDR) An Address Assignment and Aggregation Strategy*, September 1993

[16]. W. Simpson, RFC 1570 - *PPP LCP Extensions,* January 1994

[17]. W. Simpson, RFC 1661 - *The Point-to-Point Protocol (PPP),* July 1994

[18]. W. Simpson, RFC 1662 - *PPP in HDLC-like Framing*, July 1994

[19]. F. Baker, RFC 1812 - *Requirements for IP Version 4 Routers*, June 1995

[20]. S. Deering and R. Hinden, RFC 1883 - *Internet Protocol, Version 6 (IPv6) Specification*, December 1995

[21]. W. Simpson, RFC 1994 - *PPP Challenge Handshake Authentication Protocol (CHAP)*, August 1996

[22]. Y. Bernet, S. Blake, D. Grossman, and A. Smith, RFC 3290 - *An Informal Management Model for Diffserv Routers*, May 2002

Conference and Workshop Proceedings

[23]. A. T. Campbell, S. Chou, M. E. Kounavis, V. D. Stachtos, and J. Vicente, "NetBind: A Binding Tool for Constructing Data Paths in Network Processor-based Routers", *Fifth International Conference on Open Architectures and Network Programming (OPENARCH' 02)*, New York, NY, June 2002.

Other

[24]. Motorola Corp., "C-5 Network Processor Data Sheet", January 2002

[25]. Motorola Corp., "C-5 Network Architecture Guide", May 31, 2001

[26]. Motorola Corp., " Packet Over SONET Switch Application Guide", Application guide, Revision 01, June 2002

[27]. Motorola Corp., "C-Ware Software Toolset Fact Sheet", Product brief, October, 2001

[28]. IBM, "IBM PowerPC 403GCX 32-Bit RISC Embedded Controller Datasheet", January 8, 1999

[29]. PMC Sierra, "PM5351 S/UNI-Tetra Saturn User Network Interface (155-TETRA) Data Sheet", February 2002

[30]. Intel, "Intel IXP2400 Network Processor Advance Information Datasheet", Product brief, March, 2002

[31]. Intel, "Intel Internet Exchange Architecture Software Developers Kit 3.0", Product brief, March, 2002

[32]. Teja Technologies, "Performance of the Teja NP IPv4 Forwarding Foundation Application", Version 2, 24 June 2002

[33]. Agere System, "Network Processor Product Brief", Product brief, July 2002

[34]. IXP lab, Transis Laboratory at the Computer Science Department of Hebrew University of Jerusalem.

[35]. Vox Technologies Corp., "Alcatel 7420 Edge Service Router (ESR) datasheet", May, 2001.

[36]. SiGaAs Systems, "FPGA for system on chip", Homepage describing FPGA, August 2002

[37]. The ATM Forum, "UTOPIA Specification Level 1, Version 2.01", March 21 1994

[38]. NP Forum, "CSIX-L1: Common Switch Interface Specification-L1", Specification version 1, May 8 2000

[39]. Niraj Shah, "Understanding Network Processors"; version 1.0, University of California, Berkeley, September 2001

[40]. C Daniel O'Neill, "The Case for a Single Processor", Project Report, Stanford University, June 2002

[41]. Alcatel, "Network Processors: The Optimal Building Block for Next Generation IP Routers", Technical Paper, November 2000

[42]. Intel, "Intel Technology Journal – Network Processors", Issue 03, Volume 06, August 15 2002, ISSN: 1535766X

[43]. Intel, "Intel IXA SDK ACE Programming Framework", Rev. 3.3, August 2001, Document number: A71582-001

[44]. Peter N. Glaskowsky, "Intel Beefs Up Networking Line", Cahners Microprocessor, March 18 2002

[45]. Music Semiconductors, "What is CAM (Content Addressable Memory)? ", Rev. 2a, September 30 1998

[46]. Network Processing Forum, "Framework for Benchmarking Network Processors", Rev. 1.0, 2002

[47]. Transaction Processing Performance Council (TPC), "TPC Benchmark C", Standard Specification, Rev 5.0, February 26 2001

[48]. Transaction Processing Performance Council (TPC), "TPC Benchmark H", Standard Specification, Rev 1.5.0, July 12 2002

[49]. Transaction Processing Performance Council (TPC), "TPC Benchmark W", Standard Specification, Rev 1.7, October 11 2001

Internet Related Links

[50]. Network Processing Forum (NPF), http://www.npforum.org/

[51]. Standard Performance Evaluation Corporation (SPEC), http://www.specbench.org/

[52]. Asynchronous Transfer Mode Forum (ATM Forum), http://www.atmforum.org

[53]. Techfest, http://www.techfest.com/networking/wan/sonet.htm

Appendix A – Requirements for the ET-FE4 implementation

This appendix summarizes which protocols that are going to be supported and what the functionalities they will have.

