Université de Montréal

Understanding Retargeting Compilation Techniques For Network Processors

par

Jun Li

Département d'Informatique et de Recherche Opérationnelle

Faculté des Arts et des Sciences

Mémoire présenté à la Faculté des Études Supérieures

en vue de l'obtention du grade de

Maître ès Sciences (M.Sc.)

en Informatique

Octobre, 2003

Université de Montréal

Faculté des études supérieures

Ce mémoire intitulé :

Understanding Retargeting Compilation Techniques For Network Processors

présenté par :

Jun Li

a été évalué par un jury composé des personnes suivantes :

Michel Boyer
Président rapporteur

El Mostapha Aboulhamid
Directeur de recherche

Francois-Raymond Boyer
Codirecteur de recherche

Marc Feeley
Membre du jury

Mémoire accepté le : 26-10-04

Résumé

La croissance rapide du marché des télécommunications entraîne une augmentation de la demande concernant des équipements à bande passante plus large.

Les ASIPs (Application Specific Instruction-Set Processors – Processeurs à Ensemble d'Instructions Spécifiques à une Application) présentent des caractéristiques spécifiques qui mettent spécialement l'emphase sur une haute efficacité avec un coût de développement relativement bas. Ceux-ci deviennent, par conséquent, une solution parmi les plus préférées pour le traitement en réseau.

Après avoir étudié un grand nombre de NPs (Network Processors - Processeurs Réseau), nous avons remarqué que ceux-ci possèdent des fonctions spéciales (telles que : PDU à passage rapide, classification, modification, file d'attente, etc.). Ceci résulte en des fonctionnalités d'architectures particulières différentes des autres processeurs embarqués tel que les PE (Processing Element - Element de Traitement) multiples. Ceci est particulièrement vrai en ce qui concerne le parallélisme ou le pipelinage, et certaines instructions spécialisées.

Pour les raisons ci-dessus, nous faisons face à tous les défis des systèmes embarqués ASIPs et des architectures parallèles lorsque nous explorons la technologie de compilation pour NPs. Le compilateur idéal pour NPs doit avoir une excellente capacité de reciblage, une haute performance de parallélisme, et une haute qualité dans la sélection des instructions spéciales, etc. C'est la raison pour laquelle beaucoup de NPs n'ont pas de compilateurs appropriés.

Afin d'explorer la technologie de compilation pour NPs, nous avons travaillé sur une plate-forme de SoC (System on Chip - Système sur Puce) qui comprend des processeurs multiples DLX et un dispositif simple d'interconnexion. Pour notre étude, nous avons utilisé le processeur DLX comme nœud dans le NP. Nous avons réalisé un compilateur C-DLX qui est une extension du compilateur

reciblable lcc. Nous avons ajouté un nouveau *dorsal* afin de générer du code spécifique au DLX. Ce compilateur fonctionne aussi bien sur un processeur simple que sur des processeurs multiples, et certaines parallélisations manuelles simples pour la plate-forme de SoC ont été examinées avec succès. Mais, la qualité du code produit par le compilateur n'est pas aussi bonne que celle du code assembleur DLX écrit manuellement, et la parallélisation automatique n'a pas été implémentée. Beaucoup d'autres tâches ont été effectuées comprenant l'implémentation d'opérations relatives aux octets, la modification du modèle de transfert de données en anneau à jetons pour convenir au transport de données ou le traitement de paquet de bits.

Dans un proche avenir, nous ajouterons des instructions orientées paquet-de-bits au DLX, afin d'accélérer les applications du processeur réseau, qui impliquent beaucoup d'extraction et de manipulation de bits. Cependant, nous rencontrerons une difficulté dans le développement de compilateur pour produire des instructions orientées paquet-de-bits à partir d'un langage de haut niveau. Plusieurs approches ont été explorées, par exemple les CKFs (Compiler-Known Functions - Fonctions de Compilateur Connues). Nous projetons également d'explorer d'autres compilateurs tels que CoSy, SUIF et autres en vue de résoudre le problème des processus multiples dans les processeurs réseau. Ceci est dans le but de réaliser une méthodologie générale de développement de compilateurs parallèles reciblables pour des NPs.

Mots-clés : processeur réseau, compilateurs parallèles, ASIP, génération de code, compilateurs reciblables, paquet-de-bits.

Abstract

The rapid growth of the telecommunications market increases the demand for wider bandwidth equipment. As a result, high performance NPs (Network Processors), which can meet the demanded requirements are needed.

ASIPs (Application Specific Instruction-set Processors) have special characteristics which emphasize high efficiency with a relatively low development cost. Consequently, ASIPs are becoming one of the most preferred solutions for network processing.

After studying a large number of NPs, we noticed that certain special functions (e.g. fast passing PDU, classification, modification, and queuing, etc.) resulted in particular architectural features different from other embedded processors such as multiple PEs (Processing Elements). This is particularly true of parallelization or pipeline, distributed memories, and certain specialized instructions.

Due to these reasons, we experienced challenges in regards to ASIPs embedded systems and to parallel architectures when we explored NP compilation technology. The ideal compiler for NPs must have excellent retargetability, high performance in parallelization and high quality in special instruction mapping, etc., which explain why many NPs do not have suitable compilers.

To explore NP compilation technology, we worked on a SoC (System on Chip) platform which includes multiple processors and simple interconnected devices. We implemented a C-DLX compiler based on the lcc retargetable compiler, and added a new back-end to lcc. This compiler worked well when tested using our platform on a single processor and multiple processors, and simple manual parallelization for the SoC platform was successfully tested. However, the quality of the compiler-generated code is not as optimized as the hand-written DLX assembly code; automatic parallelization has not been implemented. Many other tasks were accomplished including the implementation of byte related operations

as well as modification of the data transfer model as a token-ring to suit data transportation or bit packet processing. An output device was also added in order to print information to the screen.

Bit-packet-oriented instructions will be added to DLX, in order to accelerate network processor applications, which perform much of the bit extraction and manipulation. However, difficulties in the network compiler development to generate bit-packet-oriented instructions from a high-level language will be encountered. Several approaches have already been explored, such as CKFs (Compiler-Known Functions). We also plan to investigate other compilers such as CoSy, SUIF, etc. with the purpose of exploring the multi-thread approach in network processors. Eventually, a general methodology for developing NP parallel retargetable compilers will be planned out and achieved.

Keywords: network processor, parallel compiler, ASIP, code generation, retargetable compiler, bit-packet.

Understanding Retargeting Compilation Techniques for

Network Processors

Table of Contents

Abstract ..... **V**

Acknowledgements ..... **XIV**

Chapter 1 Intruduction **1**

1.1 Motivations 1

1.2 Achievements 4

1.3 Thesis Outline 5

Chapter 2 Introduction to Network Processors **6**

2.2 Network Applications Profile 8

2.3 Network Processor Basic Architectural Features 14

Chapter 3 Compilation Technology **30**

3.1 Fundamental Conceptions of Compilation Technology 30

3.2 Basic Components of a Compiler 32

3.3 Parallel Architectures and Compilation Technology 37

3.4 Lcc Retargetable Compiler Basic Components Introduction 41

Chapter 4 Main Challenges in NPs Compilation and Related Work **444**

4.1 Network Processors Essential Functions 455

4.2 NPs' Working Mechanism and Challenges for Compilation Technology 466

Chapter 5 Implementation **555**

5.1 DLX Instruction Set Architecture 555

5.2 SoC Multiple Processors Working Platform 59

5.3 Add DLX as New Target to lcc 64

5.4 Test and Result 69

Chapter 6 Conclusion and Future Work **755**

References **800**

Appendix A: DLX instructions **855**

Appendix B: IBM PowerNP Instructions **888**

Table of Figures

Figure 1 OSI 7 layer reference model 10

Figure 2 Alchemy architecture [12] 16

Figure 3 Broadcom architecture [14] 17

Figure 4 MSP5000 architecture [20] 20

Figure 5 MXT4400 architecture [22] 22

Figure 6 NetVortex architecture [23] 23

Figure 7 IBM network processor [25] 24

Figure 8 Intel IXP1200 architecture [26] 25

Figure 9 Motorola C-5 DCP [27] 26

Figure 10 Compiler components 33

Figure 11 Lcc C compilation flow [38] 42

Figure 12 R-instruction format [48] 58

Figure 13 I-instruction format [48] 58

Figure 14 J-instruction format [48] 58

Figure 15 The architecture of DLX nodes [52] 61

Figure 16 Overview of entire platform 69

Acronyms

AH:	Authentication Header
ALU:	Arithmetic Logical Unit
ARP:	Address Resolution Protocol
ASICs:	Application Specific Integrated Circuits
ASIPs:	Application Specific Instruction-Set Processors
ATM:	Asynchronous Transfer Mode
CAM:	Content-Addressable Memory
CPI:	Communications Programming Interfaces
CRC:	Cyclic Redundant Check
DAG:	Direct Acyclic Graphs
DMA:	Direct Memory Access
DRAM:	Dynamic Random Access Memory
DSPs:	Digital Signal Processors
EPC:	Embedded Processor Complex
ESP:	Encapsulated Security Payload
ESR:	Edge Service Router
FIFO:	First In First Out
FPGAs:	Field Programmable Gate Arrays
FPSR:	Floating-Point Status Register
GPPs:	General Purpose Processors
HPF:	High Performance Fortran

IAR:	Interrupt Address Register
IDE:	Integrated Development Environment
ILP:	Instruction Level Parallelism
IP:	Internet Protocol
IPSec:	IP Security
IPv4:	Internet Protocol Version 4
IPv6:	Internet Protocol Version 6
IR:	Intermediate Representation
LAN:	Local Area Network
LER:	Label Edge Router
LEs:	Lookup Engines
LSR:	Label Switch Router
MAC:	Media Access Control
MAC:	Multiply-Accumulate
MIMD:	Multiple instructions, multiple data
MISD:	Multiple instructions, single data
MPI:	Message Passing Interface
MPL:	Message Passing Library
MPLS:	Multi-Protocol Label Switching
MTAPs:	Multi-Threaded Array Processors
NPs:	Network Processors
NPUs:	Network Processor Units
OS:	Operating System

OSCI:	Open SystemC Initiative
P2P:	Peer-to-Peer
POS:	Packet over SONET
PVM:	Parallel Virtual Machine
RISC:	Reduced Instruction Set Computers
RTL:	Register Transfer Level
SDPs:	Serial Data Processors
SIMD:	Single instruction, multiple data
SISD:	Single instruction, single data
SLAs:	Service-Level Agreements
SoC:	System on Chip
TCP:	Transport Control Protocol
TLE:	Table Lookup Engine
TOPs:	Task Optimized Processors
TSP:	Traffic Stream Processor
UDP:	User Datagram Protocol
USB:	Universal Serial Bus
VC:	Virtual Circuit
VCI:	Virtual Circuit Identifier
VLAN:	Virtual Local Area Network
VLIW:	Very Large Instruction Word
VoIP:	Voice over IP
VPI:	Virtual Path Identifier

VPNs: Virtual Private Networks

VPs: Virtual Paths

Acknowledgements

I would like to express my utmost gratitude to my director, Professor El Mostapha Aboulhamid, for all his support since I began my research in the LASSO Lab. I greatly appreciate the opportunity to work on such an interesting project under his valuable guidance. I would further like to thank him for his efforts in directing me in the way of relevant research, generating interesting ideas as well as for his kind encouragements. It would not have been possible to finish this project without his directions or reviews.

I would like to extend my thanks to my co-director, Professor Francois-Raymond Boyer of École Polytechnique de Montréal. He also provided me with significant research direction and reviewed all topics in this thesis. Without his keen guidance, this thesis would also be impossible to finish.

Special thanks go to Professor Michel Boyer and Professor Marc Feeley for their time, patience and evaluation.

Finally, I would like to thank my colleagues Luc Charest and Alena Tsikhanovich as they have built the SoC platform which is my work environment. Michel Reid and Bruno Girodias maintained all computers in working condition as well as solved all technical problems promptly. Quan Xin and Hongmei Sun, for all their fascinating discussions on the topics of this research and helping me broaden my horizons.

Chapter 1
Introduction

1.1 Motivations

In recent years, the telecommunications industry has rapidly expanded. Bandwidth needs are increasingly expanding because new media such as high quality radios, TVs, etc. over IP (Internet Protocol) are required. As a result, the market will not only require wider bandwidth links but also faster and higher level analyses of packets in routers and front-ends of server arrays. Therefore, high performance network processors may fill these needs.

An ASIP (Application Specific Instruction-Set Processor), a processor with a specific instruction set for a particular application domain is becoming the most popular solution for network processing because of its special characteristics. ASIPs lie between ASICs (Application Specific Integrated Circuits) which have the highest efficiency and GPPs (General Purpose Processors) which have the lowest development cost. As a result, ASIPs provide good balance of hardware and software to meet all requirements such as performance, flexibility, fast time response to market, power consumption, etc. Consequently, a networking application may be specified as a software programmable device with architectural features and/or special circuitry for packet processing. A network processor can be regarded as an ASIP for networking application domain [1].

As a network traffic manager, NPs (Network Processors) usually engage several processors working together. Most of NPs own the special instruction set that is designed for data forwarding and other packet oriented operations, in order to complete many diverse functions such as QoS (Quality of Service), compression, security-authentication, encryption/decryption, etc. It should be noted that NPs only run on partially protocol stacks but not all 7 layers, Since NPs are designed to handle PDU (Protocol Data Units) processing by access data stream.

Varieties of special functions of network processors are catalogued as follows and will generate different challenges which will be encountered during the development of NP compilers.

- PDUs passing swiftly: The network processor should be able to deliver the arriving data packets on time. As a PDU is coming and being sent out, the time gap is extremely short, therefore leaving the network processor little time to

accomplish the necessary tasks to be applied to an incoming PDU before re-sending it.

- PDUs classifications: When a PDU arrives, its contents should first be examined to determine the required processing followed by its retransmission. This process is used in firewalling, routing, policy enforcement, and QoS (quality of service) implementation.

- PDUs modification: A PDU may be modified by the network processor. For example, the time-to-live counter of an IP packet will be reduced, or an outgoing label will replace an incoming label in label-switched traffic. Modification is often necessary to recalculate a CRC (Cyclic Redundant Check) or checksum and headers may be removed or added.

- To queue: When the processing speed of the network processor is slower than the data arriving speed, it is then necessary to queue the PDUs. The order of retransmission may not be the same as the order of transmission. For instance, some PDUs will be dropped, and some PDUs may be prioritized over other PDUs. [2]

A network processor has to classify, modify, and queue the PDUs in order to ensure efficiency of the network. Other functions include: compression, policing, traffic metrics, security-authentication, encryption and decryption. Network processors are different from the majority of other embedded processors due to special features, namely the presence of multiple processor elements (PEs), running parallel or organized in a pipeline, distributed memories and bit packet passing. Some of the NPs, such as Agere Routing Switch Processor and Cisco's PXF, use VLIW (Very Large Instruction Word) architectures; VLIW lets these NPs taking benefit from intra-thread ILP (instruction-level parallelism) at compiling time. It may increase utilization of a hardware unit by sharing most of them but also creates more difficulties for compiler developers [1]. On the other

hand, NPs have multiple processors, particularly different kinds of processors working in a SoC (system on chip) that require compilers to have good retargetability.

The market requires a fast response time to create desired and high reliability network equipments. In order to meet this need, embedded software require appropriate compilers to stay away from slow and error prone assembly language development. However, classical compilation technology is insufficient for a particular architecture of network processors and consequently compilation technology for network processors is extremely important and in high demand. Because network processors have special features, we are faced with challenges related to ASIPs embedded systems and those related to parallel architectures when we explore compilation technology. In chapter 4, we will give an overview of related work in this area.

1.2 Achievements

We worked with a cycle-accurate SoC platform, created by our colleagues Luc Charest and Alena Tsikhanovich, which included multiple processors. It was developed using SystemC on Linux PCs. The modeled processors are DLX or ARM architectures (still in development), and both of them can work together. Each processor is an addressable device with a local memory, and processors are connected via a device called ring_device, where every pair of adjacent nodes can send and receive messages concurrently. We will provide a detailed description of the platform in chapter 4. Our colleagues worked on the platform using hand-written assembler codes but this became time-consuming when writing and debugging assembler codes.

A C-DLX compiler was implemented based upon the lcc retargetable compiler. In fact, we added a new back-end that is a DLX code generator. The compiler derived from lcc with its fast, small and convenient features will significantly improve the development and debugging time. This compiler worked well when we tested it using our platform on single processor or multiple processors and successfully tested simple parallel applications on the SoC platform with manual parallelization. We implemented byte related operations, such as LB, LBU, SB, SBU, etc., as well as modified the data transfer model to suit parallel processing. An output device was also added in order to print needed screen information. These modules and functions enable us to simplify and decrease time spent testing and debugging. However, the quality of the compiler-generated code is not as good as the hand-written DLX assembly code.

1.3 Thesis Outline

The remainder of this thesis is organized as follows: Chapter 2 briefly describes network processors (NPs) and its basic features. Chapter 3 presents an overview of compilation technology and discusses typical compiler components. Parallel compilation is then explored and the lcc retargetable compiler is introduced. Chapter 4 presents the main challenges of compilation technology in the domain of NPs and studies related works. We will explain how the DLX back-end can be implemented on lcc, and what kinds of changes have been made on our SoC platform. We also describe the benefits/weaknesses of our compiler and platform in chapter 5. Chapter 6 concludes this thesis; outlines future works as well as reviews and identifies challenges relating to NPs' compilers.

