INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

Generic Telecommunications Protocol Processor: A Programmable Architecture

by

Rawdon J. W. Taylor B. Sc.

A thesis submitted to the Faculty of Graduate Studies and Research in partial
fulfilment of the requirements for the degree of

Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering
Department of Electronics
Carleton University
Ottawa, Ontario, Canada

September 1999

**Copyright © 1999
Rawdon J. W. Taylor**

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-48462-9

Canada

The undersigned hereby recommend to the Faculty of
Graduate studies and Research acceptance of the thesis

Generic Telecommunications Protocol Processor: A Programmable Architecture

submitted by
Rawdon J. W. Taylor B. Sc.

in partial fulfilment of the requirements for the degree of
Master of Engineering

Thesis Supervisor

Chair, Department of Electronics

Carleton University
September 1999

Abstract

It has been noted that the majority of today's telecommunication ASIC's are performing some sort of protocol processing or conversion. It has been suggested that a generic, programmable, architecture could replace many of these ASICs to reduce costs, and shorten development cycles, reusing this generic processor instead of developing new ASICs.

This thesis describes a processing architecture which performs generic telecommunications protocol manipulation. Similar to a DSP processor, this architecture processes frame streams, providing programmable control for frame manipulation. This thesis offers a proof of concept for this architecture and the design for a prototype. By combining fast, dedicated hardware and slow programmable software, this processor is able to handle high-speed transmission links while supporting a wide set of protocols. For implementation purposes, the processor has been targeted for a transmission rate of 200Mbps. The design has been simulated to 50MHz in a 0.35 micron ASIC process.

Acknowledgements

I wish to acknowledge the help of my supervisor John Knight and Steve Cadieux (formally of Nortel Networks). Also, thanks to Yatish Kumar (also formally of Nortel Networks) for suggesting the idea for this thesis.

Table of Contents

List of Tables	vii
List of Figures	viii
List of Appendices	x
Acronymns and Abbreviations	xi
Chapter 1: Introduction	1
Chapter 2: Networks	5
2.1 OSI Network Model	5
2.1.1 Physical Layer.....	6
2.1.2 Data Link Layer.....	6
2.1.3 Network Layer.....	7
2.1.4 Transport Layer.....	7
2.1.5 Session, Presentation and Application Layers.....	8
2.2 Beyond the OSI Model	8
2.3 Conversion Network Scenarios	9
2.3.1 Bridge Configuration	9
2.3.2 Tunnel Configuration.....	10
Chapter 3: Protocols	11
3.1 What is in a Protocol?	12
3.1.1 Asynchronous Transfer Mode (ATM).....	13
3.1.2 Ethernet.....	16
3.1.3 Token Ring.....	17
3.1.4 HDLC.....	19
3.2 Protocols Supported in this Thesis	22
3.2.1 Unsupported Protocols.....	26
3.2.2 UTOPIA-I Interface.....	29
Chapter 4: System Integration	31
4.1 Scenario 1, 1:1	32
4.2 Scenario 2, 1:1 Embedded	32
4.3 Scenario 3, n:1	33
4.4 Scenario 4, 1:n	34
Chapter 5: Architecture	36
5.1 Architecture Overview	36
5.1.1 CPU Centric Design.....	36
5.1.2 Sub-Processor Functional Blocks Design.....	37

5.1.3 Programmable Pipelined Function Block Design	38
5.1.4 Processor Embedded Pipeline Design	39
5.1.5 Implemented Architecture	40
5.2 Conversion Example: HDLC to Ethernet	43
5.2.1 Input Frame Processing.....	44
5.2.2 Output Frame Processing.....	45
5.2.3 Processor Clock Rates.....	46
5.3 CPU	46
5.3.1 Internal Register Architecture.....	52
5.3.2 Interrupt Control	54
5.3.3 Arithmetic Logic Unit (ALU).....	56
5.4 HDLC Flag Protection	58
5.4.1 Unstuffer	59
5.4.2 Stuffer	60
5.5 CRC	61
5.5.1 CRC Theory	62
5.5.2 CRC Input	69
5.5.3 CRC Output	71
5.6 Header/Trailer	72
5.7 Buffers	73
5.7.1 Input Buffer.....	74
5.7.2 Output Buffer.....	77
5.7.3 Asynchronous Clock Domains Modifications	78
5.8 DMA Pipeline Controllers	81
5.8.1 Input DMA Controller	81
5.8.2 Output DMA Controller.....	83
5.9 I/O Interface	85
Chapter 6: Implementation Results	86
6.1 FPGA Results	86
6.2 ASIC Results	87
6.3 Verification of Architecture	88
Chapter 7: Conclusions	89
7.1 Processor Extensions and Future Development	91
Bibliography	94
Appendix A: CPU Instruction Set	97
A.1 Instruction Set:	97
A.1.1 Logical	97
A.1.2 Arithmetic	98
A.1.3 Interrupt.....	98
A.1.4 Execution Control	98
A.1.5 Other.....	99
A.2 Instruction Op-Codes	100
Appendix B: Example Frame Processing Programs	104
B.1 HDLC Frame to ATM (AAL-5) sample assembly code	104
B.2 HDLC to Ethernet sample assembly code	105

List of Tables

Table 1: OSI vs. Protocol Comparison	8
Table 2: ATM Adaptation Layer Modes	14
Table 3: UTOPIA Interface Bandwidth Options	30
Table 4: Pointer FIFO Depth Calculation Examples	77
Table 5: Condition Codes for Branch Instruction	98
Table 6: Register Selection Bits Mapping	100
Table 7: Memory Mode Bits Mapping	101
Table 8: Memory Offset Mode selection Bits	101
Table 9: Instruction Op-Codes	102
Table 10: Shift Instruction Selection Bit Mapping	103

List of Figures

Figure 2.0.1: Service Access Point Examples	5
Figure 2.1.1: OSI Network Model	6
Figure 2.3.1: Bridging Network Conversion Scenario	10
Figure 2.3.2: Tunneling Network Conversion Scenario	10
Figure 3.0.2: Protocol Family Tree	11
Figure 3.0.1: Common Frame Format	11
Figure 3.1.1: Basic ATM Cell Structure	15
Figure 3.1.2: Token Ring Network: Logical Ring Configuration	18
Figure 3.1.3: Token Ring Network: Token Passing Chain Operations	19
Figure 3.1.4: HDLC Inheritance	19
Figure 3.1.5: HDLC Link Configuration Options	20
Figure 3.2.1: HDLC Frame Format	22
Figure 3.2.2: V.120 Frame Format	22
Figure 3.2.3: Frame Relay Frame Format	22
Figure 3.2.4: LMI Message Format (Frame Relay Standard)	23
Figure 3.2.5: LAP-D Frame Format	23
Figure 3.2.6: LAP-B Frame Format	23
Figure 3.2.7: SDLC Frame Format	23
Figure 3.2.8: PPP Frame Format	23
Figure 3.2.9: IEEE 802.3 CSMA/CD(Ethernet) Frame Format	24
Figure 3.2.10: IEEE 802.4. Token Bus Frame Format	24
Figure 3.2.11: IEEE 802.5, Token Ring Frame Format	24
Figure 3.2.12: Fiber Distributed Data Interface(FDDI) Frame Format	24
Figure 3.2.13: ATM AAL-5 Mode Service: PDU Segmentation	25
Figure 3.2.14: ATM Header Format	25
Figure 3.2.15: ATM Connection Identifiers	26
Figure 3.2.16: SONET STS-1 Frame Structure	27
Figure 3.2.17: IP Frame Format	28
Figure 3.2.18: TCP Frame Format	28
Figure 3.2.19: UTOPIA-I Interface Signals	29
Figure 4.1.1: 1:1 Frame Conversion	32
Figure 4.2.1: 1:1 Embedded Frame Conversion	33
Figure 4.3.1: n:1 Frame Conversion (Reassembly)	34
Figure 4.4.1: 1:n Frame Conversion (Segmentation)	35
Figure 5.1.1: CPU Centric Architecture	36
Figure 5.1.2: Sub-Processor Functional Block Architecture	37

Figure 5.1.3: Programmable Pipelined Functional Block Architecture	39
Figure 5.1.4: Processor Embedded Pipeline Architecture	40
Figure 5.1.5: Processor Architecture	41
Figure 5.2.1: HDLC to Ethernet Conversion Process	43
Figure 5.2.2: Input Buffer and CPU Interconnection	44
Figure 5.2.3: Input Pipeline Transfer	45
Figure 5.2.4: Output Pipeline Transfer	46
Figure 5.3.1: CPU Finite State Machine	47
Figure 5.3.2: CPU FSM Inner Cycle	48
Figure 5.3.3: CPU Memory Access States	49
Figure 5.3.4: CPU FSM Outer Ring	51
Figure 5.3.5: Memory Address Space	52
Figure 5.3.6: Register Bank Size Comparison	53
Figure 5.3.7: CPU FSM Interrupt Cycle	56
Figure 5.3.8: CPU Data Path	57
Figure 5.4.1: Bit and Byte Stuffing Example	58
Figure 5.4.2: Unstuffer Data Path	59
Figure 5.4.3: Stuffer Data Path	60
Figure 5.5.1: CRC Control Matrix Example	63
Figure 5.5.2: Iterative CRC Control Signal Generation	64
Figure 5.5.3: Output CRC-16 Engine	65
Figure 5.5.4: Controlled XOR Gate	66
Figure 5.5.5: CRC-16 Controlled Input Array	66
Figure 5.5.6: Input CRC Datapath	69
Figure 5.5.7: Output CRC Datapath	71
Figure 5.6.1: Header-Trailer Datapath	73
Figure 5.7.1: UTOPIA Input Buffer	75
Figure 5.7.2: Input Buffer Start Pointer Flushing	76
Figure 5.7.3: UTOPIA Output Buffer	78
Figure 5.7.4: Asynchronous FIFO Behaviour	80
Figure 5.8.1: Input DMAC Command Options	82
Figure 5.8.2: Input DMA Controller	82
Figure 5.8.3: Input DMAC FSM	83
Figure 5.8.4: Output DMAC Command Options	84
Figure 5.8.5: Output DMA Controller	84
Figure 5.8.6: Output DMAC FSM	85

List of Appendicies

Appendix A: CPU Instruction Set.	97
Appendix B: Example Frame Processing Programs.	104

Acronymns and Abbreviations

AAL	ATM Adaptation Layer: A layer in the ATM model which accepts user PDU's and prepares them for transmission over an ATM network, and vice versa.
Address Translation	The process by which an address from one network is translated to an equivalent address in another network.
ARQ	Automatic Repeat Request: Flow control mechanism where a receiving station requests that a damaged frame be retransmitted.
ASIC	Application Specific Integrated Circuit: A custom integrated circuit designed to perform some function for use in a specific application, as opposed to generic applications.
CAM	Content Adresseable Memory: A memory device with a searcheable region and a data region. Information can be written to both areas to set up the memory. It is then possible to input a bit sequence to the search region and have the matching associated data region returned. This provides a quick and simple table lookup-type procedure embedded in the hardware of the memory device. It operates in a similar fashion to a memory cache.
CRC	Cyclic Redundancy Check: Error detection scheme. Protected messages are treated as one large bit sequence. This "number" is then divided (modulo 2) by a given polynomial. The remainder is appended to the transmitted message.
DL	Data Link: Data link layer of the OSI network model
DMAC	Direct Memory Access Controller: A device that can access memory usually only available to a CPU. Such devices are commonly used to transfer blocks of memory to I/O devices

Programmable Protocol Conversion Processor

	freeing up the CPU to perform other, unrelated, operations.
DSL	Digital Subscriber Line: A high-speed network, for the home, designed to run over a standard telephone (twisted pair) line without interfering with the functionality of the telephone. Standards exist for various uplink and downlink speeds usually between 1-3 Mbps for the downlink and 64Kbps and up for the uplink.
ED	End Delimiter: Symbol of a protocol frame to denote the end of the frame. Frequently a non-data symbol.
FCS	Frame Check Sequence: Analogous to CRC.
FDDI	Fiber Distributed Data Interface: LAN-style fiber optic network.
FSM	Finite State Machine
HDLC	High-level Data Link Control: Standard protocol used in a variety of network types. As its name suggests, it operates at the data link layer of the OSI network model.
Immediate Data	Numbers hard-coded into a program. If a program loads a register with the number 1000 say at the beginning of the program, the number 1000 is considered immediate data. It is common practice to specify addresses of I/O devices in this way, since these addresses are known and will not change.
IP	Internet Protocol: Network layer protocol for the present day internet.
ISR	Interrupt Service Routine: Software procedure executed in response to an interrupt.
LAN	Local Area Network: A network interconnecting a small cluster of computers using a shared link. E.g. ethernet and token ring.
LAP-B	Link Access Protocol B: Data link layer protocol used in ISDN and X.25/Frame Relay networks.
LAP-D	Link Access Protocol D: Data link layer protocol used in ISDN networks.
NET	Network: Network layer of the OSI network model
Non-Data Symbol	In LAN networks, all 8 bit data patterns are block encoded to 10 bit

Programmable Protocol Conversion Processor

patterns. Since not all possible 10 bit patterns can be mapped to data values, some are used for control signalling and are filtered out of the data stream before frame processing.

OSI	Organization des Standards Internationale: The International Standards Organization. A standards body.
PDU	Protocol Data Unit: Data presented to a network layer for transmission from a higher network layer.
PHY	PHYsical Layer: Lowest layer in the OSI network model.
PPP	Point to Point Protocol: Protocol to define a virtual point-to-point link between nodes on a network. Commonly used to carry Internet TCP/IP traffic.
RFC	Request For Comment: Internet standards body. Informal standards are proposed and commented on until a consensus has been reached. This consensus then becomes a de facto standard. A sort of standard of convenience.
SAP	Service Access Point: A port through which data is passed between layers in an OSI model network. A layer may have multiple service access points with its neighbouring layers.
SAR	Segmentation And Reassembly: A set of procedures performed by the ATM layer in an ATM network. AAL PDU's are segmented for transmission as ATM cells and then reassembled at the receiver back into the AAL PDU.
SD	Start Delimiter: Symbol in a protocol frame to denote the beginning of the frame. Frequently a non-data symbol.
SDLC	Synchronous Data Link Control: LAN protocol developed by IBM for use in its ring-based LAN networks. Protocol functions similarly to the HDLC protocol.
SFD	Start Frame Delimiter: Analogous to start delimiter.
SONET	Synchronous Optical Network: A family of fiber optical transmission standards ranging from 155Mbps to 40Gbps.
TCP	Transport Control Protocol: Transport layer protocol for the present

day internet.

TR

Transport: Transport layer of the OSI network model.

UTOPIA

Universal Test & Operations PHY Interface for ATM: A physical layer interface standard specified by the ATM forum. This standard allows chips from different manufacturers to interface to any PHY layer ATM device.

Chapter 1: Introduction

Networked computers communicate using a communications protocol to transfer and share information. All computers connected to the same network must use the same protocol or they will be unable to communicate properly. Problems arise when one network of computers wants to communicate with another network which uses a different protocol. Additionally, the bridge between these two networks may use a completely different protocol, specific to the transmission medium's properties. A conversion of protocols from one standard to another is necessary.

Normally, protocol conversion refers to the transfer of data between protocols operating at the same layer or level within a network. Within the context of this thesis, protocol conversion is used to refer to any modification of network data and is not limited to operations within a single network layer. For some supported conversion scenarios, this process may span multiple layers.

To convert between two different protocols, packets of information must, either be stripped down and repackaged, or encapsulated in the new protocol for transmission to the other network. This conversion/encapsulation process is complex and diverse, with many variations and options, specific to the individual networks and protocols involved.

A common approach has been to use software on standard microprocessors to perform this complex conversion process (see reference 1). The processor was programmed to perform all aspects of the frame conversion from bit stuffing and error checking to address translation and control generation/interpretation. This is a lot for one processor to perform sequentially and quickly. Much work was put in to optimizing code, to make it more

efficient. Table look-up algorithms were developed to reduce error checking times, and data was segmented into 16/32/64/etc chunks to match the hardware of the underlying processor. Advances in CPU design, especially execution speed, allowed higher speed protocols to be supported. This continued until the advent of high-speed links that went beyond the processing rate of CPU's. Functionality was lost since as inter-frame delay decreased, the CPU was spending a higher proportion of its time performing basic frame processing, rather than frame conversion.

Engineers were left with the option of using slow processors, hoping that the time between frames addressed to another network (hence needing conversion) would be long, or using specialized high-speed ASIC's. With ASIC's, engineers were able to perform specific conversions at high transmission rates. However the generic programmability was lost. With an ASIC, all processing tasks can be distributed in parallel across dedicated hardware, so to add more functionality, just add more hardware.

The major drawback of that design approach is that several ASIC's that perform similar protocol processing operations must be designed, built and inventoried to support different protocols. This represents a major investment in time, money and management resources to support all these chips. If a new conversion standard emerged or if the standard was revised, then the ASIC would have to be redesigned to account for any changes. All the old inventory would be made useless.

Today's ASIC's can handle transmission rates of over 10Gbps to support the fastest optical transmission standards. CPU design is advancing, but processing speeds are still too slow for a CPU to completely take over processing. For example, a 100MHz CPU could handle the serial bit-stream of a modem (56Kbps), or digital subscriber line (DSL) speeds of up to 1-3 Mbps. Cheap, fast DSP processors have already been used in modems and other PC communications devices. Processing ethernet traffic of 10Mbps, is probably within the domain of 100MHz CPU's too. However, as the transmission speed increases, the number of conversion options and features must decrease as the processor is forced to spend a higher proportion of the time available on basic frame processing, and less on conversion.