PPP (Supported)

It *MUST* support these protocols: IPv4, IPv6, IPCP, IPv6CP, LCP…

All protocols except IPv4 are handled as exceptions.

The LCP Packet format and the Codes below *may* be supported:

Table 8. LCP Packet Types

Code	Packet Type
1	Configure-Request
2	Configure-Ack
3	Configure-Nak
4	Configure-Reject
5	Terminate-Request
6	Terminate-Ack
7	Code-Reject
8	Protocol-Reject
9	Echo-Request
10	Echo-Reply
11	Discard-Request

LCP extensions (Optional)

The additional LCP Configuration options listed below *may* be supported:

Table 9. LCP extensions

Type	Configuration Option
1	MRU
2	Magic Number
3	Protocol field compression
4	FCS Alternatives

PPP in HDLC-like framing

On the link layer, it MUST support Octet-synchronous mode. In the HDLC frame, the address field *must* be FF (hex) and the control field *must* be 03 (hex).

- "Octet-stuffed framing" *may* be supported in the implementation with these features:
- Invalid frames, see section 4.3 in RFC 1662 [18]
- Transparency, an octet based control escape is defined
- Flag sequence 0x7E

IPCP (XScale, exception packet processing)

This configuration option *may* be supported:

Table 10. IPCP configuration options

Type	Configuration Option
2	IP-Compression-Protocol [RFC1332]

These Codes *may* be supported in the implementation:

Table 11. IPCP Packet Types

Code	Packet Type
0	Vendor Specific
1	Configure-Request
2	Configure-Ack
3	Configure-Nak
4	Configure-Reject
5	Terminate-Request
6	Terminate-Ack
7	Code-Reject

Ipv6CP (Optional)

These Configuration options *may* be supported:

Table 12. IPv6CP configuration options

Type	Configuration Option
1	Interface identifier
2	IPv6-Compression Protocol

These Codes *may* be supported in the implementation:

Table 13. IPv6CP Packet Types

Code	Packet Type
1	Configure-Request
2	Configure-Ack
3	Configure-Nak
4	Configure-Reject
5	Terminate-Request
6	Terminate-Ack
7	Code-Reject

IPv4 (Supported)

Internet Protocol must support IP, UDP and ICMP in the protocol field of the IP header. The decimal numbers are 17 (UDP), 1 (ICMP) and 4 (IP).

- These IP functionalities include:
- TOS such as Diffserv (**Not used in this release of IXP**)
- Unused IP header bits
- Unrecognised Header Option
- Fragmentation and Reassembly (Done by the XScale Core)
- Time to Live (TTL)
- IP Broadcast addressing
- Subnetting

ICMP (XScale core, exception packet)

These Types of messages *may* be supported in the implementation:

Table 14. ICMP Messages

Type	Messages
0	Echo Reply
3	Destination unreachable, only code field 0-5 and 13 are supported
4	Source Quench
5	Redirect
8	Echo
11	Time exceeded
12	Parameter problem
13	Timestamp
14	Timestamp Reply
15	Information Request
16	Information Reply
17	Address Mask Request
18	Address Mask Reply

Ipv6 (Optional)

This is only going to be implemented if time is left.

The maximum packet size *should* roughly be 2k bytes.

Protocols includes:

- Internet Control Protocol version 6 (58)
- Transport layer protocol (UDP)

Options that *may* be implemented:

- Ipv6 Router alert Option
- Definition of the Differentiated Services Field in IP headers
- Neighbour Discovery

Ipv6 Extension Headers (Optional)

It support following headers:

- Hop-By-Hop headers
- Routing Header
- Fragment Header
- Destination Option Header
- No next header

BOOTP (Optional)

May be fully supported except two restrictions:

- No BOOTP header length check
- Default value of hops threshold is 4

Appendix B – Compiler test on a Scratch ring

This section is a test of Microengine C compiler on a simple scratch ring application. Each test are summarised in one table showing the executed instructions/cycles on either the initialisation or critical phase.

Microcode

Table 15. Microcode test of Scratch ring example

Phase	Producer (ME 0)	Consumer (ME 1)
Initialisation (cycles)	14 cycles	2 cycles
Initialisation (instructions)	11	2
Producer/Consumer Critical phase (cycles)	263 cycles (7+8*32)	263 cycles (7+8*32)
Producer/Consumer Critical phase (instructions)	164 instructions (4+5*32)	164 instructions (4+5*32)

The assembler generated 21 instructions for the producer and 14 words for the consumer. It takes 11971 cycles to fill up one scratch ring and then consume it.