Chapter 2
Introduction to Network Processors

The speed of telecommunications is being changed rapidly. Changes on society and work habits impose on more and more people the requirement to be practically an "infinite" bandwidth. For examples, they consist of Voice over IP, Streaming audio, interactive video, Peer-to-Peer (P2P) applications, Virtual Private Networks (VPNs). New technologies offer more capacity and flexibility for faster and cheaper implementation of new features. All these factors support a booming IT industry, which is growing at a rate of exponential scale. These changes also have a profound effect on the way network processors are developed. A need has arisen for a new generation of network processor architecture that makes use of the technological advances, at the same time

allowing for flexibility of implementation. As for the future network architecture, both revolutionary and evolutionary development has been foreseen. The new network processor architecture must be an evolution of today's network processor. Networks are required to support new protocols, and include diverse services, security, and various network management functions, which effectively handle new applications. The increasing power of processors and continuous development of software technology creates the possibility to develop more powerful and productive applications. The new network should have the capacity, common interfaces, and protocols to support the mutual interaction in a seamless manner.

2.1 A Brief Introduction to Network Processors

"A network processor is a software programmable device with architectural features and/or special circuitry for packet processing; it is an ASIP for the networking application domain" [1]. However, network processors do not provide all solutions to network applications and this definition is reflecting the wide range of programmable architectures proposed for network processing. Therefore, network processors have similar characteristics with many different implementation choices such as network co-processors, communication processors used for networking applications, programmable devices for routing, reconfigurable fabrics and general-purpose processors for routing. Form tasks point of view, network processing responsibilities could be divided into several planes which include forwarding plane to control data packet flow, protocol conversion and a control plane to perform flow management, stacks management, routing protocols, etc.

2.2 Network Applications Profile

By decompounding network applications, we are able to study and to analyze the general network applications in detail. In is section, we are going to discus network applications performed particular operations and network processors' special architectural characteristic.

2.2.1 Network Applications

In order to study network application, the protocol standards must be fully understood. For a clear description of the NPs working mechanism, OSI stack model becomes fundamental knowledge that can be used to map NP's operation into OSI stack layers; the following is a brief description related to the 7 layers in the OSI stack model [3].

Layer 1: The physical layer is responsible for moving bits across a shared media, which is defined by the layer, over which point-to-point links are established.

Layer 2: The data link layer provides a link between two points and can create reliability on top of an unreliable physical link. The data link layer operations have been performed historically in hardware, the network processors are attacking this task. The data link layer provides error detection and correction in low levels. If a packet is corrupted, the data link layer is responsible for re-transmitting this packet.

Layer 3: The network layer is using the point-to-point communication facility, provided by the data link layer, which enables communication between any two computers. This layer is the most complex one among all the layers, because it must address the data packets. It is responsible for the routing of data to a remote

location across the network, and each packet is addressed and delivered in this layer.

Examples of layer 3 processing: IP filtering, Local Network Emulation (LANE), IP fragment re-assembly, multi-protocol over ATM, multicast forwarding and Virtual Private Networks.

Layer 4: The transport layer determines a point of access for higher applications to communicate with the other end-station. The port number and the IP address define a point of access, i.e. a socket. Both UDP and TCP provide a port number to higher layers in order to identify an individual socket. We describe parts of the protocol, which are relevant to network processors only, because much of the transport layer functionality is only executed on an end-station.

Examples of layer 4 processing: TCP stream following, Proxying, stream re-assembly, Rate Shaping, content-based routing, Port Mapping (NAT), Load balancing (LB) and QoS/CoS.

Layer 5-7: session, presentation, and application layers. The application layer may include file transfers and display formatting. An example of application layer protocol is HTTP. Protocols, which are commonly known and considered to be part of the 7 layer, actually may occupy the 5-7 layers. Most of the Session, Presentation, and Application layers tasks are executed on an end-station while network equipment in the fabric may access those layers.

Examples of layer 5 to layer 7 processing: virus detection, traffic engineering, accounting and interruption detection.

Figure 1 shows the data path to describe how network process data over OSI 7 layer model. The gray path represents a physical wire or media connecting devices, and paths in white demonstrate the virtual paths for peer level.

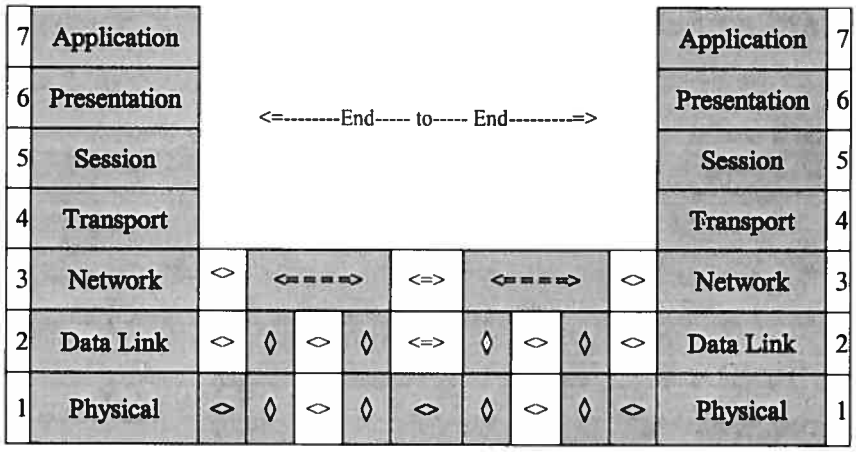


Figure 1 OSI 7 layer reference model

Here a request is made by a user who wants to go out of the network. The request comes in the Application Layer from the upward position of the Application Layer. The Application Layer encodes the request in the form, which might be understood by its peer layer protocol on the right stack. Then it is passed down by the Application Layer to the Presentation Layer, which suits with the data handed from the above layer. Next it packages it in a form, which can be understood by the peer layer protocol on the right stack and which also knows what to do with it.

The higher level data's re-encapsulation process goes on in Figure 1 down the left stack until it reaches the physical layer, which layer the data is passed by a wire, or by a wireless medium to another device, like a router. It accepts the data, brings it up to the Data Link Layer, verifying the bits being received are the transmitted bits from the left by examining the checksum. The payload is passed up by the Data Link Layer to the Network Layer, which examines the header information of the packet and decides what to do with the destination address and other header information. Then it passes to another router to its right. The packet now must travel back down this router's stack to the physical layer so that the next router may take the same process. The packet goes down and across the physical layer wire to the other router, up its layers, and back down again to the right stack. The destination host is ready to process it, taking the coming bits, and packeting them

into a frame, a cell, or a packet. If it is fine, then it is passed up to the Network Layer, which examines the header, added by the original left stack, and the header decides if the packet is good to be passed up. This process repeats with each layer. The header information is examined and is known by the respective layer in terms of how to deal with its payload. The user's original request above the Application Layer of the left stack goes the way to the top Application Layer of the remote machine finally. Either a remote user or a remote program acts on the incoming data and responds. If a response is delivered, the procedure happens again. The response goes from the right side to the Application Layer of the left stack and its user, in the end [4].

2.2.2 Protocol Standards

The following protocol descriptions highlight applications related to protocol standards across different layers but not the detail of each protocol.

- IP Security

IP Security (IPSec) includes two protocols: Authentication Header (AH) and Encapsulated Security Payload (ESP) [5]. IPSec currently supports IP version 4 packets. IPv6 and multicast support is coming later. IPSec is a layer 3 protocol that offers a platform for higher layer protocols. Higher layer protocols can define their own specific security measures. AH or ESP can be implemented in a transport mode that only protects higher layer protocols or it can be implemented in tunnel mode which protects the IP layer and higher layer protocols by summarizing the original IP packet into another packet [1].

- Asynchronous Transfer Mode

Asynchronous Transfer Mode (ATM) is a dedicated connection switching protocol. It is a hybrid form using fixed-size blocks over virtual circuit is chosen as a compromise that gives reasonably good performance for bandwidth traffic. The end stations determine a virtual circuit (VC) or throughout ATM network. As a matter of fact, it is a connection-oriented standard. There are different virtual paths (VPs) or paths between switches in the virtual circuit. To set up a virtual circuit by the control-plane functions, an ATM switch just switches ATM cells from input ports to output ports. The switching is based on referencing a table which is indexed by two fields in the ATM cells: The Virtual circuit identifier (VCI) is an 8-bit VC identifier and the Virtual path identifier (VPI) is a 16-bit VP identifier. Subsequently, a switch may update the new link and then, by changing the VPI and VCI fields, the cell can continue to travel [6].

- Virtual Local Area Network

Virtual Local Area Network (VLAN) is a group of workstations which may be distributed in several physical LAN segments. However, they can communicate with each other like they work within a LAN. Another feature to define VLAN is to check the MAC address of each frame. The switch or router will require the MAC address and verify it, this processing is called MAC layer grouping. Network layers can also be grouped by examining the network layer address so that the VLAN membership can be determined. VLAN can also be defined as an IP multicast group and in this case the router will be a key device to support IP broadcast. In another way, for a VLAN group to be determined for a frame, a unique identifier is needed in the header in order to distinguish the VLAN membership. This identifier is added by the switch [7]. User can get several benefits by using VLAN architecture, such as increasing network performance,

increasing security options, etc. VLAN also present an easy and flexible way to modify logical groups separated in deferent physical LAN segments.

- IPv4 and IPv6

Internet Protocol's (IP) main function is to move packet of data from node to node. Internet Protocol Version 4 (IPv4) is the most broadly used protocol for layer 3 communication. Routing, fragmentation and reassembly, and address resolution protocol (ARP) are the typical working sequential of IPv4. Fragmentation/re-assembly will be applied if an IP packet is bigger than the Maximum Transmission Unit (MTU) of the data link layer. ARP map IP address to physical network address [9].

"Internet Protocol Version 6 (IPv6) is the next generation of protocols designed by the IETF" [10] (The Internet Engineering Task Force) to replace the current Internet Protocol version, IPv4. IPv6 can be seen an increase in the address space from IPv4's 32-bit to 128-bits [11]. As a result, IPv6 has nearly unlimited numbers of available addresses compared to IPv4's. Routing and network auto configuration are two other improvements emphasized by IPv6.

- Transport Control Protocol

Because an unreliable medium such as IP may drop packets, a simple protocol could not solve this problem. Transport Control Protocol (TCP) provides layer 4 communications for higher layer applications over those undependable mediums such as IP. TCP's main function is to verify the correct delivery of data from client to server. TCP can detect errors or lost data and issue retransmission request to ensure that data is correctly and completely received.

A TCP header have information such as the Destination port number, the Header Length, the Source Port number, the Sequence Number and Acknowledgement Number. TCP also contains flags that serve essential functions. For example, the ACK acts as an acknowledgement validation flag, the SYN flag establishes connection, the PSH flag can ensure the receiver passes data to application without delay, the RST flag resets the connection, and the FIN flag indicates that the sender has finished sending data. Aside from this information, TCP also contains valuable information such as window size and acceptable data and information of TCP Checksum [9].

2.3 Network Processor Basic Architectural Features

There are many network processors available on the market and this section includes some of them. From usage point of view, network processors can be separated into three equipment areas that are called as core equipments, edge equipments and access equipments. These areas work for different applications and performance requirements.

Core devices are located at the center of the network. Gigabit and terabit routers are typical core devices. Core devices take important tasks in network traffic management. As a result, the performance will be a critical measurement for the core devices. Core devices have less flexibility compared to others.

Access equipments support different kinds of devices connecting the network. The majority usage of access devices is to combine big amount of traffic streams and to forward them throughout network. Modem, network card as well as computer wireless connector are examples of access devices.

Edge device is not responsible for collecting network routing information, just simply uses the routing information that comes from the up level routers. The

performance and flexibility requirement for edge devices sit between the core device and access devices. Load balancers, edge routers and firewalls are topical edge devices.

From a functionality point of view, network processors can be divided into two categories which are data plane and control plane. Data-plane network processor main task is to forward packets from the source end to the destination end. Data plane algorithms are usually implemented by parallel processors, which need to be performance optimized as they need to decode and move around large amounts of data to satisfy QoS requirements. That is why network processors optimization is focused on data-plane. Control plane network processor has lower performance requirements; they are used to control packet traffic flow. [2]

We studied different network processors' architectures and summarized them in following sections.

2.3.1 Alchemy (Au1000)

The Au1000 is a SoC design for the Internet edge device market. It is a complete SOC based on the MIPS32 instruction set. The Au1000 runs up to 500 MHz. an d its' power consumption is less than 0.5 watt for the 400 MHz version. "MIPS-compatible core provides lower system cost, smaller form factor, lower system power requirement, simpler designs at multiple performance points and shorter design cycles" [12]. In addition, there are special instructions for conditional moves, counting leading and memory prefetch. The Alchemy Au1000 can be programmed in C and support for MS Windows CE, Embedded Linux, and VxWorks operating systems. Development tools include complete MIPS32-compatible toolkit and some third party compilers, assemblers and debuggers

[12]. Manufacturer highlights the Au1000 performance in the spec sheet [12] as follow:

"Highly-integrated system peripherals include two 10/100 Ethernet controllers, USB device and host, four UARTs, AC'97 controller, two SPI/SSI interface. High-bandwidth memory buses offer 100/125 MHz SDRAM Controller and SRAM/Flash EPROM controller." [12]

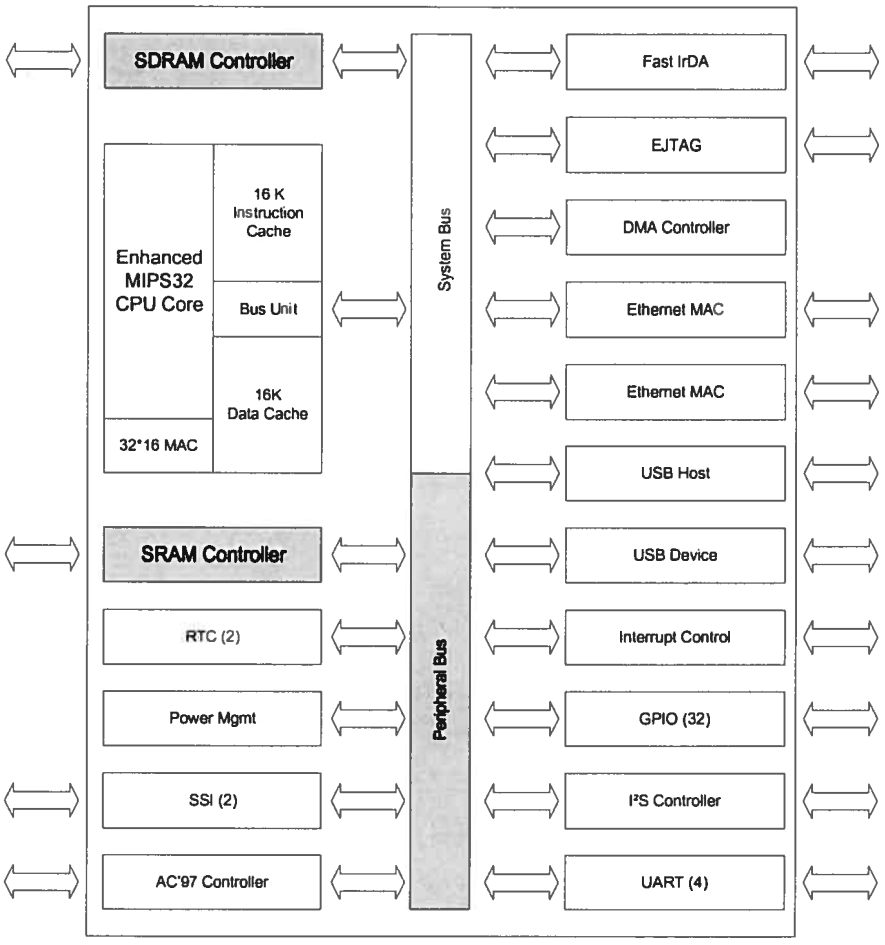


Figure 2 Alchemy architecture [12]

2.3.2 Broadcom BCM-1250

BCM-1250 is the first network processor of Broadcom. BCM-1250 is focused on layers 3-7 processing [1]. The SB-1 can execute two instructions for load or store and two instructions for ALU operations at each clock cycle. Each processor has a 32kb L1 cache and the two cores share a 4-way associative 512kb L-2 cache. Two 64-bit MIPS CPUs (SB-1) of SB-1250 can run at up to 1 GHz and do not use any special instructions for packet processing [13]. The SB-1250 also includes three on-chip Ethernet MACs and two packet FIFOs. A 256-bit bus that runs at half of the processor speed called ZBbus connects the major components of the chip. The Broadcom SB-1250 support operating system for NetBSD, Linux, and VxWorks [14].

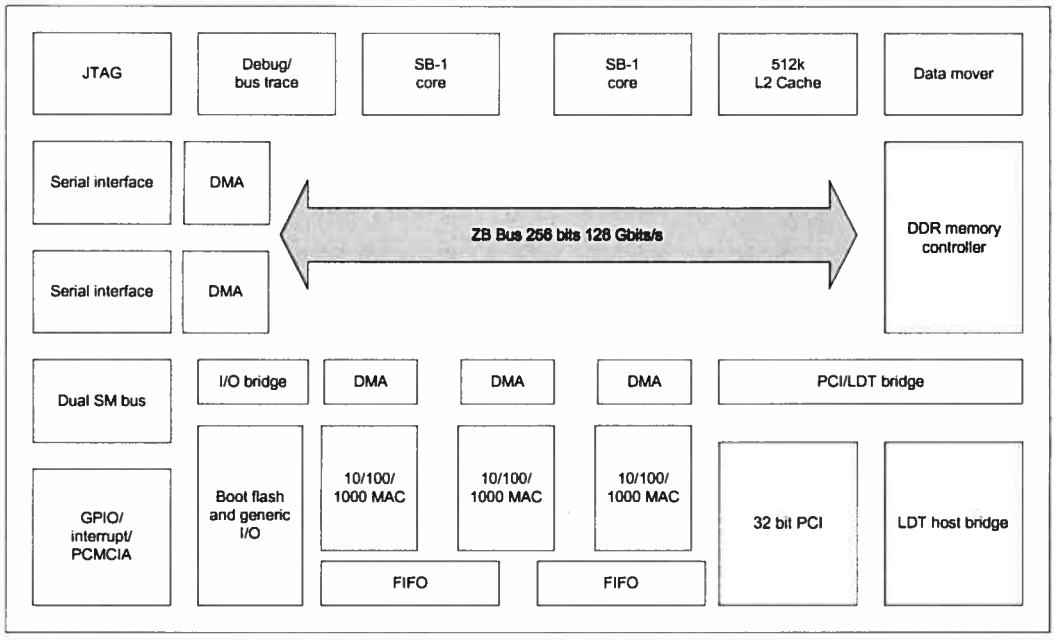


Figure 3 Broadcom architecture [14]

2.3.3 Cisco PXF

Parallel eXpress Forwarding (PXF) use a pair of ICs which are build by Cisco parallel processing technology, The Cisco PXF is an inner product for Cisco edge routers and only be used calculate layer3 data path [15]. The pair of ICs has 32 processors; each processor is a 2-issue VLIW instruction set and brings its own local memory. Particular instructions are used for packet processing. There are 8 stages in the pipeline.

