TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Integrierte Systeme

A Network Processor Architecture for High Speed Carrier Grade Ethernet Networks

Kimon Karras

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr.-Ing. Wolfgang Kellerer

Prüfer der Dissertation:

1. Univ.-Prof. Dr. sc.techn. Andreas Herkersdorf
2. Univ.-Prof. Dr.-Ing. Andreas Kirstädter, Universität Stuttgart

Die Dissertation wurde am 13.05.2013 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 24.03.2014 angenommen.

Preface

This work was performed at the Institute for Integrated Systems of the Technical University of Munich under the auspices of Prof. Dr. sc.techn. Andreas Herkersdorf and Dr.-Ing. Thomas Wild, both of whom I have to whole-heartedly thank for giving me the opportunity of working with them and for their invaluable help and guidance throughout the five years of my stay at the institute.

A further thanks goes to all the people at the institute with whom I've cooperated over the years and who have made life enjoyable both on a professional and on a personal level. A very great thanks is due to Daniel Llorente for keeping me company and bearing with me during most of this time. His contribution to this thesis is immense and without him it wouldn't have been possible. Muchas gracias, *compañero*!

Furthermore I would like to thank all the partner companies and participants of the 100GET project, which formed the basis for this disseration. It was a pleasure to work with all of you on such an interesting project and to exchange ideas with such a distiguished crowd. Special credit is due to Dr.-Ing. Thomas Michaelis, who coordinated the 100GET-E3 subproject masterfully and without whom reaching the project goals would have been that much more difficult.

Gratitude is always owned to all my friends in Munich, in no particular order, Iason Vittorias, Akis Kipouridis, Harry Daniilidis, Christian Fuchs, Theodoros Papadopoulos, Panayiotis Mitsakis, Tilemachos Matiakis and Nana Kouri, as well as many others who made sure that all work and no play didn't make Jack a dull boy after all. These five years have been some of the most amazing thus far and I have all of you to thank for it. And finally, last but not least, Katerini Papathanasiou for lending a shoulder upon which to lean on, in order to see this dissertation through.

Munich, March 2013 Kimon Karras

Abstract

Next generation core networks will raise the performance bar for processing modules to at least 100 Gbps throughput on a single port. Currently Network Processor (NP) vendors are dealing with higher throughput demands, by increasing the number of processing cores in their design. The NPU100 is a novel network processor architecture aimed at satisfying the packet processing requirements of 100 Gbps Carrier Grade Ethernet (CGE) networks, while reducing power consumption in comparison to available architecture paradigms. To accomplish this, a multi-layered, deeply pipelined and programmable architecture was proposed and implemented as an FPGA prototype. The NPU 100 architecture can be extended and tailored to the demands of various layer 2.5 network architectures.

Conventional programmable approaches, either based on massively parallel general purpose CPUs or multiple lower rate NPs, do not scale towards high speed carrier networks from a cost, performance (packet throughput) and power dissipation perspective. Architecture design and development of an integrated, programmable component that satisfies these requirements is a scientific and technical challenge.

The NPU100 architecture is based on a pipelined approach because of its inherently predictable behavior and scalable performance. In order to achieve modularity and extendibility, NPU100 processing modules can be rearranged within each configuration. The NPU100 design is divided in different levels of hierarchy, which can be replicated or arranged as required. Multiple pipeline stages were grouped into entities called mini pipelines, which were then placed in parallel to one another. A mini pipeline includes the processing modules required to complete the quintessential network processing tasks, like address look ups, packet header field extractions and insertions, and logical and arithmetic operations.

In order to provide necessary degrees of flexibility in case of changing network protocols or networking applications, the NPU100 pipeline modules are utilizing so called Weakly Programmable Processing Engines (WPPEs). A WPPE is less flexible than, for example, a RISC core. It has a reduced networking-specific instruction set (a total of 8 microcommands per module) which is, however, sufficiently flexible to adapt its functionality to layer 2.5 networking requirements. WPPEs are programmed in a high level, C-like, programming language. A compiler is provided to translate the high level code into the microcommand machine language and to assign the respective microcommands to the appropriate processing modules. WPPE microprograms can be modified during runtime via a PCI Express interface without disrupting NPU100 processing. This easy-to-use programming scheme is one of the core contributions of this work and allows the user to easily leverage the NPU100's processing resources without requiring in-depth architectural knowledge or the writing of low-level code.

The folded, parallel arrangement of NPU100 minipipelines offers significant improvement in power consumption in comparison to a conventional stretched pipelined architecture where all minipipelines are arranged in series. This is due to a combination of two factors: In case of multiple, stacked MPLS labels or generally, tunneled packet headers, a minipipeline takes care for the processing of a single label or header. The number of parallel minipipelines determines the maximum depth of packet header stacking. In case of packets where fewer (than the maximum number of) headers have to be processed, packet headers can skip the unused minipipelines which can then be dynamically clock or power gated. Second, only one label or header must be transported through each minipipeline.

This leads to a narrower data path in comparison to the stretched pipeline where all headers traverse all pipeline stages. XPower analysis revealed that the reduction in data path width resulted in a power consumption improvement of 25%, while the reduction of traversed pipeline stages can bring this figure up to 78% in the most favorable traffic scenario. The implemented FPGA prototype runs at a system clock of 66 MHz, accepting a new 192 bit packet header with every clock cycle and obtaining a throughput of up to 46 Gbps. Synthesis runs with a TSMC 65nm CMOS technology library showed that an ASIC realization of the design exceeded the 100 Gbps requirements.

Zusammenfassung

Kernnetzwerke der nächsten Generation werden das von Paketverarbeitungsmodulen verlangte Leistungsniveau auf mindestens einen 100 Gbit/s Durchsatz pro Port erhöhen. Aktuelle Netzwerkprozessoranbieter bewältigen den höheren Durchsatzbedarf durch eine Steigerung der Anzahl der Bearbeitungsmodule in deren Entwürfe. Der NPU100 ist ein neuartiger Netzwerkprozessor (NP), der danach strebt, den Paketverarbeitungbedarf der 100 Gbit/s Carrier Grade Ehernet (CGE) Netzwerke zu erfüllen. Um das zu erzielen, eine mehr-schichtige, tief gepipelined und programmierbare Architektur wurde eingeführt und als FPGA Prototyp implementiert. Die NPU100 Architektur kann leicht erweitert und an die Anforderungen der verschiedenen Schicht 2.5 Protokolle angepasst werden.

Konventionelle programmierbare Ansätze werden entweder auf eine massive Anzahl von CPUs oder auf mehreren NPs die einen niedrigen Durchsatz erreichen, basiert. Diese Ansätze skalieren sich nicht von einer Kosten-, Leistungsfähigkeit- (Paket Durchsatz) sowie auch Verlustleistungsgesichtspunkt. Der Entwurf und die Entwicklung einer Architektur für einen integrierten, programmierbaren Baustein, der alle diese Anforderungen gleichzeitig anspricht ist eine wissenschaftliche und technische Herausforderung.

Die NPU100 Architektur wird auf einer Pipeline Ansatz basiert, um deren inhärent vorhersehbares Verhalten und skalierbare Leistungsfähigkeit auszunutzen. Um die Ziele der Modularität und der Erweiterbarkeit zu erreichen, können NPU100 Module beliebig umgeordnet werden. Der NPU100 Entwurf wird in verschiedene Hierarchieebenen unterteilt, die abhängig von den jeweiligen Anforderungen redupliziert oder umgestaltet werden können. Mehrere Pipeline Stufen werden in Einheiten, die Minipipelines getauft wurden, groupiert. Die werden dann parallel zu einander geordnet. Eine Minipipeline beinhaltet alle Module, die für die Verarbeitung der wesentlichen Netzwerkaufgaben erforderlich sind, wie zum Beispiel Address Look Ups, Paketheader Feldextrahierungen und logische und arithmetische Operationen.