Non-optimised C-code

Table 16. Non-optimised C-code test of Scratch ring example

Phase	Producer (ME 0)	Consumer (ME 1)
Initialisation (cycles)	55 cycles	9 cycles
Initialisation (instructions)	39	9
Producer/Consumer Critical phase (cycles)	561 cycles (17+17*32)	460 cycles (12+14*32)
Producer/Consumer Critical phase (instructions)	364 instructions (12+11*32)	167 instructions (7+5*32)

The assembler generated 132 instructions for the producer and 51 words for the consumer. It takes 16964 cycles to fill up one scratch ring and then consume it.

Size optimised C-code

Table 17. Size optimised C-code test of Scratch ring example

Phase	Producer (ME 0)	Consumer (ME 1)
Initialisation (cycles)	22 cycles	11 cycles
Initialisation (instructions)	18	10
Producer/Consumer Critical phase (cycles)	456 cycles (8+14*32)	326 cycles (6+10*32)
Producer/Consumer Critical phase (instructions)	323 instructions (3+10*32)	99 instructions (3+3*32)

The assembler generated 53 instructions for the producer and 38 words for the consumer. It takes 16894 cycles to fill up one scratch ring and then consume it.

Speed optimised C-code

Table 18. Speed optimised C-code test of Scratch ring

Phase	Producer (ME 0)	Consumer (ME 1)
Initialisation (cycles)	22 cycles	11 cycles
Initialisation (instructions)	18	10
Producer/Consumer Critical phase (cycles)	493 cycles (13+15*32)	392 cycles (8+12*32)
Producer/Consumer Critical phase (instructions)	296 instructions (8+9*32)	99 instructions (3+3*32)

The assembler generated 57 instructions for the producer and 38 words for the consumer. . It takes 16208 cycles to fill up one scratch ring and then consume it.

Summary on scratch ring test

Table 19. Comparison between C-code and Microcode on Scratch Ring test

Phase	Microcode	Non-Optimised C-code	Size-Optimised C-code	Speed-Optimised C-code
Initialisation Producer /Consumer (Cycles)	Producer: 100% Consumer: 100%	Producer: 393% Consumer: 450%	Producer: 157% Consumer: 550%	Producer: 157% Consumer: 550%
Initialisation Producer /Consumer (Instructions)	Producer: 100% Consumer: 100%	Producer: 355% Consumer: 450%	Producer: 164% Consumer: 88%	Producer: 163% Consumer: 500%
Critical phase Producer/Consumer (Cycles)	Producer: 100% Consumer: 100%	Producer: 213% Consumer: 175%	Producer: 173% Consumer: 124%	Producer: 187% Consumer: 149%
Critical phase Producer/Consumer (Instructions)	Producer: 100% Consumer: 100%	Producer: 222% Consumer: 102%	Producer: 197% Consumer: 60%	Producer: 180% Consumer: 60%

Appendix C – Compiler test on Cell Scheduler

This section is a test of Microengine C compiler on a cell based Scheduler. Each test are summarised in one table showing the executed instructions/cycles on either the initialisation or critical phase.

Microcode

Table 20. Microcode test on Cell Scheduler

Phase	Scheduler Thread	QM Message handler Thread	Flow Control Thread	Packets in Flight Handler Thread
Initialisation (cycles)	141	4	10	7
Initialisation (instructions)	110	3	6	6
Critical phase (cycles)	44	28	-	7
Critical phase (instructions)	36	18	-	6

The assembler generated 546 instructions for the Scheduler. The total cycle time for the ingress application to send out 100 cells is 67537 cycles.

Non-optimised C-code

Table 21. Non-optimised C-code test on Cell Scheduler

Phase	Scheduler Thread	QM Message handler Thread	Flow Control Thread	Packets in Flight Handler Thread
Initialisation (cycles)	1357	23	22	27
Initialisation (instructions)	1092	15	14	15
Critical phase (cycles)	152	46	48	14
Critical phase (instructions)	98	34	36	9

The assembler generated 432 instructions for the Scheduler. The total cycle time for the ingress application to send out 100 cells is 68909 cycles.