Processing of the newer 100Mbps and 1Gbps ethernet protocols, is obviously beyond the processing capabilities of conventional 100MHz processors.

The approach of this thesis is to develop a protocol frame processor. Similar to a digital signal processor (DSP) which processes a serial bit stream, this frame processor would process a serial frame stream. With buffering and CPU manipulation of frame data at the byte or word level, protocol manipulation for conversion between two networks could be realized. Analysis of several common telecommunications protocols is made to find similarities among them. With this knowledge, a processor that would support conversion between these protocols is designed. The result of this analysis gives a processor that can be part of a system to replace all or most singular converter ASIC's.

The speeds from a few tens of Mbps through to a few hundred Mbps cannot be processed by a CPU alone, yet are well below the processing potential available through a dedicated ASIC. In this thesis, this range of bit rates is considered for a cooperative processing strategy, where CPU control is aided by sub-processing devices to help remove some of the processing load, while still maintaining the decision power of a CPU. The processor incorporates dedicated hardware to deal with common protocol processing operations, and a central programmable microprocessor to perform the specific conversion/encapsulation details. This compromise between "fast", dedicated hardware, and "slow", programmable software allows this processor a compromise of both worlds: the ability to handle some high-speed communications, and still comprehend a diverse set of protocols.

For this thesis, a digital processor was developed using the Verilog hardware definition language. Frames are processed in a pipeline fashion with a central processing unit overseeing the manipulation process. The input and output pipelines are under the control of sub-processors, essentially leaving the on chip CPU performing a management function. The CPU only interacts with frames directly to manipulate specific bit or byte fields (e.g. address translation). The pipeline performs functions that are common to many protocols, or that would put too great a load on the CPU, for example CRC checking and generation and HDLC bit/byte stuffing and unstuffing. The processor's interface has been targeted to

handle a 200Mbps input stream. The pipeline and CPU run at a frequency of 50MHz.

This processor has been verified using Verilog simulation. Attempts were made to implement portions of the design in an FPGA, but unfortunately, a working prototype was not completed.

Development of the processor in a 0.35 micron ASIC process was also attempted. The great length of time between fabrication runs prevented actual fabrication of the processor. however back annotated timing of the synthesized design was possible. Synthesis of the individual parts was successful at twice the frequency required for the intended interface. As such is was expected that the fully synthesized processor would match timing for the requisite frequency. without problems. This design does not stress the limits of the given ASIC technology.

Chapter 2: Networks

This chapter is meant to provide some background and vocabulary about computer networks and the various layers of control associated with them. All networks operate using various layers arranged in a hierarchy. Each layer is responsible for accepting data from higher layers, manipulating it and then sending it down to the next lower layer for transmission. Additionally, the reverse is also required, namely, accepting received data from a lower layer, processing it and then presenting the result to the next higher layer. There may be multiple access points between layers for multiple application processes or multiple networks interfaces. Each access point is generally referred to as a service access point (SAP), see figure 2.0.1.

Figure 2.0.1: Service Access Point Examples

The number of access points between layers is variable. Multiple service access points could mean multiplexing traffic between multiple network interfaces, the sharing of a single network interface with multiple processes or sharing multiple interfaces with multiple processes.

2.1 OSI Network Model

The OSI network model, shown in figure 2.1.1, is a model of how user applications communicate across a computer network. Each layer of the model is tasked with performing various operations to coordinate reliable data transfer across the network. Each

layer is examined below.

Figure 2.1.1: OSI Network Model

The dotted lines represent the virtual path of data at each layer. Sent data is passed down the hierarchy to the physical layer where it is passed between network entities. Received data is processed up the hierarchy in a similar fashion.

2.1.1 Physical Layer

This layer is only aware of an unstructured bit stream, the concept of user data does not exist. Standardized options at this layer include voltage level, swing, bit duration, and other mechanical requirements of the physical link. Additionally, it defines any procedures required to set up and run the link. Example protocols at this level is SONET and RS-232.

2.1.2 Data Link Layer

The Data Link layer interfaces with the raw bit stream from the physical layer and separates out blocks of data called frames. Using frames, the Data Link layer provides synchronization, error and flow control across the physical link. When errors are detected, frames are retransmitted. Thus, higher level layers can assume virtually error-free transmission. When data is transmitted through several tandem links, errors are still

possible, and are uncontrolled at this level. Higher level protocols must account for this. Example protocols that operate at this layer are HDLC, LAP-B, LAP-D and LLC.

2.1.3 Network Layer

This layer is meant to provide some transparency for data transport across a network. Generally, data link layer protocols are only aware of either point-to-point links or small clusters of nodes in a local area network (LAN). Communication between nodes not directly connected or nodes from different clusters is the responsibility of this layer, namely inter-network communication. In this way, all layers above the Network layer (i.e. Transport, Session, etc.) are unaware of the specific protocols used to transfer data, whether a particular node is local or not, or for that matter, the nature of the underlying network. This makes it easy to change the network characteristics without completely redesigning nodes. Example protocols of this layer include the Internet Protocol (IP) and the X.25 layer 3 protocol.

2.1.4 Transport Layer

This layer provides a reliable mechanism for the exchange of data between computers in different networks. It ensures that data units are delivered error-free, in sequence and with no losses or duplications. Depending on how extensive the services of the Network layer are, the Transport layer may provide very little in addition to these services, or if the Network layer is unreliable, it may need to provide extensive error checking and flow control. An example of a protocol operating at this layer is the Transport Control Protocol (TCP).

Protocol Name	Associated OSI Layer(s)
HDLC	DL
LAP-B	DL
LAP-D	DL
Ethernet	DL, PHY
Token Ring	DL, PHY
SDLC	DL, PHY
Token Bus	DL, PHY
FDDI	DL, PHY
ATM	TR, NET
Frame Relay	NET, DL, PHY
SONET	PHY
UDP	TR
TCP	TR
IP	NET
PPP	DL

Table 1: OSI vs. Protocol Comparison

Not all protocols were developed with the OSI network model in mind. As such, most do not fit into any one particular layer. Additionally, some protocols are intended to be multi-layer and have functionality defined for several layers (e.g. Frame Relay and ATM).

2.1.5 Session, Presentation and Application Layers

These layers are primarily responsible for providing a standard interface for applications to use and otherwise interact with the network. They allow multiple applications to initiate separate connections with remote computers, communicate through the connection, despite differences in architecture, and then close the sessions, when transfers are complete. These layers are mainly software based, and provide software control hooks for applications. This thesis deals mainly with Data Link, Transport and Network layer protocols so a further discussion of these layers will not be made.

2.2 Beyond the OSI Model

With today's Internet, the OSI model is unsuccessful in fully describing it. The major problem is that some layers have been sub-divided, especially the data-link layer. The HDLC protocol is being used by higher level protocols to encapsulate their frames for

transmission over high-speed networks. Specifically, the PPP protocol is being used to build virtual point to point links over large multi-point networks (references 8, 26, 27 and 28). This sub-layering causes many variations in implementation creating many “home-grown” proprietary network solutions.

Reference 10 opens the door to running an ethernet over an ATM network. This completely flips the OSI network model upside-down by running a data-link/physical layer protocol over a transport/network layer protocol. These kinds of exotic network solutions demand new kinds of functionality from network devices.

With a programmable frame processor, network device functionality is not as fixed, and can be adapted to new situations.

Within the context of this thesis, *protocol conversion* is used to refer to any modification of network data and is not limited to operations within a single layer and may span multiple layers. This is in contrast to a more specialized use of the phrase to denote data transfer between same-layer network protocols. This looser definition allows more complex processing options to be explored.

2.3 Conversion Network Scenarios

Below are two examples of network configurations which would require protocol conversion, encapsulation or manipulation. A network entity which provides access to external networks is usually called a gateway. It is usually the job of the gateway to perform the conversion, encapsulation or manipulation.

2.3.1 Bridge Configuration

In this configuration (figure 2.3.1), the gateway machine is a physical part of two different networks. The gateway has two network interfaces and two network identities, one for each network.

Figure 2.3.1: Bridging Network Conversion Scenario

For this configuration, the gateway node is a part of both networks: physically and logically.

2.3.2 Tunnel Configuration

For this configuration (figure 2.3.2), there are essentially two gateways, one for each network. The two gateways are connected to each other via another network. The two connected networks could operate using the same protocol, with the connecting network being different. Several variations exist. The main difference from bridging is that frames entering the tunnel arrive at the other end unchanged. It is as if they have never been modified.

Figure 2.3.2: Tunneling Network Conversion Scenario

For this configuration, there is a gateway node for each network. The gateways are connected via a third network.

Chapter 3: Protocols

The point of a protocol is to transfer data from one point in the network to another. Transfer is either reliable, using some sort of error checking, or unreliable, assuming that the transporting network is (almost) error free. Usually the transferred data is sandwiched between a header and a trailer (see figure 3.0.1), although some protocols have only one. Header and trailer contain information about the packet that they are attached to, such as:

Figure 3.0.1: Common Frame Format

source/destination address, error checking, protocol control, flow control, padding, packet size and sequencing. Generally, addressing and control information is found in the header, while error control and padding are found in the trailer. There are notable exceptions to this common configuration, but it was this typical field arrangement that was used to architect the processor's design.

Figure 3.0.2: Protocol Family Tree

Family tree of protocols commonly used in various types of networks. Not all protocols listed operate at the same level (e.g. Internet traffic is carried over any of ISDN, LAN or high speed networks).

Figure 3.0.2 shows various protocols grouped by the types of networks each is employed in. ISDN networks are generally used to provide network services to the home, usually over a standard telephone (twisted pair) line. ISDN networks are designed to carry “low speed” traffic, such as voice although data can also be carried. It operates in a circuit switched mode (as opposed to packet switched like ATM). To achieve higher speeds, several ISDN channels can be multiplexed together.

LAN's are primarily found in small offices and businesses, although they are becoming more common in homes. Ethernet is probably one of the cheapest networks to build, and hardware is easy to come by. A LAN is primarily meant to carry data traffic. There is little coordination of timing across the network, and no guarantees for quality of service. For example, a process cannot reserve a certain amount of bandwidth for timing critical applications (such as voice). The solution to these problems has been to increase bandwidth so that there is an ample supply available for all competing processes, however this is not a solution for busy, heavily populated networks. This network type is best suited to non-critical applications like file transfer, and message passing.

High speed networks were initially developed for long distance telephone companies. These large trunk lines connected several smaller, regional networks together to form one large network. Their high speed makes them a high cost to build and maintain. SONET networks allow only point to point links, thus there are no routing options. The basic high quality of service makes error checking less important, compared with slower, cheaper media. ATM networks usually run on a backbone of SONET links and can provide a variety of controls for types of services. One can allocate dedicated bandwidth for voice or video teleconferencing-type applications, or use a less restrictive service for non-critical applications like file transfer or video playback. Frame Relay allows similar services as compared to ATM, but is older, and frame sizes are not fixed to such a small size.

3.1 What is in a Protocol?

A protocol is more than just its frame format. The frame format is how input data is to be formatted, and how this data is packaged and transmitted over a network. Some protocols

have several layers of control associated with them to ensure data is correctly transmitted. Others have no associated control, and never attempt to check if messages get through. Some protocols are designed for use with very specific transport technologies, and thus have control data fields that address problems or hazards associated with a particular technology. Most other protocols ignore the underlying architecture, and therefore assume nothing about its characteristics. Below are outlined a few protocol standards to illustrate the range of differences between them. Much of the information in the following sections has been taken from references 2, 32 and 33.

3.1.1 Asynchronous Transfer Mode (ATM)

In ATM networks, all data is transmitted as cells. Cells travel over a connection between to network nodes. A node is a termination point on a network, presumably the point where a user can interface with the network. A connection is made up of several point-to-point links. A channel is an addressing mechanism of the ATM network to set up connections between nodes. Many channels can exist of a particular link, and may use all or some of the available bandwidth of that link. In this way, multiple channels can share a link.

In ATM transmission, a connection must be specified between two stations. They use a particular channel to communicate with each other. No other station may use that channel until it is freed. In this way, stations receive a call to begin transmission, an acknowledgement is given, and then communication can begin. The network is responsible for assigning an available channel for the two stations to communicate through. There is no unannounced communication.

Within a network, all channels are multiplexed together. The data cells of various channels are interleaved across the links in the network. Data is routed through the network at the cell level, with the final destination node accumulating and reassembling its cells. Channels that have been allocated higher bandwidth are allowed to inject more cells per unit time than other channels. If the total bandwidth allocated to a particular channel is not used, the extra is given to other channels temporarily, until needed by the original channel. Those channels not specifically allocated any bandwidth accept the best available service

at the time. This is accomplished by the channel presenting a cell to the network as often as it is allowed. This leads to bursty traffic as bandwidth becomes available, and then later removed.

Once a connection is established, the way in which data is transmitted over that connection can vary. The channel can be of a constant or variable bit-rate. Cells can be addressed to specific services or left unaddressed, with some higher level, embedded protocol coordinating all data transfer requests. The table below summarizes the three major ATM adaptation layer (AAL) services.

Service	Bit-Rate	Connection Control
AAL-1	constant	Connection Oriented
AAL-3/4	variable	Connectionless or Connection Oriented
AAL-5	variable	Connectionless

Table 2: ATM Adaptation Layer Modes

Connection control refers to how cells are transmitted within a virtual channel. For a connection oriented link, payloads are numbered, and are marked as to which process they belong to. With several processes sharing a link, their traffic is interleaved over the network, requiring it to be sorted and reordered at the receiving end. Connection oriented communication is useful for long messages, such as file transfer, or timing critical traffic, such as voice or video conferencing.

For connectionless links, cells arrive unannounced. If several processes share the link, their individual protocol data units (PDU's) must be distinguishable or sent serially, in order to avoid confusion. Connectionless communication is useful for short messages, where connection control overhead is not necessary, and takes too much time to process efficiently.

Figure 3.1.1: Basic ATM Cell Structure

There are two basic layers to this protocol: the ATM layer and AAL layer. The ATM layer is responsible for transmitting, receiving and routing standard ATM cells. These cells have 5 bytes of header and 48 bytes of payload (see figure 3.1.1, above). No cell has any more or less. Cells are padded if necessary to make them exactly 53 bytes long. The ATM layer is responsible for carrying out the routing of cells based on the destination address. Cells are transmitted in order, but can occasionally be lost or jumbled. There is no guarantee that data is transmitted correctly or at all.

The AAL layer adapts input data to the ATM network, and presents accumulated payloads at the output. Generally this is done by encapsulating the data in one of three AAL services: AAL-1, AAL-3/4 or AAL-5. Each service provides very different options, to best match a specific type of input data.

AAL-1 assumes a connection oriented, constant bit-rate data source. Cells are numbered as they are transmitted to ensure that data is received in the correct order. The cell numbering requires one byte of payload space and increases the effective cell overhead. It is expected that only one process will be generating traffic since it must be at a constant rate. It is possible, however, to multiplex several slower speed links into one high speed link for transmission over an AAL-1 channel, but this multiplexing must be controlled

outside the AAL interface.

The AAL-3/4 service is a variable bit-rate service, and can assume either a connection oriented or connection less channel. Multiple processes can share this channel, with their cells interleaved over it. Cells are numbered and processes are differentiated by an additional 10 bits of addressing (using up some of the payload space within a cell). As well, there is a CRC-10 check made on the payload data (again at the cost of payload space). The total available payload is reduced from 48 to 44 bytes. Many designers view this service as too overhead intensive given the effective payload size.

AAL-5 is variable bit rate and connectionless oriented. If multiple processes are sharing a channel, only one PDU from a process is transmitted at once. Since the cells contain no other distinctions, multiplexing traffic would lead to confusion as to which cells belong with which process. All received cells are assumed to be of the same PDU until the end cell is signalled. A CRC-32 check is performed over all cells in the PDU, rather than on each individual cell. This reduces the per cell processing overhead while still providing some error checking and high throughput.

In all cases, damaged cells are dropped from the network, with the assumption that some higher or lower layer protocol is watching out and will retransmit data as necessary. The physical layer is not specified, but is usually assumed to be SONET. More about ATM protocols is discussed in section 3.2 and in reference 39.

3.1.2 Ethernet

While there are about three unique ethernet-type standards, they all behave in more or less the same way. All stations on a network transmit data on a shared bus. If two stations happen to transmit at the same time, both immediately stop transmitting and wait a random amount of time before trying to transmit again. No station begins to transmit if it detects another station already transmitting. Stations continue trying to transmit a frame until no other station interrupts the transmission. Basic error checking is performed on all received

frames, with damaged frames being dropped from the network. This protocol is similar to ATM in its lack of responsibility about the transmission of frames. Since frames cannot be received out of order, there is no provision in this protocol for numbering frames. Due to the nature of the shared bus, all stations receive all frames, and discard those that are not destined for that node. This allows one to easily provision a broadcast address to which all stations respond to, a service not easily available in an ATM network.

All data transmitted on the network is 8/10B block encoded. That is, every 8 bit pattern is mapped into a 10 bit pattern. This is to help keep data randomized, and to aid in clock recovery. Obviously, not all the 10 bit patterns are used: only 256 of the 1024 are mapped for data. Some of the remaining 768 patterns are used for special signalling purposes (e.g. start delimiter and end delimiter) and are referred to as non-data symbols. These non-data symbols are used to provide protocol-control information to other stations without sending a separate control frame (as the HDLC protocol does). These symbols are filtered out before data is presented to higher network layers since they cannot be mapped into meaningful data.