Um die von verschiedenen Netzwerkprotokole beziehungsweise Netzwerkanwendungen benötigte Funktionalität zu erzielen, wurden beim Entwurf der NPU100 Pipelinemodulen so genannte Weakly Programmable Processing Engines (WPPEs) eingesetzt. Eine WPPE ist nicht so flexibel wie, zum Beispiel, ein RISC Prozessorkern. Die verfügt über einen eingeschränkten netzwerk-spezifischen Befehlsatz (insgesamt 8 Microcommands pro Modul), der sich aber als ausreichend erweist, um die Schicht 2.5 Netzwerkanforderungen abzudecken. WPPEs werden durch eine high-level, C-ähnliche Programmiersprache programmiert. Ein Compiler wird zur Verfügung gestellt, der den high level Code in Microcommand Maschinensprache umwandelt und die Microcommands den entsprechenden, passenden Modulen

zuweist. WPPE Microprogramme können während der Laufzeit über eine PCI Express Schnittstelle geändert werden, ohne dass die Bearbeitung des NPU100 unterbrochen oder beeinflusst wird. Dieses einfach zu benutzen Schema ist einer der Kernbeiträge dieser Arbeit und erlaubt dem Benutzer den problemlosen Einsatz der vom NPU100 angeboten Ressourcen ohne tiefreichende Kenntnisse der Architektur und ohne low-level Code zu schreiben.

Die gefaltete, parallele Organisierung der NPU100 Minipipelines bietet eine erhebliche Verbesserung der Verlustleistung im Vergleich zu einem konventionellen, gestreckten Pipelinearchitektur, in der alle Minipipelines nacheinander geordnet sind. Dies passiert wegen zwei Faktoren: Wenn mehrere, gestapelte MPLS Label, oder allgemeiner getunnelter Paketheader, sich im Labelstapel befinden, dann ist jeweils eine Minipipeline für die Verarbeitung eines Labels beziehungsweise Paketheaders zuständig. Die Anzahl der parallelen Minipipelines legt die maximale Tiefe des Labelstapels fest. Falls Pakete, von der wenigere (als die maximale erlaubte Anzahl) Headers bearbeitet werden müssen, ankommen, dann können Paketheader die unbenötigte Minipipelines überspringen. Diese Minipipelins können dann clock oder power gated werden. Ferner, nur ein Label beziehungsweise Paketheader muss durch jede Minipipeline mitgeführt werden. Infolgedessen, ergibt sich ein schmällerer Datenpfad daraus im Vergleich zu der gestreckten Pipeline, in der alle Paketheader alle Minipipelines durchlaufen müssen. Eine XPower Analyse zeigte, dass die Verringerung des Datenpfadbreites zu einem Verlustleistungersparniss von 25% geführt hat. Durch die reduzierte Anzahl der Minipipelinedurchläufe kann im optimalsten Szenario diese Zahl bis auf 78% steigern. Der FPGA Prototyp läuft bei einer 60MHz Frequenz und verarsbeitet einen neuen 192 Bit Paketheader in jedem Takt. So wird ein Durchsatz von 46 Gbit/s erreicht. Eine Synthese mit einer TSMC 65nm CMOS Bibliothek zeigte allerdings, dass eine ASIC Realisierung des Entwurfes den 100 Gbit/s Zieldurchsatz weit überschreiten wird.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Goals of the thesis . . . 3
- 1.3 Structure of the thesis . . . 4

2 Fundamentals and Related Work **7**
- 2.1 Carrier Grade Ethernet Networks . . . 8
- 2.2 Carrier Grade Ethernet Protocols . . . 9
  - 2.2.1 Multiple Protocol Label Switching - Transport Profile (MPLS-TP) . 9
  - 2.2.2 Provider Backbone Bridging - Traffic Engineering (PBB-TE) . . . 15
- 2.3 Carrier Grade Ethernet Node Architecture . . . 17
- 2.4 Processing Architectures Overview . . . 19
  - 2.4.1 Network Processors . . . 20
  - 2.4.2 Other Processing Concepts . . . 27
  - 2.4.3 Useful Conclusions . . . 28
- 2.5 Programming Network Processors . . . 29
- 2.6 Hashing . . . 30
- 2.7 Interfaces . . . 32
  - 2.7.1 PCI Express . . . 32
  - 2.7.2 Interlaken . . . 34

3 NPU100 Architecture **35**
- 3.1 NPU100 Outer Layer . . . 36
  - 3.1.1 Header-Payload Split Concept . . . 37
  - 3.1.2 NPU100 Internal Header . . . 39
- 3.2 A Folded Pipeline Architecture for Low Power Consumption in High Speed Carrier Grade Network Processors . . . 40
  - 3.2.1 Architectures Considered . . . 42
  - 3.2.2 Architecture Evaluation . . . 50
- 3.3 A Distributed Programming Paradigm for Pipelined Network Processors . 64

4 NPU100 System Implementation **69**
- 4.1 Outer Layer Implementation . . . 69
  - 4.1.1 Outer Layer Module overview . . . 69
  - 4.1.2 Header-Payload Split . . . 72
  - 4.1.3 Packet Reassembly . . . 75
  - 4.1.4 Main Memory . . . 83

4.2 NPU100 Processing Pipeline 84
4.2.1 Dispatcher 84
4.2.2 Label Reassembly 88
4.2.3 Label Buffers 91
4.2.4 Pipeline Modules 98
4.3 NPU100 Programming Concept 120
4.3.1 Addressing Scheme and Programming Table Access 121
4.3.2 Hardware Modules 124
4.3.3 Programming the NPU100 125

5 NPU100 Implementation Results and Testbed 141
5.1 Implementation Results 141
5.1.1 Resource Consumption 141
5.1.2 Power Consumption 144
5.1.3 Performance 149
5.1.4 Scalability 151
5.2 NPU100 Test Bed 153
5.2.1 NPU100 Test Bed Description 153
5.2.2 NPU100 Verification and Test Bed Results 158

6 Conclusions 161

Bibliography 163

List of Figures

1.1 Drivers for a Reduced Reliance on Scaling as a Pillar for Power Reduction in Modern ICs . . . 2

2.1 MPLS Label Field Structure . . . 10
2.2 Overview of the MPLS Forwarding Process . . . 11
2.3 Example of an MPLS-TP LSP with one Tunneling Layer . . . 12
2.4 Example of an MPLS-TP LSP with two Tunneling Layers . . . 13
2.5 Overview of the MPLS Generic ACH Message . . . 14
2.6 Overview of the PBB-TE Frame Development over the Years . . . 16
2.7 Processing Steps Required by a PBB-TE Frame . . . 17
2.8 Overview of an Example 100 Gbit/s CGE Node Architecture . . . 18
2.9 Netronome NFP-32xx Overview [13] . . . 21
2.10 Block Diagram of the New Generation IBM NPU [37] . . . 22
2.11 Block Diagram of the Freescale QorIQ AMP Series T2080 Communications Processor . . . 23
2.12 Architectural Overview of the Cisco ASR 1000 Embedded Services Processor [24] . . . 24
2.13 Block Diagram of the LSI APP3300 NP Family [26] . . . 25
2.14 Overview of the DataFlow Architecture used in the Xelerated NPs [62] . . . 26
2.15 1:1 and M:1 Hash Mapping Schemes . . . 31
2.16 PCI Express Layer Stack . . . 33

3.1 High-level Block Diagram of the NPU100 System . . . 36
3.2 Block Diagram of the NPU100 Outer Layer, which Shows how the Header is Copied and Sent to the Pipeline core while the Complete Packet is Stored in the Packet Memory. . . . 39
3.3 Structure of the NPU100 Internal Header . . . 39
3.4 Overview of a Conventional Pipeline Architecture . . . 42
3.5 Block Diagram of the Registered Architecture . . . 43
3.6 Block Diagram of the Folded Pipeline Architecture . . . 45
3.7 NPU100 Mini-Pipeline Stages Derived from CGE Protocol Requirements . . . 48
3.8 General NPU100 Pipeline Word Structure, along with MPLS-TP and PBB-TE Variants . . . 50
3.9 Folded Architecture Dynamic Register Power Consumption Reduction Relative to the Conventional Pipeline . . . 53
3.10 Registered Architecture Dynamic Register Power Consumption Reduction Relative to the Conventional Pipeline . . . 54
3.11 Implementation of a Registered Architecture Prototype . . . 59

3.12 Registered Architecture Resource Consumption and Frequency for Various PE Numbers . . . 60
3.13 Folded Pipeline Architecture Modelled in xPower . . . 61
3.14 Block Diagram of the NPU100 Configuration Architecture . . . 65
3.15 Forwarding and QoS Table Entry Generation Process . . . 66
3.16 Microcommand Programming Information Generation Process . . . 66