Size optimised C-code

Table 22. Size optimised C-code test on Cell Scheduler

Phase	Scheduler Thread	QM Message handler Thread	Flow Control Thread	Packets in Flight Handler Thread
Initialisation (cycles)	1092	22	22	22
Initialisation (instructions)	849	15	15	15
Critical phase (cycles)	99	36	32	9
Critical phase (instructions)	74	25	26	6

The assembler generated 305 instructions for the Scheduler. The total cycle time for the ingress application to send out 100 cells is 74261 cycles.

Speed optimised C-code

Table 23. Speed optimised C-code test on Cell Scheduler

Phase	Scheduler Thread	QM Message handler Thread	Flow Control Thread	Packets in Flight Handler Thread
Initialisation (cycles)	1092	22	22	22
Initialisation (instructions)	867	15	15	15
Critical phase (cycles)	96	33	27	8
Critical phase (instructions)	73	27	23	6

The assembler generated 309 instructions for the Scheduler. The total cycle time for the ingress application to send out 100 cells is 68161 cycles.

Summary on Cell Scheduler test

Table 24. Comparison between microcode & C-code for Cell Scheduler

Phase	Microcode	Non-Optimised C-code	Size-Optimised C-code	Speed-Optimised C-code
Initialisation (Cycles)	Sc.: 100% QM: 100% Pif: 100%	Sc.: 962% QM: 575% Pif: 386%	Sc.: 775% QM: 550% Pif: 314%	Sc.: 775% QM: 550% Pif: 314%
Initialisation (Instructions)	Sc.: 100% QM: 100% Pif: 100%	Sc.: 993% QM: 500% Pif: 250%	Sc.: 772% QM: 500% Pif: 250%	Sc.: 788% QM: 500% Pif: 250%
Critical phase (Cycles)	Sc.: 100% QM: 100% Pif: 100%	Sc.: 345% QM: 164% Pif: 200%	Sc.: 225% QM: 129% Pif: 129%	Sc.: 218% QM: 118% Pif: 114%
Critical phase (Instructions)	Sc.: 100% QM: 100% Pif: 100%	Sc.: 272% QM: 189% Pif: 150%	Sc.: 206% QM: 139% Pif: 100%	Sc.: 203% QM: 150% Pif: 100%

Appendix D – Compiler test on the OC-48 POS ingress application

This section is a test of Microengine C compiler on a simple scratch ring application. Each test are summarised in one table showing the executed instructions/cycles on either the initialisation or critical phase.

Microcode

The RX Buffer was almost full (96%) most of the time.

Table 25. Microcode test on the OC-48 POS ingress application

Microengine	Executing (%)	Aborted (%)	Stalled (%)	Idle (%)	Rate (Mbps)
1 (Packet Rx)	74.4	13.6	0.1	11.9	446.2
2 (IPv4 fwd)	64.2	17.0	0.0	18.8	385.3
3 (IPv4 fwd)	70.8	15.8	0.0	13.4	424.6
4 (QM)	64.2	28.2	0.0	7.7	385.0
5 (Scheduler)	69.5	30.5	0.0	0.0	417.0
6 (IPv4 fwd)	69.1	15.5	0.0	15.4	414.4
7 (IPv4 fwd)	64.5	14.8	0.0	20.6	387.2
8 (CSIX Tx)	75.1	13.8	0.0	11.0	450.9

The assembler generated 4216 instructions for the ingress application. It takes 3983 cycles to process only one packet. The total cycle time for the ingress application to receive 604 packets and send out 500 CSIX frames are 60797 cycles. Receive rate was 2464 Mbps and transmit rate was 3241 Mbps.

Size optimised C-code

The RX Buffer was full (100%) most of the time.

Table 26. Size optimised C-code test on the OC-48 POS ingress application

Microengine	Executing (%)	Aborted (%)	Stalled (%)	Idle (%)	Rate (Mbps)
1 (Packet Rx)	75.7	21.3	1.1	1.8	454.5
2 (IPv4 fwd)	65.0	14.8	0.0	20.1	390.3
3 (IPv4 fwd)	63.2	14.5	0.5	21.7	379.2
4 (QM)	68.7	22.5	0.0	8.8	412.2
5 (Scheduler)	71.2	28.8	0.0	0.0	426.9
6 (IPv4 fwd)	66.0	14.9	0.3	18.8	396.1
7 (IPv4 fwd)	69.8	15.4	0.0	14.8	419.0
8 (CSIX Tx)	79.9	12.8	0.0	7.3	479.4

The assembler generated 4072 instructions for the ingress application. It takes 4873 cycles to process only one packet. The total cycle time for the ingress application to receive 694 packets and send out 500 CSIX frames are 72658 cycles. Receive rate was 2335 Mbps and transmit rate was 2676 Mbps.