There are several physical layers defined for this protocol including coax and twisted pair. Each specifies further requirements pertaining to the electrical signalling specification required for each medium.

3.1.3 Token Ring

This protocol is similar to ethernet, however it has one major difference. Stations may not transmit unless it is their turn to transmit. In the network there is a token. Stations only transmit when they have the token. After a transmission, they pass the token on the next station in line to get the token. In this way, communication is very controlled, and the bus is shared fairly (see figure 3.1.2).

Figure 3.1.2: Token Ring Network: Logical Ring Configuration

When stations are added or removed from the network, they must be added or removed from the token chain as well. A complex state-machine controls how stations communicate collectively to add or remove themselves from the network. Provisions are also needed in case a station with the token crashes, and the network is left without a token (see figure 3.1.3). Timers are required to determine a "timeout" condition when stations then arbitrate for who gets the token. This timer is based on how long it takes for a frame to propagate around the ring, and it reset when a new frame is detected. Additionally, any holes in the token passing chain (due to crashed or isolated systems) must be healed to maintain the ring nature of the network. In this protocol, all frames are seen by all stations. Stations ignore frames meant for other destinations. This protocol also uses 8/10B block encoding of all transmitted data and has several physical layers defined.

Figure 3.1.3: Token Ring Network: Token Passing Chain Operations

3.1.4 HDLC

Perhaps one of the most complex protocols due to its flexibility, the HDLC protocol state machine can behave like any one of ATM's AAL services. Connections between machines can be ordered up, and then torn down after transmission is complete or communication can be routed to its destination without any formal connection. During transmission, frames can be numbered and acknowledgements received for each number before more frames are sent. Various windowing options can also be employed to allow multiple frames to be transmitted at once while waiting for acknowledgements. Because of its generality, it is left to specific implementations to define a physical layer, and its requirements, see figure 3.1.4 for a list of common HDLC derivative protocols.

Figure 3.1.4: HDLC Inheritance

Due to the flexible nature of the HDLC protocol, it has been used in many different types of networks. The more popular options have been standardized under various names, as shown in the figure. In some cases, not all functions of the HDLC protocol have been inherited. For example, Frame Relay incorporates HDLC framing, but also defines higher layer functionality.

The HDLC standard defines three station types (see figure 3.1.5):

- **Primary:** Issues commands to secondary stations. A primary station has operational control of all links.
- **Secondary:** Responds to primary station commands. Under link control of primary station. Each secondary station is connected to the primary via a separate logical link.
- **Combined:** A station which can behave as both primary and secondary.

Figure 3.1.5: HDLC Link Configuration Options

The links between stations are full or half-duplex and can be one of two configurations:

- **Unbalanced:** Link consists of one primary and multiple secondary stations.
- **Balanced:** Link consists of two combined stations. The link is point-to-point.

There are data transfer modes defined for each link configuration. For the Unbalanced mode, data transfer options are:

- **Normal response mode (NRM):** Primary initiates data transfer. Secondary only responds to poll command from primary.
- **Asynchronous response mode (ARM):** Secondary station may initiate/request a data transfer, although primary maintains control over the link. This configuration is rarely used.

For a balanced link configuration, there is only one data transfer mode defined, the asynchronous balanced-mode (ABM). In this mode, either combined station may initiate

a data transfer without permission from the other.

The control field in the HDLC frame is used to define three frame type:

- Information (I) Frame: Used to transmit user data
- Supervisory (S) Frame: Used to provide automatic repeat request (ARQ) flow control
- Unnumbered (U) Frame: Supplemental link control functions (e.g. link configuration)

In an I-frame, the control field is used to denote the ordering frames, and to acknowledge which frames have been received. In this way some ARQ responses are piggy-backed onto a return-link frame (e.g. sequence number acknowledgement), however if none are available, use of an S-frame is required. Unnumbered frames are used to configure link options, such as those listed above (e.g. NRM, ARM).

All HDLC related protocols have flag sequences at the beginning and end of all frames. The flags are used to delimit frames in the serial bit stream. The flag sequence is 01111110 or h7E¹. When a link is idle, a constant stream of flag sequences are generated. If a flag byte is generated as a part of data, this could cause problems at the receiver which would incorrectly detect the data and interpret it as the end flag sequence. To prevent this from occurring, a technique called bit stuffing is used. Bit stuffing involves monitoring the transmitted bit stream and inserting zero bits when ever a flag sequence might occur. These bits are inserted in a deterministic way so that they can be removed at the receiver.

The HDLC protocol was developed for use in a serial transmission system where the amount of data exchanged was measured in bits, not bytes. With newer highspeed links, data is processed in bytes or words to speed up CPU-type processing. The data is only serialized at the physical layer. Most high speed protocols assume an integer number of bytes to transmit. The standard defines stuffing for octet oriented links (high speed), as byte stuffing. In this case, the byte stream (as opposed to bit stream) is monitored for the

¹. This notation denotes a hexadecimal number (base 16).

flag character h7E. When a flag character is found, the two byte sequence: h7D, h5E is used to replace it. To protect the protection sequence, when h7D is detected, the two byte sequence: h7D, h5D is used to replace it. In this way, the flag sequence never occurs except at the beginning and end of a frame.

For a more indepth description of HDLC stuffing, please refer to section 5.3.

3.2 Protocols Supported in this Thesis

The HDLC protocol (figure 3.2.1) was supported because it is the base format for many derivative protocols used in the ISDN and LAN networks (e.g. LAPB, LAPD, SDLC and PPP). The protocol standard allows for many possible variations and implementations. It is because of this flexibility, that this protocol is in use in so many different systems, all-be-it under different names. The protocols in figures 3.2.2 through 3.2.8 are all HDLC derivatives and their frame structures are demonstrative of their parentage. Each frame is encapsulated between two flag-bytes. They have the destination address first, followed by a control field. All data within the frame is protected using 2-4 bytes of CRC. Perhaps the most famous protocol in this family is frame relay. The inclusion of the HDLC protocol greatly diversifies the protocol base supported by this processor. Its frame structure exemplifies the common frame structure used for this processor design. Additionally, the

Figure 3.2.1: HDLC Frame Format

Figure 3.2.2: V.120 Frame Format

Figure 3.2.3: Frame Relay Frame Format

Figure 3.2.4: LMI Message Format (Frame Relay Standard)

Figure 3.2.5: LAP-D Frame Format

Figure 3.2.6: LAP-B Frame Format

Figure 3.2.7: SDLC Frame Format

HDLC protocol is commonly used for encapsulation purposes over high speed links in conjunction with internet protocols. see references 13, 14, and 26.

Point-to-point protocol (PPP) support was considered important because of its popularity for encapsulating other protocols (e.g. IP over ATM), see references 8, and 25-31. It was found that its frame structure is also a derivative of HDLC, with its address field fixed to the "all stations" address and control field set to the unnumbered information (UI) command. The protocol field is essentially an HDLC protocol address extension, and is used to denote the type of PDU encapsulated in the data field. The PPP protocol is

Figure 3.2.8: PPP Frame Format

commonly used by higher layer protocols to establish a virtual point-to-point link across networks. It's ideal for this kind of operation since it provides no address, yet allows multiple protocols to be carried (identified with the protocol field), and somewhat reliably (CRC error protection). It is protocols like this one which are not easy to fit into an OSI

model view of networks.

The popularity of Ethernet and other IEEE 802.x protocols (e.g. Token Ring and FDDI) makes it an ideal candidate for inclusion in the processor's supported protocol base. LAN frame structures (figures 3.2.9 through 3.2.12) are not based on the generic HDLC frame as all the previous cases have been. These frame structures follow the generic frame structure of figure 3.0.1, with a note of the extra field of packet size in its header section. Generally, these protocols are line encoded before transmission over a physical medium (e.g. 8B/10B). The extra unused line codes are then used for special signalling purposes so that these signals will not be confused with data. These non-data signals can be used as start/end frame delimiters, synchronization preamble between frames, and frame/protocol status. Non-data fields are shaded in the frame format figures 3.2.9 through 3.2.12. These

Figure 3.2.9: IEEE 802.3 CSMA/CD(Ethernet) Frame Format

Figure 3.2.10: IEEE 802.4, Token Bus Frame Format

Figure 3.2.11: IEEE 802.5, Token Ring Frame Format

Figure 3.2.12: Fiber Distributed Data Interface(FDDI) Frame Format

ten bit patterns must be removed prior to processing since there is no 8 bit representation

for them.

Figure 3.2.13: ATM AAL-5 Mode Service: PDU Segmentation

An obvious exception to the generic frame format is the ATM protocol, which contains error control information in the header of each cell. As well, in AAL-5 mode, there is an additional CRC-32 check on the accumulated payload data of a series of cells, rather than on a cell by cell basis (see figure 3.2.13).

Figure 3.2.14: ATM Header Format

The HEC byte is a CRC-8 error protection code for the four header bytes.

Referring to figure 3.2.14, the destination address of each cell is composed of the virtual path identifier (VPI) and virtual circuit identifier (VCI). A virtual path may contain many virtual circuits (see figure 3.2.15).

Figure 3.2.15: ATM Connection Identifiers

The payload contents are identified by the payload type identifier (PTI). Cells with their cell loss priority (CLP) bit set are dropped by the network interface if it is too busy to process them. This is to prevent lost or excessive numbers of cells from congesting the network. The header is protected by a header error code (HEC), using a CRC-8 polynomial.

Since the ATM protocol is very popular for high speed links, the inclusion of it in the list of supported protocols was considered important.

3.2.1 Unsupported Protocols

Protocols such as TCP, IP, UDP and SONET are unsupported by this processor (at least directly). The main reason for dropping SONET support was one of speed. The frame interfaces will not support speeds beyond OC-3 (155 Mbps) since it is rated at only 200Mbps. Additionally, the SONET frame header is not similar to other supported protocols (see reference 40). In SONET frames, the header information is interspersed with the information payload at regular intervals, rather than all of it at the beginning. Referring to figure 3.2.16, the bytes are transmitted row by row with each 3 byte header section interspersed between 87 bytes of payload. The flexible hardware to support this type of transfer option was complex and could have degraded processor performance. Therefore, it was decided that this processor would be best at encapsulating protocols, grooming traffic, for transmission over a SONET network, and that all SONET processing would be performed by other, more dedicated hardware, processing frames in series with this

1	1	1	87 Bytes
---	---	---	----------

UDP standards are defined in references 19 through 21.

Figure 3.2.17: IP Frame Format

As defined in Internet RFC 791, reference 19.

Figure 3.2.18: TCP Frame Format

As defined in the Internet RFC 793, reference 20.

One careful note must be made about protocol compliance. The processor architecture of this thesis mainly deals with frame modification, commonly referred to in this thesis as protocol conversion. This is the process of taking one frame's data and conforming it to a new protocol frame standard. Any line encoding, clock recovery, token passing or other protocol state machine behaviour is not intentionally supported by the processor. It is expected that the processor will communicate with other external logic, via its I/O port or shared memory (i.e. memory mapped I/O), to help perform these other protocol mechanisms, but will not be responsible solely for their computation. As such, it is expected that a certain amount of external logic will be required in the system to allow for

proper interfacing to the physical layer characteristics of any protocol. It is for this reason that an I/O interface was included as part of the processor design. Any special signalling to be made by the physical layer can be communicated separately from the data stream via the I/O port.

3.2.2 UTOPIA-I Interface

The UTOPIA level 1 standard is defined by the ATM forum as a standard way to interconnect chips from different vendors. The standard defines only the interface between the two chips: signals, polarity, and clocks. No data formats, or other framing is defined or implied. As shown in the figure below (3.2.19), the interface defines a data bus (8-bits), two acknowledge strobes (Ready and Cell Available) and a start of cell (SOC) signal for frame deliniatiation.

Figure 3.2.19: UTOPIA-I Interface Signals

The interface is completely synchronous. All signals are sampled on the rising clock edge. The CLAV signal (Cell Available) is an indication from the transmitting device that it has a cell available for transmission in its buffer. The READY signal from the receiving device indicates that it is ready to begin the reception of cells into its buffer. When these two signals are in their active states, data is passed every clock edge between the two devices over the data bus. Either device can pause transmission if its buffer becomes full/empty. In this way, these two signals essentially throttle the data rate through the interface according to the processing rates of the two chips. The Data and Clock signals are defined at various ratings to handle a variety of bit-rates. The most basic is an 8-bit data bus running at 25MHz for a total bandwidth of 200Mbps. This is enough bandwidth to handle the primary SONET optical rate of OC-3 (155Mbps). The standard defines the following combinations:

Data Bus Width (bits)	Clock Rate (MHz)	Total Bandwidth (Mbps)
8	25	200
8	33	264
16	33	528
16	50	800

Table 3: UTOPIA Interface Bandwidth Options

For this particular processor the first option was chosen as a likely implementation target. As will be discussed later in the architecture chapter, increasing the clock frequency here can significantly impact the performance of the processor.

By standardizing the interface for the processor, it was hoped that this would make it easier to integrate it with other components at the board level. Additionally, by using the UTOPIA standard, an obvious migration path exists for accelerating the design to handle higher-speed communications, by widening the data bus, and increasing the clock frequency (see references 35 and 36).

The implementation of UTOPIA used for this processor is not completely standard. UTOPIA is meant for use in an ATM network, where the size of a cell is known and fixed. Since this processor is meant to handle protocols where the frame size is variable and unknown, some redefinition of the acknowledgement signals is necessary. The UTOPIA standard assumes that if a device stops transmission in the middle of a cell, that the receiver will be keeping a count of this, and wait for the remainder of the cell before processing occurs. The implementation used in this thesis relies on the fact that a frame will be transmitted wholly into the input buffer. If the CLAV strobe is negated, this signals the end of the frame, and that no new frame is available. Negating the CLAV signal during a frame transfer will result in splitting the frame into two. It is this distinction of the CLAV signal that differs from the UTOPIA standard defined by the ATM forum.

Chapter 4: System Integration

Four major frame manipulation scenarios were developed in the course of this thesis. This processor was designed with the purpose of supporting these four scenarios:

1) 1:1

- Incoming frames are received, checked for errors, stripped, repackaged and sent
- One output frame is generated for every input frame. No segmentation or concatenation is performed.

2) 1:1 Embedded

- Incoming packets have multiple protocols layered on top of each other.
- Outermost layer is checked for errors, desired sub layer is extracted, stripped, repackaged and sent
- Possible embedding of packet into multiple protocols provided only the outermost layer requires CRC code generation

3) n:1

- n frames are reassembled to form a single large frame
- This large frame is then transmitted or stripped, repackaged and then sent

4) 1:n

- Incoming frames are segmented into n smaller sub-frames. Each sub-frame is packaged and sent.

Most protocol conversions can be categorized into one of these four scenarios. The scenarios are mainly concerned with how the input frame is received and what data inside it is transferred. Especially for embedded protocols, what part is user data and what part is overhead can get a little grey. For the most part, the final layer of data to be passed on to the next host is considered data, whether it is internally encapsulated or raw user data.

4.1 Scenario 1, 1:1

Scenario one, shown in figure 4.1.1, exemplifies a simple protocol conversion where one frame is directly converted into another. Any destination address information would be translated to the new system, and the enclosed data segment would be repacked with the new protocol format's header and trailer information. The change in the size of the frame is the difference between the headers and trailers of the two protocols. It is assumed that the maximum frame size of the networks is identical, or that converted frames are never too large for the destination network.

Figure 4.1.1: 1:1 Frame Conversion

4.2 Scenario 2, 1:1 Embedded

Scenario two, shown in figure 4.2.1, exemplifies a system where incoming frames have multiple layered protocols. The output format may be one of the sub-layers alone, or repackaged contents of the data segment embedded within the original frame. For CRC error checking, only the outer-most layer needs to be checked. Once it has passed the error check, all the data contained within the outer layer can be assumed valid. After error checking, sub-layers can be "stripped off" simply by moving the start and end pointers denoting the limits of the frame in memory. The data to be used can easily be picked out in this way for direct output or for encapsulation in another protocol.

Figure 4.2.1: 1:1 Embedded Frame Conversion

4.3 Scenario 3, $n:1$

Scenario three, shown in figure 4.3.1, represents a reassembly function. In this case several frames are compiled to form a single large frame which is then transmitted. In this case, incoming frames can be queried in the input buffer to determine start and end pointers of the data to be concatenated. The frame data can then be appended to any existing data in main packet memory. If header information needs to be checked for errors, two receive instructions can be issued, one for header processing, diverting the header bytes to some unused memory space, and the second to concatenate the data. If the segmented frames are of equal size, then frames may be received out of sequence provided some sequence control information is included in the header or trailer.

When the last frame has been received, the reassembled frame can be encapsulated in another protocol layer, or simply presented to the output for further processing down stream.

Figure 4.3.1: n:1 Frame Conversion (Reassembly)

4.4 Scenario 4, 1:n

The last scenario, shown in figure 4.4.1, describes a segmentation scheme. If the two networks have different maximum frame sizes, then the larger of the two must have its frames segmented to be properly transmitted to the other network. If no segmentation standard exists, then large frames must be dropped or truncated; simple tasks, using pointer manipulation.