4.1 Overview of the Architecture of the NPU100 Outer Layer with Bus Widths and Operating Frequencies . . . 70
4.2 Utilized Link Throughput and Required NPU100 Internal Bus Frequency per Packet Size . . . 72
4.3 Overview of the Header Payload Split Submodule . . . 73
4.4 State Diagram of the Header-Payload Split Module. Solid Lines Show Transitions During Data Input, Dashed Lines Transitions when a Packet is Finished and no New Packet is to Start in the Following Clock Cycle and Dotted Lines transitions when a Packet is Finished and is Trailed by a New Packet in the Next Clock Cycle. . . . 76
4.5 Packet Reassembly Submodule Block Diagram . . . 77
4.6 Header Alignment Process Example in the Packet Reassembly Module . . . 78
4.7 Process Determining the Label Composition of the Reassembled Packet Header Depending on the Sequence of Operations Performed in the Pipeline Core . . . 79
4.8 Example of an LSP in which the packets enter a double tunnel at node 5 . . . 80
4.9 Example Packet Reassembly in a Pop, Pop, Swap and Push Scenario . . . 81
4.10 Example Packet Reassembly in a Swap, Push, Push Scenario . . . 81
4.11 Packet Reassembly Process Illustrating Two Cases, the Top One where the Output Packet Grows in Comparison to the Input One, and the Bottom One where it Shrinks . . . 82
4.12 Overview of the Dispatcher Design, Showing the Three Parallel Pipelines which Make up the Module . . . 86
4.13 Dispatcher Label Buffer Input Vector for Various Values of the Maximum Label Size Generic and the Actual Label Size Currently Being Processed in the Pipeline . . . 87
4.14 Overview of the Label Reassembly Design with the Three Parallel Pipelines Demarcated . . . 90
4.15 Overview of the Label Buffer Time Slot Concept . . . 93
4.16 Overview of the Buffer Design for Three Mini Pipelines and Thirteen Mini Pipeline Stages . . . 96
4.17 Reassembly Buffer Cell Block Diagram . . . 97
4.18 Overview of the Reassembly Buffer Structure for Two Minipipelines and Five Stages per Mini Pipeline . . . 99
4.19 ISU bit Selection and Field Extraction, Followed by the Processing of Data in the PE and Recombination of Processing Results and Original Register Data in the OSU . . . 100

4.20 Example of the Processing Results Being Reinserted into the Pipeline Word by the OSU . . . 101
4.21 Microcommand Table Entry Structure for the ISU and OSU . . . 105
4.22 Block Diagram of the Forwarding Lookup Module . . . 106
4.23 Performance of Three Prominent Hash Algorithms for Various Bin and CAM Sizes . . . 109
4.24 Block Diagramm of the Black Sheep Memory Block . . . 111
4.25 FPGA Resource Consumption for Various CAM Sizes . . . 111
4.26 Block Diagram of the Forwarding Lookup Write Control Architecture . . . 115
4.27 Overview of the QoS Parameter Lookup Operation . . . 118
4.28 Overview of the Insert Operation . . . 120
4.29 NPU100 Configuration Addressing . . . 121
4.30 Block Diagram of Hardware Modules Making Up the PCI Express Interface 124
4.31 Example of the Definition of a Field in the PIF . . . 127
4.32 Example of a QoS Lookup Table Entry Definition in the LIF . . . 129
4.33 Flow Diagram of the CPF Compilation Process . . . 132
4.34 Examples for Different Task Allocation to the PEs . . . 134
4.35 Further Examples of Different Task Allocation to the PEs . . . 135
4.36 Example of Statement Mapping to PEs with the MP Loop Being Used . . . 137
4.37 Allocation of QoS Subtable Entry Words to Programming Words . . . 138

5.1 Resource Use Break Down between the Various NPU100 Modules . . . 143
5.2 Resource Use Break Down between the Various Pipeline Core Modules . . . 143
5.3 Resource Use Break Down between the Various Mini Pipeline Modules . . . 144
5.4 Dynamic Power Consumption per Packet Size for Loop Run Scenarios One and Two . . . 146
5.5 Dynamic Power Consumption per Packet Size for Loop Run Scenarios Three and Four . . . 146
5.6 Dynamic Power Consumption per Packet Size for Traffic Scenarios I through IV as Detailed in Table 5.3 . . . 147
5.7 Dynamic Power Consumption per Packet Size for PBB-TE Processing . . . 148
5.8 Critical Path Constraining the Performance of the NPU100 . . . 150
5.9 Reduction of the Critical Path by Pipelining . . . 151
5.10 Die Shot of a Fully Routed Modify Module . . . 152
5.11 Dispatcher and Label Reassembly Module Resource Consumption for 4 and 8 Mini-Pipelines . . . 153
5.12 Overview of the NPU100 Testbed Setup . . . 155
5.13 Ethernet Interface Module Chain Block Diagram . . . 156

List of Tables

2.1 Meaningful MPLS-TP Operation Sequences 14

3.1 Required Processing Stages for CGE Protocol Processing 47
3.2 Percentage of Labels to be Processed for Different Traffic Scenarios 53
3.3 Summary of the Qualitative Analysis's Findings 57
3.4 Estimation of the Activity Factor for the NPU100 Modules 62
3.5 Power Consumption Data for Conventional and Folded Pipeline Architectures 62
3.6 Resource Consumption for Conventional and Folded Pipeline Architectures 63
3.7 Impact of the Pipeline Width on the ISU/OSU Latency 64

4.1 Supported MPLS-TP Operations and Required Label Reassembly Action . 91
4.2 LUT Use and Combinatorial Path for Various number of Inputs, Input Width and Granularity Combinations in the ISU 103
4.3 LUT Use and Combinatorial Path for Various Number of Inputs, Input Width and Granularity Combinations in the OSU 104
4.4 NPU100 Configuration Address Assignment 123

5.1 Detailed Overview of the Resource Consumption for the Various NPU100 Modules 142
5.2 Meaningful MPLS-TP Operation Sequences for which Power Consumption was Calculated 145
5.3 Percentage of Labels to be Processed for Different Traffic Scenarios for which Power Consumption was Calculated 146
5.4 Detailed Overview of the Power Dissipation for the Various NPU100 Modules 149
5.5 Resource Consumption of the NPU100 Testbed Ethernet Block 157

Notations

Abbreviations

ALU	Arithmetic Logic Unit
APS	Automatic Protection Switching
ATM	Asynchronous Trasnfer Mode
CAM	Content Addresable Memory
CAPEX	CAPital EXpenditure
CGE	Carrier Grade Ethernet
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DA	Destination Address
DPI	Deep Packet Inspection
ECMP	Equal Cost Multi Path
FCAPS	Fault, Configuration, Accounting, Performance and Security
FIFO	First in First Out
FSM	Finite State Machine
G-ACH	Generic Associated CHannel
GAL	Generic Associated channel header Label
GPU	Graphics Processing Unit
IP	Internet Protocol
LAN	Local Area Network
LSB	Least Significant Bit
LIFO	Last In First Out
LSP	Label Switching Path
MAC	Medium Access Control
MP	Mini Pipeline
MPLS	Multi Protocol Label Switching
MPLS-TP	Multi Protocol Label Switching - Transport Profile
MSB	Most Significant Bit
NP	Network Processor
NSN	Nokia Siemens Networks
OS	Operating System
PBB	Provider Backbone Bridging
PBB-TE	Provider Backbone Bridging - Traffic Engineering
PCB	Printed Circuit Board
PHB	Per Hop Behavior

PHP	Penultimate Hop Popping
PCIe	PCI Express
PE	Processing Engine
PnP	Plug and Play
PPR	Post Place and Route
QoS	Quality of Service
RISC	Reduced Instruction Set Computer
SA	Source Address
SIMD	Single Instruction Multiple Data
SLA	Service Level Agreement
TLV	Type Length Value
TLP	Transaction Level Packet
VLAN	Virtual Local Area Network
WAN	Wide Area Network
XML	eXtensible Markup Language