The PXF is benefited from certain features of distributed architectures such as parallel processors. A processor using a parallel array could accelerate a switching path. Another architectural feature, which has been used to improve performance, is distributed memory access and assigns memory to each column of processors as well as allocates independent memory to each processor. Distributed memory access also uses multiple memory banks within a per-column array to optimize the memory access. Modularization of functionality was also used to improve performance. PXF distributes data structures across multiprocessor arrays and by doing so, the distributed task-oriented memory resources have been used to allow parallel processing to fulfill complex tasks [15]. It should be noticed that VPN Acceleration Module (VAM) is not compatible with the PXF processor. The PXF supports Cisco's internal operating system, IOS.

2.3.4 EZchip NP-1

The NP-1 is a single-chip solution developed by Ezchip Technologies Inc., it based platforms can be developed to implement switching, routing, and QoS functions as well as flow based traffic policing, URL switching, and security application. NP-1 is a 10-Gigabit 7-Layer Network Processors.

The NP-1 use EZchip own Task Optimized Processors (TOPcore), TOPcore using customized instruction set and data path for each packet-processing task in order to achieve best performance. TOPs are arranged in a pipelined way and are composed of TOPparse, TOPsearch, TOPresolve, and TOPmodify.

Ezchip has their own development toolkit-Ezdesign, which allows NP-1 application developer to write, test and debug programs when implementing customized applications. EZdesign includes Assembler/Compiler, Cycle accurate Simulator, Traffic and Structure Generators, Applications Library and a Debugger [17].

2.3.5 Xelerated Packet Devices X40 and T40

Xelerated's network processor is based on a fully programmable pipeline. The core block is the PISC (Packet Instruction Set Computing). By using PISC technology, Xelerated provides two network processor chips. The first one is Packet Processor that is called X40, and the second one is Traffic Manager that is called T40.

"The X40 Packet Processor includes 10 pipeline stages and also includes 384k counters and 128k meters for traffic metering and conditioning decisions making. The Xelerator T40 enables advanced traffic management at 40 Gbps wire speed. The forwarding plane of the T40 includes Rx ports, a programmable input pipeline, a queue Engine, a programmable output pipeline, and Tx ports. PISCs can also take tasks such as protocol encapsulation, statistics counting and classification" [19].

Xelerated full production line includes X40 packet processor, T40 traffic manager, Xelerator control plane software, Xelerator forwarding plane applications and development tools. A user needs to program X40 and T40 separately since each pipeline stage executes a different program.

2.3.6 BRECIS MSP 5000

BRECIS provides a network processor that is called MSP5000, which includes a MIPS CPU, two 10/100 Ethernet MACs, a Security Engine, a Voice Engine, and a Packet Engine [20].

MSP5000 offers voice and packet processing software that is supported by multi-service platform applications. BRECIS' development toolkit is called FastStart, which includes a development board and support for Linux, and VxWorks operating systems.

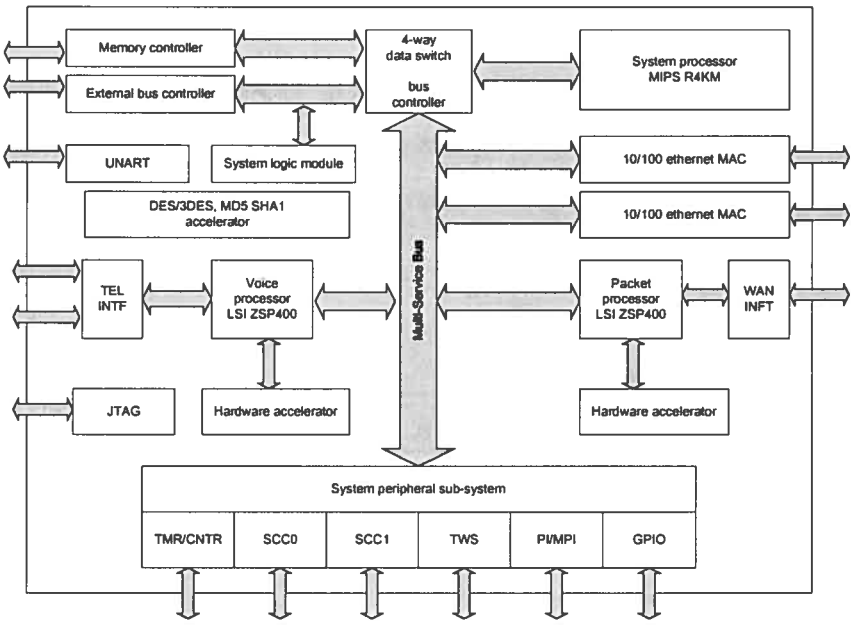


Figure 4 MSP5000 architecture [20]

2.3.7 Mindspeed MXT4400

The MXT4400 Traffic Stream Processor (TSP) platform is offered by Mindspeed Technologies. It can serve wide applications rang from OC-3 to OC-48. To save system developing time, Mindspeed Technologies provides PortMaker software for the MXT4400. PortMaker is an application collection for AAL5 SARing (PortMaker-SAR) and ATM traffic shaping, policing and OAM cell processing (PortMaker Cell Traffic Manager). The source code is opened to customers who want to do further development.

"TSP Board Development Kit provides a chip model, test bench and diagnostic code for hardware design verifications. The TSP Design Toolkit is a powerful development environment for customers modifying PortMaker source code or developing custom applications." [22].

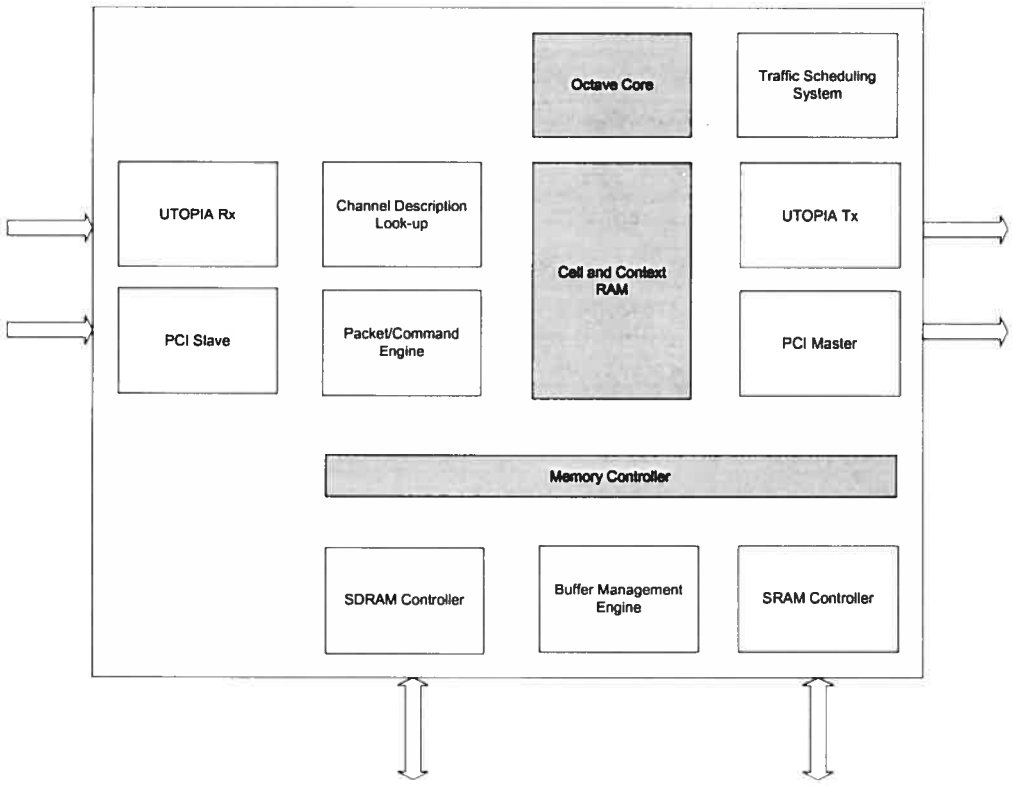


Figure 5 MXT4400 architecture [22]

2.3.8 Lexra NetVortex PowerPlant

Lexra's NetVortex PowerPlant [23] processor integrates sixteen LX8380 32-bit RISC processors on a chip that speed up to 420-MHz clock frequency. Each processor runs a modified instruction set that is based on the MIPS-I instruction set. The instruction set has been extended by Lexra to provide specific support for many functions required from routers. "The new instructions have been added to the MIPS-I instruction set in the LV8380 are optimized for operations such as bit field manipulation, packet checksum, and support for multithreading."[24].

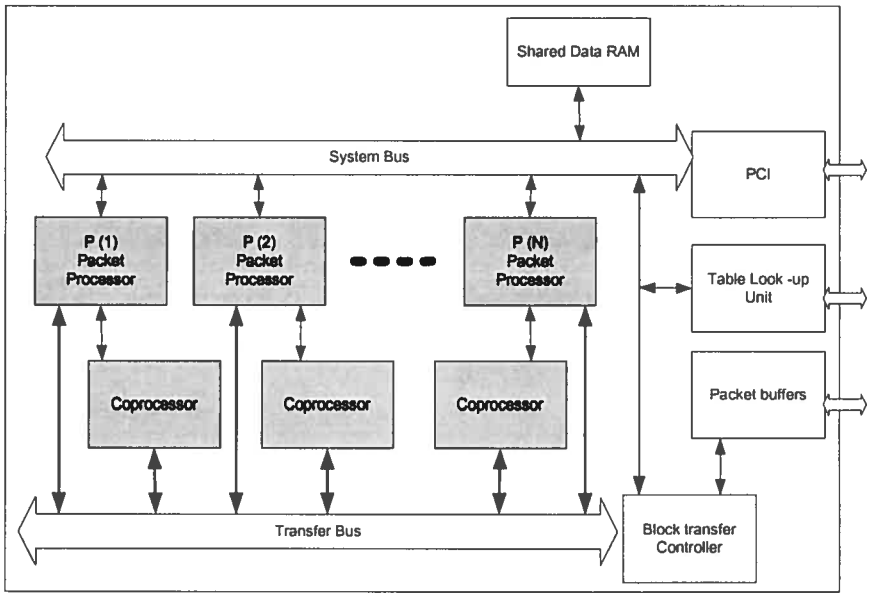


Figure 6 NetVortex architecture [23]

2.3.9 IBM PowerNP

IBM PowerNP provides a multiprocessor solution including an Embedded Processor Complex (EPC) which has a PowerPC core, 7 dedicated co-processors and up to 16 protocol processors. PowerNP supports Packet over SONET (POS) and gigabit Ethernet at 2.5 Gbps. [25]. The 7 co-processors have special functions such as data store, Checksum, Enqueue, Interface, String Copy, Counter and Policy. The IBM Code Development Suite includes an assembler, a debugger and a system simulator.

PowerNP's Functional blocks includes Physical MAC multiplexer that is responsible for moves data between physical layer devices and the PowerNP, and Switch interface which "supports two high-speed data-aligned synchronous link

(DASL) interfaces, standalone operation (wrap), dual-mode operation (two PowerNPs interconnected), or connection to an external switch fabric." [25].

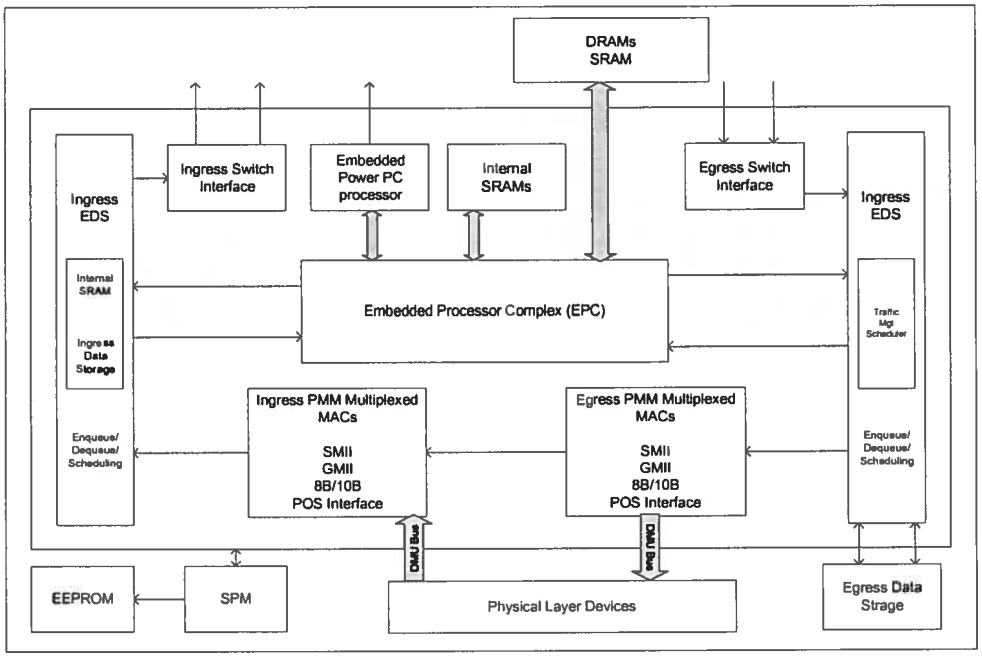


Figure 7 IBM network processor [25]

2.3.10 Intel IXP1200

The Intel IXP1200 can support packet rate up to 2.5 Mpackets/s. The IXP includes a StrongARM controller and 6 micro-engines. Each micro-engine has up to 4 threads hardware support; therefore, there are total 24 threads running on the chip to perform packet processing. IXP 1200 uses IX 64-bit bus that provides high bandwidth connection in the middle of StrongARM controller, memory, micro engines and other devices.

The development tool is called IXDP1200 Advanced Development Platform, which includes available evaluation software, example designs, utilities and libraries, ATM/OC-3 to fast Ethernet IP router example design, WAN/LAN access switch example design, and Windows NT integrated development environment for embedded Linux. Unfortunately, before releasing the C compiler, all programming must be completed by macro-assembly, which is a very difficult task since there are 6 micro-engines working in parallel. However, Intel offers the Integrated Development Environment (IDE) which can provide an assistant to program the device making debugging easier because of "its configurable simulation environment and its visualization" [26].

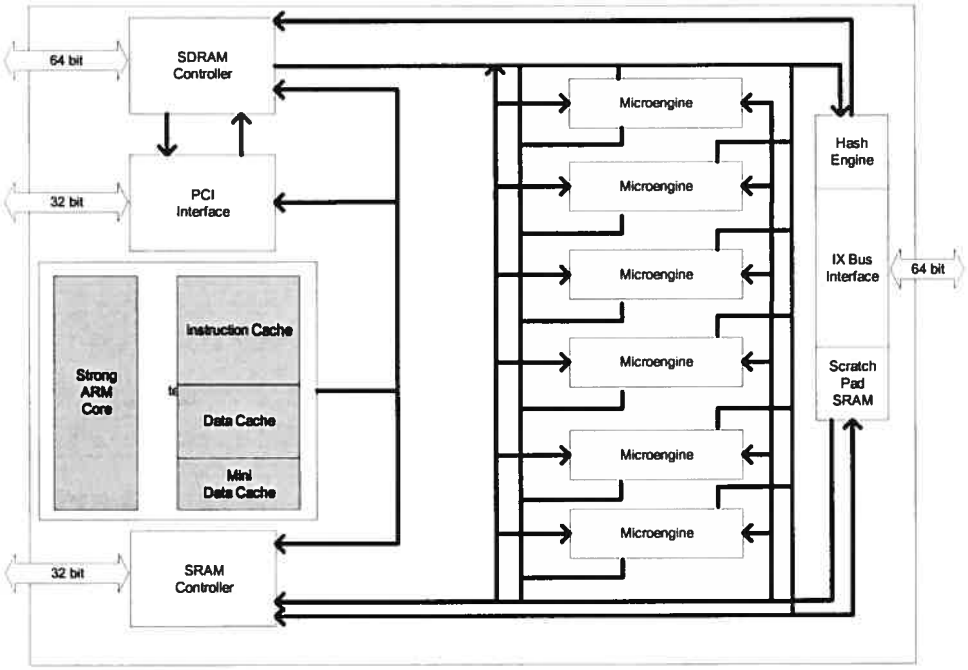


Figure 8 Intel IXP1200 architecture [26]

2.3.11 Motorola C-5 DCP

"The Motorola C-5 DCP is a SoC NP including multiprocessor, it is designed for layers 2-7 processing speed rate up to 2.5 Gbps. 16 channel processors and 5 co-processors for system management. Five co-processors are executive processor, table lookup unit, fabric processor, queue management processor and buffer management processor.

Each channel processor includes a RISC core and two parallel serial data processors which perform communication function. The RISC cores are responsible for classification, policy enforcement, and traffic scheduling" [27].

The C-5 DCP development tools include a C/C++ compiler and users' C-Ware Communications Programming Interfaces (CPI) which abstracts common network task building blocks. [28].