Speed optimised C-code

The RX Buffer was full (100%) most of the time.

Table 27. Speed optimised C-code test on the OC-48 POS ingress application

Microengine	Executing (%)	Aborted (%)	Stalled (%)	Idle (%)	Rate (Mbps)
1 (Packet Rx)	75.7	21.3	1.3	1.8	453.9
2 (IPv4 fwd)	63.5	14.6	0.2	21.6	381.2
3 (IPv4 fwd)	62.1	14.5	0.9	22.5	372.5
4 (QM)	68.2	23.4	0.0	8.5	408.9
5 (Scheduler)	71.0	29.0	0.0	0.0	426.1
6 (IPv4 fwd)	64.8	14.9	0.6	19.7	388.7
7 (IPv4 fwd)	68.5	15.4	0.1	15.9	411.3
8 (CSIX Tx)	79.2	13.0	0.0	7.9	475.1

The assembler generated 4088 instructions for the ingress application. It takes 4902 cycles to process only one packet. The total cycle time for the ingress application to receive 715 packets and send out 500 CSIX frames are 75034 cycles. Receive rate was 2322 Mbps and transmit rate was 2577 Mbps.

Appendix E – Configure the Ingress Application

The ingress application uses two devices:

- Receive device, SPHY 32-bit wide bus using POS3 at a line rate of 301 Mbps. The receiving buffer size is 256 bytes.
- Transmit device, SPHY 32-bit wide bus using UTOPIA level 2 at a line rate of 601 Mbps. The transmitting buffer is 4096 bytes.

The following directories are included in the Ingress application:

/building_blocks – Here are all the microblocks stored

/dispatch_loop – Here are the files for building up the dispatch loop stored. This directory also includes IPv4 microblock root file used. The files are used in an example application made by Intel.

/include – Here are hardware definition files stored.

/library – Here are library files for both data plane and microcode stored.

/list – Here is the list file created in the assembler process stored.

/log – Here are files for logging device ports on the MSF stored.

/scripts – Here are the script files for debugging used. In this case, it uses seven script files: *aal5_tx_init, dbcast_init, dl_system_h, pos_ipv4_system_setup, qm_init, rtm_init, rtm_routes.*

The *dl_system_h* file only includes definitions for the other script files and therefore it must be executed first. The main file (*pos_ipv4_system_setup*) is the file to execute all other script files and must be executed last.

/streams – Here are the packet generated files stored. The file *ppp_ip_42b_port_0* is used here. Note, that this file is modified from the original stream file used in an existing applications provided by Intel.

In the root directory, there are 5 files included: one output file (.uof), three project files, and also a file (*system_init*) used to define all scratch rings and microengines to use in the project.

Predefinitions for each Microengine on the ingress application

Microengine 0:0 – Packet (POS) Rx

SPHY_1_32,RFC_POS_COUNTERS,PPP_RECEIVE

Microengine 0:1 – IPv4 forwarding

MICROENGINE,MICROCODE,CHIP_VERSION=IXP2XXX,RFC1812_SHOULD,META_CACHE_SIZE=8,IPV4_START_ME,DL_NEXT_ME=0x01,IP_HDR_OFFSET=2,RFC2644_CHECKS,NEXTHOP_INFO_SRAM,DBCAST_TABLE_BLOCK_SIZE=8,PROCESS_CONTROL_BLOCK,PPP_RECEIVE,PPP_HDR_SIZE=2

Microengine 0:2 – Cell based QM

No predefinitions

Microengine 0:3 – Cell based Scheduler

No predefinitions

Microengine 1:0 – AAL5 Tx

IXP2400,FIRST_AAL5_TX_ME,NEXT_AAL5_TX_ME=0x10,COUNTERS

Microengine 1:1 – Not used

No predefinitions

Microengine 1:2 – Not used

No predefinitions

Microengine 1:3 – Not used

No predefinitions

Appendix F – Configure the Egress Application

The egress application uses two devices:

- Receive device, SPHY 32-bit wide bus using UTOPIA level 2 at a line rate of 601 Mbps. The receiving buffer size is 256 bytes.
- Transmit device, MPHY 16-bit wide bus using POS3 at a line rate of 301 Mbps. The transmitting buffer is 4096 bytes.

The following directories are included in the Egress application:

/building_blocks – Here are all the microblocks stored

/dispatch_loop – Here are the files for building up the dispatch loop stored.

/include – Here are hardware definition files stored.

/library – Here are library files for both data plane and microcode stored.

/list – Here is the list file created in the assembler process stored.