1 Introduction

1.1 Motivation

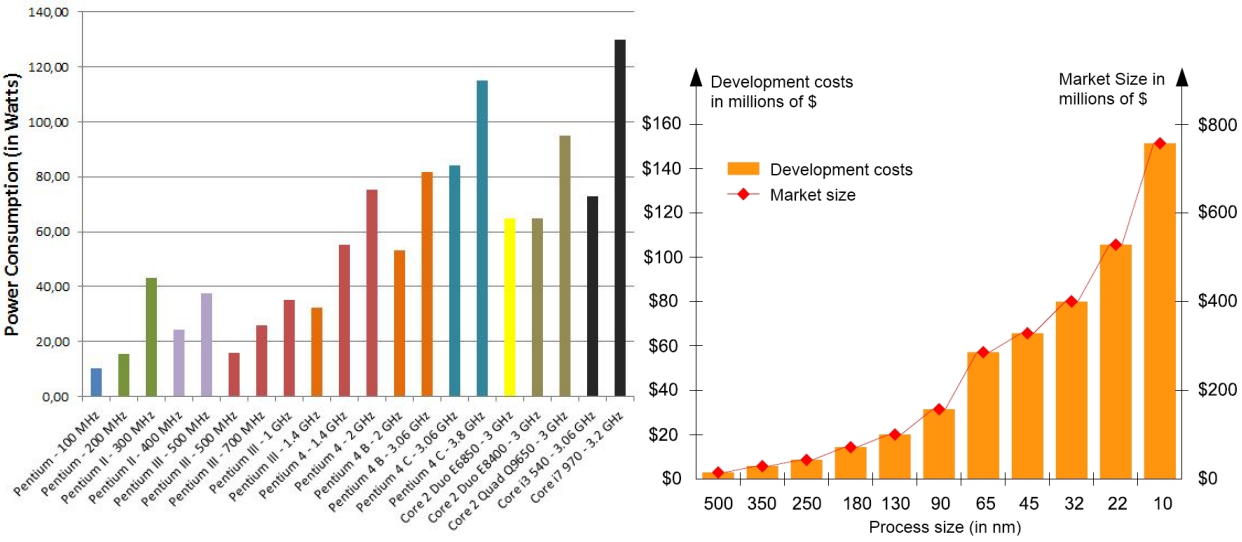
The semiconductor industry has gone from a nascent to a multi-billion dollar industry within the last 40 years. One of the main reasons behind this transformation was the constant reduction in the feature size of transistors, which has led silicon chips to become one of the most ubiquitous items on the planet. The reason for this is that reducing transistor size was a win-win solution. It allowed for shrinking an integrated circuit and its operating voltage, without any obvious drawbacks, despite the considerable, ever-increasing effort that has to be put in accomplishing these die shrinks. As equation 1.1, which is used to calculate the dynamic power consumption of a CMOS circuit, shows, reducing the voltage supply (V) in particular but also the capacitance (C) severely limits dynamic power dissipation. From the remaining variables f is the operating frequency of the circuit and a is the switching factor, a design dependent constant which is used to indicate how often do bits toggle on average in a given circuit.

$$P = aCV^2f \tag{1.1}$$

Based on this principle, ICs continued to grow and become more complex, provide all sorts of functionality and permeate almost every aspect of modern life. The wake up call came in 2004, when Intel's ultra deeply pipelined Prescott Pentium 4 core, proved to be extremely power hungry (to say nothing of under performing). This was the first time that a process shrink (in this case from 130nm in the previous Northwood core Pentium 4 family to the 90nm Prescott) failed to deliver where power consumption reduction is concerned. Figure 1.1a clearly demonstrates this trend for Intel CPUs (used here just as a reference). Power consumption clearly increases more or less linearly within CPUs of the same architecture, however when a process change occurs there is almost always an impressive drop in power consumption. This trend was broken with the Pentium 4C, as mentioned above. From this point on, the process gains from a die shrink are reduced considerably. In fact, power consumption remains more or less steady, which means any gains are offset by the increase in circuit complexity.

Further exacerbating the issue are the mounting development costs for each new process. Figure 1.1b clearly illustrates this exponential rise. Combining these two facts, one is led to the conclusion that an always increasing amount of money has to be spent to realize new processes, which however fail to deliver the enormous benefits of the past.

A direct consequence of these facts is that it is now up to the engineers to devise new architectural solutions in order to reign in power consumption, while maintaining a constant performance increase. This inevitable fact has to be weighed in together with two other contemporary trends.



(a) Power Consumption for Various Intel CPU Models **(b)** IC Development Costs for Various Processes [56] Spawning Several Generations

Fig. 1.1: Drivers for a Reduced Reliance on Scaling as a Pillar for Power Reduction in Modern ICs

The first is that the constantly expansive scope of Internet services leads to companies putting together massive data and storage centers, which in turn require unprecedented power levels to operate and cool. Given that these data centers are based on huge numbers of various components operating in parallel (e.g. CPUs for data centers or hard disk drives for storage ones), even a modest reduction in the power consumption of any one component (which is however used hundreds or even thousands of times in the center) results in a considerable power gain for the user.

This is indeed the case for high-end core network routers. Take for example the Cisco CRS-1 ([51]) chassis, boasting a maximum throughput of 1.28 Tbit/s (this depends on the linecards used of course, the maximum possible value is provided here), which in that case would consume a whopping 13.6kW! Taking into account that in a typical backbone system the linecards consume about 50% of the power and 45% thereof is dissipated in the various linecard ICs ([53]), meaning predominantly the NPs (but also additional ICs, like signal processors, A/D converters, etc.), that leaves us with 191W consumed just by the NPs on a single linecard. Given that there are thousand of core routers with multiple line cards each, which have to support ever increasing throughput, it is evident that there is every reason to believe that their impact on power consumption will only grow over time. Indeed, predictions [20] verify that the power dissipation and the carbon footprint of core networks will grow considerably over time. Power consumption has become a major hassle for everyone in the business, from vendors (who have a hard time selling equipment which is difficult to accommodate due to its power guzzling) to governments which have to deal with the increased environmental awareness of their constituencies.

Indeed, ecological factors have, at this juncture in time, asserted a relative preeminence and the corollaries of this diffuse into all areas of human activity. With climate change signs evident around the world, pressure is put on all stake holders to act on this. And this

in turn affects all aspects of science and engineering, whether it is governments shifting funds to environmentally beneficial projects or companies looking to adopt a greener profile by reducing their carbon footprint.

The message is thus clear: in the foreseeable future the world needs to see a tangible reduction in power consumption and to bring this about, circuit designers have no choice but to innovate on an architectural level. The networking space is no different, since increased link throughput drives the need for more performance from packet processing ICs. Higher performance typically means higher power consumption unless clever architectural innovations are employed. This thesis tries to address this by delivering an innovative architecture for next generation network processing at 100 Gbit/s, which provides significant reduction in power consumption while offering high throughput and sufficient programming flexibility for its target space.

1.2 Goals of the thesis

The goal of this thesis was the definition of a high performance, low power, modular carrier grade Ethernet Network Processor (NP) architecture and its implementation into an FPGA-based prototype in order to verify its functionality. More specifically, the aim was proving the feasibility of designing an NP, which could provide all the functionality required by contemporary and future Carrier Grade Ethernet (CGE) protocols, while at the same time reducing power consumption and remaining user friendly and easy to program. This would in turn significantly sink power dissipation in CGE routers, first by reducing the number of components per router (by requiring only one NP per 100 Gbit/s port) and at the same time making sure that this NP consumes significantly less power than network processors based on current architectural paradigms.

Reducing power consumption can be done in a multitude of not mutually exclusive ways in modern circuits as equation 1.1 shows. Lower operating voltage yields the highest benefits, this however is process dependent. Reducing the frequency is another, which was put to use in this design, but the key factor which we utilize in our approach is limiting the switching factor of the circuit through various techniques. The approach followed was two-fold, meaning an attempt was made to reduce both the number of bits that flow through the processing core and the frequency at which these bits toggle, thus reducing the switching activity in the network processor and bringing power consumption down.

In summary, the goal of this dissertation is to define a network processor architecture which exhibits the following characteristics:

- Allows for at least 100 Gbit/s of throughput for typical CGE protocols on an ASIC implementation.
- Significantly reduces power consumption in comparison to existing architectures.
- Provides for an extensible, flexible and modular design, which can be easily modified to tailor it to new demands.
- Incorporates programmable modules, which satisfy the required functionality, while enabling the adaptability of the NP to new protocols.