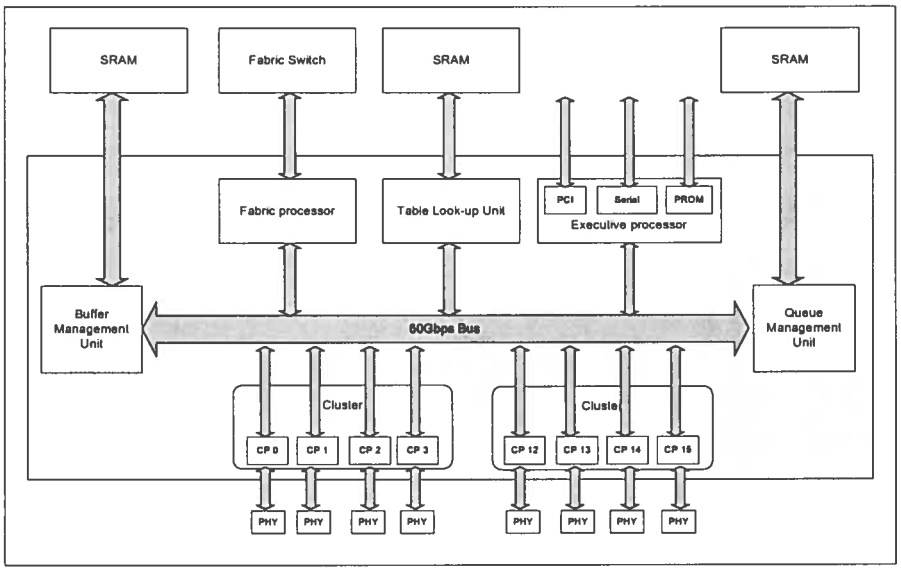


Figure 9 Motorola C-5 DCP [27]

Through the previous NP overview, we can conclude that the network processors are multiple parallel ASIPs which work as an embedded system. The network processors have special features such as multiple processing elements (PEs) which work in parallel by using distributed/shared memory and special instructions.

Most network processors use parallel processing through multiple Processing Engines (PEs) and their architecture can be divided into two types: one is parallel architecture, and the other one is pipeline stage architecture. Parallel NP architecture usually contains bit manipulation circuits to increase packet processing ability; many of these PEs are put into one SoC chip to increase physical space efficiency. In parallel mode, the task scheduler is responsible for transferring packets to different PEs. It keeps track of which PE is available and then sends it a task. Parallel NPs can produce higher bandwidth networking appliances because of their architecture features.

In pipelined architecture, each processor is responsible for a specific packet processing assignment; communication between processors utilizes pipelined way and is very similar to data flow processing: a PE sends a packet to the next PE as soon as a PE finishes the packet processing. Motorola's C-5 DCP is an example of this kind of architecture.

Although NPs have basic features such as special function units, on-chip interconnect methods and the variation in memory architectures. NPs also have other features, One of these common features is a number of distributed processing elements which process packets in a parallel fashion in order to exploit

the parallel nature networking applications. Most companies provide their own programming model which exposes various elements of the hardware. This allows users to be able to map a domain specific language and then automatically map down to machine codes to be implemented in the hardware.

Special instruction is the interface to operate specialized hardware. IBM PowerPC is an example; it defines special instructions such as cache block touch instructions for loading data before the data is needed. This special instruction reduces the memory latency.

In most network-transfer formats, bit-fields are packed together to save bandwidth, however, normal GPP instructions cannot handle these fields efficiently. As a result, Bit-packet-oriented instructions [29] are needed to accelerate network packets processing. Unfortunately, after using this kind of instructions, a big challenge to compilation technology will appear since it is very hard to find which sequence of instructions can map to a bit-packet instruction.

Table 1 is a summary of the basic features of NPs. We can summarize them as follows: NPs are multiple processors working in parallel or in pipeline. Normally they have special instructions and shared (or distributed) memory. These features bring more challenges to compilation technology that will be discussed in chapter 4.

Table 1. Typical NPs comparison

NPs	Intel IXP1200	Lexra NetVortex	Ezchip NP-1	IBM PowerNP	Motorola C-5 DCP
Compilers	C compiler Assembler	C	C/assembler	Assembler only	C/C++
Capacity	2.6 Gbps	>10 Gbps	10Gbps	8Gbps	5 Gbps
OSI layers	2-7	*	2-7	2-5	*
Multi-PE	6 programmable processors	RISC with special instructions	64 task oriented processors	16 processors	16 channel processors
PE Interconnect	FBI Fast Bus Interface	64 bit Vortex bus 427 MHz	*	*	60 Gbps bus
No. of PEs	6	2	64	16	16
Threads/PE	4	2	64	16	16
Memory	Shared 4 kb SRAM FIFOs	1-8	*	2	4

* Means no specification

Chapter 3
Compilation Technology

3.1 Fundamental Conceptions of Compilation Technology

In the early stages of the computer age, many software systems were developed by using assembly language to achieve efficient execution. However, it brought much more workload in coding, debugging and maintenance stages. Presently, most modern computer systems (both of OS and application) are developed by using high-level languages in order to obtain better coding efficiency, higher algorithms abstraction and easier maintenance.

High-level language programs are more human friendly, but they cannot run directly on hardware processors and must therefore be translated into a

corresponding machine code before it is executed. As a result, translators are required in this situation. Here we provide an overview of the internal structure of translators and classify them into several categories.

The development of a translator involves at least three languages, which are source language, target language and host languages. The host language is used to implement the translator, the source language is the resource code to be translated, and the target or object language is the one to be generated. The host and object languages remain hidden usually from a user of the source language. A translator may be defined as a formal function. Its domain is a source language and its range is contained in an object or target language. [30]

One well-established class of translators is the assembler, which maps low-level language instructions into machine language code that can later be executed on a specific processor. The machine-level instructions are mapped 'one-for-one' by individual source language statements. Another is the macro-assembler, which is a variation of the assembler as it also maps low-level language instructions into machine code. Some macro statements may map into a sequence of machine-level instructions effectively providing a text replacement facility to suit the users.

The compiler is another well-established class of translators which maps high-level language instructions into machine code and subsequently machine code can be executed. An individual source language statement usually maps into many machine-level instructions.

The pre-processor maps a superset of a high-level language into the original high-level language or performs simple text substitutions. The high-level translators are useful as components of a two-stage compiling system. Those translators map one high-level language into another high-level language.

The decompiler is a translator which takes a low level language as the object code and then regenerates the source code at a high-level language, but it is difficult to

recreate the original written source code although this can be done successfully for the production of assembler level code.

Microcomputers and embedded systems often use cross-compilers, which generate code for computers and systems other than the host machine since they are too small to allow the translators to generate code on themselves. Self-resident translators are those translators that generate a code for their host machines. The cross-translators may bring additional problems during the transfer of the object code from the donor machine to the object machine that is set to execute the translated program.

The machine code, which is the output of some translators, is loaded in a machine at fixed locations for execution. The other load-and-go translators initiate execution of this machine code but many translators do not produce the fixed address machine code and produce a semi-compiled, binary symbolic or re-locatable form which are closely related to it. Starting from a mixture of source languages it is possible to develop a composite library of special purpose routines as a frequent use for this. The linkers are designed to link together the routines compiled code in this way. It could be regarded as the final stage for a multi-stage translator [31].

3.2 Basic Components of a Compiler

A typical compiler can be divided into two parts that are front-end and back-end. As shown in Figure 10, the front-end is an analyzer that generates an internal representation, which gives correct meanings of the source code. The back-end continually transforms internal representation to the target code. The most important requirement to a compiler is to work correctly which means generated codes must expressing the same effects as specified by the source code.

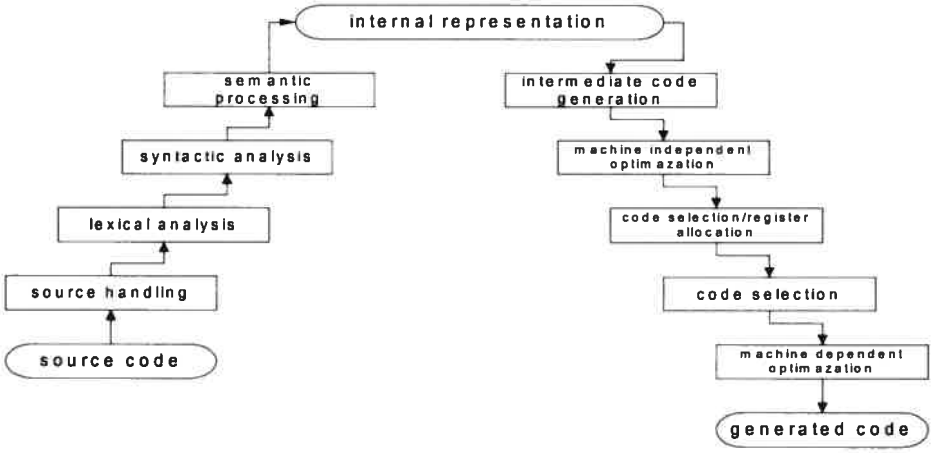


Figure 10 Compiler components

Before it begins analyzing, the front-end normally tokenizes the source codes program and also identifies program constructs. Moreover, the front-end will verify the input code related semantics to ensure they are correct.

The back-end will generate target instructions. In order to perform this activity, the synthesizer component needs to choose a wide range of available instructions and data addressing formats. One way to optimize execution on the target machine is to perform register allocation. By using this method, variables may frequently be accessed from registers rather than accessed from the slower memory to accelerate program-running time. In order to improve the performance of generated code many other optimization techniques may be used to complete this task. These techniques include constant folding, strength reduction, algebraic simplification, tail recursion elimination, copy/constant propagation, common sub expression elimination, dead code elimination; loop unrolling, leaf procedure optimization, invariant code motion, hardware loop optimization, register coloring, resource-based scheduling and peephole optimizations [30].

3.2.1 Source Handling

One of the most important issues for a compiler is to detect as many program errors as possible in running. Thus, source handling becomes very important and its function is to process source code and to supply the lexical analyzer with an input stream of characters from the source code. As a very important feature, a source handler normally includes a suitable interface to report syntax errors. Since the incorrect program fragments have been reported on a regular basis, the source handling would ensure that a diagnostic message reporting correctly. As a matter of fact, most of the errors could be detected in a single compiler run [32].

3.2.2 Lexical Analysis

The lexical scanner plays a very important role in lexical analysis. In order to identify the words, the lexical scanner takes the sequence of characters and it must obey the common rule that a user is not allowed to use system-reserved words as user defined identifiers. For example, a C lexical scanner will identify the string *void* which is a reserved word as a keyword, as a result, we cannot use *void* as a user identifier. However, the string *Number_of_CPU* is not a language-reserved word so it would be identified as a user declared identifier such as variable, constant, type name, etc.

Tokens may finally be represented by an integer sub-range. Lexical scanners usually attempt to determine the longest symbol to match a token. For instance, string "= =" could have been interpreted as two "=". However, because the lexical scanner will use it's rule to determine the longest string, it is matched as "= =". Therefore, the string "*int_number*" will be interpreted as a user declared identifier but not be interpreted as two identifiers. One of the identifier is the reserved word *int,* and the other identifier is a user declared identifier*_number*.

From a language design point of view, certain scanners are not only performing above-mentioned functions but also requiring to providing context sensitive

information. As in C or C++, the scanner must verify the symbol table in order to distinguish user defined type names with variable names. However, this function cannot be fulfilled correctly by checking the spelling or consequences. More importantly, the symbol table constructed has to be accessed during the semantic analysis stage [32].

3.2.3 Parsing

The parser's functions include recognizing a token or word sequence as a valid sentence in the language. In the mean time, these words are organized in a tree to perform representation of the phrase. There may be two ways for the parser to interface with semantic analysis and code generation routines. As a result of a parser, an explicit parser-tree records the structure of the recognized program. Consequently, semantic analysis and code generation will proceed through this tree.

Another situation is that the parser may at appropriate intervals call semantic analysis and code generation routines directly. In this case, a simple one-pass analysis is sufficient and therefore, an explicit parse tree representation is unnecessary. Calling semantic analysis and code generation routines have the overall effect of traversing a virtual tree from time to time but without the overheads of constructing one [32].

3.2.4 Semantic Processing

Semantic analysis verifies the language components recognized during parsing. It should be emphasized that the language components must have well-defined meanings and do not break any language restrictions. As a matter of fact, most languages impose a declaration before the use of a rule [32].

Many languages also include the ideas of scope rule and declaration blocks. In such block-structured languages, the scope of a symbol is limited to the block within which it is declared. The attributes for an identifier declaration are recorded and each event is subsequently examined. The name analysis via an appropriate symbol table organization to reflect program structure and scope rule as adopted in the language. Type compatibility checks may then be performed. When completing the analysis of a scope, the associated identifier entries may be deleted if they are no longer accessible. For each identifier occurrence, semantic processing would determine if an identifier was declared. If so, locating its declaration will disclose its class, type, and the validity of the current usage. At the end of the semantic analysis phase, the meanings of the program have been determined and code generation may thus proceed.

3.2.5 Code Generation

Code generation is the process of implementing an evaluation procedure for the original source program; for example, a particular target computer or environment. The difficulty of the code generation is relative to the difference between the mechanism and structures in the source language and corresponding availability in the target machine. The first important aspect of code generation is the design of a suitable run-time environment. As a result, the source language features and mechanisms are easily supported and eventually implemented in the target environment. These target environments all have their own special machine architectures, register organizations or addressing formats [33]. Execution of the resultant code must produce the same effects as that specified in the original program. This task engages three steps which are memory mapping, register allocation and instruction selection.

3.3 Parallel Architectures and Compilation Technology

A computational problem may be solved by a set of co-operating processors, workstation networks and other embedded systems. The computational resources, processors, memory, input and output bandwidth are collected together. The parallel approach splits the task into smaller subtasks and works simultaneously by coordinated and efficient workers. The granularity of subtasks should be optimized to find a balance between the number of subtasks and the communication frequency.

The computer architectures can be classified as serial and parallel architectures according to the Flynn's taxonomy [34], [35]. The concepts, instruction stream and data stream are the basis of this classification. A computer is operated by a series of instruction sequence and date stream. A data stream is a sequence of data which is manipulated by the instruction stream. There are four categories of combination of instruction stream and data stream but there are no practical examples of the MISD (Multiple Instruction and Single Data) case. SISD (Single Instruction, Single Data): Most of the serial computers belong to this class. Computers of this class can decode only a single instruction in unit time although instructions can be pipelined. Under the direction of a single control unit, a SISD computer may have multiple functional units. SIMD (Single Instruction, Multiple Data): Vector processors and processor arrays belong to this class. A processor array contains many arithmetic-processing units and each one is able to fetch and manipulate its own data as well as execute a single stream of instructions. Multiple processing units manipulate different data and a single operation is in the same state of execution in any time unit. MIMD (Multiple Instruction, Multiple Data): Most multi-computers belong to this group. Every processor executes its own instruction stream with its data stream.

Memory architectures determine the way of processors' communication, so that they can affect writing parallel programs. There are three primary memory

architectures that are shared memory, distributed memory and hierarchy of memory. Shared memory is shared between multiple processors although they work independently, but only one processor at a time accesses the shared memory which is synchronized. The bandwidth of memory is limited by the access speed; if the number of processors is small, it will be fast. The increasing number of processors is dependent on the increasing of bandwidth. Distributed memory: Each of the multiprocessors operates with its own memory independently and data is shared through a communication network and synchronized through message passing. Hierarchy of memory is a combination of shared and distributed memory architecture that contains several shared nodes interconnected by a network. The inter-processor communication is required to transmit data and information between processors and synchronizes node activities.

To program parallel computers, the common approaches are message passing and data passing. Message passing means that each processor has its private local memory, and cooperation is obtained by exchanging messages. A 'receive' operation matches a 'sent' operation. The message passing programming is conducted by linking and making calls to libraries which manage data exchange between processors. The three message passing parallel libraries will be discussed later, they are message passing interface, parallel virtual machine, and message passing library. Data parallel means that data is distributed across processors. Each processor works on a different part of the same data structure. All passing messages are generated by a compiler and they are invisible to the programmer; they are commonly on the top of the message passing libraries. A program is written by using data parallel constructions and compiled by a data parallel compiler, the program is compiled into a standard code with a message passing library call. The High Performance Fortran (HPF) and Connection Machine Fortran (CMF) are two examples of data parallel languages and will later be discussed.

The Parallel programming necessitates a different approach for solving problems; this approach is distinguished from sequential algorithms. There are four distinct stages: Partitioning, Communication, Agglomeration and Mapping (PCAM) in the process of designing a parallel application. I. Foster described them in [34] as follows:

- Partitioning: It decomposes the computation and the data operation into small tasks. We can use domain or functional decomposition in the partition step. The number of processors in the target computer is ignored. Attention is focused on recognizing potentials for parallel execution.
- Communication: This determines the communication, which is required to coordinate the task execution. It chooses the algorithm and the appropriate communication structure.
- Agglomeration: These are the structures of communication and task, determined in the first and second design stages. They are estimated and evaluated according to the performance requirements and implementation costs to decide whether the tasks must be necessary combined.
- Mapping: The full processor utilization and less communication costs are the goals to be obtained for each task design.

Message Passing Programming

The parallel programming model generously used today perhaps is message passing. It creates multiple tasks; a unique name and local data encapsulated identify every task. Send to and receive the messages from the named tasks make the tasks interacting. The programs in the message passing model are written in common high-level languages like C or Fortran.

Several message-passing libraries are widely used today. Programs call library routines to send and receive messages.

Parallel Virtual Machine (PVM)

The feature of PVM is that this library can be used in a heterogeneous hardware environment but communication is slower than the other two following libraries. Some modern features such as dynamic task creation and groups have been implemented. Today, there is a public domain message passing library and nearly all hardware platforms are implemented. Originally, it was developed at the Oak Ridge National Laboratory for clusters of workstations [35].

Message Passing Interface (MPI)

This is a standardized library and is widely used. MPI is available for many hardware platforms and is promising since the number of tasks is fixed while a program is executing unlike PVM.