It is most likely that all segmented frames will be sent to the same destination. The Header/Trailer unit was implemented for situations such as this. Now, much processing time is saved when most duplicate address or control information is automatically added to every sent frame, without CPU intervention.

Figure 4.4.1: 1:n Frame Conversion (Segmentation)

Chapter 5: Architecture

5.1 Architecture Overview

Several architectures were investigated in the research of this thesis. With each architecture, the associated pros and cons are listed, as well as why the implemented design was chosen above it.

5.1.1 CPU Centric Design

This architecture is the ultimate in programmability. Referring to figure 5.1.1, the UTOPIA interfaces at the input and output serve only to format data into bytes or words and to write the data to memory. The CPU performs all functions sequentially, namely: bit/byte unstuffing, error detection (CRC), header/trailer modification, error code generation, bit/byte stuffing and any other required functions. This architecture is used in reference 1.

Figure 5.1.1: CPU Centric Architecture

The throughput of this design would probably be quite low seeing as the conversion is handled entirely by the CPU. Heavily stuffed frames may lead to larger processing times. CRC checking/generation requires processing the whole frame, and is very time consuming. Performing, for example, both CRC checking and bit unstuffing would require processing the frame sequentially twice, although more advanced VLIW, or superscalar CPU structures could probably perform both these operations at the same time. A large

amount of memory would be required to hold the programme for the CPU, although the total memory requirement compared to the selected architecture is much less. This architecture places performance squarely on the CPU. While there are fast tricks for performing CRC error checking (see references 6 and 17) and such, this type of architecture cannot keep up with high speed protocols. This is the base architecture that is to be replaced by this thesis.

5.1.2 Sub-Processor Functional Blocks Design

This design improves upon the last one by employing dedicated sub-processing blocks to perform common functions, see figure 5.1.2. Each block is able to search through packet memory and read the required information that it needs. The CPU is still performing some byte manipulation, however it is relieved of the burden of performing certain functions, which can be done faster by the dedicated sub-processors than by the CPU itself.

The sequential delay (latency) for any particular frame is large if, for example, both stuffing and CRC generation must be done. However this architecture allows multiple frames to be processed at once. Perhaps up to five frames can be processed, one by each of the four functional blocks and one by the CPU: this is a major speed advantage. Encapsulation of a PDU in more than one layer, where each layer has a CRC code, is possible in this architecture, although this operation would slow down the frame processing rate.

Figure 5.1.2: Sub-Processor Functional Block Architecture

The major drawback of this design is memory. For this design, a large, multiport RAM is required. Each port must allow both read and write operations. Ideally, a RAM on the order of 7 ports would be required! It would be possible to implement this architecture using several 2-port RAMs by virtually segmenting the central packet memory. Each module would then be restricted to accessing a single 2-port RAM at any time. This would lead to large memory segments to account for large frames. If, for example, 4K frames are supported, then the central memory is 7 times that in size (i.e. 28K). Consider also the number of address and data buses that must be routed and multiplexed.

Another problem with this architecture is the concatenation and segmentation of frames. Concatenation would be tricky if each frame must be checked before concatenation. At some point, a module must be able to copy data from one segment to another. This copying could kill all the speed advantages of this architecture. If frames are not copied, then the maximum number of concatenatable frames is the number of memory segments, an unrealistic restriction. Also, if header or trailer information must be stripped from the frames after processing and before concatenation, this adds to the complexity of frame processing for this architecture.

There is also a lot of duplicate hardware in the design. Each functional block has a DMA controller embedded in it for accessing the central packet memory. This design therefore sacrifices area and increased complexity for speed.

The complex memory interface, the large number of address and data buses criss-crossing the processor and no simple method of frame concatenation or segmentation led to this design being dropped in favour of the current architecture.

5.1.3 Programmeable Pipelined Function Block Design

This architecture is similar to the previous one with one major change, the individual functional blocks are now connected in a pipeline, see figure 5.1.3. In this way, frames can be pipelined into packet memory with all the necessary processing performed along the way. The pipeline is somewhat programmeable so that unused modules can be bypassed

to allow for faster processing. The modules are still able to directly address the central packet memory to perform their function directly, or to create a smaller pipeline with other modules in series. This last option is particularly useful for embedding protocols in multiple layers.

Figure 5.1.3: Programmable Pipelined Functional Block Architecture

By pipelining, functions appear to operate in parallel, thus speeding up the frame processing rate of the processor.

This architecture suffers the same problems of memory design as the previous one. In order to properly take advantage of the existing hardware, the memory must be modifiable in several different locations simultaneously. By allowing the frames following the pipeline path to bypass unused modules, the time saved is only one or two clock cycles. Removing this bypass option would increase latency, but this is not a pipeline issue. Throughput is the critical measure of any telecommunications protocol processing device. So long as a device can keep up with the input frame rate, latency is not usually an issue.

The pipeline feature of this design was carried forward to the final architecture, however due to the complex memory interface required, the use of internal DMA controllers in each module was abandoned.

5.1.4 Processor Embedded Pipeline Design

This architecture takes pipelining to an extreme, see figure 5.1.4. With the CPU embedded

in the pipeline, frames are processed on a byte-by-byte basis. The CPU must be able to perform instructions every clock cycle, thus either a VLIW or superscalar processor is needed. Some buffer memory is required at the UTOPIA interfaces so that new frames can be queried prior to processing.

Figure 5.1.4: Processor Embedded Pipeline Architecture

The major drawback of this system is that it is unable to support supplementary communications outside the processor as pipeline bandwidth usage increases. At some point the processor will be so busy trying to figure out what to do with frames that it will be unable to respond to external requests for information or control or at least if it does, the input buffer would be in danger of being overrun. Additionally, the design constraints on the CPU are quite extreme. It is quite beyond my experience to design such a processor given the time constraints surrounding this thesis.

As shown in the earlier designs a central packet memory, reduces the constraints on the CPU. This allows it to execute instructions at a slower pace and still keep up with the incoming frame rate. If the pipeline runs at a faster frequency than in input buffer, then it is possible for the CPU to catch up with the frame rate between frames, if it is forced to respond to some external interrupt.

5.1.5 Implemented Architecture

Figure 5.1.5 shows the general breakdown of the implemented processor's architecture. In order to reduce the computational burden on the CPU, several common frame manipulation functions are implemented in hardware in a pipeline fashion. Each function is able to operate iteratively as frames are transferred, byte-by-byte, through the pipeline, from the

input to the output. In this way, each module in the pipeline views the entire frame one byte at a time. All frames must follow the complete pipeline, with no option to bypass unused modules. Modifications and additions to the frame are made on the fly to each byte as they progress through the pipeline. By pipelining these functions, frame processing time is saved since the pipeline modules effectively operate in parallel. Latency is increased as a result, but this is not as important as throughput. The CPU is left with the duty of coordinating the transfer of frames, and any byte manipulation not already performed by the pipeline (e.g. address translation).

Figure 5.1.5: Processor Architecture

From a datapath perspective, the processor consists of an input pipeline, an output pipeline and a central processing unit (CPU). As frames are received, they are stored temporarily in the input buffer. The CPU signals the top frame in the buffer to be processed, and the

frame is transferred, byte-by-byte, through the input pipeline. The pipeline performs the operations of CRC error checking and HDLC bit/byte unstuffing. The processed frame is then deposited in the central packet memory for manipulation by the CPU. Any segmentation or reassembly functions can be performed on the data segment using pointer manipulation, with no need to move packets about in main memory. Since the input buffer can be addressed for reading by the CPU, efficient use of the receive command is possible, so that successive data segments can be automatically concatenated in packet memory.

To increase the throughput of the processor, packet memory is dual ported, allowing simultaneous read and write. In this way, a frame can be received while a processed frame can be transmitted. Memory contention can occur when the CPU attempts to access memory while a transfer is in effect. The pipeline controllers account for this and stall a pipeline when contention occurs. However, this reduces the throughput of the processor, so further measures have been taken. The packet memory is split into two equal halves. Each half allows a simultaneous read and write. In this way, the CPU can read from the receiving memory segment, and write to the transmitting segment.

Once a frame is in the central packet memory, the CPU is able to manipulate the frame directly. Usually, the data segment of the input frame is left untouched, and only the header/trailer bytes are manipulated to conform to the output frame format. Again, for reassembly/segmentation functions, frames are referred to in memory using start and end pointers. When sending a frame, the start and end segments of the sent frame are all that is needed. In this way, it is very easy to segment or concatenate sequential frames in memory. Once all necessary changes are completed, the packet is then set up for processing in the output pipeline.

The output pipeline functions mirror those of the input in that they generate CRC error information for appending to the outgoing frame, and perform bit/byte stuffing for HDLC flag protection. Additionally, the pipeline can add preprogrammed bytes before and after the frame as extra header or trailer. Once the entire frame has passed through the output pipeline, it is deposited in the output buffer where the UTOPIA-I interface transfers the

frame outside the processor.

The ordering of modules in the pipeline is important. Since stuffing is the last thing done to a frame prior to any transmission encodings, it should be the first thing undone for the receive line. One is unable to perform a CRC check on a stuffed frame, at least to do so would be very slow. The header/trailer module is incorporated at the start of the output pipeline so that the added bytes can be incorporated in the CRC code and stuffing. Obviously, any reordering would limit the usefulness of the header/trailer module.

5.2 Conversion Example: HDLC to Ethernet

The example conversion shown in figure 5.2.1 is presented to give an idea how the various functional blocks of the pipeline interact to perform frame conversion.

Prior to processing the HDLC frame, it must be unstuffed and then checked for errors. Ethernet framing requires no stuffing. We will assume that both networks have the same maximum frame size and that all processor options have been previously set.

Figure 5.2.1: HDLC to Ethernet Conversion Process

The data is extracted from the HDLC frame using simple pointer manipulation. During this process, the data is checked for errors and the HDLC address is translated to its Ethernet equivalent. The extracted data is repackaged into an Ethernet frame and transmitted onto the network.

5.2.1 Input Frame Processing

External logic buffers the input frame from the network and presents it to the UTOPIA interface of the processor. The frame is received and its start and end memory locations are recorded. The frame waits in the buffer for CPU to issue a receive command (see figure 5.2.2).

Figure 5.2.2: Input Buffer and CPU Interconnection

Start and end pointers for the top frame of the input buffer are mapped as dedicated registers in the CPU. Using these two registers, the CPU can query the contents of the top frame prior to processing.

The CPU reads the frame's address and control information and issues the required receive command (i.e. unstuff the whole frame and use all of the frame's data for the CRC check), see figure 5.2.3. While the transfer from the buffer to packet memory takes place, the CPU can begin address translation and any other necessary administrative tasks. To avoid memory contention, and stalling the input pipeline, the translated addresses are written to the Header/Trailer unit (using the CPU data bus, rather than the packet memory bus) for prepending to the ethernet frame.

Figure 5.2.3: Input Pipeline Transfer

Frames are transferred from the input buffer to packet memory using the input DMA controller

When the transfer is finished and the CRC module has not indicated any errors, the CPU reads back the last memory address from the DMA controller to get the end pointer of the frame. Since the frame size has changed due to unstuffing, this pointer could not have been computed earlier. At this point:

- the frame has been unstuffed and checked for errors.
- its address translated.
- the control field checked to determine that this frame is to be converted and not dropped (e.g. supervisory frame meant for only the HDLC network) and
- the CPU has pointers to the start and end of the frame in packet memory.

5.2.2 Output Frame Processing

Next, the CPU computes the size of the frame using the start and end pointers and inserts the result after the data in packet memory. A send command is issued using the HDLC frame's start and end pointers with offsets to select only the data segment of the frame. The output DMA controller is instructed to transfer the memory contents between the two memory pointers and adding header and CRC code (with no stuffing) (see figure 5.2.4). The frame is transferred to the output buffer where the UTOPIA interface presents the completed ethernet frame to the physical ethernet layer for transmission across the network.

Figure 5.2.4: Output Pipeline Transfer

Frames are transferred from packet memory to the output buffer using the output DMA controller. No write pointer is necessary since the output buffer has its own internal FIFO mechanism. The output buffer is not addressable by the CPU (i.e. it cannot read the memory contents).

5.2.3 Processor Clock Rates

The UTOPIA-I standard is defined to operate at a frequency of 25MHz. The internal pipelines have been designed to operate at at least 50MHz. This doubling of the clock rate is to account for processing delays from the CPU. If more time is required to modify the packet for retransmission, this cuts in to the speed advantage of the pipeline, reducing the effective transfer rate through the whole processor. So long as this effective transfer rate stays above 25MHz, the input cannot overrun the processor. As frames become larger, the time allotted for CPU manipulation increases by the ratio of the two clocks times the number of bytes in the frame (see equation 5.1). With the frequencies given above, this "saves" two clock cycles for every byte in the frame. These "saved" clock cycles are then "spent" during the manipulation of the frame (i.e. 3-6 cycles per instruction). In this way, the speed requirements of the CPU are reduced. More internal cycles are spent transferring the frame through the pipeline functional blocks, than manipulating it with the CPU.

$$N_{\text{cycles}} = \frac{f_{\text{Pipe}}}{f_{\text{Buffer}}} \times N_{\text{Bytes}} \quad (5.1)$$

5.3 CPU

The CPU is state machine based (see figure 5.3.1), where most instructions execute within 3-6 cycles or states. The only exception to this are the two macro Send and Receive

instructions which require up to 11 clock cycles to complete. This is due to the number of address pointers that must be computed and passed to the pipeline controllers.

Figure 5.3.1: CPU Finite State Machine

One clock cycle per state/bubble.

Referring to figure 5.3.2, the inner cycle of the fetch, decode and execute states is able to perform all instructions using registers only for operands and destination. The number of required cycles for execution increases for memory related instructions since operands must be fetched from memory and memory addresses must be calculated from immediate

offsets, etc.

Figure 5.3.2: CPU FSM Inner Cycle

Basic instruction execution cycle: fetch new instruction, decode the instruction, and then execute it. This cycle assumes that all data is in registers. The fetch/decode/execute cycle handles interrupt vector table lookup.

The CPU supports three memory access modes:

- Direct
 - The programmer has hard coded an immediate address value.
 - e.g. `MOVE #13,[1234]`
- Register indirect
 - Address is specified using the contents of a register.
 - e.g. `MOVE #13,[R5]`
 - e.g. `MOVE #13,[R5+2]`
- Register indexed
 - Address is computed using the contents of two registers added together.
 - e.g. `MOVE #13,[R5+R7]`
 - e.g. `MOVE #13,[R5+R7+3]`

Referring to figure 5.3.3, these three memory modes can require up to three clock cycles in addition to the basic three. To help speed up processing, a data fetch is performed automatically in the decode state. While the exact instruction may not yet be known, it is likely that additional data is necessary. These additional operands are encoded sequentially after the instruction in instruction memory.. If the data recorded by the auto-fetch is not

needed, it is discarded later. By performing an automatic data fetch in the decode state, the processor is able to quickly begin address calculation. The auto-fetch is meant to fetch operands in the follow order of preference:

- Immediate data used in the execution state
- Immediate offset data to a register(s) for register indirect memory access
- Direct address

This is the primary reason why there are two direct memory access in figure 5.2.3. In the top case (states 2 and 4), the auto-fetch reads immediate data for use in the execute state. A second read is necessary to fetch the direct address (immediate data always preceeds all address data). In the lower case (states 2 and 5), the auto-fetch reads the direct address, so no additional fetch is necessary.

Figure 5.3.3: CPU Memory Access States

All instructions which use memory locations as operands follow this state diagram. The general execution flow is: calculate address (if necessary), fetch memory operand, and finally execute and write-back of the result to memory (unless destination is a register). One important thing to note about this diagram is that in the decode state, there is an automatic fetch to instruction memory. This for fetching immediate data potentially for use in address calculation. In some cases, this may be a dummy read. The square brackets, [], denote a memory reference.

Note 1: Move instructions with memory as destination do not need to have memory contents fetched

Note 2: The difference between the two [Direct] memory references is that in the top case, there is other immediate data being fetched first (e.g. #13 in the instruction MOV #13,[1234]). Thus a second read is necessary to fetch the direct address. An example of the lower case would be MOV [1234],R5

The following is a description of each state:

- Decode
 - Auto-fetch immediate data. Determine memory mode and use of auto-fetched data.
- Fetch Imm Offset
 - If register offset has not already been fetched in decode state, then fetch it.
 - Begin address calculation of offset and first register and store partial sum
- Memory Address Calculation
 - Final memory address calculation state
 - If only using one register (register indirect), add the register and the fetched offset.
 - If using two registers(register indexed), complete partial sum using the required second register.
- Fetch Direct Address
 - Fetch direct memory address specified in instruction.
- Memory Fetch
 - Using the calculated/fetched memory address, the memory contents at the address are fetched.
- Execute
 - Execute the required instruction using the fetched data.
 - If desired, write the result back to memory.

The outer ring of the CPU FSM (figure 5.3.4) automates the programming of the pipeline DMA controllers. The send instruction passes a control word and two memory pointers (start and end). Each memory pointer must be calculated using direct or any register memory mode. The receive instruction is the same except that an additional memory pointer must be calculated to specify the destination start address in packet memory, hence the last three states of the ring). This last address can be specified using any memory mode, hence the more complicated branching structure allowing short cuts through unnecessary parts of the FSM.