/log – Here are files for logging device ports on the MSF stored.

/scripts – Here are the script files for debugging used. In this case, it uses five script files: *aal5_rx_init, aal5_rx_hash_table_init, dl_system_h, pos_ipv4_system_setup, qm_init*.

The *dl_system_h* file only includes definitions for the other script files and therefore it must be executed first. The main file (*pos_ipv4_system_setup*) is the file to execute all other script files and must be executed last.

/streams – Here are the packet generated files stored. The file *min_pkt_2_cells* is used here. Note, that this file is modified from the original stream file used in an existing applications provided by Intel.

In the root directory, there are 5 files included: one output file (.uof), three project files, and also a file (*system_init*) used to define all scratch rings and microengines to use in the project.

Predefinitions for each Microengine on the egress application

Microengine 0:0 – Not used

No predefinitions

Microengine 0:1 – Not used

No predefinitions

Microengine 0:2 – Not used

No predefinitions

Microengine 0:3 – AAL5 Rx

ME_NUMBER=0x03,ME_BEGIN=0x03,ONE_ME_AAL5_RX,HASH_LOOKUP,IXP2400, AAL5_RX_COUNTERS,META_CACHE_SIZE=8

Microengine 1:0 – Packet based QM

No predefinitions

Microengine 1:1 – Packet based Scheduler

No predefinitions

Microengine 1:2 – PPP encapsulation

No predefinitions

Microengine 1:3 – Packet (POS) Tx

THIS_ME=MPHY16_PACKET_TX_FIRST_ME,SCHEDULER_ME=0x11

Appendix G – Stream files used in Ingress and Egress flow

Ingress application

```
# IXP1200 Developer Workbench Data Stream File
# Format Version 6.00
#********  Do not edit this file ***********

# Begin Data Stream ppp_ip_42b_port_0
STREAM_TYPE = POS IP
# Begin POS IP Frame
# Begin Ppp Header
16_BIT_PROTOCOL = TRUE
INCLUDE_ADDRESS = FALSE
INCLUDE_CONTROL = FALSE
FIXED_DATA = 0021
# End Ppp Header
# Begin Ip Packet
# Begin Ip Header
COMPUTE_CHECKSUM = TRUE
COMPUTE_PACKET_LENGTH = TRUE
FIXED_DATA = 450000190000000004069961188776381887763 8
# End Ip Header
# Begin Data Payload
USE_FILL_PATTERN = TRUE
FILL_PATTERN_TYPE = 13
FILL_START_VALUE = 0
DATA_PAYLOAD_SIZE = 5
DATA_PAYLOAD = 0001020304
# End Data Payload
# End Ip Packet
# Begin PPP Trailer
4_BYTE_CHECKSUM = TRUE
COMPUTE_CHECKSUM = TRUE
FIXED_DATA = f40ed4f9
# End PPP Trailer
# End POS IP Frame
# Begin POS IP Frame
# Begin Ppp Header
16_BIT_PROTOCOL = TRUE
INCLUDE_ADDRESS = FALSE
INCLUDE_CONTROL = FALSE
FIXED_DATA = 0021
# End Ppp Header
# Begin Ip Packet
# Begin Ip Header
COMPUTE_CHECKSUM = TRUE
COMPUTE_PACKET_LENGTH = TRUE
FIXED_DATA = 450000240000000004069956188776381887763 8
# End Ip Header
# Begin Data Payload
USE_FILL_PATTERN = TRUE
FILL_PATTERN_TYPE = 13
FILL_START_VALUE = 0
DATA_PAYLOAD_SIZE = 16
DATA_PAYLOAD = 000102030405060708090a0b0c0d0e0f
# End Data Payload
# End Ip Packet
# Begin PPP Trailer
4_BYTE_CHECKSUM = TRUE
COMPUTE_CHECKSUM = TRUE
FIXED_DATA = aea42256
```

```
# End PPP Trailer
# End POS IP Frame
# Begin POS IP Frame
# Begin Ppp Header
16_BIT_PROTOCOL = TRUE
INCLUDE_ADDRESS = FALSE
INCLUDE_CONTROL = FALSE
FIXED_DATA = 0021
# End Ppp Header
# Begin Ip Packet
# Begin Ip Header
COMPUTE_CHECKSUM = TRUE
COMPUTE_PACKET_LENGTH = TRUE
FIXED_DATA = 450000300000000000406994a1887763818877638
# End Ip Header
# Begin Data Payload
USE_FILL_PATTERN = TRUE
FILL_PATTERN_TYPE = 13
FILL_START_VALUE = 0
DATA_PAYLOAD_SIZE = 28
DATA_PAYLOAD = 000102030405060708090a0b0c0d0e0f101112131415161718191a1b
# End Data Payload
# End Ip Packet
# Begin PPP Trailer
4_BYTE_CHECKSUM = TRUE
COMPUTE_CHECKSUM = TRUE
FIXED_DATA = 53447803
# End PPP Trailer
# End POS IP Frame
# Begin POS IP Frame
# Begin Ppp Header
16_BIT_PROTOCOL = TRUE
INCLUDE_ADDRESS = FALSE
INCLUDE_CONTROL = FALSE
FIXED_DATA = 0021
# End Ppp Header
# Begin Ip Packet
# Begin Ip Header
COMPUTE_CHECKSUM = TRUE
COMPUTE_PACKET_LENGTH = TRUE
FIXED_DATA = 450000560000000000406992418877638818877638
# End Ip Header
# Begin Data Payload
USE_FILL_PATTERN = TRUE
FILL_PATTERN_TYPE = 13
FILL_START_VALUE = 0
DATA_PAYLOAD_SIZE = 66
DATA_PAYLOAD =
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f20212223242
5262728292a2b2c2d2e2f303132333435363738393a3b3c3d3e3f4041
# End Data Payload
# End Ip Packet
# Begin PPP Trailer
4_BYTE_CHECKSUM = TRUE
COMPUTE_CHECKSUM = TRUE
FIXED_DATA = eb3709b1
# End PPP Trailer
# End POS IP Frame
# End Data Stream
```