Message Passing Library (MPL)

Message Passing Library was developed by IBM for the SP-2 supercomputer as a native library and is highly optimized for it. MPL has a similar function to the MPI. In the new version of its operating system and parallel environment of SP-2 it resembles the MPI. It seems as though the IBM supporting MPI and the MPL will be replaced by it [35].

Data Parallel Programming

Data parallelization is another commonly used parallel programming model. It is derived from the application to multiple elements of a data structure. An example is to add the same number to all elements of an array. A sequence of such operations is contained in a data parallel program. Each operation of the data element is an independent task. The natural units of a data parallel program are small. The programmer should provide explanation for how tasks are partitioned, and how data are distributed over processors. Then a normal programming language C or Fortran are translated from the data parallel program with calls to a message passing library or communication through shared memory.

The following two data-parallel languages, based on Fortran, are widely used:

High Performance Fortran (HPF)

HPF is a well-standardized language [36]. Many commercial High Performance Fortran compilers are available. A public one is called Adaptor. The Fortran 90 syntax plus High Performance Fortran data distributing directives and some new language constructs are included.

Connection Machine (CM) Fortran

Connection Machine Fortran was developed for the Connection Machine originally and now spread away and its architecture is independent [37].

3.4 Lcc Retargetable Compiler Basic Components Introduction

Lcc is an open-source ANSI-C compiler developed by C.W. Fraser and D.R. Hanson [38]. Lcc has been ported to several machine platforms such as Alpha/OSF, x86/Win32, x86/Linux, Sparc/Solaris, and MIPS/IRIX. In this project, we use lcc as the base for our DLX compiler since lcc has been widely used and it is simple.

The compilation flow of lcc can be summarized in Figure 11.

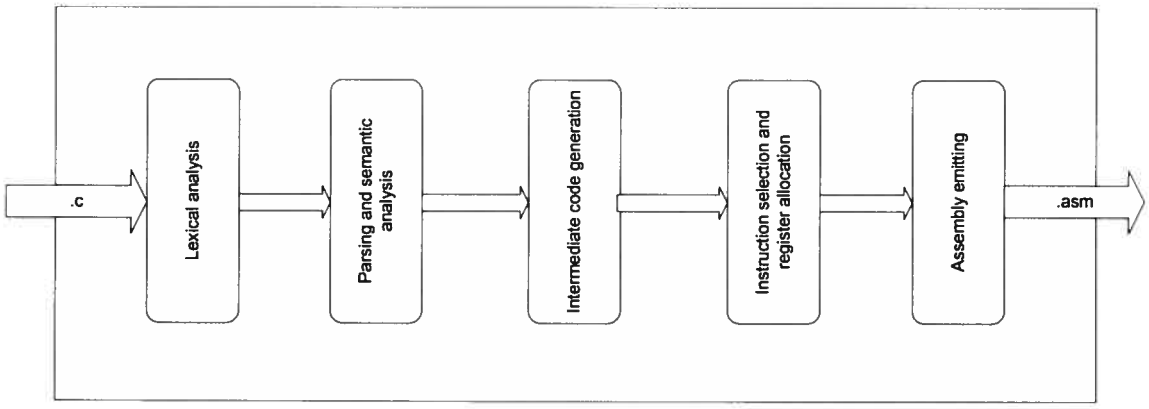


Figure 11 Lcc C compilation flow [38]

Lcc is developed under traditional compiler techniques. The front-end is responsible for lexical, syntactic, and semantic analysis as well as performs some machine independent optimizations. As a matter of fact, both the lexical analyzer and the recursive-descent parser are hand-written programs [38].

The front-end is target-independent but the back-end is target-dependent. An efficient interface packages the front-end and the back-end to be a single program. This interface includes several shared data structures, functions and a DAG language. Examples of interface functions are: emit function prologue, define global, emit data, etc. The DAG language encodes the executable code from a source program by using a smaller language than typical intermediate language which is similar to what is used in other compilers. The authors wrote the lexical scanner and parser programs by hand. These programs are smaller and more efficient than a lex and yacc based on implementation which are tools used to generate lexical analyzers and parsers [30]. First, a C program is translated into syntax trees by the scanner and the parser. Second, from DAG, the syntax trees will be translated into lcc's intermediate representation (IR). In order to simplify

the code generation, the DAGs will be decomposed into trees. Finally, code generation and register allocation will be performed on the trees.

Lcc's code-generator generator is called *lburg* and is based on a dynamic programming algorithm. With the fragment patterns of individual instructions, *lburg* maps the program's IR trees and chooses the most suitable matches consistent with the overall tiling cost of the tree. Furthermore, the algorithm can generate an optimal matching for the IR trees in linear time. One of the features worth mentioning is that the lcc's register allocator uses a simple labeling method that does not utilize graph coloring.

Lcc's back-end is its most interesting part since it shows the results of the design choices that were made to enhance retargetability. Lcc is smaller than the other open source compilers such as gcc [39], and is quicker than gcc. However, the quality of lcc's generated code is lower since lcc does not make extensive back-end code optimization. The code size and execution speed of the lcc-generated code is lower than gcc generated code by an average of 10% [40].

In this project, we selected porting lcc to DLX generally because it's small and easy to implement. It would be much more difficult to port other compilers such as gcc. Our research group needed the C-DLX compiler to test our multi-processor SoC platform when we began this project. In other words, we are not compilation technology specialists; porting lcc is a first start before we explore more compilers.

Chapter 4
Main Challenges in NPs Compilation and Related Work

The most popular usage of network processors is employed as a traffic manager in switches, routers and other network equipments to manage the data packet stream. The traffic manager checks, adds and modifies header of PDUs and puts PDUs in a queue according to PDUs priority weight if necessary. The routing and time schedule are also made by the traffic manager [2].

Network processors own special functional features different from other processors. These features bring different challenges to network processor compiler design. We will discuss these challenges and related work in the rest of this chapter.

4.1 Network Processors Essential Functions

Network processor architecture is different from the general-purpose processors (GPP), since NPs play special role in network traffic management. Like we mentioned before, it works as traffic manager to handle PDUs with the aim of finishing variety functions like quality of service (QoS), encryption, decryption, security-authentication, etc. The NPs implement partially protocol stacks, only those parts of a protocol that require direct access to the data stream are to be implemented by NPs [2].

Let us assume a traffic manager uses 90% of time to run 10% of a network protocol. The workload is partitioned with that a network processor run 10% of the protocol by using 90% of time, GPP run the 90% of workload by using 10% of time. According to Amdahl's law [41], the overall acceleration is $1 / ((1 - f) + f / s)$, where f is the fraction of the program enhanced by a speedup of s. In this example, since f is equal to 0.90, the overall speedup is 5.3 if NPs run 10 times faster. As a result, NPs performance is a significant factor in overall network performance since making a GPP five times faster could be much more expensive.

A variety of special functions of network processors are implemented by the development of compilation technologies. They are listed as follows:

The network processor should be able to process and deliver out the incoming PDUs in a very short time gap. The contents of a PDU should first be examined to determine which processing must be done when a PDU arrives. This process is used in firewalling, routing, policy enforcement, and QoS implementation. A PDU should be modified by the network processor. These modifications include time-to-live counter reducing, an outgoing label replacing an incoming label in the label-switched traffic, adding or removing a header. When the processing speed of network processor is slower than the data arriving speed, queuing the

PDUs is thus necessary. The order of retransmission may not be the same with the order of transmission. Some PDUs will be dropped, and some PDUs may be prioritized over other PDUs [2].

4.2 NPs' Working Mechanism and Challenges for Compilation Technology

Some modern GPP, just like Intel's Pentium or AMD's Opteron, are not only devoted to the increase of manipulating speed but also to maintain the ease of programming. A GPP operates by a sequence of instructions which give the impression that the instructions are executed one by one without discontinuity. Actually, the GGP takes a sequence of instructions, analyses them, cuts them into parts, handles multiple registers' copies, caches memory and then executes the multiple instructions in parallel.

NPs have special features different from other embedded processors such as bit packet passing, distributed memories, multiple PEs (processor elements) working in parallel or pipeline. Some of NPs, such as Agere Routing Switch Processor, Brecis' MSP5000, and Cisco's PXF, use VLIW architectures.

Because of these special features, we are faced with challenges to the ASIPs embedded systems and challenges to parallel architectures when we explore compilation technology. In this chapter, we examine related works about how to resolve the above challenges.

4.2.1 Multiprocessor Architecture with Distributed Memory

There are some common themes shared by a wide variety of network processor architectures, specifically multiprocessing. Many individual processors are embedded in a network processor and cover a wide range in complexity up to

C++ programmable RISC processors. They divide up processing using different techniques and have different internal data flows. Here we call the individual processing units the RISC processing elements (PEs) although there are a variety of names used, from pico-processors to RISC cores, by the provider.

A multiprocessor is made up of many PEs which are not very powerful. However, it has more processing power than a single processor is constructed by the same resources. The design of multiprocessors makes the multiprocessor architecture have the potential to multiply the amount of processing time that a NP can devote to a protocol data unit by the numbers of processing elements. The speed of an individual general-purpose processor is the most important specification and the network processor designers optimize the number of PEs with their size and power.

The PEs' programming differs from one kind of device to another and its sophistication also varies. For example, the Motorola C5 network processor possesses 16 RISC cores, supported by a C++ compiler and IBM's PowerNP device has 8 protocols PEs and 2 pico-code engines. It can run two threads with context switch time. However, an assembly language should be used to program the engine with sound architecture knowledge of the processor.

Raja Das developed methods which deal with parallel loops [42]. Only in the presence of regular array reference patterns within the loops, the communication optimizations could be performed by the existing systems, such as Fortran D, etc. Irregular array references, often contained in the parallel loop nests cannot be analyzed at compile time. Parallel loops and loops that contain reduction type output dependencies are dealt with this method. Loops not containing cross-processor, loop-carried dependencies or cross-processor loop-independent dependencies could be performed by this method. Cross-processor dependence means its end points cross processors. Loop-carried dependence has a write to a location in an iteration and is at the same location followed by a read in a later

iteration. Loop-independent dependence has a write to a location in an iteration and followed by a read to the same location and iteration. Across the nodes to be partitioned the array, performed computations on a part of it by each processor, data parallelism is established, and cross processor loop independent dependences will not occur.

For irregular references to reduce communication costs, runtime optimization techniques have been developed. They are reasonable partitioning of data and computational work, combining element messages into a large message to reduce the number of messages transmitted and eliminating redundant communication of elements.

A prototype compiler, Arguably Fortran (ARF), was developed to automatically perform these optimizations. A simplified Fortran 77 program, enhanced with specifications for distributed data, is accepted by ARF and outputs a program to execute on the nodes of a distributed memory directly. The ARF classifies the array references as regular or irregular by partitioning computations and analyzing them. The runtime optimizations are performed for the irregular references to reduce communication costs.

At the NASA Langley Research Center Hampton (ICASE), a Parallel Automated Runtime Toolkit (PARTI) is used to implement the runtime optimizations. The runtime pre-processing procedures are performed by the compiler. These procedures include supporting a shared name space, providing the infrastructure needed to implement non-uniform data mappings efficiently, coordinating inter-processor data movement, managing the storage, accessing to and coping of off-processor data.

There are two distinct layers in the compiler. A compiler carries out program transformations to the PARTI primitives by embedding calls in the original program, and it is located in the top layer. The library of PARTI runtime

procedures is at the bottom layer. It is designed to support irregular patterns of distributed array accesses efficiently. Each distributed array element is assigned to an arbitrary processor and the whole distributed arrays can be partitioned in a non-uniform manner. The operations of these procedures are off-processor data fetching, storing and accumulating off-processor memory locations. Using embedded procedures, all distributed memory accesses are carried out by generating a multi-computer program [42].

Significant advantages are offered by the distributed memory message passing multi-computers over shared-memory multiprocessors. However, much more work must be done in order to compose programs that release all computational power from them. The lack of a single global shared address space is a main reason. The PARAllelizing compiler for Distributed memory General-purpose Multi-computers (PARADIGM) [43] addresses a need for efficient parallel programming to replace the manual distribution of codes and data on processors and management of communication among tasks. The sequential programs are converted by an automatic means, paralleled by compiler dependence analysis and compiled for execution on distributed memory multi-computers.

Both structured and unstructured parallel numerical applications are the target of the PARADIGM compiler written for Fortran 77 and high performance Fortran[43]. Using a parallel compiler, the sequential Fortran for regular application is paralleled automatically. On the program, several compiler transformations are performed by the PARADIGM compiler and the efficient message is generated.

Many compiler optimizations are performed automatically by the PARADIGM compiler, for example, message vectorization, chaining and aggregation. In addition to performing traditional compiler optimizations, the PARADIGM is unique since it is able to perform automatic data distribution for regular

computation, to exploit simultaneously functional and data parallelism, and in iterative application to exploit regularity in irregularity.

PARADIGM is a multifunctional parallel compiler for distributed-memory multi-computers. For regular computations, it can distribute program data and perform a variety of communication optimizations automatically for regular computations and use compiler and run-time techniques to provide support for irregular computations. It has a wide range of applications as a compiler in the field of distributed-memory multi-computers [43].

4.2.2 Bit Packet Processing

J. Wagner and R. Leupers introduced the design of a C compiler using an application specific instruction set processors (ASIPs) for telecom applications [44]. The use of ASIPs in embedded system design is quite common but the compiler supporting them is extremely desirable since the compilers are required to avoid time-consuming and error-prone assembly programming. The traditional compiler techniques could not fully develop the functions of ASIPs. The more dedicated code generation and optimization techniques are demanded. To meet high code quality for embedded systems, a variety of highly machine-specific techniques is developed and promises to generate high-quality machine codes similar to the hand-written assembly code. However, the increased compilation time seems inevitable; the efficient bit-level processing is a selection for some ASIC designing. Unfortunately, the highly application specific hardware has low flexibility. The network processors (NPs), as a specialized class of ASIPs, represent a promising solution because NPs instruction sets could be tailored towards efficient communication protocol processing. NPs may be designed to have the ability to process bit packets with variable length. The memories of the standard processors as a transmitter or a receiver have a fixed word length. The packets meant to be transmitted at the beginning of a transmission are aligned at the word boundaries of the transmitter. These words to be sent into the send

buffer should be packed in to the bit stream format required by the network protocol. On the receiver side, the bit stream format packets have to be extracted reverse after the transmission over the communication channel and be aligned at the receiver memories, the word length of which may even be different from the memories of transmitter. A fixed word length of memory of transmitters and receivers show that a relatively expensive processing may be required on both sides while the standard processors are being used.

Developing an efficient C compiler for an advanced NP architecture is a design challenge since the dedicated bit-packet oriented instructions are not easily generated from a high-level C language. The classical techniques are not good solutions to this problem. The required machine-specific code generation techniques are implemented and the bit packet processing is made available to the programmer at the C level. The register allocator is designed to handle the variable-length bit packets in registers.

The target machine is Infineon NP [44] which possesses 12 general-purpose registers with special extensions for bit-level data access and its core shows a 16-bit RISC-like basic architecture. The ALU computations can be performed by the NP instruction set on bit packets. Any bit index sub-range of a register may store a packet which may cover two different registers. Therefore, instead of the fixed machine word length, the variable packet lengths can be adapted within registers. The bit streams in memory should be the first to be loaded into registers. For each specific application the size and position of the different bit field are statically known from the C source code.

The NP instruction set permits the specification of offsets and operand lengths within registers to enable packet-level addressing of unaligned data. If CMD represents the assembly command, *reg1.off* and *reg2.off* are the argument

registers with offset and operand lengths within registers. Then the general instruction format is as such:

CMD reg1.off, reg2.off, width

Where the *width* denotes the bit width of the operation to be performed, the register number, offset, and the packet bit width addresses a corresponding bit packet. If the register word length is not enough to span the packet, two registers should be spanned over without increasing the access latency so that two offsets and one width parameter for instructions are enough [44].

In C language, the bit packet-level addressing can be expressed by a complex shift and masking scheme only. Therefore, it is inconvenient and lacks readability and maintainability. The code may be machine-dependent or depend on the word length of the processor. By means of special instructions for packet-level addressing, the shift and mask operations are avoided by the NP instruction set due to their high cost. The compiler-known functions (CKFs) [44] were introduced in the C compiler to make the bit manipulation visible to the programmer. The compiler maps calls to CKFs into fixed instructions or instruction sequences and may be considered as C-level macros whit no any calling overhead. If a suitable set of simulation functions is provided for the CKFs, there is no any machine-dependent for the NP for C code written, as well as compiled to other machines. The packet-level addressing can be implemented within a single instruction by NP. A packet access (PA) CKF was introduced:

PA (int op, int var1, int off1, int var2, int off2, int width)

Where "op" represents the operator, "var1" and "var2" are the operands, "off1" and "off2" are the operand packet offsets and "width" is the packet bit width.

Use of CKF significantly benefits the programmer as it permits him to use high level language constructs for control code and loops as well as performs address generation and registers allocation; keeping the code reusable and saves development time [44].

4.2.3 Specialized Processors

The network processor compiler, like most other compilers, consists of a front-end and a back-end. The front-end takes an input high-level language source code and generates an intermediate representation (IR) which is independent of the target machine. The back-end takes the machine-independent IR and translates it into a machine-specific optimized assembly code. The retargetability is an easy translating source code to the specialized assembly. If one compiler possesses this feature, we call it retargetable compiler.

LANCE [45] is a retargetable platform used by J. Wagner and his colleagues for implementing a C NP compiler. The company web site introduces it as follow: As a front-end, it compiles ANSI C code into a machine-independent IR. IR can be read, written and manipulated through a C++ API. LANCE includes a back-end interface. The main features of the LANCE system are ANSI C front-end and IR optimizations, executable IR in low-level C syntax, C++ API library for IR access and manipulation, generation and visualization of control/data flow, and interface functions for retargetable assembly code generation, compatible with popular code generators like OLIVE or IBURG. The OLIVE tool is an extension of IBURG contained in the SPAM compiler which is a retargetable optimizing compiler for embedded fixed-point processors. The back-end of the SPAM [46] compiler consists of two components; the first one is a set of data structures that store the various representations of the source program and the second is a suite of retargetable algorithms which perform code generation and machine-dependent code optimization. The code selection and register allocation modules are two parts of the C compiler back-end. The code selector maps the Data Flow Trees

(DFTs) into assembly code by using the technique of tree pattern matching with dynamic programming. Hence, an optimal code selection is obtained.