- Implements an interface to write programming information to the NP processing modules and its forwarding and QoS lookup tables.
- Provides clear and comprehensive means to generate the programming data for the processing modules.

Finally, the resulting design, which encompasses all of the above elements, must be implemented into a functional test bed using an FPGA device, in order to verify its functionality, measure its performance parameters and evaluate the results.

1.3 Structure of the thesis

The next chapter of this thesis recapitulates already available knowledge about the core areas of this work. This includes on the one hand fundamental information which is required in order to understand the challenges and limitations of the design decisions made further along in this thesis, as well as research in areas congruent with this work, upon which it is based and in which it provides improvements in specific areas. These include an introduction into Carrier Grade Ethernet and the protocols used therein, followed by a description of the typical architecture of a 100 Gbit/s CGE node. An extensive look into various architectures for packet processing, as well as into the different programming concepts used to get these architectures working is provided. Furthermore, an overview into hashing, which finds widespread use in network processor lookups and an overview of two interconnection technologies, the PCI Express link and the Interlaken are given.

Chapter 3 builds upon the NP architecture introduction done in chapter 2 and describes the NPU100 concept along with the proposed system architecture. At first, an overview of the NPU100's outer layer is given, including the Header-Payload split concept and the modules that enable the information exchange between the NPU100 and the neighboring line card components. Subsequently, the two main advantages of the NPU100 are presented, that is, its low power architecture and how this was achieved through an investigation of possible improvements points and the implementation and comparison of various solutions and the programming concept used to write programming information into the NP, which provides an easy to use, programmer-friendly tool chain to leverage the processing capabilities of the design.

Chapter 4 then describes the implementation of all the NPU100 modules. It starts by describing the implementation of the outer layer of the NP, and then the focus shifts to the pipeline core modules. The requirements for each of them are provided first and then an overview of the implementation is given. The level of detail depends on the importance of the module and the challenges encountered during its implementation. After fully covering the NP modules, the programming subsystem is described, starting with the hardware side, which includes the Xilinx PCI Express endpoint block and custom modules which receive the data from it and forward it to the needed NPU100 modules. The software is then explained in detail. Here the process of reading the user defined configuration files for the NPU100 is described together with the compiler software which converts them into the programming words that are then fed to the NPU100 over the PCI Express link.

Upon completion of the NPU100 description, chapter 5 provides a detailed look on the results of this thesis. Initially, detailed results from the implementation itself, focusing on resource and power consumption, performance and scalability followed by a description of the FPGA-based test bed realized to verify the functionality of the design.

Finally, chapter 6 concludes this work by providing a summary of the key points, findings and scientific contributions, as well as providing pointers to future improvements that could be done to further enhance it.

2 Fundamentals and Related Work

The work presented herein draws on a multitude of disjoint scientific areas and touches upon several sub fields of electrical engineering in varying degrees. This chapter attempts to extricate the relevant parts of each domain and present them in a homogeneous manner, thus laying the groundwork for the comprehension of the work that follows.

Any successful design is always the result of the harmonious confluence of the state of the art with a sound understanding of the requirements that it must satisfy. As a result of this it is necessary to correctly understand the functionality of Carrier Grade Ethernet networks and of the protocols that operate in them. Sections 2.1 and 2.2 sheds light on this.

Network protocols aside, an important element that plays a part in defining a network processor is the environment in which it must operate, meaning the node of which it is a part and with which it exchanges data in order to accomplish its task. This environment consists of various modules typically included in the NP IC itself (e.g. MAC interfaces), additional circuitry found on the linecard, on which the NP is typically located and finally the complete node itself. Thus, section 2.3 positions the NPU100 in the node and on the linecard and describes which other components are at play and how the NPU100 interacts with them.

Apart from the CGE protocols requirements and the node structure, understanding the current state of the art in packet processing architectures must be the starting point of an investigation to improve upon their performance. The design of network processors is inherently interdisciplinary, since it involves intimate understanding of systems architecture and design, tightly coupled with extensive knowledge of networking on various levels. This chapter attempts to provide an introduction to network processor architectures in section 2.4 by presenting an overview of the architecture of several commercial NPs in order to gain insight into the real world products that are available on the market. The overview does not limit itself strictly to architectures tried and tested in the NP domain, but also considers other possibilities, which have not succeeded in making inroads in this field. As such, a summary of various works on the performance evaluation of alternative concepts, such as graphics processor use for packet processing is given.

Another, often overlooked component in NP design is the programming. Typically only relegated to an afterthought, the methods used in it so far, their deficiencies and the potential for improvement, are analyzed in section 2.5.

Hashing, which finds application in NP lookups is a vast scientific area the surface of which is only scratched in section 2.6.

Last but not least, section 2.7 provides a brief overview of PCI Express (2.7.1) and Interlaken (2.7.2), two communication protocols used to connect the design described in this work with external components.

2.1 Carrier Grade Ethernet Networks

Telecom networks are typically subdivided into two categories, the core and the access network. The latter refers to the part to which the end users connect and which acts as the middleman between them and the core which then takes care of long-haul transport. The term backbone network is also used, usually to identify a core network in an enterprise setting. The core is essentially a mesh of switches, which enable the transmission of very high throughput aggregated traffic from one end to the other. The core itself is divided into two parts, the edge devices and the core ones, with the former being burdened with the classifying and aggregating packet flows as they arrive in the network, and the latter having to forward the aggregated flows through it. The work presented here only concerns the core devices. Core networks differentiate from edge networks by using a distinct set of protocols and technologies, such as SONET, DWDM and ATM, with Ethernet starting to slowly drift into the picture.

Ethernet is without a doubt the indispensable network technology, having triumphed over all contesters and prevailed in the local and metro networks. Notwithstanding the significance of this event, complete dominance is now in sight, as Carrier Grade Ethernet (CGE) standardization is moving along rapidly. CGE is an umbrella term under which a number of initiatives attempting to endow Ethernet with the carrier-grade features it lacks, are housed. As a result of their work, Ethernet has been slowly moving into position to infiltrate the core network domain for the past ten years or so. Key to this were the introduction of higher data rates and the additional of features which enabled the interoperability with carrier-grade transmission systems (e.g. SONET/SDH).

The question that begs itself then, is what does Ethernet lack in order to become a carrier-grade protocol. The short answer is the features that ensure the reliability and robustness required to deliver the QoS and security levels required by typical Service Level Agreements (SLAs) between providers. To elaborate upon this, the core deficiencies which are being addressed in current 100 Gbit/s CGE standardization are [31]:

- Wide area scalability. Ethernet provides for a very big (at least 2^{48}) addressing space, which does not allow for separation between customer and provider domains, leading to the overflowing of provider routers with customer addresses.
- Resilience and fast network failure recovery. The currently used mechanism requires the exchange of the spanning tree with a new one and thus falls short of the required protection speeds required by WANs.
- Advanced traffic engineering. Only rudimentary capabilities have been introduced with the passage of time (e.g. 802.1P [1]), however Ethernet still lacks the fine tuning of traffic that is an indispensable feature of core networks.
- Operation, Administration and Maintenance (OAM) capabilities, which allow the exploitation of traffic engineering to deliver services as specified by the SLAs.

The final unknown variable that remains is motivation. Why is CGE such an enticing concept? The answer is unoriginal and straightforward: reduced costs through simplicity.

Ethernet dominates the LANs, which means that almost all traffic is generated on Ethernet platforms and all content is delivered to Ethernet platforms. Hence any intermediate protocols that are used along the way are naturally seen as necessary complications that increase complexity. Currently these include the SONET dominated WANs, MPLS, to a lesser degree ATM and of course IP. Eliminating the just named protocols could lead to considerable savings, which the authors of [38] calculate to reach 40% in port-count and 20-80% in CAPital EXpenditure (CAPEX) in comparison to non-Ethernet alternatives.

2.2 Carrier Grade Ethernet Protocols

The next section provides a relatively short overview of the two protocols poised to be used in future CGEs, MPLS-TP [48] and PBB-TE [2]. Every attempt was made to keep the descriptions short and concise and as a result a lot of parts have been omitted for brevity's sake. The focus was kept on the areas that constitute the points of interest for a hardware implementation. The reader is encouraged to read the detailed protocol standards and accompanying, rich literature if a deeper look into the two protocols is called for.