Egress application

```
# IXP1200 Developer Workbench Data Stream File
# Format Version 6.00
#********  Do not edit this file ***********

# Begin Data Stream min_pkt_2_cells
STREAM_TYPE = ATM AAL5
# Begin Atm Frame
# Begin Atm Header
AUTOMATIC_PTI = TRUE
FIXED_DATA = 00100010
# End Atm Header
# Begin AAL5 Trailer
FIXED_DATA = 6a6a6a6a6a6a6a6a6a6a6a0000001ddd01bb9c
# End AAL5 Trailer
# Begin CS-SDU Information Field
# Begin Ip Packet
# Begin Ip Header
COMPUTE_CHECKSUM = TRUE
COMPUTE_PACKET_LENGTH = TRUE
FIXED_DATA = 4500001d0000000004068bda0a00010120000001
# End Ip Header
# Begin Data Payload
DATA_PAYLOAD_SIZE = 9
DATA_PAYLOAD = 000102030405060708
# End Data Payload
# End Ip Packet
# End CS-SDU Information Field
# End Atm Frame
# Begin Atm Frame
# Begin Atm Header
AUTOMATIC_PTI = TRUE
FIXED_DATA = 00100010
# End Atm Header
# Begin AAL5 Trailer
FIXED_DATA = 00000028948f7178
# End AAL5 Trailer
# Begin CS-SDU Information Field
# Begin Ip Packet
# Begin Ip Header
COMPUTE_CHECKSUM = TRUE
COMPUTE_PACKET_LENGTH = TRUE
FIXED_DATA = 450000280000000004068acf0a00020120000001
# End Ip Header
# Begin Data Payload
DATA_PAYLOAD_SIZE = 20
DATA_PAYLOAD = 000102030405060708090a0b0c0d0e0f10111213
# End Data Payload
# End Ip Packet
# End CS-SDU Information Field
# End Atm Frame
# Begin Atm Frame
# Begin Atm Header
AUTOMATIC_PTI = TRUE
FIXED_DATA = 00200020
# End Atm Header
# Begin AAL5 Trailer
FIXED_DATA =
6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a000
000345660dd36
# End AAL5 Trailer
# Begin CS-SDU Information Field
```

```
# Begin Ip Packet
# Begin Ip Header
COMPUTE_CHECKSUM = TRUE
COMPUTE_PACKET_LENGTH = TRUE
FIXED_DATA = 4500003400000000040689c30a00030120000001
# End Ip Header
# Begin Data Payload
DATA_PAYLOAD_SIZE = 32
DATA_PAYLOAD =
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
# End Data Payload
# End Ip Packet
# End CS-SDU Information Field
# End Atm Frame
# Begin Atm Frame
# Begin Atm Header
AUTOMATIC_PTI = TRUE
FIXED_DATA = 00300030
# End Atm Header
# Begin AAL5 Trailer
FIXED_DATA =
6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6
a6a6a6a6a6a6a6a6a0000005a99ebaefa
# End AAL5 Trailer
# Begin CS-SDU Information Field
# Begin Ip Packet
# Begin Ip Header
COMPUTE_CHECKSUM = TRUE
COMPUTE_PACKET_LENGTH = TRUE
FIXED_DATA = 4500005a000000000406889d0a00040120000001
# End Ip Header
# Begin Data Payload
DATA_PAYLOAD_SIZE = 70
DATA_PAYLOAD =
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f20212223242
5262728292a2b2c2d2e2f303132333435363738393a3b3c3d3e3f404142434445
# End Data Payload
# End Ip Packet
# End CS-SDU Information Field
# End Atm Frame
# End Data Stream
```