Figure 5.3.4: CPU FSM Outer Ring

To simplify the memory interface, all instruction memory references are in words. That is, all memory offsets, direct references and immediate operands are 16-bits. Additionally, all instruction/data memory references are to 16-bit memory locations. Writing an 8-bit register out to instruction/data memory results in a 16-bit value being written. No byte oriented operations will be allowed to this portion of memory. All addressable buffer memory is byte-oriented. All word operations are truncated to conform to 8 bits. Internal memory mapped I/O locations vary from 1 to 16-bits in size. For these internal locations, only the required bits are captured from the data bus. See figure 5.3.5.

Figure 5.3.5: Memory Address Space

Packet memory is implemented as two independent segments of dual port RAM.

The CPU instruction set is CISC based, and has been optimized with the application of protocol manipulation in mind. The CPU has the standard allotment of move, logic, arithmetic, shift, and branching instructions (see references 5, 9 and 34) with a few changes. There are no multiply or divide instructions since these are not needed for byte manipulation, they require large amounts of hardware, and operate slowly. Also, two specialized instructions for checking and generating ATM AAL-1 sequence number protection were added. Finally, send and receive instructions were added to automate the programming of frame transfers to and from the buffers. These two instructions compute and transfer the address and control information to their respective pipeline direct memory access controller. A complete list of instructions and all modes of operation is listed in Appendix A.

5.3.1 Internal Register Architecture

The CPU is able to address 16-bits of address space, half of which is allocated to instruction/data memory. The remaining 32Kbytes is reserved for device and buffer addressing (all I/O is memory mapped), see figure 5.3.5. The register bank contains eight registers: where two are 8-bit and six are 16-bit registers, see figure 5.3.6. Of each group,

two are subdivided in two again, thus the two 8-bit registers can be addressed as four 4-bit registers, and two of the 16-bit registers can be addressed as four 8-bit registers.

This arrangement of registers can allow simplification of byte manipulation. For example, in a control byte, there may be a 4-bit counter, and four bits of other control information. This control byte could be read into one of the 4/8-bit registers and then have it's 4-bit counter incremented separately while logic operations are used to query the other control bits. These two operations can proceed independently with the final results already spliced together. The completed control byte can then be written back to the frame in memory. This subdivision of registers removes the necessity of copying the register twice and masking out the other bits not relevant to the test or computation at hand. It also helps reduce the number of registers needed to perform a particular conversion function.

Figure 5.3.6: Register Bank Size Comparison

Registers 1, 3, 7 and 8 are shadowed so that during interrupt servicing, an alternate set of these registers are used. This is done to avoid implementing a cumbersome stack structure. By not shadowing all registers, unshadowed registers can be used to pass data back and forth between the main program and the ISR.

Four of the 16-bit registers (R5-R8) are designed to be used as pointers with respect to the send and receive instructions. R7 and R8 are meant to be used as start and end pointers while R5 and R6 can be used as index pointers based on the first set or as start and end pointers in their own right. The use of pointers to denote frame boundaries within memory removes the need to transfer frames more than necessary, and simplifies segmentation and reassembly operations. As well, the length of any frame can be computed easily by subtracting the two pointer values.

There are two other special registers in the CPU which are not writeable: IS and IE. They denote the start and end memory locations of the top frame in the input buffer. These two registers can be used to reference the top frame for format queries or for receive commands. They are updated between instructions and are fed directly from the input buffer.

5.3.2 Interrupt Control

For control purposes, the CPU has an interrupt mechanism. Only one layer of interrupts is allowed, that is, only one interrupt can be serviced at once, and all other interrupts must wait until after the service routine has ended before they can be serviced. This one layer of interrupt was chosen principally for speed. Since this processor is designed for use in high-speed communication systems, most interrupt service routines will have to be short and any corrupted frames are simply dropped, rather than fixed, a fast, low latency interrupt mechanism was necessary. The use of register shadowing was chosen to give low latency in the interrupt response. With no stack to contend with, register values are left as they are, and a separate set of registers is used exclusively by the interrupt service routine. Upon completion of the interrupt, the processor switches back to its main register bank. This "expensive" hardware option greatly decreases the interrupt latency of the processor by avoiding a costly series of memory writes. As well, there is no stack pointer or other stack logic to maintain or update, no specialized stack instructions to support, and no stack memory to allocate. The programmer is able to use the full 32Kbytes available for instructions and data storage.

It is occasionally useful for the main program to communicate information to an interrupt

service routine, or to have a routine take over processing at some point and query the main program's situation or status. As such, not all registers are shadowed. In fact several are left unprotected so that they may be used to pass information back and forth between the interrupt service routines and the main program. This also means that those registers which are protected have constant values from one interrupt to the next. If, for example, only one interrupt service routine is running on the processor, then a counter or other status information could be placed in a shadowed register. Upon termination of the service routine, the CPU would resume using its normal set of registers, and the counter/control data would remain, protected, in the shadowed register until the interrupt service routine is triggered again. In a complete system, the CPU could trigger an interrupt service routine in response to CRC errors. By keeping the count of errors in a shadowed register, the increment is accomplished much faster, than fetching and writing it to memory. When the count gets to a certain value, the routine could notify other devices in the system that excessive CRC errors have occurred, and that receiver performance should be increased to compensate. Many such inter-process communication scenarios can be explored with this architecture.

Thus with this interrupt register shadowing mechanism and short interrupt service routines, multiple interrupts makes little sense in this architecture considering the cost in hardware complexity, size and speed.

The CPU can be interrupted based on the following signals:

- External interrupt
- New frame in input buffer
- CRC Error in frame just transferred to packet memory
- ATM HEC error in ATM cell just transferred to packet memory
- Output buffer full

Figure 5.3.7: CPU FSM Interrupt Cycle

- #1:** During the fetch state, the interrupt signal is acknowledged, and program control switches to interrupt mode (i.e. shadow registers are enabled).
- #2:** The interrupt vector table is consulted during the decode state to load the new instruction pointer. Instruction pointer now points to the beginning of the ISR.
- #3:** CPU begins execution of the ISR by fetching the first instruction.

When an interrupt is signaled, the CPU finishes executing its present instruction, and beginning with a new fetch cycle, fetches the start address of the service routine from its interrupt vector table (see figure 5.3.7). The CPU then branches to the fetched start address to begin the interrupt service routine. Thus the maximum latency is 12 cycles: 10 cycles of latency, assuming the interrupt was signaled just after a receive instruction was fetched and decoded, plus 2 cycles for fetching the start address from the interrupt vector table.

5.3.3 Arithmetic Logic Unit (ALU)

The ALU of the CPU is able to perform 4-, 8- and 16-bit operations. The support of 4-bit operations was included so that increments of small counters were possible. It is common in several protocols to include a mod-8 or mod-16 frame count along with other signal and status bits in a control field, usually a byte long. Performing the counter update using only 4 bits protects the other status bits from corruption should the counter value overflow.

Figure 5.3.8 below shows the complete CPU architecture from a data path perspective.

Figure 5.3.8: CPU Data Path

This is the general data path architecture of the CPU (control signals not shown for clarity). The arithmetic logic unit (ALU) is able to drive values into all registers (including temporary ones), the data bus, the address bus and feedback register (for use in further computations). The control logic bus is used to generate data, for shift instructions from the op code, and addresses for the send and receive instructions.

5.4 HDLC Flag Protection

Flag protection (accomplished by bit or byte stuffing) is performed using the Stuff and Unstuff pipeline modules. HDLC bit/byte stuffing is supported since the HDLC protocol is the basis for many different protocols. HDLC or PPP (an HDLC variant) encapsulation is also a common procedure in several Internet request for comments (RFC's), a sort of standard by consensus (e.g. IP or ATM over SONET). In HDLC-like protocols, special flag bytes are always at the beginning and end of every frame. These flag sequences should never occur within the frame, otherwise the receiving station will incorrectly interpret the frame as two separate frames. As the frame is transmitted, it is checked for the flag sequence, and if found, additional bits or bytes are inserted in the frame to prevent accidental flag detection. These stuffed bits or bytes must be removed prior to error checking, or any other frame processing, see figure 5.4.1.

Figure 5.4.1: Bit and Byte Stuffing Example

With bit stuffing, a non-integer number of bytes can be generated. With byte stuffing, the frame will always remain an integer number of bytes in length, but more bits are transmitted than with bit stuffing.

There are two types of stuffing: bit and byte (see reference 12). Byte stuffing is meant for high-speed "octet oriented" links and involves inserting a two-byte sequence when either the HDLC flag sequence or the protected flag sequence is detected. Byte stuffed frames are always an integer number of bytes in length. While this makes the frame longer (in terms of bits) compared to a bit stuffed frame (potentially taking longer to transmit), the processing requirements for unstuffing are reduced. This is very important for high speed links where frame processing is likely to occur using multiple bytes at a time and high

throughput is paramount.

Bit stuffing was included for completeness and to support lower speed links. In this case, a zero bit is inserted after every string of five 1's. With bit stuffing it is possible to have a frame that is not an integer number of bytes. This can lead to smaller frames than with byte stuffing, however every bit must be monitored at the receiver for unstuffing. This granularity is not possible with high speed links. Stuffing was also discussed in section 3.1.4

Both the stuffing and unstuffing modules can be bypassed if the HDLC protocol is not in use. In that case, the modules merely add a one-cycle latency delay through the pipeline.

5.4.1 Unstuffer

The unstuffing module's basic operation is controlled by its internal mode register and bypass signal, from the input DMA controller. The mode register sets which unstuffing unit (bit or byte) processes bytes from the frame. The bypass signal is used to reroute bytes around the unstuffing modules, as shown in figure 5.4.2.

Figure 5.4.2: Unstuffer Data Path

Bypass control is flexible enough to accommodate ATM cell processing. For the input pipeline, the DMA controller can allow unstuffing to occur for:

- the whole cell,
- just the header of the cell,
- or just the payload of the cell (ignore the first 5 header bytes).

This is useful, for instance, in HDLC encapsulated protocols of ATM networks where the encapsulated HDLC frame has been segmented into several ATM cells. The incoming segmented packet can be examined for stuffing on a cell-by-cell basis so that unstuffed segments can be automatically concatenated in memory.

5.4.2 Stuffer

The stuffing module (shown in figure 5.4.3) in the output pipeline allows for stuffing control along similar lines. Stuffing control is centred around the header-trailer module. The header and trailer bytes can be included or excluded from the stuffing, as can the data in between. ATM mode is unsupported since there is no guarantee that the selected 48 payload bytes will remain that number if stuffing occurs.

Figure 5.4.3: Stuffer Data Path

To perform, for example, an HDLC encapsulation, followed by ATM cell segmentation, would require two processors in series, one performing the encapsulation, and performing

the ATM cell segmentation. Supporting this kind of operation would require the processor to monitor when data would exceed the 53-byte cell limit. At this point the extra payload would be prepended to the next cell's payload, and the subsequent extra bits shuffled down from cell to cell. This control would have to be sophisticated enough to generate additional cells, as well, to compensate for the increased protocol data unit size. The way in which padded cells are handled is dependent on the type of ATM service being used as well (i.e. AAL-1, AAL-3/4 or AAL-5). This extra processing would require quite complex dedicated hardware for this specific application. The added complexity to support this one application was considered too much. If small changes were needed in the way the data was processed, this specialized ATM mechanism would be useless. Thus, it was deemed beyond the scope of this processor to support this kind of specialized mechanism.

5.5 CRC

The CRC modules on the input and output pipelines are able to perform both 16- and 32-bit CRC computations, using any 16- or 32-bit polynomial. The two modules are independent allowing each to take on a different size and polynomial compared to the other. As bytes progress through the pipeline, each new byte is passed through the module with no changes, while its value is used to compute a new CRC value. This process is similar to a standard serial CRC implementation except that the hardware has been modified to process 8-bits at a time to increase bandwidth. The method used is based work from references 16, 17 and 24. Due to this structure, the CRC module does not need access to the entire frame at any one time, bytes are simply passed through it one at a time, and the CRC computation proceeds iteratively in a pipeline fashion. For both CRC modules, it is expected that once the CRC polynomial has been set, it will not be changed. Changing the CRC polynomial during a computation, while possible, is not expected to give meaningful results.

For ATM cell processing, an ATM header error code (HEC) CRC module was incorporated in the design (see reference 15). When in ATM cell processing mode, the CRC module counts the first four or five bytes sending them to the ATM HEC checking/generator unit. ATM cell payloads can either bypass or be processed by the main CRC modules. In this

way ATM cell headers and payloads are processed separately.

5.5.1 CRC Theory

The main CRC engines are based on mathematical theory. Each bit in the internal CRC computation register is the modulo-2 sum of several bits of the input and computation register. Which bits are XOR'ed for each bit of the register is controlled a polynomial control matrix. This control matrix is very large, using 128 control signals for CRC-16 and 256 for CRC-32.

The mathematics around generating CRC structures is well defined. Generalized equations exist for CRC engines of n^{th} order polynomials that can compute using k bits at a time.¹

For the equations below:

- G is the generating polynomial of order n .
- $X(t)$ is the contents of the CRC computation register at time t .
- D is the polynomial control matrix, defiend as shown.
- $Z(t)$ is contents of a k -bit register of input bits at time t and.
- t is the time index, incremented by k for each computation.

$$\begin{aligned}
 G &= [g_0 \ g_1 \ g_2 \ \dots \ g_{n-1}] \\
 Z(t) &= [z_t \ z_{t+1} \ \dots \ z_{t+k-1}] \\
 X &= [x_0 \ x_1 \ x_2 \ \dots \ x_{n-1}] \\
 D &= \begin{bmatrix} G \\ GT^1 \\ GT^2 \\ \dots \\ GT^{k-1} \end{bmatrix} \quad T = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 1 \\ g_0 & g_1 & g_2 & g_3 & g_4 & \dots & g_{n-1} \end{bmatrix}
 \end{aligned} \tag{5.1}$$

¹. See reference 16

The generalized equation is:

$$X(t+k) = [x_0 \ x_1 \ \dots \ x_{n-k-1}] [0_{n-k} \ I_{n-k}] \oplus ([x_{n-k} \ x_{n-k+1} \ \dots \ x_{n-1}] \oplus Z(t)D) \quad (5.2)$$

Where: 0_{n-k} is a zeros matrix of size $(n-k) \times (n-k)$

I_{n-k} is an identity matrix of size $(n-k) \times (n-k)$

$\oplus$ means sum modulo-2

For the specific case of $k=8$ used in this processor, the equation for an n -bit polynomial is

$$X(t+8) = [0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ x_0 \ \dots \ x_{n-1}] \oplus [u_0 \ \dots \ u_7]D \quad (5.3)$$

where $u_i = x_{i+n-8} \oplus z_{i+7-i}$

The U matrix is an modulo-2 sum of the input matrix and the last eight bits of X ; a simplification of the right-hand side of the equation.

For illustrative purposes, figure 5.5.1 shows the results for a 4th order polynomial processing 2 bits at a time.

The following example is for a 4th order polynomial ($n=4$) processing 2 bits ($k=2$) at a time.

$$G = \begin{bmatrix} g_0 & g_1 & g_2 & g_3 \end{bmatrix} \quad T = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ g_0 & g_1 & g_2 & g_3 \end{bmatrix}$$

$$D = \begin{bmatrix} G \\ GT \end{bmatrix} = \begin{bmatrix} g_0 & g_1 & g_2 & g_3 \\ g_0g_3 & g_0 \oplus g_1g_3 & g_1 \oplus g_2g_3 & g_3 \oplus g_3g_3 \end{bmatrix}$$

$$= \begin{bmatrix} g_0 & g_1 & g_2 & g_3 \\ g_0g_3 & g_0 \oplus g_1g_3 & g_1 \oplus g_2g_3 & 0 \end{bmatrix}$$

$$X(t+2) = [X_0 \ X_1] \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \oplus [X_2 \ X_3] \oplus [Z_t \ Z_{t+1}]D$$

$$= [0 \ 0 \ X_0 \ X_1] \oplus [X_2 \oplus Z_t \ X_3 \oplus Z_{t+1}]D$$

Figure 5.5.1: CRC Control Matrix Example

In hardware, the D matrix is computed iteratively, where each iteration performs a matrix multiplication by T as is shown in figure 5.5.2. For a limiting case where only a few polynomials can be chosen, significant optimization can occur with the propagation of constants through these matrices. This optimization would reduce the size and complexity of the CRC engines and lead to increased performance.

Figure 5.5.2: Iterative CRC Control Signal Generation

From equation 5.1, note that the D matrix is formed using only the polynomial coefficients. Thus once the polynomial has been set, the D matrix will not change; it will not need to be recomputed. Thus in equation 5.3, the new value for the CRC register (n bits) depends only on itself (n bits) and the input (k bits). For each CRC computation, the D matrix determines which input bits are combined with the X register to produce a new value for X , see figure 5.5.3. For a 32-bit polynomial, there are 256 control signals required to account for all possible polynomials (128 for 16-bit polynomials). The implementation for equation 5.3 is illustrated in figure 5.5.5 where the D matrix is implemented by controlled XOR gates (figure 5.5.4).

Figure 5.5.3: Output CRC-16 Engine

Figure 5.5.4: Controlled XOR Gate

Referring to the previous figure, D determines whether an input bit affects an X register bit:

X is the register bit associated with a particular row

U is the summed input bit and register bit for a particular column

D is the control signal for the intersection of a particular row and column

Figure 5.5.5: CRC-16 Controlled Input Array

This figure has been adapted from figure 2 in reference 16. D input to controlled XOR gate not shown for clarity, see figure 5.5.4.