This section describes the main functions of the protocols, which, being the dominant CGE protocols, dictate the requirements for the NP architecture. Any NPU must be able to provide layer 2.5 functionality, allowing for the forwarding of both MPLS packets and PBB frames. Layer 2.5 is an unofficial term, which is however widely used to categorize protocols that provide functionality that lies between traditional switching (Layer 2) and routing (Layer 3). MPLS-TP and PBB-TE will form the basis of the functionality that will be then expanded to encompass a wider functional palette, so as to support the processing of future protocols, which share the basic functional principles of the two afore mentioned ones.

2.2.1 Multiple Protocol Label Switching - Transport Profile (MPLS-TP)

MPLS (Multi Protocol Label Switching) is a protocol, which allows the transmission of packet- and circuit-based protocols, including IP, ATM and Ethernet, through a single encapsulation layer.

MPLS is in principle a further evolution of the concepts, which were already in use in the Frame Relay and ATM protocols, sidestepping functions which are not relevant today (e.g. cells of a fixed length) and reducing control overhead. However, useful ATM functionality like traffic shaping and out-of-band control signaling was preserved. MPLS-TP [18] is a currently under standardization transport profile for the MPLS protocol, which adds improvements and additions focused in the OAM area.

MPLS-TP considerably simplifies the forwarding mechanism of the original MPLS protocol by eliminating Penultimate Hop Popping (PHP) [48], LSP Merge [29] or Equal Cost MultiPath treatment (ECMP) [55]. These were stripped from the protocol in order to streamline is functionality and increase performance. It directly inherits the pseudo wire architecture [22], but most importantly, it introduces considerable enhancements to areas critical for core networks. These features are:

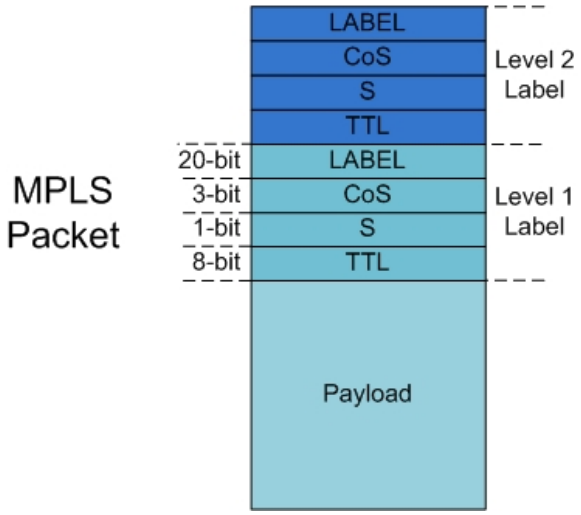


Fig. 2.1: MPLS Label Field Structure

- The major addition of OAM characteristics, necessary for performance monitoring and traffic management.
- A Generic Associated Channel (G-ACh) through which all management related messages are transmitted in-band with the data packets.
- Data plane fault protection schemes similar to those available in traditional transport networks.
- Network synchronization, a feature important for various carrier grade application such as mobile backhaul ([27]) and TDM circuit emulation.

At the core of MPLS processing lies the label stack. The label stack consists of 32 bit labels with the payload following after the end of the stack as shown on figure 2.1 for a two label stack. Each label is made up of four fields. The Label field (not to be confused with the label, which encompasses all four fields), consumes the top 20 bits of the label and uniquely identifies the Label Switched Path (LSP) path to which this packet flow belongs. The LSP is simply the MPLS term for the route the packet needs to follow inside the MPLS network. The 3 bit Class of Service (CoS) defines the QoS parameters for this packet, in MPLS jargon the Per Hop Behavior (PHB) from which the Forwarding Class for this packet is derived. The S bit demarcates the end of the label stack. The label, whose S bit has the value of one is the last one and after this label, the upper layer packet header or the payload follows. Finally the 8 bit Time To Live (TTL) field prevents loops from forming by setting an upper limit to the number of hops that this packet can go through.

Figure 2.2 illustrates the MPLS standard's guide to how MPLS processing should be performed [48], whence the prospective designer must derive the functionality of an MPLS node.

Forwarding an MPLS packet consists of two initially parallel strands which are interwoven at a later stage of the processing. The separation here is purely logical and is not meant to imply any level of compulsory parallelism in the implementation. The same is valid for the order in which the processing steps are performed.

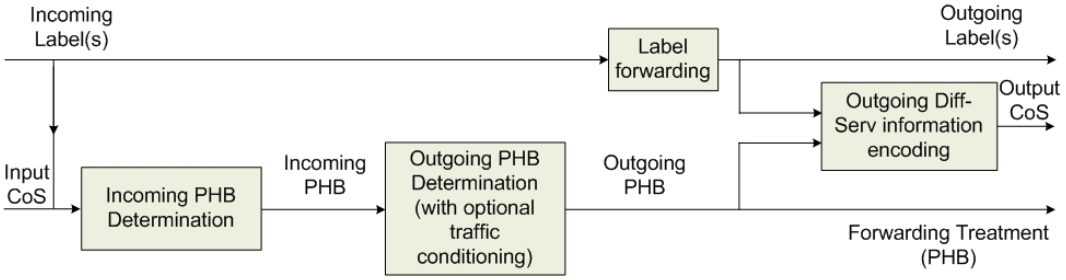


Fig. 2.2: Overview of the MPLS Forwarding Process

The top strand corresponds to the Label processing, which determines the operation that will be performed and the output ports for this packet. This realizes the "Label Forwarding"functionality as shown on the figure. The bottom strand involves the determination of the QoS parameters for this packet. These are divided into two sets, the Incoming PHB and the Outgoing PHB. The former refers to the parameters for the input queuing system and the latter to the parameters for the output queuing system. These are determined by using the CoS and the Label field. MPLS allows for different so called CoS to PHB mappings per LSP, which in practice is translated to different correspondence between CoS and PHB values for every label. This means that for label X, CoS value Y maps to a specific QoS policy, say Z, whereas for a label X', the same CoS value Y can (but does not have to) map to QoS policy Z'. Thus, the label and the CoS are used to determine the two PHBs values which essentially define the treatment the packet will receive from the queuing system. The optional nature of this step allows a packet to have matching input and output PHB values. Finally, the label and outgoing PHB fields are used in determining the CoS field value for the outgoing packet.

The theoretical forwarding model described thus far makes a poor job in explaining how packets actually flow in an MPLS network. MPLS-TP functionality is in reality straightforward when simple forwarding is done. When a packet is received in a node, the label is looked up in the routing table. This lookup might return multiple suitable entries, each of which indicates a different next hop and/or QoS parameters. The protocol does not specify which criteria are to be used to select one of these entries, thus leaving it up to each implementation. The label is swapped, the new QoS parameters are encoded in to the outgoing label and the packet is forwarded. QoS is required when MPLS is used in conjunction with DiffServ [17]. This represents a common scenario, where a packet enters the core network, is assigned to an LSP at the networks edge and then follows this path until exiting the core network and being stripped of the MPLS header.

This however describes only one side of the coin, since the prime MPLS-TP characteristic which has a significant impact on the processing architecture is that labels can be stacked on top of each other to create tunnels and thus any node may possibly need to process more than one label to forward a packet, something that is only determined by examining the previous label in the stack. Figure 2.3 illustrates a conventional MPLS-TP Label Switching Path, in which packets reach node 5 and are then redirected in to a tunnel through node 7 to their final destination node (node 10), instead of taking the path through node 8. Creating tunnels provides the network engineers with an easy method to redirect packets belonging to specific flow to alternative routes, which can be used to facilitate load balancing in case