We have described a number of related works about retargetability, multiple processors with distributed memories and special instructions mapping technology separately such as CKF methods to process bit packet [44]. The usage of these special features is explicitly stated in the source C code. This allows ease of expression for bit-packet operations. We might use these technologies to solve NPs compiler problems.

We have general ideas about how compilers solve the above mentioned problems in diverse ways. For the reason of better understanding how a retargetable compiler can be used to target a NP with ASIP architecture, we added DLX as a new back-end in lcc since DLX has been used for our multi-processor platform. We built a C compiler that might save our colleagues a significant amount of time in hand-writing DLX assemble code, with the purpose of improving the development and debugging time. We could easily add other network processors such as targets since the DLX instruction set came from many real world RISC processors. The compiler is based on lcc because it has fast, small and convenient features. On the other hand, it also has obvious weaknesses, such as the quality of compiler-generated code which is not well optimized. We will explain in detail how DLX back-end may be implemented on lcc and what kind of changes will be made on our SoC platform in the following chapter.

Chapter 5
Implementation

We have introduced a fundamental understanding about network processors and compilation technology. This chapter will further discuss how we implemented a C/DLX compiler based on lcc retargetable compiler and how we made modification to our SoC platform.

5.1 DLX Instruction Set Architecture

The DLX processor comes from a combination of ideas from other load/store RISC architecture. The DLX architecture was selected based on observations about most commonly used in programs. DLX offers a good studying

architectural model since DLX comes from real popular machine architectures and because it is easy to understand [47].

We used DLX to construct a SoC platform since it is a clear and simple RISC architecture. Moreover, it is easy to add new instructions because it has common features that come from many real RISC architectures. DLX is a simple load/store instruction set; it is designed in pipelining approach. DLX includes several simple formatted instruction sets which are very easy to follow and has good organization as a compiler target.

The DLX Instruction Set Architecture (ISA) contains 32 (R0-R31) 32-bit general-purpose registers. Registers R1-R30 are real general-purpose registers. Register R0 always contains zero. Register R31 is used for saving the return address for the Jump And Link (JAL) instructions [48].

The DLX ISA also has 32 single-precision floating-point (32-bit) registers (F0-F31). These registers can also be addressed as pairs (two consecutive registers, the first one being even-numbered) to form 16 double-precision floating-point (64-bit) registers. DLX ISA has three specific registers: Program Counter (PC), Interrupt Address Register (IAR) and Floating-Point Status Register (FPSR). In DLX ISA, a word is defined as 32 bits and a byte is 8 bits. Memory is byte addressable and word storage adheres to the big end in byte ordering [49].

The DLX instruction set architecture includes five pipeline stages. These stages include Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory (MEM), and Write Back (WB). As in the first stage, IF is in charge of obtaining instructions from memory. Then ID is responsible for choosing the operand registers, decoding the instruction and examining branch conditions. After Instruction Decode, it comes to arithmetic and logical operations. Moreover, memory address calculation will be processed at this stage. The fourth stage will be MEM stage and it will access data memory; either reading or writing the data

into its memory. The final stage is to write the results calculated by EX or read by MEM. Occasionally, it can also write to the destination register when necessary [49].

There are four classes of instructions: Load/Store instruction, ALU Operations instruction, Branches/jumps instruction and Floating-point operations instruction.

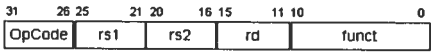
Load/Store instructions set for any of the GPRs or FPRs may be loaded and stored except that loading R0 has no effect.

All ALU instructions are register-register instructions. Register-register instructions include ADD, SUB, AND, OR, XOR, etc. These instructions use to two registers as input operators. If the condition is true in the compare-instruction, these instructions place a "1" in the destination register; otherwise they place a "0" in the destination register.

Branches/Jumps instruction may test the register source for zero or nonzero while the branch condition is specified by the instruction. Floating-point instructions include ADDF, ADDD, MULTF, MULTD, DIVF, DIVD.

To know the detailed instructions, see appendix A: DLX Instructions Set. All DLX instructions are located in one of three types: R type, I-type, or J-type. Figure 12, Figure 13, and Figure 14 show these three type instructions and give related examples [48].

- R-type (register)
  rd ← rs1 funct rs2



- e.g. ADD R8, R17, R18 # R8 = R17 + R18

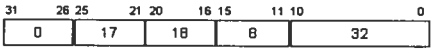
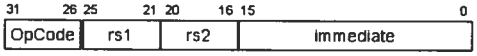


Figure 12 R-instruction format [48]

- I-type (immediate)



e.g.

```
ADDI   R8, R17, -44        # R8 = R17 -44
LW     R8, -44(R17)        # R8 = M[R17 - 44]
BEQZ   R4, label           # if( R4 = 0 ) go to label
```

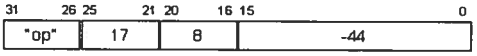
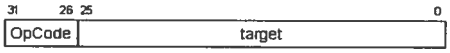


Figure 13 I-instruction format [48]

- J-type (jump)



e.g.

```
jump    label              # call label ,
                             R31 = pc + 8
```

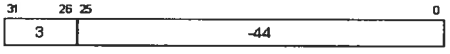


Figure 14 J-instruction format [48]

5.2 SoC Multiple Processors Working Platform

Our cycle-accurate SoC platform is based on SystemC which is an open source class library in C++. In this project, we modified several parts of the platform which allowed us to explore object oriented design methodology used in system design field. Before we describe our work in detail, we will glance at SystemC and object oriented design.

5.2.1 Fundamental Object Oriented Design Characteristics

Object-Oriented Design is a design method in which a system is modeled as a collection of co-operating objects and individual objects are treated as instances of a class within a class hierarchy [50]. Object-Oriented Design methodology requires user spending more time during the system design stage in front of implementation. However, it benefits us a lot by its fundamental characteristics such as encapsulation, abstraction and reusability.

Encapsulation (or information hiding) could keep a system design away from too interdependent and too intermingle. Encapsulation allows user to modify a single part of the system to realize bug fixing, performance improving or system changing rather than change many place. Encapsulating functions could reduce compile time, support testing and allow subsystems taking advantage of independently and asynchronously. Object-oriented design strongly supports code reusability. It allows a particular object model to be written once and reused in numerous places. Object-oriented design also supports localization change, which is possible to take full advantage of code reusability, significant reducing code size [50].

5.2.2 Brief Introduction to SystemC

SystemC is an open source class library in C++ which is a good tool to develop cycle-accurate or more abstract models of software algorithms, hardware architectures and system level designs. SystemC is supported by the Open SystemC Initiative (OSCI). More details about SystemC can be found in [51], the following is a brief introduction from [51]. One characteristic of SystemC is that it is an interoperable modeling platform which allows seamless tool integration [51]. SystemC supports Register Transfer Level (RTL), behavioral level and system levels design abstraction; sometimes the SystemC is used as standardization of a C/C++ hardware design methodology while a class library and a simulation kernel are employed. OSCI is an independent none-profit organization composed of a broad range of semiconductor companies, university embedded software developers, design automation tool vendors and individual. In conclusion, SystemC has all the C++ advantages such as data abstraction, modularity and object orientation. Furthermore, SystemC creates a general design environment consisting of C++ libraries models and tools. This feature lets SystemC is becoming a very popular language in hardware/software co-design field.

We, the users of SystemC, could write the SystemC models at the system level, behavioral level or RTL level by using an open source SystemC class library which provides two important advantages. One is that SystemC provides the implementation of many types of objects that are hardware-specific modules, ports and clocks. Other one is it contains a small kernel for process scheduling. SystemC code could be compiled and linked together with the class library with any standard C++ compiler [51].

5.2.3 Multiple Processors Platform

Our colleagues have modeled a system on chip (SoC) platform, including multiple processors. It was developed using SystemC on Linux PCs. The modeled processors are DLX or ARM architectures (still in development) and both of them can work together. Each processor is an addressable device and processors are connected via a device called *ring_device* [52]. Figure 15 shows the architecture of a DLX processor and its adjacent nodes. We made some changes due to the testing for our compiler.

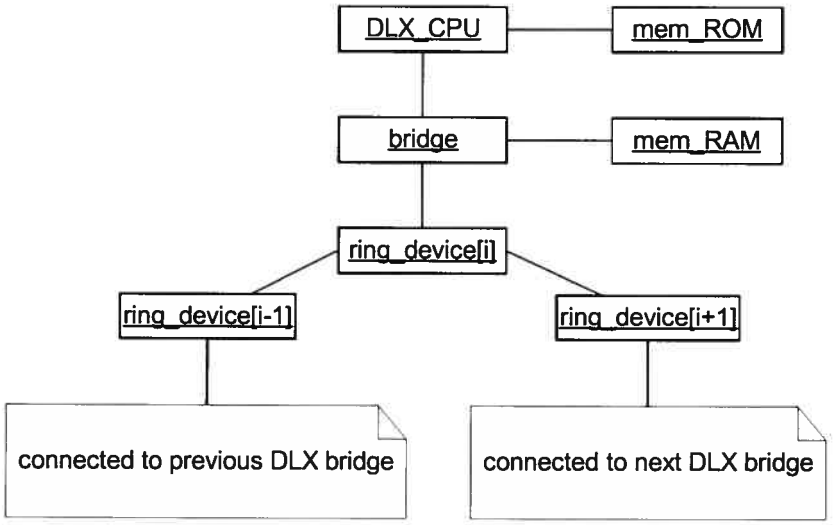


Figure 15 The architecture of DLX nodes [52]

The memory modules are located next to the processors, but we plan to have both distributed and shared memory mechanisms in the future version of the network platform.

Our platform consists of a number of DLX processors which are connected through a ring. Every pair of neighbor nodes is able to receive and send messages at the same time. In a platform which contains n processors, a message sent from node i to node j run through the path (i, i+1 mod n, i+2 mod n... j-1 mod n, j) [52].

As we previously mentioned, the DLX model was developed consisting of five pipeline stages. Each stage was modeled using an SC_METHOD construct:

- IF (Instruction Fetch): The IF stage is responsible for getting the instructions out of the ROM (program memory).

- ID (Instruction Decode): The ID stage is responsible for selecting the operand registers, decoding the instructions, and evaluating the branching condition.

- EX (Execution): The EX stage is responsible for arithmetic and logical computations as well as memory address calculation.

- MEM (Memory): The RAM (data memory) access is performed at the MEM stage.

- WB (Write Back): When it is necessary, results will be written back to the destination register.

In this platform, each processor processes both the RAM and ROM memories; we define the size as 1024 (4096 bytes). Memory size is not important in this project because they are big enough to test our compiler. Both Mem_RAM and Mem_ROM memories are modeled using a common memory abstracted class. This design respects the communality and variation principle described in [53].

We have modified memory from word align to byte align; this makes the model closer to the DLX architecture description. Furthermore, we added byte operations and half word operations such as load byte (LB), load byte unsigned (LBU), load half word (LH), load half word unsigned (LHU), store byte (SB) and store half word (SH).

The ring device of processor exchanges data packets with two neighbors through the Addressable_Device interface, the exchange operation on the ring is

unidirectional and cannot be blocked. This operation performs at each clock cycle. An example is given underneath.

When DLX node number 'i' wants to send a message to a DLX node number 'j', the first step is taking the ring_device status register (at address 0x3000) and wait until the transmit register available. If so (transmit register is true), the next step for the DLX number 'i', is to write the message and the address of destination (in this case is 'j') in the appropriate registers. As mentioned, the messages already looping on the ring have the right of way on the new incoming messages from a DLX processor. Since no more than 'n' (total nodes number in the ring) messages can be located on the ring at the same time, definitely, each message can be delivered to the target processor after most 'n' steps. These two conditions make sure that there is no deadlock except invalid destination addresses. When a free slot is available on the ring, the new message is transferred from the RingDevice registers to the next RingDevice in the loop. When a message reaches the appropriate RingDevice it is sent to the RingDevice receive registers so the polling DLX j processor can read the incoming data. Finally, a new space on the ring is made "available" allowing the ring interconnect to accept a new message.

Each processor accesses a bridge; the bridge controls the memory access commands from the DLX and either sends them as messages over the ring or as an access to the local RAM memory. Processors can also access other addressable devices via the bridge. The multiple DLX processors exchange messages on the ring using a memory-mapped mechanism:

- 0x0000-0x2FFF: RAM
- 0x3000: Transport status
- 0x3004: Receive address
- 0x3008: Receive data
- 0x300C: Transmit data
- 0x4001: Port to output device (new added).

- 0x5001: Data exchange interface to other device (new added)

For testing purposes, we added two addressable devices to the platform. One is an output device (address 0x4001) for outputting characters to the screen and other one is a data interface (address 0x5001).

The output device is directly a character to the screen because that SystemC does not have output interface. We can easily print necessary information to screen by using this device. For the data interface we used two First In First Out (FIFO) data buffers to exchange data with other devices.

As previously mentioned, SystemC is an object-oriented modelling language based on C++ [51]. It was easy to implement these two modules with the methods inherited from class Addressable_Device that is derived from SC_METHOD of SystemC.

5.3 Add DLX as New Target to lcc

Our colleagues worked on the platform using a hand-written assembly code but this time consuming in regards to writing and debugging assembly code. For this reason, a suitable compiler was necessary.

As we already mentioned, lcc is a retargetable compiler for ANSI C. It has been ported to the VAX, SPARC, MIPS, X86, and other target processors. LCC is a small, fast C compiler now available on most popular operating systems [54]. The compiler is based on lcc, because lcc has fast, small and convenient features, but lcc has obvious weaknesses such as the quality of the compiler-generated code which is not well optimized, and lacks of parallel work. Similar to most other compilers, the lcc compiler is subdivided into two parts: a front-end and a back-end. The front-end is responsible for source code analysis, generation of an

Intermediate Representation (IR), and machine-independent optimizations. The back-end maps the machine-independent IR into machine-dependent assembly code. It is labeled as retargetable since we can easily add one or more different target code generators to the back-end.

The lcc compiler back-end can be divided into two parts: code selection and register allocation. The code selector maps the intermediate representation (trees or directed acyclic graphs), generated by the front-end, using the back-end interface into DLX instructions. The register allocator maps all the virtual registers to physical registers. We used the MIPS [55] code generator as a model and made some changes to suit the DLX instruction set. Perhaps this work is not much from a compilation technology's point of view but it is especially important to us since we need a correct compiler to test and modify our SoC platform to save time and avoid mistakes from hand-written codes. It also helps us to explore compilation technology step by step.

Instructions Selection

The instruction selectors in lcc are automatically generated from compact specifications by the program *lburg*. That is, one gives grammar to *lburg* to partition the IR tree and it generates the C code for the back-end. A tree parser accepts a subject tree of intermediate code and partitions it into chunks that correspond to DLX assembly instructions.

The interface between lcc's target-independent front-end and its target-dependent back-ends consists of a few shared data structures, 18 functions, and a 36-operator DAG language, which encodes the executable code from a source program. Most of the functions (15) are simple, e.g., they emit function prologues, define global, lay out data, etc.

A symbol's name, class, and type fields give its name, its storage class, and its type respectively. Fields and types irrelevant to register allocation have been omitted.

In a DAG node, the *kids* point to the operand nodes, and the *syms* point to symbol table entries for those operators that take symbols as operands. Count holds the number of references to this node from kids in other nodes. The *x* field is the back-end's 'extension' to nodes and it holds the per-node which is target dependent data that the back-end needs to generate code.

reg holds the number of the registers allocated to this node. *rmask* is 1 if the node needs an ordinary register, and 3 if it needs a register pair, n*ext* points to the next node in the linearized forest.

The *op* field holds an operator. The last character of each is a type suffix, such as: C means character, S means short, I(int), U(unsigned), P(pointer), F(float), D(double) and V(void) [56].

Tree grammar is the core of *lburg* which is the back-end generator in lcc. Tree grammar is a list of rules which contains four parts. First is a non-terminal that replaces the part of the tree if the rule is applied. Then, a tree-matching expression (non-terminals and IR nodes) specifies where the rule can be applied finally, it specifies what assembler instructions must be added to make the transformation and their cost (what the compiler tries to minimize; size or number of cycles).

DLX non-terminal list as below:

acon:	address constants
addr:	address calculations from registers
addrr:	address calculations from immediate values
con:	constants
reg:	computations that result to a register

```
stmt:    computations done for side effects
```

The above non-terminals give a high level overview of the tree grammar used for mapping to DLX assembler instructions. Here are some actual rules:

```
reg:    BCOMI4( reg )      " xori r%c,-1\n"                          1
reg:    BCOMU4( reg )      " xori r%c,-1\n"                          1
reg:    NEGI4 ( reg )      " sub r%c,r0,r%0\n"                       1
...
stmt:   EQI4 ( reg, reg )  " seq r3,r%0,r%1\n  bnez r3, %a\n"        2
stmt:   GEI4( reg, reg )   " sge r3,r%0,r%1\n  bnez r3, %a\n"        2
stmt:   GTI4( reg, reg )   " sgt r3,r%0,r%1\n  bnez r3, %a\n"        2
...
xor:    XORI( reg, reg )   " r%0,r%1 "                               0
reg:    ADDI( xor, reg )   " xadd r%c ,%0, r%1\n"                    1
```

The elements in the first column, like *reg* and, *stmt*, are non-terminals. The second column has tree nodes that written in uppercase and operands types in parentheses (here, operands are non-terminal reg). In the third column, inside double quotation marks are the assembler code templates. The final column's numbers are the optional cost.

Register Allocation

The register allocator is a small part written in C, using predefined lcc register management functions. The front-end passes a forest of dags to the back-end. The

production back ends traverse each dag to select suitable instructions, but the sample back-end emits a naive code so it does little during this first pass.