CRC codes can detect the following types of errors:

- all single-bit errors
- all double-bit errors, as long as $G(x)$ has a factor with at least three terms (e.g. x^n+x^2+1)
- any odd number of errors, as long as $G(x)$ contains the factor $(x+1)$
- any burst error for which the length of the burst is less than the length of the polynomial
- most larger burst errors

CRC codes are susceptible to the addition or deletion of zeros from the start or end of a frame. This is due to a modulo-two summation characteristic, that adding several zeros to an existing sum, yields no change. Thus if the CRC register is all zeros at the start, leading zeros will not change the register. If a received frame is identical to the transmitted one except for extra zero bits at the beginning, the CRC checker would not detect this as an error. In order to guard against these errors, the CRC register can be preset to all ones, and then the final CRC result inverted (1's complement) before transmission. At the receiver end, its register would also be preset to all ones. The final receiver CRC register result is non-zero, but unique for each polynomial, and needs to be programmed.

For the general case, the remainder, $R(X)$, of message $M(X)$, of length k bits, when divided by polynomial $G(X)$ of order n is:²

$$R(X) = \left[X^n M(X) + X^k \frac{L(X)}{G(X)} \right] \quad (5.4)$$

$$\text{Where } L(X) = X^{n-1} + X^{n-2} + \dots + X + 1$$

$R(X)$ is then inverted prior to transmission. This mode of operation can be selected in both CRC modules.

There are several other methods for computing CRC codes. This more brute force method provides a simple, iterative architecture, that can support all possible polynomials. It is

². Equation taken from reference 32

possible to use look-up tables to speed up the computation, or to use a simpler polynomial to give a first approximation of the CRC result, and then use another polynomial on the CRC result to get a proper check.

The less computationally intensive method would be to use look-up tables when computing the next value for the CRC register (see references 6 and 17). In this case, a memory of all possible inputs compared with all possible register contents would be pre-programmed. The speed of such a system can be much faster than one using direct computation, limited only by the access time of the memory device. However, this method requires a very large ROM. For a CRC-16 module, a 24 x 16 RAM (24 address bits, with each location being 16 bits wide) is necessary; the 24 comes from 8 bits for the input, and 16 bits for the CRC register. Such a RAM is impossible to build, presently. The method proposed in reference 17 offers several observations about CRC computation. These observations point to ways of making the look-up table smaller. The method proposes using the U bits from equation 5.3 as the table look-up value. The returned result is XOR'ed with a shifted version of the CRC register. In this way, only an 8 bit value is necessary to address into the RAM, to provide the value needed for the final XOR'ing. Thus, the RAM size is reduced to 8 x 16. The optimizations presented in this reference were taken advantage of in the direct computation version, with the RAM contents being calculated on the fly, rather than beforehand. For the implementation methods attempted in this thesis, synthesis of RAM's was difficult. As such, the direct computation was chosen above this table look-up method.

Another method, proposed in reference 7, is to split the computation into two steps. Use a simpler polynomial to make a first approximation of the result, and then use another polynomial to adjust the approximation to the exact CRC result. It is possible to use certain factors of a polynomial and use fewer terms (spacing the terms out by factors of 8 for 8-bit processing). By computing with fewer terms, the CRC computation logic is simplified. The final result can then be used with the remaining factors to compute the final CRC result. Not all polynomials may benefit from this type of operation, and for the "small" polynomial lengths supported in this processor, it is not necessary to gain this speed advantage. This two-step process is more suited to CRC-64 or CRC-128 where the speed

advantages of processing multiple bits simultaneously are offset by the large amounts of hardware necessary for processing 128 bits.

5.5.2 CRC Input

For the input CRC pipeline module (see figure 5.5.6), ATM header bytes are not passed on through the pipeline by default (this is to aid in cell reassembly functions), however this behaviour can be overridden. In this way, the header is automatically stripped from the cell so that during cell reassembly, cell payloads can automatically be appended together, without shifting payloads around in memory to remove ATM headers. The remainder of the cell (i.e. the payload) can be processed through the CRC-16 or CRC-32 engine. An error in the HEC is signalled to the CPU by way of condition code and an interrupt. In this way the CPU can react to an corrupted ATM cell and drop it from the stream, usually by flushing the cell from the input buffer, and performing pointer management for any other cells affected by the corruption.

Figure 5.5.6: Input CRC Datapath

Frames can flow through either the CRC-16 or the CRC-32 engines. Both engines are fully programmable, and compute using any given polynomial of its respective length. For ATM cell processing, the header is protected using a CRC-8 polynomial. ATM headers are directed to the CRC-ATM module for processing. In this way an ATM header can be checked for CRC errors separate from its payload.

The main CRC engine can carry its computation over from frame to frame. When a receive

command is given, the control word specifies whether to reset the CRC register contents or not for the next frame. If frames are independent of each other this would be the normal operation, however, for segmented frames where the last frame in the sequence contains the CRC error check, it is necessary to carry the CRC computation over the series of frames until the last frame has been received. The assumption is that all frames will be received in the correct order; otherwise the CRC computation will be incorrect.

An example of where this kind of processing is necessary is for the ATM AAL-5 service where a large packet is run through a CRC-32 engine and appended with CRC result. This packet is then segmented into ATM cells. Received cells can be set up to have their header checked and stripped by the ATM module, and their payloads reassembled in main packet memory after being run through the CRC-32 engine. When the last cell has been received, the CRC module indicates success with an interrupt and by setting a condition flag. In either case, the CPU is notified of the CRC result and can process the reassembled packet accordingly.

The CRC module can be programmed to either preset or reset the CRC registers. CRC checking is performed by comparing the CRC register to a check register programmed by the CPU. For basic CRC use, this register should be programmed with all zeros, however there are mechanisms to also protect against the addition or deletion of leading or trailing zeros. In this case, all frames will generate the same, non-zero, value in the CRC register. This polynomial-specific value should be programmed in the check register for proper error detection.

On a side note, the CPU can read the CRC register contents, one word at a time. This is mainly a SAR feature, but it could be potentially used for CRC error correction for some low speed applications. Other possible applications include protocol encapsulation or the cells in frames CRC checking procedure as discussed in references 22 and 41. For protocol encapsulation, preformatted frames could be input to the processor and received through the input CRC generator. The CRC result could then be fetched and appended to the frame before it is further encapsulated with another protocol and transmitted out the output

pipeline. With the large number of bytes added to every frame, this application can only be performed with transmission links well below 200 Mbps or in parallel with other processors.

5.5.3 CRC Output

For the output pipeline, the CRC module computes the error code while the frame is being transferred to the output buffer (see figure 5.5.7). Normally, the entire frame is processed, with options of excluding any autogenerated header or trailer from the header/trailer module. In ATM mode, an ATM HEC is generated using the first four bytes sent to the CRC engine. Whether the first four bytes are generated from the Header/Trailer module, from main packet memory, or a combination of both, does not matter. After four byte transfers, the pipeline automatically stalls transmission by one clock cycle to allow insertion of the HEC into the byte stream. As before, the remaining bytes of the payload can either pass through the main CRC engine or bypass the module completely.

Figure 5.5.7: Output CRC Datapath

As in the input pipeline case, this output CRC module can carry its CRC computation over several frames. The send instruction indicates whether to append the CRC result to the end of the sent packet, and whether to reset the CRC register before sending the next frame. As mentioned in the input case, this gives support for the ATM AAL-5 service by allowing

ATM cells to be generated and have the payloads run through the CRC-32 engine, while the headers are not. The final 44-byte payload is appended with the four bytes from the CRC computation to give a complete cell payload (48 bytes). There is no way to read the output CRC result internally. The result can only be appended to the end of an outgoing frame. It is this end of frame insertion limitation that prevents support of protocol formats such as IP, TCP and UDP. These protocols require that error code be inserted ahead of where processing occurs, requiring a buffer. This buffer is somewhat arbitrary in size, and was left out of the implementation in this thesis.

The CRC module is able to operate in several different modes. Specifically, the CPU can specify whether to preset or reset the CRC registers. Additionally, the final CRC result can be inverted prior to transmission in order to protect against the erroneous addition or deletion of leading or trailing zero bits.

5.6 Header/Trailer

The Header/Trailer module adds a fixed number of bytes to the head and/or tail of each sent packet. Like all other modules, it can be put into bypass mode, where no bytes are added to the sent packet. This module allows faster processing, for example, of packets with canned headers (e.g. a series of ATM cells going to the same destination). The same header information does not have to be written to memory every time a packet is sent, instead the header information is written once to the Header/Trailer module, and after that, every sent packet can include this pre-programmed header data. A block diagram is shown in figure 5.6.1.

The present implementation allows for 14 bytes of header, and 2 bytes of trailer. This was chosen to allow for an entire Ethernet header and trailer to be stored since Ethernet had the largest size for these fields of all supported protocols. The 16-bytes of memory used for this is a small amount and can easily fit into an FPGA or ASIC. The number of stored bytes could be easily increased to account for larger frame headers, at the expense of resources.

Figure 5.6.1: Header-Trailer Datapath

The FIFO is used to buffer frame bytes while header is being injected into the byte stream. Additionally, should the pipeline stall for some reason, the FIFO can keep data flowing from packet memory until either the FIFO fills, or the pipeline resumes.

The module also has a 16-byte FIFO so that as header bytes are injected into the byte stream, data bytes can be buffered from memory into the queue, ready for transmission. This means that the queue can feed the pipeline bytes if a CPU read operation interrupts the flow of data into the output pipeline. The size of the FIFO was chosen so that the entire header memory could be injected, without stalling the data transfer.

5.7 Buffers

Each buffer has a UTOPIA-I interface on its external end to facilitate the input and output of frames from the chip. The buffer size of 8Kbytes was chosen as a reasonable maximum frame size. Frame sizes beyond a couple of thousand bytes usually involve more complex error checking, usually at least CRC-64. Since this processor only supports up to CRC-32, it was decided that this size of buffer was reasonable.

The UTOPIA-I standard was defined by the ATM forum as a chip interconnect bus standard. In its cell variation, entire frames are buffered outside the processor until the processor signals that space is available for a new frame. As mentioned in the beginning of this thesis, the UTOPIA implementation used in this thesis is slightly modified. For the input, frames must be transferred wholly into the processor, so that the start and end of the frame in the buffer can be recorded. This is not always the case in that an external device

could pause transmission of a frame. In order to simplify the interface, this possibility has been removed. The interface is 8-bits wide and runs at 25MHz giving a bandwidth of 200Mbps, easily enough bandwidth to handle an OC-3 signal (155Mbps). The signaling scheme is a basic handshake where the processor can interrupt the transfer at any time, and the external UTOPIA-I device signals to transfer only when it has buffered an entire frame or in the case of a receiver, can buffer a whole frame.

One note about clocking strategies must be made. The implementation in this thesis is premised on the fact that the UTOPIA 25MHz clock is derived from the system 50MHz clock. Full and empty signals are generated asynchronously from the FIFO pointers, and no provision has been made for the two clocks to be asynchronous. This limits the application of the design. Suggestions to correct this oversight are discussed below in section 5.7.3.

The original processor design had two onboard FIFO buffers of 8K each. Due to the limitations of given FPGA technologies, this memory was moved off chip with only the address and control signals at the interface. Ideally, if integrated, this buffer memory would be included on-chip.

5.7.1 Input Buffer

Referring to figure 5.7.1, this 8Kbyte buffer is essentially a FIFO with a UTOPIA interface controlling write access, and selection circuitry for read access. Any segment of the buffer can be read without removing bytes from the queue. This is to allow the CPU to perform an inspection of the control information in a frame to determine how to process it. Additionally, it allows the user to issue several receive commands for a single frame without removing it from the queue until the last transfer is complete. FIFO read access is split between the CPU and the input DMAC. A CPU read operation overrides a DMAC read and the fetched data is written out to the CPU input data bus. DMAC reads are associated with transfers of frames from the buffer to packet memory. Data from this read operation is output to the HDLC flag unstuffer module, the next module in the input pipeline. Frames are only flushed from the input buffer after a receive command from the

DMAC or by an explicit CPU control signal.

Figure 5.7.1: UTOPIA Input Buffer

The input buffer keeps a list of pointers to the start of each frame in buffer memory. This is to delimit received frames in the input buffer, and provide the start and end memory address of the top frame in buffer memory to the CPU. The CPU uses these two pointers for specifying what parts of a frame are to be received from the buffer into packet memory.

The start pointers are kept in a separate FIFO ("Start Ptr FIFO" in the figure). As a frame is flushed from the buffer, the top start pointer is pulled out of this second FIFO and placed in the IS (start pointer) register. The next start pointer (now the new top pointer) is read (but not removed) and is used to calculate the end pointer (IE register) since the start of one frame implies the end of the previous frame. This is illustrated in figure 5.7.2 below.

Figure 5.7.2: Input Buffer Start Pointer Flushing

As a frame is flushed from the data FIFO, the start pointer for the next frame is removed from the start pointer FIFO and captured in the IS register. The new top of the start pointer FIFO is then fed through a subtractor to the IE register. In this way, the start pointer FIFO only contains start pointers to frames other than the top one. Since the end of a frame can be found from the start of the next frame, storage space is saved by only storing one pointer, and computing the other.

The depth of the start pointer FIFO determines the maximum number of frames that can be buffered in the system. A depth of 256 entries was chosen for the start pointer FIFO. This depth of queue allows up to 256 32-byte frames, or 154 ATM 53-byte cells. While filling the queue completely might suggest that the processor is unable to handle the input frame rate, in the case of frame reassembly, several frames could be pulled out of the buffer in quick succession for the construction of a new, larger frame. This ample storage space also allows the processor to become temporarily busy with, for example, a rare external interrupt condition where frames are not processed while the interrupt is being serviced. In this case, there is room for the incoming frames to be buffered for when the processor is able to resume its normal frame processing operations.

Buffer Size (Bytes)	# ATM Cells (Size/53)	FIFO Depth (D) ($2^n > \text{Size}/53$)	# Address Bits ($\log_2 D$)
16K	309	512	9
8K	154	256	8
4K	77	128	7
2K	38	64	6
1K	19	32	5

Table 4: Pointer FIFO Depth Calculation Examples

The size of the buffer and the pointer FIFO depth are related based on the maximum number of ATM cells that can be held in the buffer at any one time. In order for the input buffer memory to be fully used, the size of the pointer FIFO was calculated by determining how many ATM cells would fit in the allocated memory space, then figure out the lowest power of two that would allow the addressing of at least that many frames. Should the input buffer memory depth be halved to 4Kbytes, then the maximum number of ATM cells is 77. This leads to a choice of 128 for the pointer FIFO depth. Choosing 64 instead, uses less hardware, and could lead to more speed since there would be fewer bit lines, but this comes at the price of having part of the input buffer memory unuseable when used in small sized frame networks such as ATM.

5.7.2 Output Buffer

Much like the input buffer, the output buffer (shown in figure 5.7.3) is FIFO based with a UTOPIA interface controlling read-access off-chip. To conform to the UTOPIA interface, the FIFO is 9 bits wide: 8 bits for data, 1 bit for the Start Of Cell (SOC) signal. Frames must still be delimited in memory, but address pointers are not needed. The read circuitry merely waits for a frame to be written to the buffer. Once this is done, it transmits the data off-chip according to the UTOPIA protocol. The 9th bit is used as a frame delimiter and avoids a secondary FIFO, as used in the input buffer case.

This buffer is not addressable by the CPU. This is to ensure that frames are transmitted sequentially, and in order from the FIFO. There is no priority transmission mode where a

new frame for transmission can be inserted at the front of the queue. The CPU can only suspend or resume transmission operations through control signal interaction with the output buffer module.

Figure 5.7.3: UTOPIA Output Buffer

5.7.3 Asynchronous Clock Domains Modifications

The initial implementation of both buffer modules assumed that the UTOPIA clock was derived from the system clock. This implementation lacks flexibility should this processor ever to be implemented as a commercial product.

The problem with having an asynchronous relationship between the two clocks is the generation of signals using registered values from both clock domains. In this case, the signals are full and empty flags for the FIFO's using the read and write pointers (each pointer is updated using a different clock). If the clocks are asynchronous, then their edges are not aligned. An update of the read pointer can come just before a write operation. Normally, a write must be inhibited if the FIFO is full (this is to prevent overrun). If the read pointer increments, the full flag will consequently update. The result of the pointer comparison may complete just as the write event occurs, potentially sending registers into a metastable state, or the full signal may be in the process of fluctuating causing an improper value to be used. In short, signal glitches can occur close to a clock edge causing unpredictable and incorrect behaviour. Below are outlined two design strategies to correct for an asynchronous clock relationship.

The first strategy is to use Gray code the read and write pointers and then pass them across the clock boundary and perform the empty or full comparison within the clock domain that it is being used in. Generally one wants to inhibit writes when the FIFO is full, and reads when the FIFO is empty. To inhibit writes, for example, the design should pass the read pointer across the clock boundary (using a flip-flop to harden the signal to the new clock domain). The resynchronized read pointer would then be compared with the existing write pointer to generate a full signal for use within the write clock domain. A similar procedure would be necessary to generate the empty signal in the other clock domain.