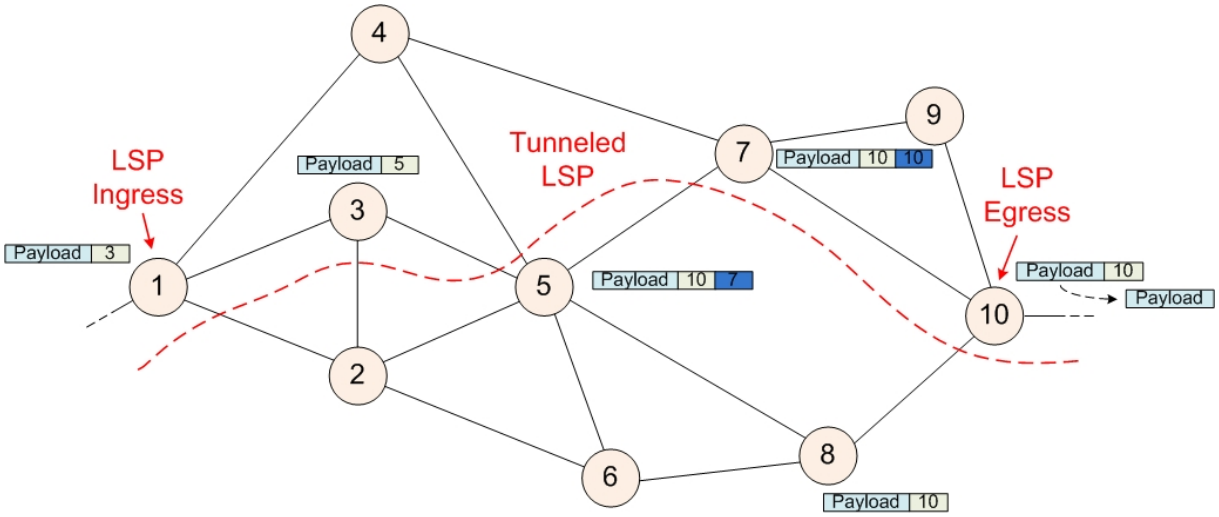


Fig. 2.3: Example of an MPLS-TP LSP with one Tunneling Layer

of congestion or fast redirection in case of a link failure. The tunnel can be set up without any change to the remainder of the LSP. In the example cited above, only the nodes 5,7 and 10 need to be updated when the tunnel is set up. Nodes 1, 3 and 8 remain oblivious to the change.

Notice that some nodes have to process two labels now. When the packet arrives at node 5, the label is looked up in the routing table. The result of the lookup determines that the current label must be swapped and that a subsequent operation needs to be performed. The nature of this future operation is not known at the time and can only be determined by performing an additional lookup using the new label. The subsequent lookup determines that a new label has to be pushed on to the stack. Similarly when the packet arrives at node 10, the label 10 is looked up. The result of the lookup is the removal of the label, and then the lower layer label 8 is looked up and removed as well (in this case both tunnels end in node 10. If this was not the case and only the one tunnel ended there, label 8 would have been swapped).

Hence in both cases, two operations are needed here, something that cannot be anticipated before processing the first label. This issue is further exacerbated by the fact, that theoretically any number of labels may be added to or removed from the stack. Figure 2.4 provides an example, which extends the previous one, by adding an additional tunneling layer at node seven. Thus now node 10 would have to pop all three labels (and process the payload -which in this context will most probably be the upper layer header- after that, to determine what to do next with the packet). The unpredictable processing coupled with an unlimited stack depth introduces a level of non-determinism, since it is impossible to anticipate the number of processing steps that each packet requires.

Of equal significance in the confines of this work is the processing done on each label by each node. MPLS supports three operations: Swap - the exchange of the current top label in the stack for a new one, Push - the addition of a label at the top of the stack and Pop - the removal of the top label. Each of these operations corresponds to an action that produces some meaningful sense within the given network architecture, hence:

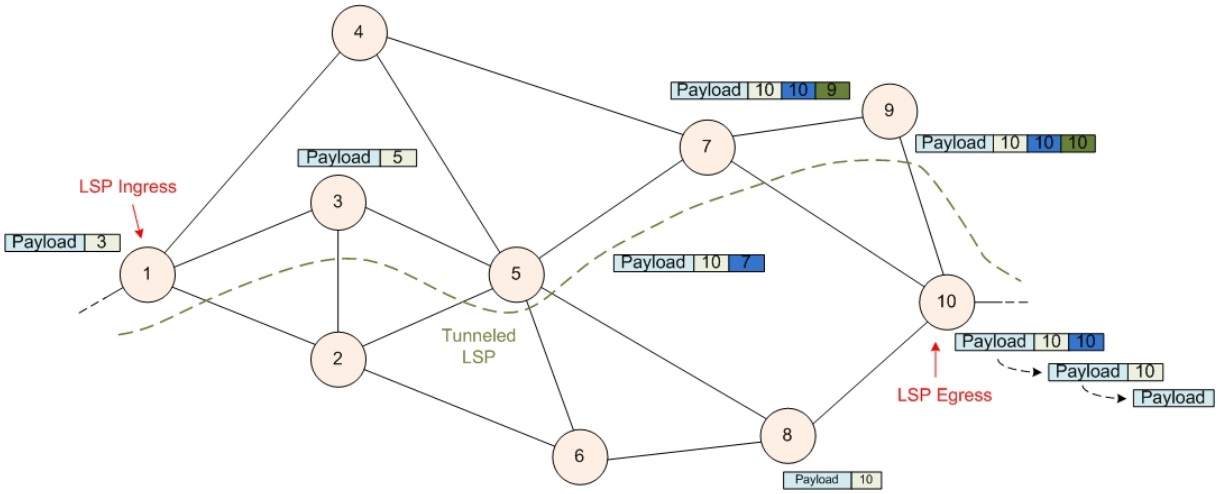


Fig. 2.4: Example of an MPLS-TP LSP with two Tunneling Layers

- swap denotes a simple forwarding action. The old label is replaced by a new one.
- push marks the beginning of an LSP. This could be either the LSP, which the node enters when arriving at the edge of the MPLS network, or could simply be an MPLS tunnel. In the former case the label pushed is the first and only MPLS label on the stack, while in the latter there are more labels beneath it.
- pop is the opposite operation from the push and designates the end of a tunnel or the exit from the MPLS network as a whole.

Although these three operations may be arbitrarily combined according to the MPLS standards, it makes little sense to do so in a practical environment. For example pushing a label and then removing it immediately produces no effect and thus only wastes processing resources. Respectively a swap must typically come before a push. The example on figure 2.3 illustrates this. Node five is the tunnel entrance and in that node the second label was swapped from 5 to 10 so that it points to the end of the tunnel and then label with the value 7 -the next hop- was pushed on top of it. Had this not happened the packet would have arrived at node 10, the top label would have been popped and the node would have encounter a label which was either unknown to him or even worse, corresponded to some other node. This is possible since labels in MPLS have node-local significance, which means that a label with a value X will be part of a different LSP in each node. This is in stark contrast to protocols like PBB-TE, where the addresses have global significance and thus a route is identified by a single address in all the nodes it traverses. Table 2.1 provides a listing of sequences of MPLS operations that make sense. Up to four operations are considered, the limit for the implementation performed in this work as will be explained in the subsequent chapters.

In all but the most uncommon MPLS scenarios one swap and push, pop and swap or a simple swap operation will be the typical tasks of an NP processing MPLS packet, however it would would sanguinely naive to assume that the most obscure cases will never occur. Thus any aspiring MPLS NP must be able to cope with all these scenarios.

Operations	Explanation
Swap	Simple packet forwarding in an MPLS LSP
Swap and Push	Tunnel entrance in an MPLS LSP
Swap and Push x 2	Entrance into two tunnels at the same node
Swap and Push x 3	Entrance intro three tunnels at the same node
Pop and Swap	Exit from a tunnel in an MPLS LSP
Pop x2 and Swap	Same as above but for two tunnels
Pop x3 and Swap	Same as above but for three tunnels
Pop, Swap and Push	Exit from a tunnel and at the same time entrance in a new one at the same node
Pop x2, Swap and Push	Exit from two tunnels and at the same time entrance in a new one at the same node
Pop, Swap and Push x 2	Exit from a tunnel and at the same time entrance into two new ones at the same node