After code selection, all back ends linearize the forest and make two passes over the resulting list, which is linked through the *x.next* fields in each node. The first pass allocates registers and the second emits the final code.

Lcc's register allocation strategy is simple: it traverses the linearized forest and allocates registers to each node. The count field tells when all of the references to a node have been processed. For each node, the registers used by its children are released by *putreg*, which decrements count and frees the register only when the last reference is removed by clearing the appropriate bits in the global variable *rmask*, where *rmask*&(1<<r) is 1 if register r is busy. Subsequently a register or register pair is allocated to a node by *getreg*. A register is spilled even if its value is already available elsewhere in memory and even if it would be cheaper to re-compute than to spill and re-load.

The register allocator frees registers as soon as possible. If the available registers are exhausted, it is often because there are multiple references to the nodes holding the registers which arise from common sub expressions and from multiple assignment, augmented assignment, and the operators ++ and --.

The DLX instruction set gives a few constraints: R0 is always 0 and R31 only contains jump and link instructions' return address. The assembler reserves R1 for pseudo-instructions. R2 and R3 are reserved by convention for return values. R26 and R27 are reserved for Operating System (OS). R4-R7 are for procedure arguments. R8-R15, R24, and R25 are scratch registers. R16-R23 are for register variables. R28 is used as a global pointer. R29 is the stack pointer, and R30 is for compiler temporary values.

5.4 Test and Result

With a new added output device and data interface, especially they work with our compiler and assembler, the entire idealized platform structure is shown in Figure 16.

The entire design flow is described as follows: write applications using C code as a replacement for assembly code, in order to avoid hand writing deficiencies such as long developing time, difficulty with debugging, etc. Then it compiles C code to assembly code. It translates assembly code to binary code by using EBEL DLX assembler which was developed by Etienne Bergeron and Eric Lesage [57]. It loads the binary code to ROM (program memory) which can be run on our multiple processors platform. It is very easy to print information to screen through the output device and it might successfully exchange data flow with other devices by using data interface.

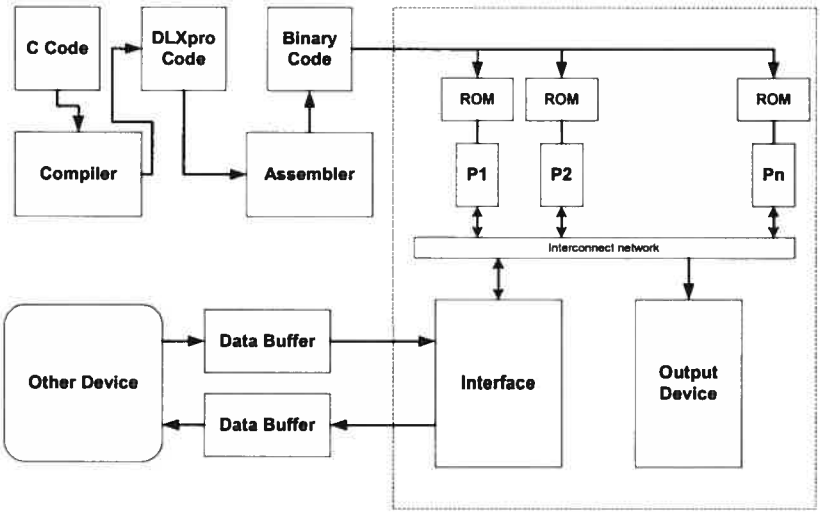


Figure 16 Overview of entire platform

We also used some special pure C functions to implement basic I/O functions, mathematical functions, and conversion functions without C libraries. These functions include *getaddress, getdata, gputadress, putdata, getchar,*

string_to_integer, printString, sqrt, etc. All of these functions are small and easy ported.

By using these functions and modules we can conveniently implement some library functions such as *printf*, but we often do not need a printing function that is as complex and big as *printf*.

The following example shows how the above-mentioned functions and modules work: Suppose we need to print "Hello" to the screen; using the printString function, we could write the C code as follows:

```
void printString(char *p)
{
  while(*p)  *((unsigned int*)0x4001) = (*(unsigned char*)p++);
}
void main()
{
  printString("Hello");
}
```

Compiling the above code we got the corresponding DLX assembler code:

```
        addi    r29, r0, 1000          we define stack here
        jal     main
        trap    0
        nop
        nop
printString:
        j       L.3
L.2:    addi    r24, r4, 0
        addi    r4, r24, 1
        addi    r15, r0, 0x4001
        lbu     r24, (r24)
        sw      (r15), r24             access to a addressable device using
                                       memory mapped mechanism
L.3:    lb      r24, (r4)
```

```
    sne     r3, r24, r0
    bnez    r3, L.2             need optimisation
    addi    r24, r0, 0x4001
    sw      (r24), r0
L.1: jr     r31
main:
    addi    r29, r29, -24
    sw      16(r29), r31
    addi    r24, r0, L.6
    sw      -4+24(r29), r24
    lw      r4, -4+24(r29)
    jal     printString
L.5: lw     r31, 16(r29)
    addi    r29, r29, 24
    jr      r31
L.6:
    byte    72, 101, 108, 108, 111, 0
```

We ran the above assembler code on the network platform model using a single DLX processor and got the expected results but the code is not optimal. The "sne" in *printString* could be removed by changing the condition on the next line [58].

We have performed simple tests for a multiprocessor with manual parallelization. Restricted by the testing platform memory model we loaded the same code to all RAM and ROM but each processor can only run the indicated program fragment that is distinguished by identifying the node number. See the following example:

```
/* this is for test simple parallel on DLX platform
0X3000 is base ring device address
0X10(16) is data out_port offset
0X0C(12) is address out_port offset
0X08(8) is data get_port offset
*/
void trans(X, addr)             /* transfer X to number "add" node */
```

```
{
*((int *)0x3010)=X;
*((int *)0x300C)=addr;
}

int get_data()
{
return (*( int*)0x3008);
}

int get_addr_self()
{
return *( int*)0x3004;
}
void main()
{ int a1=2, node_No, tran_No;
node_No=get_addr_self();
a1+=node_No;
if ( node_No==0)
trans(a1, 2);
if( node_No==2)
a1+=get_data();
}
```

In the above simple C program, we can easily identify the current DLX node number by calling function get_addr_self(). We got the equivalent DLX assembly code as follow by compiling this C code.

```
addi r29,r0,1000
jal main
trap 0
nop
nop
trans:
addi r2


4, r0, 0x3010
sw (r24), r4
addi r24, r0, 0x300c
sw (r24), r5
L.1:
jr r31
get_data:
addi r24, r0, 0x3008
lw r3,(r24)
L.2:
jr r31
get_addr_self:
addi r24, r0, 0x3004
lw r3,(r24)
L.3:
jr r31
main:
addi r29, r29, -32
sw 16(r29), r30
sw 20(r29), r31
addi r24, r0, 2
sw -4+32(r29), r24
jal get_addr_self
addi r30,r3,0
lw r24,-4+32(r29)
addu r24,r24,r30
sw -4+32(r29), r24
sne r3,r30,r0
bnez r3, L.5
lw r4,-4+32(r29)
addi r5, r0, 2
jal trans
L.5:
L.4:
lw r30,16(r29)
lw r31,20(r29)
addi r29, r29, 32
jrr31
```

We ran the above assembler code on the network platform model using DLX processors and got the expected results but the generated code is not optimal. For instance, the "sne" in *printString* could be removed by changing the condition on the next line. We have also tested certain programs using the mentioned method and we found they worked well on our platform even though we made several changes for this multiple DLX processors platform. This means the compiler works as a debugging tool to refine our simulation environment.

Indeed, the lcc compiler for the DLX target is fully functional. The performance of the generated code successfully passed testing on a single processor and multiple processors by manual parallelism giving the expected results. Thus, our colleagues can write applications by using C instead of DLX assembly. Although the generated code is not as good as a hand writing code and it lacks of parallelism, it could allow faster developing time and also avoid hand written mistakes which would eventually save the developing period in the whole SOC platform project.

Chapter 6
Conclusion and Future Work

With fast development in the telecommunication industry, bandwidth becomes a key element of the market. The market requires wider bandwidth switches, routers, load balancers and other high speed network equipments. In order to satisfy this demand, high performance network processors are studied and developed to meet the requirements of the market.

One of the most popular solutions for network processing is ASIP. ASIP is an instruction set processor typical for a particular application domain and has become a widely accepted solution because of its special characteristics. By overcoming the weaknesses of other solutions such as ASICs' higher development cost and GPPs' lower efficiency, ASIPs get the balance with many aspects. For instance, ASIPs could provide good balance both in hardware and software to meet all requirements in terms of performance, flexibility, and fast time to market. Therefore, a network processor can be defined as the ASIP for the networking application domain.

NP works as a network traffic manager in charge of PDUs processing at high speed with the purpose of complete many different functions such as QoS, compression, security-authentication, encryption/decryption, etc. we have discussed protocol stacks and showed the data path "from end to end" In chapter 2, It shows clearly that network processors do not run all of protocol stacks since the NP's main function is data forwarding.

Special functions of network processors are enumerated in chapter 2. A network processor may perform some prompt operations such as classification, modification, queuing, and buffer management. Other operations include security with the content of encryption, decryption, and authentication; and operations like policing, compression, and traffic metrics, etc.

Detailed explanation of fast passing PDU, classification, modification, and queuing also has been described in chapter 2. We know that the network processor has very little time to operate on a PDU because data arrives at a high-speed rate and it needs to be quickly dispatched. Examinations have to be done to determine what type of processing should be performed on a PDU. Modification

explains how a PDU may be modified. For instance, an incoming label will be changed to an outgoing label in label switching traffic. Moreover, PDU's header may be added or removed. Frequently, CRC recalculation or checksum replacing may be involved in the modification step. Queuing is needed when the data's arriving speed is faster than the processing speed. Transmission or retransmission of a PDU is not directly forward, as a result, some PDUs may be prioritized over others, On the other hand, certain PDUs may be dropped.

Special functional features of NPs were also discussed. Those features cause NPs to have particular architecture characteristics different from other embedded processors such as multiple processor elements working on a parallel or pipeline, special instructions, distributed or shared memories.

Furthermore, the market requires fast reaction time with good reliability in network equipment field. In network processing systems, embedded software requires compilers to avoid slow and error-prone development in assembly language in order to meet market needs. As a result, classical compiler technology cannot provide sufficient solutions for the particular architecture of network processors. Thus a very special code generation technology must be used to fully develop the network processor's performance. Moreover, multiprocessor and other kinds of processors may be used in one system. As a result, particular compilers for NPs are needed.

A good NPs' compiler must have excellent retargetability, high performance in parallelization, high quality in special instruction mapping, etc. We face all challenges to ASIPs embedded systems and to parallel architectures when we explore compilation technology for NP. We analyzed NPs basic attribute, reviewed related work about parallel compilation, retargetable compiler, compiler

technology for distributed or shared memory architecture and special instruction mapping particularly in bit packet processing. We obtained a lot good ideas related to NPs compilation technology after discovering a number of related works.

Our colleagues, Luc Charest and Alena Tsikhanovich, have modeled a system on chip platform, which includes multiple DLX processors and simple interconnect devices. The platform was developed using SystemC on Linux PCs. Other students of our lab are still working on the ARM processors modeling and AMBA bus modeling. A detailed description of the platform was presented in chapter 5.

We implemented a C-DLX compiler based on lcc retargetable compiler. In addition, a new back-end that is a DLX code generator we made worked well when tested using our platform on a single processor or multiple processors. Furthermore, we tested simple parallel work in the SoC platform with successful result. We implemented many other functional models such as the byte related operations and modified data transfer model as a token ring to suit data transportation. We also put an output device that can print needed information to screen without using stand C libraries.

We achieved a fundamental understanding of how a retargetable C compiler like lcc can be used to target an ASIP like DLX architecture through this project. We could easily add other network processors as targets since the DLX instruction set came from many real world RISC processors. The compiler based on lcc with its fast, small, and convenient characteristics significantly improved the development and debugging time. We also presented our tested compiler on our network platform with expected results. Examples are illustrated to show how the specific functions and modules are written to act as some of the standard C libraries.

These functions and modules will play an important role in testing and code writing in terms of time saving and simplified work. The quality of the compiler-generated code is not as good as the hand-written DLX assembly code.

In the future, we plan to explore CoSy DSP compiler development system [59], and use SUIF parallel mechanisms to try to solve the multiple-thread problem in network processors. We also plan to add special instructions such as bit-packet-oriented instructions to DLX in order to speedup network processor applications which performs bit extraction and manipulation. However, we will encounter difficulty in the network compiler development to generate bit-packet-oriented instructions from a high-level language. There are several approaches that have already been explored such as Compiler-Known Functions (CKFs) [44]. Eventually, we plan on achieving a general methodology of NPs parallel compilation developing.

References

[1] N. Shah, "Understanding Network Processor" Dept. EECS, UC, Berkeley. September 2001.

[2] D. Herity, "Network Processor Programming", Silicon & Software Systems, Dublin, Ireland Jul 31, 2001.

http://www.embedded.com/story/OEG20010730S0053

[3] M. Rose, *The Open Book: A Practical Perspective on OSI* Prentice-Hall, 1990.

[4] M. Egan, "Networking Models"

http://mike.passwall.com/networking/netmodels

[5] N. Doraswamy and D. Harkins, *IPSec: the new security standard for the Internet, intranets, and virtual private networks.* Prentice Hall. 1999.

[6] P. Loshin, *Essential ATM Standards: RFCs and protocols made practical.* John Wiley & Sons. 2000.

[7] VLAN Information http://net21.ucdavis.edu/newvlan.htm

[8] E. Rosen, A. Viswanathan, R. Callon. *Multiprotocol Label Switching Architecture* Internet Request for Comments. January 2001.

[9] D. Comer, D. Stevens, *Internetworking with TCP/IP Volume II: ANSI C Version: Design, Implementation, and Internals* Prentice Hall. 1994.

[10] http://www.ipv6.org/

[11] B. Cole, "Accelerated Intros 128 bit IPv6 Protocol Stack"

http://www.iapplianceweb.com/

[12] Alchemy Semiconductor, Inc. "The Alchemy Au1000 Internet Edge Processor." Product brief. 2000.

[13] SiByte. "SB-1250 Data Sheet" http:// www.sibyte.com

[14] SiByte, Inc. "SB-1 CPU fact sheet" http:// www.sibyte.com

[15] Cisco Systems. "Parallel eXpress Forwarding in the Cisco 10000 Edge Service Router" White Paper. October 2000.

[16] EZchip Technologies. "Network Processor Designs for Next-Generation Networking Equipment" White paper. December 1999.

[17] EZchip Technologies. "EZchip Technologies Software Development Suite Now Available For Its 10-Gigabit 7-Layer Network Processor." Press Release. January 17, 2001.

[18] T. Eklund, "The World's First 40Gbps (OC-768) Network Processor." Presentation. Network Processor Forum. June 14, 2001.

[19] Xelerated Packet Devices. "Xelerator™ X40 Packet Processor." Preliminary Product Brief. June 2001.

[20] M. Ngo, "Introducing the BRECIS Multi-Service Processor™." Presentation. Network Processor Forum. June 14, 2001.

[21] Quantum Effect Devices. "QED RISCMark." Product Sheet.

[22] Maker. "MXT4400: Traffic Stream Processor." Product Brief. 1999.

[23] http://www.lexra.com

[24] B. Gelinas, P. Alexander, C. Cheng, W. Patrick Hays, K. Virgile, W.J. Dally, "NVP: A Programmable OC-192c Powerplant." Presentation. Network Processor Forum. June 14, 2001.

[25] IBM Corp. "PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors" 2000.

[26] B. Cole, "Intel net processor boosts clock, adds C compiler." EE Times. February, 20, 2001.

http://www.eetimes.com/story/OEG20010220S0029.

[27] Motorola Corp. "C-5 Digital Communications Processor." Product Brief. May 4, 2000.

[28] D. Husak & R. Gohn, "Network Processor Programming Models: The Key to Achieving Faster Time-to-Market and Extending Product Life." White Paper. Motorola Corp. May 4, 2000.

[29] J. Wagner and R. Leupers, "C Compiler Design for a Network Processor." *IEEE transaction on computer-aided design of integrated circuits and systems* VOL.20, NO. 11, November 2001.

[30] A. V. Aho, R. Sethi, J. D. Ullman, *Compilers –principles techniques, and tools* Addison-Wesley Publishing Company, 1986.

[31] P.D. Terry, *Compilers and Compiler Generators - an introduction with C++*, 2000.

[32] J.K. Gough, "Syntax Analysis and software tools" Addison-Wesley Publishing Company, 1988.

[33] R. Gregerich, S.L. Graham, "Code Generation: concepts, tools, Techniques" Springer-Verlag 1991.

[34] I. Foster, *Designing and Building of Parallel Programs*. 1995.

[35] M. Quinn, *Parallel Computing : Theory and Practice*. McGraw-Hill, 1994.

[36] "High performance fortran - language specification. Technical report", Rice University, Houston Texas, 1993.

[37] O. Plachy, "Parallelization of Sequential Code - MPL, HPF and Automatic Parallelization" 1997.

[38] C.W. Fraser and D.R. Hanson, *A retargetable C compiler : design and implementation*, Redwood City, CA: Benjamin/Cummings Pub. Co., 1995.

[39] R. Stallman, "Using and Porting the GNU Compiler Collection(GCC), Calif.: Morgan Kaufmann Publishers, 1997.

[40] http://www.q-software-solutions.com/lccwin32/

[41] G. Amdahl, "Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities,"

[42] R. Das, J. Wu, J. Saltz, H. Berryman, S. Hiranandani, Distributed Memory Compiler Design For Sparse Problems.

[43] P. Banerjee, J. Chandy, M. Gupta, E. Hodge, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. The Paradigm Compiler for Distributed-Memory Multicomputers. IEEE Computer, 28(10):pp37--47, 1995.