Using Gray Code for the read and write pointers means that only one bit changes whenever the pointer updates. In this way only one pointer bit is changing at the clock edge. Either the old value or the new value is captured and no other. This means that the full or empty signal is either based on the old pointer value, or the new pointer value. This presents no problem to the functionality of the FIFO. For the write case, using an old read pointer means that the FIFO will think that it is more full than it really is. FIFO overrun is not possible. Similarly, checking for an empty FIFO using an "old" write pointer means that the FIFO thinks that it is more empty than it really is. In either case, the full or empty flag could trigger prematurely, but this behaviour would be corrected during the subsequent clock cycles when the newer pointer value is successfully passed across the clock boundary. Gray coding increases the complexity of the comparison and increment hardware, however it is necessary to avoid the problems outlined above.

A subtle problem occurs in the FIFO when it is almost empty. As shown in the figure 5.7.4, it is necessary to prevent the FIFO from reading and writing to the same memory location. In this case, a byte may be left in the FIFO, unable to be removed otherwise. However, both the UTOPIA interface and the internal pipeline signal the end of a frame. Using this information, and provided the synchronized pointer has remained constant, it would be permissible to remove the last byte safely.

Figure 5.7.4: Asynchronous FIFO Behaviour

The write pointer is passed across the clock boundary when it is captured by the synchronizing flip-flop. The synchronized value is then compared with the read pointer to check for the an empty condition. Special circuitry is necessary to remove the final byte from the FIFO if no other bytes are input to the buffer. Metastability is unlikely so only one synchronizing flip-flop is used.

Since the frequencies used in this processor are probably well below the maximum toggle rate of the flip-flop, the chance of it going metastable is small compared to the bit error rate. In this case, it is safe to use a single flip-flop rather than two (as one would need to protect against metastability). Should this unlikely even actually occur, the worst case would be a FIFO overrun or underrun. These events will probably generate frames which are in error, and the design should eventually correct itself as new valid frames are entered into the buffer. As mentioned before, frames which are in error are usually dropped from the network.

While the first strategy is sufficient for the output buffer, a second strategy is also possible for the input. In the input buffer, data is entered byte-by-byte, but is removed frame-by-frame. The key difference from the output buffer is that the read pointer update is a rarer event than the write pointer update. In this case, it would be easy to warn the write control circuitry that a read pointer update is about to occur, and that writes should be inhibited at this point. Once the read pointer has been updated, and resynchronized to the write clock domain, writes to the buffer can be resumed, provided that the buffer is not full.

The combination of these two clocking strategies should be sufficient to handle asynchronous clock relationships for the processor.

5.8 DMA Pipeline Controllers

Each side of the pipeline is under independent control from the CPU. Each side is programmed by the CPU and left to perform its transfer while the CPU busies itself with other tasks. This is the principle reason for the existence of these sub-processors, to relieve the CPU of major processing tasks so that its processing power is reserved for other functions. The controllers act as direct memory access (DMA) devices, transferring blocks of data between memory domains. They also set pipeline options as frame processing occurs. For instance, during ATM cell processing, internal counters count off the five bytes of header to process them separately from the payload. The available options for each side of the pipeline are discussed in more detail below.

5.8.1 Input DMA Controller

This pipeline controller (figure 5.8.2) is relatively simple compared to its output cousin. The DMAC requires four words from the CPU to begin processing:

- control (see figure 5.8.1),
- start pointer,
- end pointer and
- destination (start) pointer.

The frame contents delimited by the start and end pointers are then processed by the pipeline, according to the options set in the control word. Processed bytes are written to packet memory, successively, starting at the given destination pointer. When the end of the frame in memory is reached, the controller signals last byte and the various functional blocks of the input pipeline perform any finishing processing as the end signal propagates down the pipeline. The DMA state machine is shown in figure 5.8.3.

7	6	5	4	3	2	1	0
Mode	Unstuff Data	Unstuff ATM Header	Perform CRC Check	Keep ATM Header	N/A	N/A	Flush Packet

Figure 5.8.1: Input DMAC Command Options

Mode defines the transfer option as normal or ATM. Those options referring to ATM headers are ignored in normal mode. In normal ATM cell processing, the header is stripped from the cell to facilitate reassembly, this behaviour can be overridden using bit 3. Normally a packet is flushed from the buffer after transfer, however if a frame requires multiple receive commands to properly process it, the flush packet signal can be delayed until the last receive command.

Figure 5.8.2: Input DMA Controller

Figure 5.8.3: Input DMAC FSM

FSM remains in Start state until the all memory pointers have been loaded, whereupon it moves to the transfer state. Processing stops when the read pointer equal the end pointer. An internal reset is triggered when the CPU begins to reprogram the DMAC. This reset clears a few internal counters, and triggers the reset of the pipeline modules as necessary.

5.8.2 Output DMA Controller

This pipeline controller (see figure 5.8.5) is easier to program than the input since there is no need to specify a destination. The destination is always the output buffer at the end of the pipeline. Therefore, program the controller only three words are required:

- control word (see figure 5.8.4).
- start pointer and
- end pointer.

The memory contents between the start and end pointers in packet memory are then transferred through the pipeline to the output buffer. Pipeline options are set by the control word. This pipeline controller is more complex than its cousin in the coordination of the pipeline since it must deal with the addition of header and/or trailer bytes to the frame. This requires some end of frame signals to be delayed until trailer bytes have been inserted. This control is handled by extra states in the state machine, see figure 5.8.6. as compared to the input controller.

Figure 5.8.4: Output DMAC Command Options

Mode defines the transfer option as normal or ATM. Those options referring to ATM headers are ignored in normal mode. Normally, a frame is available to the outside world, immediately after being inserted into the buffer. This may not be the case if a particular frame format requires that several send commands be given. In this way, the frame can remain in output buffer memory until it is completely constructed. The end command can be delayed until the final send command.

Figure 5.8.5: Output DMA Controller

Figure 5.8.6: Output DMAC FSM

This FSM operates in the same fashion as the input DMAC, however the timing of certain control signals is changed since extra bytes may be added by the header/trailer module. As such, there are two extra states to create the early and delayed signal behaviour.

5.9 I/O Interface

While not an important section of the processor design, it is a required module. It is expected that this processor will be a part of a larger system and working with other processing elements. The I/O interface would allow communication between the processor and other elements. The bus protocol used here is a standard strobed input/output bus. Two 8-bit input port and two 8-bit output ports (which can each be addressed as one 16-bit port) comprise the I/O interface. Each 8-bit port has its own strobe to signal the capture or driving of data. It is expected that this particular module may change as more development in the processor's design is made. A more robust, or addressable bus protocol may be necessary for a high-speed system of interconnected chips (e.g. VME or PCI).

Additionally, since most internal communication between the CPU and pipeline modules is in the form of memory mapped I/O, connecting other processing devices into a shared memory domain might be another way to facilitate inter-chip communication.

Chapter 6: Implementation Results

This chapter describes the results of the verilog HDL code developed for this thesis. The code was targeted to an FPGA and an ASIC process providing insight into the complications of the design. A complete listing the of the processor verilog code is included with this thesis in CD format.

6.1 *FPGA Results*

The results from FPGA implementation provided much useful information about synthesis issues. The back annotation simulation flow was well defined within the FPGA developement tools. As such, several design faults were corrected at this stage. Apart from pure verilog simulation, FPGA back annotation gave a reliable way help verify synthesis results. The Xilinx 4000XL FPGA family was explored for use in prototyping.

It soon became obvious that one single FPGA would be insufficient. The speed target of 50MHz for most of the design was impossible unless module functionality was reduced, or fewer modules were implemented within a particular FPGA. As such, a multiple FPGA solution seems feasible with one or two pipeline modules per FPGA, and one large FPGA for the CPU. The major cause of timing constraint failure was routing delay. This was especially pronounced in the CRC modules (over 75% of the delay path in some cases). The implemented CRC controlled grids consumed the majority of the FPGA's routing resources. As a result several smaller, stripped down versions of the CRC engines were implemented when combined with other pipeline modules.

The prototype board available for use contained only one FPGA (Xilinx 4062XLA), mounted on a PCI card, with some onboard DRAM. This developement board was delivered six months late, and provided only single port RAM. Attempts to implement

parts of the design failed mainly due to the PCI interface. The interface was not hardwired, and needed to meet a 33MHz clocking constraint. It was not possible to route partial designs to this speed, and running at a slower clock rate only complicated the interface (clock boundary problems). Additionally, when compiling some of the demo designs, it was not possible to meet the timing constraints. Other users of this FPGA board have experienced the same problems, and no solution has yet been reported.

6.2 ASIC Results

When FPGA results did not look promising, the design was ported to a 0.35 micron process. Unfortunately, the next fabrication date was set six months later, so a design fabrication was not possible.

Using synthesis timing-analysis tools, the design was verified at 50MHz. It is important to note that all pipeline modules could be synthesized to 100MHz (twice the specification) within this ASIC process. The CPU was synthesized to a more modest 70MHz. With double the necessary slack it is expected that it would be possible to design the layout for the processor and meet the 50MHz timing specification in preparation for fabrication. Even if the auto-place & route tool is unable to route the design according to timing constraints, there exists sufficient slack within the system to compensate for this. No doubt additional commands or compile-time is all that is needed in order to pass the design's timing constraints.

Verilog back annotation of these tools has been hampered by tool and library upgrades as well as network problems. The majority of verification had to be performed at the pre-synthesis level.

The processor synthesized to roughly 65,000 gates with no buffer ram included in that figure. The process used for synthesis had no library RAM cells, so the small 16x8 FIFO's within the pipeline and the pointer FIFO in the input buffer were instantiated using flip-flops. Given this size, it would be reasonable to build a bidirectional processor with onboard buffer RAM as a single chip.

6.3 Verification of Architecture

Verification of the processor was performed using Verilog simulation. Verification centred mainly around pipeline module functionality. Each module has been tested individually, and together under pipeline control. Various conversion scenarios were simulated at this level, including ATM segmentation and reassembly (SAR), ethernet to HDLC, and ethernet to token ring.

For individual testing, the strategy was to test the basic functionality, and any extreme input combinations for the conversion scenarios mentioned above. Most pipeline modules only react to certain kinds of data, and otherwise just register input to output. This behaviour simplified testing considerably. The CRC modules are based heavily from mathematics, and are built using for loops and parameterized modules. As such, after a few polynomials were tested and verified, it was deemed most likely that all other polynomials would work. Since the structure is so regularized and generic, it is unlikely that one polynomial should work, and not another. Perhaps the most complex modules in the pipeline to verify were the pipeline controllers (DMAC In/Out). These sub-processors had to perform correctly under all conversion scenarios. The orchestrating of control signals to the various pipeline modules was quite complex, and all variations of the control register were tested. Additional testing was performed with respect to the timing of the end of frame signal, propagating through the pipeline, and its interaction with the various pipeline control signals. Basic functionality of the CPU instruction set was performed as well, including the development of an assembler.

Verilog code development was performed with synthesis in mind. Periodically, throughout the development of this processor, Verilog code was synthesized to assure that synthesis was possible, and that results were reasonable. For smaller sub-modules, visual inspection was possible, while for larger modules, such as the CRC modules, back annotated verilog code was used (FPGA only).

The use of dedicated ASIC's for protocol processing developed out of a need for throughput to handle high-speed networks. This led to converter ASIC's being designed for use in all aspects of protocol conversion. The advantages of a generic, programmeable architecture have been pointed out by others and are obvious.

The major innovation of this thesis is an architecture to perform this conversion for a wide variety of protocols, at a reasonable throughput. This architecture is a mix of dedicated hardware for high-speed processing, and programmable CPU control to allow for decision making within the conversion process. A dedicated ASIC is able to outperform this architecture in terms of throughput, but is unchangeable, and not dynamic. The incorporation of CPU control slows down dedicated processing, but not to anywhere near the level of unassisted CPU processing.

With certain protocols being used outside of their usual roles in networks, (e.g. PPP and HDLC) a frame processor can be useful. Non-programmable devices may make certain assumptions about a network that they are connected to, and this may lead to problems with interfacing. Able to adapt to new roles with new software, a frame processor can become an essential building block of a new network device design.

An investigation of protocol standards showed that many protocol frame formats are similar and follow predictable patterns. This allowed development of a programmeable pipeline to handle common, processing tasks, leaving an on chip CPU time to control the specifics of the conversion at hand.

While pipelined frame processing is not a new concept, aspects of the pipeline as presented

in this thesis are. The general purpose CRC modules were adapted from existing designs for dedicated CRC engines. The reprogrammability of the module for any polynomial is unique to this thesis.

The Header/Trailer module is also an innovative part of the pipeline, allowing seamless addition of bytes to the data stream without CPU intervention. Also, the 1:2 ratio of the interface clock to the pipeline clock is important. This ratio creates slack in the processor's pipeline to allow for complex manipulation of frames while still allowing the processor to keep up with the input data stream. With asynchronously related clocks, the interface clock could be increased to provide higher throughput for cases where the amount of CPU-frame interaction is limited. This creates a non-standard interface, but this illustrates the flexibility of the processor.

The development of the instruction set for the on chip CPU of this thesis was placed under much scrutiny.

- Standard instruction sets from generic processors were analyzed and unnecessary instructions were removed.
- Additional specialized instructions were added to aid in protocol processing, and pipeline programming and control.
- An assembler was developed using a simple search and replace algorithm.

Another innovative part of the CPU is its partial integration with the pipeline.

- The mapping of memory pointers to dedicated registers within the CPU helps to streamline input buffer access.
- This also saves CPU processing time by removing the need to fetch and store these pointers manually.

The CPU has registers ranging from 4 to 16 bits. The four-bit registers are also mapped as eight-bit registers. This allows a byte to be loaded into one of these registers so that each half segment can be processed independently without affecting the other. Many of the remaining design choices in the CPU were based on existing techniques for standard generic processors. Those techniques which emphasized speed were selected and implemented. For example, the use of register shadowing instead of a stack, for interrupt

processing, and the auto-memory fetch during the decode state of the instruction cycle to speed up instruction processing.

The processor architected in this thesis was simulated and implementations were attempted in an FPGA and ASIC process. The results from the ASIC simulation and layout are promising, showing the processor running at 50MHz with an area of just over 65K gates.

7.1 Processor Extensions and Future Development

The following is a list of possible extensions to the processor to increase its functionality, speed or protocol base:

- CRC external bypass option (for external hardware error protection)
- Incorporate more error checking methods in CRC modules
- Variable ATM header size
- Move to 16-bit pipeline with UTOPIA-II interfaces.
 - Modification of all parts of the pipeline to perform 16-bit operations to double the pipeline bandwidth
- Use of a content addressable memory (CAM) structure to facilitate fast address translation, etc.
 - Possible swap-type move instruction
 - branch using fetched data as address
- Additional CPU interrupt conditions for critical errors
- Modify design to allow for a generic "drop in" CPU module
- Include additional pipeline modules to provide more processing options

Modification of the CRC module to support checksum error checking would allow the processor to support some processing of TCP/IP-type frames. A more rigorous design approach may be necessary in order for the system to support the generation of this type of frame protection. This operation could be implemented by using an external bypass option for the CRC module. In this case, the pipeline would be diverted to an external error checking device. The device would indicate errors for error protection mechanisms

unsupported by the processor. From the pipeline's perspective, the CRC module is still doing the error checking, the CRC module has merely *contracted* the job out to external hardware. More information on how the internet protocol's checksum is computed is available in reference 4. Support of other, more lower level error protection methods such as Reed Solomon and error correcting codes would be useful.

Allowing a flexible ATM header size could lead to a more generic system. The incorporation of an internet checksum in the CRC module could lead to an auto-reception-type instruction for TCP or IP frames. The header could be processed separately from the data with the execution of only one instruction by the CPU.

To increase processor performance, rather than just increasing the clock frequency, it would be possible to modify the internal pipeline to handle information in word, rather than byte format. Most modules would require little redesign to allow for a 16-bit pipeline. Some care would have to be taken for frames that do not end on a word boundary. Migration to a UTOPIA-II interface would also improve performance, continuing the trend to a 16-bit wide pipeline.

Use of a CAM structure would significantly reduce frame processing times, especially in address translation. This would be equivalent to using a hash table. As it stands, most addresses are converted using a form of table look-up. A CAM would reduce this look-up time by combining the lookup and conversion steps into one. Choice of size and depth would be very important so as to simplify its programming and maximize its usefulness. Additionally, it could be used to perform a number of simultaneous test and comparison instructions where the returned result from the CAM is a memory offset or interrupt vector to a procedure for controlled program execution (i.e. new CPU instructions).

During the development of the processor, certain redundancy and reliability issues were ignored. Interrupt conditions for critical errors, such as FIFO overrun, were not included. While the processor pipeline protects itself against this type of situation, if it were to occur, the CPU would need to take over and flush the FIFO to ensure that corrupt data was not

passed on. More of this kind of recovery control is needed for a complete commercial processor design.

From a broader design perspective, providing the option of inserting a standard CPU macro or embedded core, would significantly reduce development costs for this processor. For companies not willing to design their own CPU, this option would be most advantageous. This would remove the need to develop, support and train software programmers on assemblers and compilers, a major cost savings. Additionally, CPU functionality could virtually be guaranteed with an independent CPU contractor providing support. Many of the CPU's interactions with pipeline devices are memory-mapped I/O, making this part of the integration easier. The use of special dedicated CPU registers (i.e. IS and IE) and specialized instructions can usually be accommodated by embedded CPU designers. If register remapping is not possible simple circuitry could be devised to process offsets to these registers to provide functionality, and help offset some of the inherent loss in efficiency. Additionally, assemblers for the embedded core would have to be somewhat retargetable to account for the added instructions.