Appendix H – Test specification on IPv4 microblock

Test on IPv4 forwarding mechanism

This test is to see if the IPv4 microblock processes the header validation correctly according to RFC 1812 [19] MUST & SHOULD statements. Eighteen cases are tested below, and in all cases, 10 IP packets of size 31 bytes (See first file in Appendix E for ingress part) have been changed to the specific incorrect value stated on each case. The IPv4 forwarding microblock has counters in SRAM to show if a packet are dropped, forwarded, set as exception, etc.

Case 1 – Packet size is less than 20 bytes

Change the packet size from 25 (Without PPP header and PPP trailer) to 1.

Result: Okay, the failure packets where marked as dropped.

Case 2 – Wrong number in version field

Change the packet version to 3.

Result: Okay, the failure packets where marked as dropped.

Case 3 – Packet with header length < 5

Change the packet header length to 3.

Result: Okay, the failure packets where marked as dropped.

Case 4 – Packet with header length > 5

Change the packet header length to 7.

Result: Okay, the failure packets where marked as exception.

Case 5 – Packet with total length < 20 bytes

This case checks the same as Case 1 above.

Result: Okay, the failure packets where marked as dropped.

Case 6 – Packet with invalid checksum

Change the calculated checksum to 0x1000

Result: Okay, the failure packets where marked as dropped.

Case 7 – Packet with destination address equal to 255.255.255.255

Change the destination address in the IP header to 255.255.255.255

Result: Okay, the failure packets where marked as exception.

Case 8 – Packet with expired TTL

Change the TTL in IP header from 4 to 1.

Result: Okay, the failure packets where marked as exception. Note that the packets forward counter is also incremented. TTL is checked after updating counter.

Case 9 – Packet length < total length field

Change the packet length to a bigger value than 25 bytes.

Result: Okay, the failure packets where marked as exception.

Case 10 – Packet with source address equal to 255.255.255.255

Change the source address in the IP header to 255.255.255.255

Result: Okay, the failure packets where marked as dropped.

Case 11 – Packet with source address equal to zero

Change the source address in the IP header to 0.0.0.0.

Result: Okay, the failure packets where marked as dropped.

Case 12 – Packet with source address of form {127, <any>}

Change the source address in the IP header to 127.x.x.x

Result: Okay, the failure packets where marked as dropped.

Case 13 – Packet with source address in Class E domain

Change the source address in the IP header to 240.x.x.x

Result: Okay, the failure packets where marked as dropped.

Case 14 – Packet with source address in Class D (multicast domain)

Change the source address in the IP header to 224.x.x.x

Result: Okay, the failure packets where marked as dropped.

Case 15 – Packet with destination address equal to zero

Change the destination address in the IP header to 0.0.0.0

Result: Okay, the failure packets where marked as dropped.

Case 16 – Packet with destination address of form, {127, <any>}

Change the destination address in the IP header to 127.x.x.x

Result: Okay, the failure packets where marked as dropped.

Case 17 – Packet with destination address in Class E domain

Change the destination address in the IP header to 240.x.x.x

Result: Okay, the failure packets where marked as dropped.

Case 18 – Packet with destination address in Class D (multicast domain)

Change the destination address in the IP header to 224.x.x.x

Result: Okay, the failure packets where marked as exceptions.

Test on PPP classify mechanism

This test is to see if the IPv4 microblock processes PPP header classifies correctly according to Appendix A. It should only support PPP protocol IPv4. Other protocols such as IPv6, IPCP, IPv6CP, and LCP are set as an exception packet. Unknown protocols should be dropped. To test this, generate six different equal-size packets: One IPv4, one IPv6, one LCP, one IPCP, one IPv6CP, and one LCP packet. Then run the packets sequent through the application. The IPv4 counters show if the packets are forward, dropped, or set as an exception.

The result shows that only the IPv4 packets are passed through the pipeline. The exception packets are dropped by the dispatch loop due to the exception handling is not supported now by the core.