[44] J. Wagner and R. Leupers, "C Compiler Design for a Network Processor." *IEEE transaction on computer-aided design of integrated circuits and systems* VOL.20, NO. 11, November 2001.

[45] LANCE - Retargetable C compiler, http://www.icd.de/es/lance/lance.html

[46] SPAM, http://www.princeton.edu/~mescal/spam/

[47] J.L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Seconded, Morgan Kaufmann Publishers, 1995.

[48] http://www.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/DLX.html

[49] P.M. Sailer and D. R. Kaeli, *The DLX Instruction Set Architecture Handbook,* Morgan Kaufmann Publishers, Inc. 1996.

[50] G. Booch, *Object-oriented analysis and design with applications*, 2nd ed., Redwood CA: Benjamin/Cummings, 1994.

[51] Open SystemC Initiative (OSCI), Functional Specification for SystemC 2.0, http://www.systemc.org, 2001.

[52] L. Charest, E.M. Aboulhamid, C. Pilkington, and P. Paulin, "SystemC Performance Evaluation Using A Pipelined DLX Multiprocessor," Proceedings of Design and Test in Europe Designers' Forum, Paris, pp. 8-12, 2002.

[53] J. Coplien, *Multi-Paradigm Design for C++*

[54] C. W. Fraser, D. Hanson, *A Retargetable C Compiler: Design and Implementation,* The Benjamin/Cummings Publishing Company, Inc. 1994.

[55] G. Kane, *MIPS RISC Architecture*, Englewood Cliffs, NJ: Prentice Hall, 1989.

[56] C. W. Fraser and D. R. Hanson, "A code generation interface for ANSI C", Software—Practice& Experience, 1991.

[57] E. Bergeron and E. Lesage, *EBEL-DLX,*

http://www.iro.umontreal.ca/~bergeret/EBEL-DLX/, 2001.

[58] J. Li, F.R. Boyer, and E.M. Aboulhamid, "Retargetable C Compiler for Network Processors," Proceedings of 6th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2002), Orlando, Fl. pp. 435, 2002.

[59] CoSy compiler development system, http://www.ace.nl

Appendix A: DLX instructions

Description	Mnemonic	Operands
load		
byte	LB	rd, offset(rs1)
byte unsigned	LBU	
halfword	LH	
halfword unsigned	LHU	
word	LW	
SP floating-point	LF	
DP floating-point	LD	
store		
byte	SB	offset(rs1), rd
halfword	SH	
word	SW	
SP floating-point	SF	
DP floating-point	SD	
move		
GPR to special register	MOVI2S	rd, rs1
special register to GPR	MOVS2I	
single FPR to single FPR	MOVF	
double FPR to double FPR	MOVD	
single FPR to GPR	MOVFP2I	
GPR to single FPR	MOVI2FP	
integer arithmetic		
add (signed)	ADD	rd, rs1, rs2
add unsigned	ADDU	
subtract (signed)	SUB	
subtract unsigned	SUBU	
multiply (signed)	MULT	
multiply unsigned	MULTU	
divide (signed)	DIV	
divide unsigned	DIVU	
add immediate (signed)	ADDI	rd, rs1, immediate
add unsigned immediate	ADDUI	
subtract immediate (signed)	SUBI	
subtract unsigned immediate	SUBUI	
logical		
and	AND	rd, rs1, rs2
or	OR	
xor	XOR	
and immediate	ANDI	rd, rs1, immediate
or immediate	ORI	
xor immediate	XORI	
load high immediate	LHI	rd, immediate
shift		
left logical	SLL	rd, rs1, rs2
right logical	SRL	
right arithmetic	SRA	
left logical immediate	SLLI	rd, rs1, immediate

right logical immediate	SRLI	
right arithmetic immediate	SRAI	
set-on-comparison		
less than	SLT	rd, rs1, rs2
greater than	SGT	
less than or equal to	SLE	
greater than or equal to	SGE	
equal to	SEQ	
not equal to	SNE	
less than immediate	SLTI	rd, rs1, immediate
greater than immediate	SGTI	
less than or equal to imm.	SLEI	
greater than or equal to imm.	SGEI	
equal to immediate	SEQI	
not equal to immediate	SNEI	
floating-point arithmetic		
add SP floating-point	ADDF	rd, rs1, rs2
add DP floating-point	ADDD	
subtract SP floating-point	SUBF	
subtract DP floating-point	SUBD	
multiply SP floating-point	MULTF	
multiply DP floating-point	MULTD	
divide SP floating-point	DIVF	
divide DP floating-point	DIVD	
convert		
SP to DP floating-point	CVTF2D	rd, rs2
SP floating-point to integer	CVTF21	
DP to SP floating-point	CVTD2F	
DP floating-point to integer	CVTD21	
integer to SP floating-point	CVT12F	
integer to DP floating-point	CVT12D	
set-on-comparison		
less than SP floating-point	LTF	rs1 rs2
less than DP floating-point	LTD	
greater than SPFP	GTF	
greater than DPFP	GTD	
less than or equal to SPFP	LEF	
less than or equal to DPFP	LED	
greater than or equal to SPFP		GEF
greater than or equal DPFP	GED	
equal to SP floating-point	EQF	
equal to DP floating-point	EQD	
not equal to FP	NEF	
not equal to DPFP	NED	

Description	Mnemonic	Operands
jump		
jump	J	name
jump and link	JAL	
jump register	JR	rs1
jump and link register	JALR	
branch		
on CPR equal to zero	BEQZ	rs1, name
on CPR not equal to zero	BNEZ	
on FP status register true	BFPT	name
on FP status register false	BFPF	
special		
trap	TRAP	name
return from exception	RFE	
no operation	NOP	

Appendix B: IBM PowerNP Instructions

(Grey part is reserved bits)

Integer Arithmetic Instructions

Name	0-5	6-10	11-15	16-20	21	22-30	31
addx	**31**	**D**	**A**	**B**	OE	**266**	**Rc**
addcx	**31**	**D**	**A**	**B**	**OE**	**10**	**Rc**
addex	**31**	**D**	**A**	**B**	**OE**	**138**	**Rc**
Addi	**14**	**D**	**A**	SIMM			
addic	**12**	**D**	**A**	**SIMM**			
addic.	**13**	**D**	**A**	**SIMM**			
addis	**15**	**D**	**A**	**SIMM**			
addmex	**31**	**D**	**A**	**00000**	**OE**	**234**	**Rc**
addzex	**31**	**D**	**A**	**00000**	**OE**	**202**	**Rc**
divwx	**31**	**D**	**A**	**B**	**OE**	**491**	**Rc**
divwux	**31**	**D**	**A**	**B**	**OE**	**459**	**Rc**
mulhwx	**31**	**D**	**A**	**B**	**0**	**75**	**Rc**
mulhwux	**31**	**D**	**A**	**B**	**0**	**11**	**Rc**
mulli	**07**	**D**	**A**	SIMM			
mullwx	**31**	**D**	**A**	**B**	**OE**	**235**	**Rc**
negx	**31**	**D**	**A**	**00000**	**OE**	**104**	**Rc**
subfx	**31**	**D**	**A**	**B**	**OE**	**40**	**Rc**
subfcx	**31**	**D**	**A**	**B**	**OE**	**8**	**Rc**
subficx	**08**	**D**	**A**	**SIMM**			
subfex	**31**	**D**	**A**	**B**	**OE**	**136**	**Rc**
subfmex	**31**	**D**	**A**	**00000**	**OE**	**232**	**Rc**
subfzex	**31**	**D**	**A**	**00000**	**OE**	**200**	**Rc**

Integer Compare Instructions

Name	0-5	6-8	9	10	11-15	16-20	21-30	31
cmp	**31**	**crfD**	**0**	**L**	**A**	**B**	**0000000000**	**0**
cmpi	**11**	**crfD**	**0**	**L**	**A**	SIMM		
cmpl	**31**	**crfD**	**0**	**L**	**A**	**B**	**32**	**0**
cmpli	**10**	**crfD**	**0**	**L**	**A**	UIMM		

Integer Logical Instructions

Name	0-5	6-10	11-15	16-20	21-30	31
andx	31	S	A	B	28	Rc
andcx	31	S	A	B	60	Rc
andi.	28	S	A	UIMM		
andis.	29	S	A	UIMM		
cntlzwx	31	S	A	00000	26	Rc
eqvx	31	S	A	B	284	Rc
extsbx	31	S	A	00000	954	Rc
extshx	31	S	A	00000	922	Rc
nandx	31	S	A	B	476	Rc
norx	31	S	A	B	124	Rc
orx	31	S	A	B	444	Rc
orcx	31	S	A	B	412	Rc
ori	24	S	A	UIMM		
oris	25	S	A	UIMM		
xorx	31	S	A	B	316	Rc
xori	26	S	A	UIMM		
xoris	27	S	A	UIMM		

Integer Rotate Instructions

Name	0-5	6-10	11-15	16-20	21-25	26-30	31
riwimix	22	S	A	SH	MB	ME	Rc
riwinmx	20	S	A	SH	MB	ME	Rc
riwnmx	21	S	A	SH	MB	ME	Rc

Integer Shift Instructions

Name	0-5	6-10	11-15	16-20	21-30	31
slwx	31	S	A	B	24	Rc
srawx	31	S	A	B	792	Rc
srawix	31	S	A	SH	824	Rc
srw x	31	S	A	B	536	Rc

Floating-Point Arithmetic Instructions

Name	0-5	6-10	11-15	16-20	21-25	26-30	31
fadd x	63	D	A	B	00000	21	Rc
fadds x	59	D	A	B	00000	21	Rc
fdiv x	63	D	A	B	00000	18	Rc
fdivs x	59	D	A	B	00000	18	Rc
fmul x	63	D	A	00000	C	25	Rc
fmuls x	59	D	A	00000	C	25	Rc
fres x 1	59	D	00000	B	00000	24	Rc
frsqrte x 1	63	D	00000	B	00000	26	Rc
fsub x	63	D	A	B	00000	20	Rc
fsubs x	59	D	A	B	00000	20	Rc
fsel x 1	63	D	A	B	C	23	Rc
fsqrt x 1	63	D	00000	B	00000	22	Rc
fsqrts x 1	59	D	00000	B	00000	22	Rc

Floating-Point Multiply-Add Instructions

Name	0-5	6-10	11-15	16-20	21-25	26-30	31
fmadd x	63	D	A	B	C	29	Rc
fmadds x	59	D	A	B	C	29	Rc
fmsub x	63	D	A	B	C	28	Rc
fmsubs x	59	D	A	B	C	28	Rc
fnmadd x	63	D	A	B	C	31	Rc
fnmadds x	59	D	A	B	C	31	Rc
fnmsub x	63	D	A	B	C	30	Rc
fnmsubs x	59	D	A	B	C	30	Rc

Floating-Point Rounding and Conversion Instructions

Name	0-5	6-10	11-15	16-20	21-30	31
fctiw x	63	D	00000	B	14	Rc
fctiwz x	63	D	00000	B	15	Rc
frsp x	63	D	00000	B	12	Rc

Floating-Point Compare Instructions

Name	0-5	6-8	9-10	11-15	16-20	21-30	31
fcmpo	63	crfD	00	A	B	32	0
fcmpu	63	crfD	00	A	B	0	0

Floating-Point Status and Control Register Instructions

Name	0-5	6 7 8	9 10	11-13	14 15	16-20	21-30	31
mcrfs	63	crfD		crfS	0 0	00000	64	0
mffs x	63	D		00000		00000	583	Rc
mtfsb0 x	63	crbD		00000		00000	70	Rc
mtfsb1 x	63	crbD		00000		00000	38	Rc
mtfsf x	31	0	FM		0	B	711	Rc
mtfsfi x	63	crfD	0 0	00000		IMM	0 134	Rc

Integer Load Instructions

Name	0-5	6-10	11-15	16-20	21-30	31
lbz	34	D	A	d		
lbzu	35	D	A	d		
lbzux	31	D	A	B	119	0
lbzx	31	D	A	B	87	0
lha	42	D	A	d		
lhau	43	D	A	d		
lhaux	31	D	A	B	375	0
lhax	31	D	A	B	343	0
lhz	40	D	A	d		
lhzu	41	D	A	d		
lhzux	31	D	A	B	311	0
lhzx	31	D	A	B	279	0
lwz	32	D	A	d		
lwzu	33	D	A	d		
lwzux	31	D	A	B	55	0
lwzx	31	D	A	B	23	0

Integer Store Instructions

Name	0-5	6-10	11-15	16-20	21-30	31
stb	38	S	A	d		
stbu	39	S	A	d		
stbux	31	S	A	B	247	0
stbx	31	S	A	B	215	0
sth	44	S	A	d		
sthu	45	S	A	d		
sthux	31	S	A	B	439	0
sthx	31	S	A	B	407	0
stw	36	S	A	d		
stwu	37	S	A	d		
stwux	31	S	A	B	183	0
stwx	31	S	A	B	151	0

Integer Load and Store with Byte Reverse Instructions

Name	0-5	6-10	11-15	16-20	21-30	31
lhbrx	31	D	A	B	790	0
lwbrx	31	D	A	B	534	0
sthbrx	31	S	A	B	918	0
stwbrx	31	S	A	B	662	0

Integer Load and Store Multiple Instructions

Name	0-5	6-10	11-15	16-31
lmw 1	46	D	A	d
stmw 1	47	S	A	d

Integer Load and Store String Instructions

Name	0-5	6-10	11-15	16-20	21-30	31
lswi 1	31	D	A	NB	597	0
lswx 1	31	D	A	B	533	0
stswi 1	31	S	A	NB	725	0
stswx 1	31	S	A	B	661	0

Memory Synchronization Instructions

Name	0-5	6-10	11-15	16-20	21-30	31
eieio	31	00000	00000	00000	854	0
isync	19	00000	00000	00000	150	0

lwarx	31	D	A	B	20	0
stwcx.	31	D	A	B	150	1
sync	31	00000	00000	00000	598	0

Floating-Point Load Instructions

Name	0-5	6-10	11-15	16-20	21-30	31
lfd	50	D	A	d		
lfdu	51	D	A	d		
lfdux	31	D	A	B	631	0
lfdx	31	D	A	B	599	0
lfs	48	D	A	d		
lfsu	49	D	A	d		
lfsux	31	D	A	B	567	0
lfsx	31	D	A	B	535	0

Floating-Point Store Instructions

Name	0-5	6-10	11-15	16-20	21-30	31
stfd	54	S	A	d		
stfdu	55	S	A	d		
stfdux	31	S	A	B	759	0
stfdx	31	S	A	B	727	0
stfiwx 1	31	S	A	B	983	0
stfs	52	S	A	d		
stfsu	53	S	A	d		
stfsux	31	S	A	B	695	0
stfsx	31	S	A	B	663	0

Floating-Point Move Instructions

Name	0-5	6-10	11-15	16-20	21-30	31
fabs x	63	D	00000	B	264	Rc
fmr x	63	D	00000	B	72	Rc
fnabs x	63	D	00000	B	136	Rc
fneg x	63	D	00000	B	40	Rc

Branch Instructions

Name	0-5	6-10	11-15	16-20	21-29	30	31
b x	18	LI				AA	LK
bc x	16	BO	BI	BD		AA	LK
bcctr x	19	BO	BI	00000	528		LK
bclr x	19	BO	BI	00000	16		LK

Condition Register Logical Instructions

Name	0-5	6-8 9 10	11-13 14 15	16-20	21-30	31
crand	19	crbD	crbA	crbB	257	0
crandc	19	crbD	crbA	crbB	129	0
creqv	19	crbD	crbA	crbB	289	0
crnand	19	crbD	crbA	crbB	225	0
crnor	19	crbD	crbA	crbB	33	0
cror	19	crbD	crbA	crbB	449	0
crorc	19	crbD	crbA	crbB	417	0
crxor	19	crbD	crbA	crbB	193	0
mcrf	19	crfD 0 0	crfS 0 0	00000	0000000000	0

System Linkage Instructions

Name	0-5	6-10	11-15	16-20	21-29 30	31
rfi 1	19	00000	00000	00000	50	0
sc	17	00000	00000	000000000000000 1		0

Trap Instructions

Name	0-5	6-10	11-15	16-20	21-30	31
tw	31	TO	A	B	4	0
twi	03	TO	A	SIMM		

Processor Control Instructions

Name	0-5	6-8 9 10	11 12-15	16-19 20	21-30	31
mcrxr	31	crfS 0 0	00000	00000	512	0
mfcr	31	D	00000	00000	19	0
mfmsr 1	31	D	00000	00000	83	0
mfspr 2	31	D	spr		339	0
mftb	31	D	tpr		371	0
mtcrf	31	S	0 CRM	0	144	0
mtmsr 1	31	S	00000	00000	146	0
mtspr 2	31	D	spr		467	0

Cache Management Instructions

Name	0-5	6-10	11-15	16-20	21-30	31
dcba 1	31	00000	A	B	758	0
dcbf	31	00000	A	B	86	0
dcbi 2	31	00000	A	B	470	0
dcbst	31	00000	A	B	54	0
dcbt	31	00000	A	B	278	0
dcbtst	31	00000	A	B	246	0
dcbz	31	00000	A	B	1014	0
icbi	31	00000	A	B	982	0

Segment Register Manipulation Instructions.

Name	0-5	6-10	11	12-15	16-20	21-30	31
mfsr 1	31	D	0	SR	00000	595	0
mfsrin 1	31	D	00000		B	659	0
mtsr 1	31	S	0	SR	00000	210	0
mtsrin 1	31	S	00000		B	242	0

Lookaside Buffer Management Instructions

Name	0-5	6-10	11-15	16-20	21-30	31
tlbia 1,2	31	00000	00000	00000	370	0
tlbie 1,2	31	00000	00000	B	306	0
tlbsync 1,2	31	00000	00000	00000	566	0

External Control Instructions

Name	0-5	6-10	11-15	16-20	21-30	31
eciwx	31	D	A	B	310	0
ecowx	31	S	A	B	438	0