It has also been suggested that the inclusion of scrambling, encryption and interleaving of frames would enhance the usefulness of this processor. While these issues were not addressed in this thesis, these sorts of extensions to the processor pipeline would be easy to implement. The interface between all pipeline modules is standardized, and the pipeline controllers are programmed to rely on feedback from the pipeline, rather than on an internal time delayed response based on the number of pipeline modules.

Bibliography

- 1) Akhan, M. B. and Tedekcioglu, O. A., A RISC based protocol converter for IBM mainframes. *Microprocessors and Microsystems* 19(4):223-228, May 1995.
- 2) Black, Uyless, "Data Link Protocols" Englewood Cliffs, New Jersey, Prentice Hall, 1993.
- 3) Braden, R., Editor, "Requirements for Internet Hosts -- Communication Layers". Internet Engineering Task Force, RFC 1122, October 1989.
- 4) Braden, R.T., Borman, D.A., Borman, Partridge, C., "Computing the Internet Checksum", RFC 1071 , September 1988.
- 5) Feldman, J. and Retter, C., "Computer Architecture: A Designers Text Based on a Generic RISC" Toronto, McGrawhill, 1994.
- 6) Feldmeier, D. C., Fast Software Implementation of Error Detection Codes. *IEEE Transactions on Networking* 3(6):640-651, December 1995.
- 7) Glaise, R. J., A two-step computation of cyclic redundancy code CRC-32 for ATM networks. *IBM Journal of Research & Development*, 41(6):705-9, November 1997.
- 8) Gross, G., Kaycee, M., Li, A., Malis, A., Stephens, J., "PPP Over AAL5", RFC 2364, July 1998.
- 9) Heath, Steve, "Microprocessor Architectures RISC, CISC and DSP 2nd edition" Toronto, Butterworth-Heinemann Ltd., 1995.
- 10) Heinanen, J., "Multiprotocol Encapsulation over ATM Adaptation Layer 5". RFC 1483, July 1993.
- 11) Hornig, Charles, "A Standard for the Transmission of IP Datagrams over Ethernet Networks", RFC 894, April 1984.
- 12) ISO standard 3309:1993, "Information technology -- Telecommunications and

information exchange between system -- High-level data link control (HDLC) procedures -- Frame structure"

- 13) Laubach, M., Halpern, J., "Classical IP and ARP over ATM", RFC 2225, April, 1998.
- 14) Murakami, K., Maruyama, M., "MAPOS - Multiple Access Protocol over SONET/SDH Version 1", RFC 2171, June 1997.
- 15) Ng, Seong L. and Dewar, B., Parallel realization of the ATM cell header CRC. *Computer Communications* 19(3):257-263, March 1996.
- 16) Pei, T. and Zukowski, C., High-Speed Parallel CRC Circuits in VLSI. *IEEE Transactions on Communications*, 40(4):653-657, April 1992.
- 17) Perez, A., Wismer and Becker, Byte-wise CRC Calculations. *IEEE Micro*, 3(3):40-50, June 1983.
- 18) Postel, J. "A Standard for the Transmission of IP Datagrams over IEEE 802 Networks", RFC 1042, February 1988.
- 19) Postel, J., "Internet Protocol", RFC 791, September 1981.
- 20) Postel, J., "Transmission Control Protocol", RFC 793, September 1981.
- 21) Postel, J., "User Datagram Protocol", RFC 768, August 1980.
- 22) Roginsky, A., Christensen, K. J. and Polge, S., Efficient computation of packet CRC from partial CRCs with application to the Cells-In-Frames protocol. *Computer Communications* Vol. 21: ppg. 654-661, 1998.
- 23) Romkey, J., "A Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP", RFC 1055, June 1988.
- 24) Sait, S. and Hasan, W., Hardware Design and VLSI Implementation of a Byte-Wise CRC Generator Chip. *IEEE Transactions on Consumer Electronics*, 41(1):195-200, February 1995.
- 25) Simpson, W., Editor, "The Point-to-Point Protocol (PPP)", RFC 1661, July 1994.
- 26) Simpson, W., "PPP in Frame Relay", RFC 1973, June, 1996.
- 27) Simpson, W., "PPP in X.25", RFC 1598, Mach 1994.
- 28) Simpson, W., "PPP over SONET/SDH", RFC 1619, May 1994.
- 29) Simpson, W., Editor, "PPP in HDLC Framing", RFC 1661, December 1993.

-
- 30) Simpson, W., Editor, "PPP LCP Extensions", RFC 1570, January 1994.
 - 31) Sklower, K., Lloyd, B., McGregor, g., Carr, D., Coradetti, T., "The PPP Multilink Protocol (MP)", RFC 1990, August 1996.
 - 32) Stallings, William, "Handbook of Computer Communicatoin Standards Vols. 1-3" New York, New York, Macmillan Publishing Company, 1987.
 - 33) Stallings, William, "ISDN An Introduction" New York, New York, Macmillan Publishing Company, 1989.
 - 34) Vranesic, Zvonko, and Zaky, Safwat. "Microcomputer Structures" Toronto, ON, Saunders College Publishing. 1989.
 - 35) UTOPIA-I standard, ATM Forum
 - 36) UTOPIA-II standard. ATM Forum

World Wide Web references:

- 37) Agrawala, Ashok K. [agrawala@cs.umd.edu]. "CSMC 417" [<http://www.cs.umd.edu/class/spr1998/cmcs417-0101/98S17/index.htm>]. March 1998.
- 38) CISCO Systems Inc. "Internetworking Technology Overview" [<http://www.jenntel.com/Tutorials/routing/75818.htm>]. 1995.
- 39) InfoWin. "ATM References" [<http://www.dit.upm.es/infowin/atmeurope/atmrefs.html>]. 1999.
- 40) International Engineering Consortium. "Web ProForum Tutorial: SONET" [<http://www.webproforum.com/sonet/index.html>]. September 1999.
- 41) Roberts, Lawrence G. [lroberts@packet.cc]. "Request for Coordination of Cells In Frames Specification" [<http://www.ziplink.net/~lroberts/Atmf-961104.html>]. August 1996.

Appendix A: CPU Instruction Set

A.1 Instruction Set:

The following notation has been used for operands. If no key words are specified, the instruction has no operands.

- **reg:** register (one of 8 general purpose registers)
- **imm:** immediate data (user specified number)
- **mem:** memory address (direct, register indirect, or register indexed)

All instructions follow the format of:

instruction source, destination

For some instructions this means that data from both the source and destination locations is used in the calculation and the result stored in destination. A '/' between key words indicates possible options for the source or destination operand. No instruction is allowed to specify two memory locations as operands.

A.1.1 Logical

- **And** reg/imm, reg/mem
- **Or** reg/imm, reg/mem
- **Xor** reg/imm, reg/mem
- **Not** reg/imm, reg/mem
- **Shift left** imm, reg/mem
- **Shift right** imm, reg/mem
- **Arithmetic shift left** imm, reg/mem
- **Arithmetic shift right** imm, reg/mem
- **Rotate left** imm, reg/mem
- **Rotate right** imm, reg/mem

- **Bit Test** imm, reg/mem
- **Test** imm, reg/mem

A.1.2 Arithmetic

- **Add** reg/imm/mem, reg/mem
- **Subtract** reg/imm/mem, reg/mem
- **Compare** reg/imm/mem, reg/mem

A.1.3 Interrupt

- **Software interrupt** imm
- **Return from interrupt**
- **Set interrupt mask**
- **Enable interrupts**
- **Disable interrupts**

A.1.4 Execution Control

For the following table, the following flag codes are defined:

- Carry: C
- Overflow: V
- Negative: N
- Zero: Z
- DMA Transfer complete: D
- CRC Error: E
- New Input Frame: F
- HEC Error: H

Branch Condition (16-bit offset unless otherwise specified)	Condition
Branch always (direct or 16-bit offset)	$l = 1$
Carry clear (higher or same)	$c = 0$
Carry set (lower)	$c = 1$
Zero (equal)	$z = 1$
Not zero (not equal)	$z = 0$

Table 5: Condition Codes for Branch Instruction

Branch Condition (16-bit offset unless otherwise specified)	Condition
Greater than or equal to zero	$n^v = 0$
Greater than zero	$z \mid (n^v) = 0$
Higher	$c \mid z = 0$
Less than or equal to zero	$z \mid (n^v) = 1$
lower or same	$c \mid z = 1$
Less than zero	$n^v = 1$
Negative	$n = 1$
Positive	$n = 0$
Overflow set	$v = 1$
Overflow clear	$v = 0$
Input DMA transfer complete	$d = 1$
Input DMA transfer not complete	$d = 0$
Input CRC error	$e = 1$
No input CRC error	$e = 0$
New input frame	$f = 1$
No new input frame	$f = 0$
Input transfer complete with CRC errors	$d \ \& \ e = 1$
Input transfer complete with no CRC errors	$d \ \& \ \sim e = 1$
Input transfer complete with HEC errors	$d \ \& \ h = 1$
Input transfer complete with no HEC errors	$d \ \& \ \sim h = 1$
Either CRC or HEC errors	$e \mid h = 1$
No CRC or HEC errors	$e \mid h = 0$
Input transfer complete with no CRC or HEC errors	$d \ \& \ \sim(e \mid h) = 1$
Input transfer complete with either CRC or HEC errors	$d \ \& \ \sim(elh) = 0$
HEC errors	$h = 1$
No HEC errors	$h = 0$

Table 5: Condition Codes for Branch Instruction

A.1.5 Other

- **Move** imm. reg/mem
- **Move** reg. mem.reg
- **Move** mem. reg
- **Disable input/output UTOPIA interface**
- **Flush input/output frame from buffer**
- **Pause input/output pipeline**

- **Abort input/output pipeline transfer**
- **SNP check** reg/mem
- **SNP generate** reg/mem
- **SEND** mem1, mem2
- **RECEIVE** mem1, mem2, mem3

A.2 Instruction Op-Codes

For the following table of op-codes, the following conventions are used.

- cc Condition code. Selects which branch instruction to execute
- R Register selection bit (three bits)
- M Memory mode selection bit (three bits)
- mod Memory offset mode (two bits)
- W Byte (0) or Word (1) operation
- D Direction. 0 means MMM to RRR. 1 means RRR to MMM
- I Immediate value used to determine the size of shift.
- S Shift operation selection bit. Determines which shift operation to perform
- b Bit test bit encoding. Binary value of bit position to test
- Imm Immediate. Immediate data is referenced with this instruction.
- INT Interrupt
- R_S Start register selection bit. 0 = R5, 1=R7
- R_E End register selection bit. 0=R6, 1=R8

The three register selection bits are mapped as follows:

RRR	W=0	W=1
000	R1H	R1
001	R1L	R2
010	R2H	R3
011	R2L	R4
100	R3H	R5

Table 6: Register Selection Bits Mapping

RRR	W=0	W=1
101	R3L	R6
110	R4H	R7
111	R4L	R8

Table 6: Register Selection Bits Mapping

The available memory modes as selected by the three memory bits are:

M_1M_2	$M_3=0$	$M_3=1$
00	R5	R6+R8
01	R6	R5+R7
10	R7	IS
11	R8	IE

Table 7: Memory Mode Bits Mapping

Note: An instruction cannot specify IS or IE as a destination

The memory offset mode (mod) bits are decoded as follows. Their effect is applied to the memory referenced specified by the memory modes bits.

mod	Effect
00	No effect
01	Direct memory reference
10	Add 16-bit offset
11	Treat memory mode bits as register selection bits

Table 8: Memory Offset Mode selection Bits

Note: Memory mode selection bits are ignored when a direct memory reference is selected.

Op-codes:

Instruction	Byte 1								Byte 2							
AND	0	0	0	0	0	0	D	W	R	R	R	mod	M	M	M	
AND Imm	0	0	0	0	0	1	0	W	0	0	0	mod	M	M	M	
OR	0	0	0	0	1	0	D	W	R	R	R	mod	M	M	M	
OR Imm	0	0	0	0	1	1	0	W	0	0	0	mod	M	M	M	
XOR	0	0	0	1	0	0	D	W	R	R	R	mod	M	M	M	
XOR Imm	0	0	0	1	0	1	0	W	0	0	0	mod	M	M	M	
NOT	0	0	0	1	1	0	0	W	0	0	0	mod	M	M	M	
ADD	0	0	1	0	0	0	D	W	R	R	R	mod	M	M	M	
ADD Imm	0	0	1	0	0	1	0	W	0	0	0	mod	M	M	M	
SUB	0	0	1	0	1	0	D	W	R	R	R	mod	M	M	M	
SUB Imm	0	0	1	0	1	1	0	W	0	0	0	mod	M	M	M	
COMPARE	0	0	1	1	0	0	D	W	R	R	R	mod	M	M	M	
COMPARE Imm	0	0	1	1	0	1	0	W	0	0	0	mod	M	M	M	
SNPCHK	0	0	1	1	1	0	0	W	0	0	0	mod	M	M	M	
SNPGEN	0	0	1	1	1	0	1	W	0	0	0	mod	M	M	M	
MOVE	0	1	0	0	0	0	D	W	R	R	R	mod	M	M	M	
MOVE Imm	0	1	0	0	0	1	0	W	0	0	0	mod	M	M	M	
SHIFT	1	1	0	0	S	S	S	W	I	I	I	mod	M	M	M	
BRANCH	0	1	0	1	1	cc	cc	cc	cc	cc	0	0	0	0	0	
BIT TEST	0	1	1	0	0	0	b	W	b	b	b	mod	M	M	M	
TEST	1	0	0	1	0	0	0	W	0	0	0	mod	M	M	M	
SEND	0	1	1	0	1	0	0	0	0	0	mod _{RS}		mod _{RE}		R _S	R _E
REC	0	1	1	1	0	0	0	mod _{IS}		mod _{IE}		mod		M	M	M
FLUSH REC	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
FLUSH SEND	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
SET INT MASK	1	0	0	0	0	1	0	0	INTERRUPT MASK							

Table 9: Instruction Op-Codes

Instruction	Byte 1								Byte 2							
ABORT REC	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
ABORT SEND	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
INT RETURN	1	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0
SOFTWARE INT	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
PAUSE REC	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
PAUSE SEND	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
DISABLE REC	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
DISABLE SEND	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
DISABLE INT	1	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0
ENABLE INT	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 9: Instruction Op-Codes

The shift instruction selection bits are defined as follows:

SSS	00	01	10	11
0	SHL	SHR	ASR	ROL
1	ROR	na	na	na

Table 10: Shift Instruction Selection Bit Mapping

In instruction memory after the op-code, all immediate, offset, direct memory and control data must be encoded. As such, instruction opcode size can vary from two bytes to ten bytes.

Appendix B: Example Frame Processing Programs

B.1 HDLC Frame to ATM (AAL-5) sample assembly code

```
START      BNP  START                      ; wait for new packet
           REC.CRC.BIT [IS], [IE], [R7]
LOOP1      CMP [DMAC_MEM_LOC], #4          ; wait for header to be processed
           BL LOOP1
           MOV [R7+2], R8                  ; load 16-bit control field
           CMP R8, #CONTROL_VALUE          ; make sure this frame is to be processed
           BNE BADFRAME
           MOV [R7], R8                    ; load 16-bit address
```

<address translation occurs here. Series of compare or table-lookup instructions>

<new address is written to header module>

```
LOOP2      BNFR LOOP2                      ; wait for complete reception of frame
           BFE START                      ; check for CRC errors
           MOV [DMAC_MEM_LOC_END_PTR], R8 ; load end pointer
TEST       MOV R7, R5                      ; copy start pointer
           SUB R8, R5                      ; subtract end pointer from copy
           CMP R5, #40                     ; test if we have less than one cell's worth of data left
           BLE TAG                          ; jump to TAG if we do
           ADD R7, #48                     ; else send complete cell, and move start pointer
SEGMENT    SEND.CRC.ATM.Header [R5], [R5+48] ; send ATM cell
           BRA TEST                        ; branch back for next cell
TAG        MOV #00, [R8+1]                ; write AAL-5 trailer
```

```

SUB #48,R5                ; find out size of pad needed
MOV R5,[R8+3]             ; write pad size to frame
MOV #ATMHEADER [HEADER_MEM_LOC] ; set last frame bit
SEND.CRC.ATM.HEADER [R7], [R7+48] ;send padded ATM cell
ADD #OFFSET,R7           ; shift to other packet memory domain
BRA START
BADFRAME ABORT.REC        ; bad frame, so abort transfer
FLUSH.REC                 ; flush bad frame from FIFO
BRA START                 ; get new frame
    
```

B.2 HDLC to Ethernet sample assembly code

```

START    BNP START
          REC.CRC.BIT [IS], [IE], [R7]
LOOP1    CMP [DMAC_MEM_LOC], #4      ; poll DMAC status reg to detmine
          BL LOOP1                  ; when header has been processed
    
```

<address translation takes place here. New address is written to header module>

```

LOOP2    BNFR LOOP2
          BE START
          MOV [DMAC_MEM_LOC_END_PTR], R8      ; load end pointer
          MOV R7,R5                          ; copy start pointer
          SUB R8,R5                          ; subtract end pointer to get length
          SUB R5, #6                          ; subtract size of header and trailer
          MOV R5, [R7+2]                     ; write data length size to frame
          SEND.CRC.HEADER [R7+2],[R8-2]      ; send frame
          ADD #OFFSET,R7                     ;shift to other packet memory domain
          BRA START
    
```