Tab. 2.1: Meaningful MPLS-TP Operation Sequences

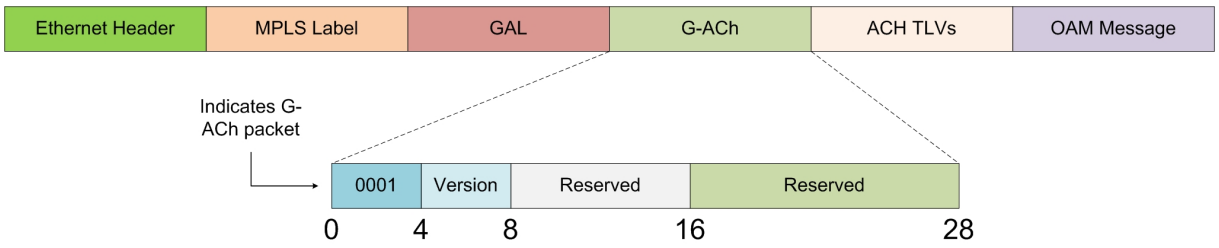


Fig. 2.5: Overview of the MPLS Generic ACH Message

Efficiently communicating management information between the data and the management planes is a cornerstone requirement of carrier grade networks. MPLS-TP implements an in-band communications channel for all sorts of OAM, Automatic Protection Switching (APS) and Fault, Configuration, Accounting, Performance and Security (FCAPS) messages. This is done through the Generic Associate Channel (G-ACh) and the Generic Associated Channel Header Label (GAL), which are described in detail in [19]. The structure of such a message is shown in figure 2.5. The initial part is a typical CGE MPLS packet, with the Ethernet header and underneath it one MPLS label. The destination node for this OAM packet strips the header to reveal the GAL, which indicates that this is a packet for the management plane. This wraps up the data plane's side of the processing. The rest of the message consists of the G-ACh, ACH Type Length Values (TLVs) and the OAM message itself. The G-ACh essentially identifies the protocol that is encapsulated in the OAM channel, the TLVs provide the means to recognize extended the information provided by the G-ACh, while the OAM message provides the payload for the message. Any NP design supporting MPLS-TP must be able to identify the GAL and sent this packet automatically to the port corresponding to the management plane. This complicates processing because it demands that the prospective NP supports one additional label stack layer in order to be able to process the GAL label.

To summarize, the key aspects of MPLS-TP for core network forwarding are:

- potentially infinite number of labels to process, with no possibility to snoop ahead in order to determine this number beforehand.
- multiple lookup entries per label.
- different CoS-to-PHB mappings per label.
- possibly different input and output QoS treatment.
- OAM message passing through the G-ACh scheme.

These requirements formed the basis of the NPU100 design, along with those of the PBB-TE protocol described in the next section.

2.2.2 Provider Backbone Bridging - Traffic Engineering (PBB-TE)

PBB-TE is a further development of the Ethernet Protocol, a full history of which can be found in [30]. Ethernet started out as a shared medium protocol with limited LAN applications and has now come full circle to almost monopolize practically every network type apart from the core networks, in which it has been slowly but steadily making inroads in the past years, something that the standardization of CGE Ethernet protocols and the coming into view of 100 Gbit/s link speeds should only help precipitate.

Figure 2.6 attempts to tell Ethernet's tale through the evolution of the Ethernet frame over the past 40 years. The simplicity of the initial Ethernet frame header shown on the far left attests to the frugal beginnings of the protocol itself. Only the source and destination address (SA and DA respectively, each 48 bits) and the Ethertype fields, which designates the protocol encapsulated in the Ethernet header, are present. With each successive version of the protocol additional features were added to cope with the demand for more services and more complex capabilities.

The first major update came in the mid-1990s with the introduction of VLANs, characterized by the customer VID (or C-VID) field and officially known as IEEE 820.1Q [9]. This field is 12 bits long and thus allows up to 4096 VLANs. VLANs have several uses, some of them being to multiplex services by allowing differentiation of channels over the same physical path or the creation of interest groups by assigning output ports to a specific VLAN. The other new addition is the Q-Tag, which addressed the need to specify different QoS parameters for each packet.

The next evolution were the Provider Bridges or IEEE 802.1AD, which stemmed from the need to separate customer and provider address space. This was due to the explosion in Ethernet deployment, caused by the development of optic fiber interfaces for Ethernet switches. The solution given was to provide one more VLAN and Q-TAG pair, dedicated to the service provider (the S-VID and S-TAG on the figure) and relegate the original pair, now known as C-VID and C-TAG to customer use. Though this allowed for the decoupling of services and more importantly of their administration, the problem that loomed on the horizon was that the S-VID field only allowed for 4096 provider services, which promptly ran out.

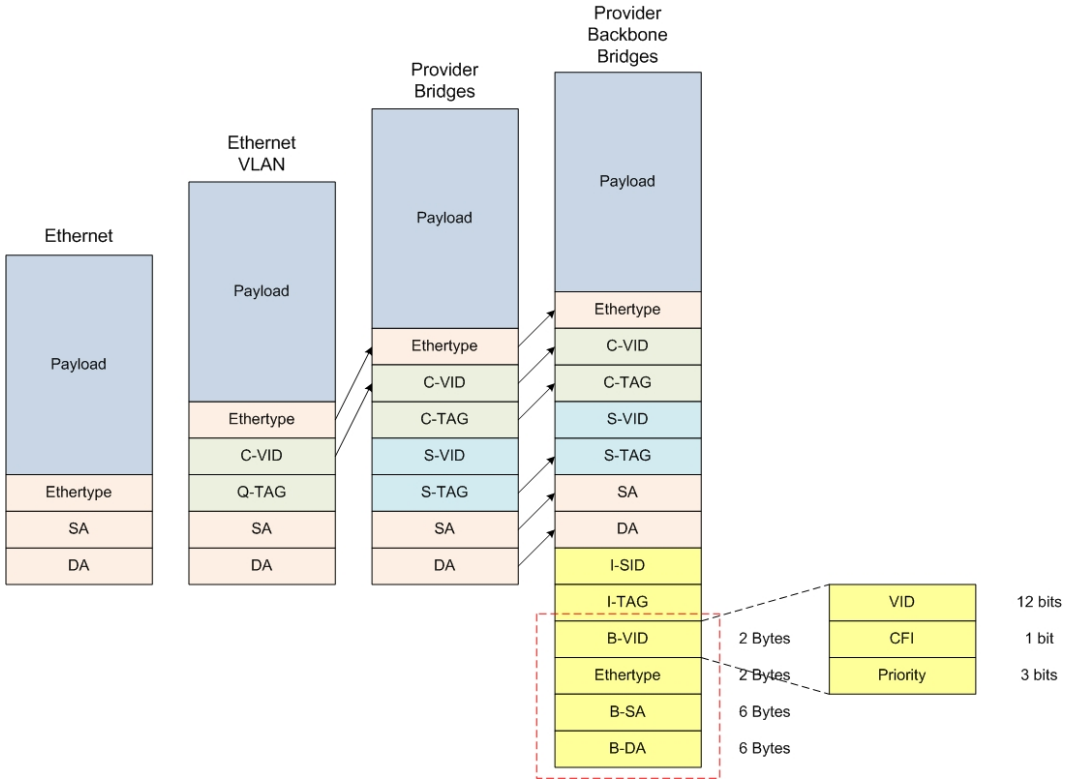


Fig. 2.6: Overview of the PBB-TE Frame Development over the Years

In an effort to provide a decisive solution to this problem Provider Backbone Bridging (IEEE 802.1AH) was promulgated. It completely encapsulates the customer header in a provider header an thus mitigates the above issue. The provider header added is an 802.1Q header with a service identifier field (I-SID) added. This decouples the services from the topology, which the B-VID now exclusively signifies. This allows for efficient demarcation of the consumer and provider networks. PBB was designed to avoid overflowing core network nodes with a multiple of edge node addresses (and thus demanding huge lookup tables) by encapsulating the edge network Ethernet frame in a core network header. Thus, only the addresses assigned to core routers need to be known throughout the core network.

PBB-TE is a considerable simplification in comparison to the PBB protocol from which it was derived. Its goal is the complete separation of the control and data planes, something that is achievable through the elimination of the flooding, learning and active topology functionalities. The absence of the learning functionality means that all lookup tables are programmed through the management plane. The frame header and forwarding mechanism however are identical to the PBB protocol. PBB packet forwarding requires only a subset of the functionality of MPLS, since there is no header stacking and no label swapping. However, the length of the combined B-DA and B-VID fields allows for a huge pool of addresses something that translates into large routing table size and lookup complexity.

Figure 2.7 illustrates the processing steps for a PBB-TE frame. The area crossed out with the red dashed line refers to queuing and metering functionality, which is outside the scope of this thesis. PBB frame processing consists thus of a sequence of four steps.

1. Active Topology Enforcement refers to the assignment of a frame to a Spanning

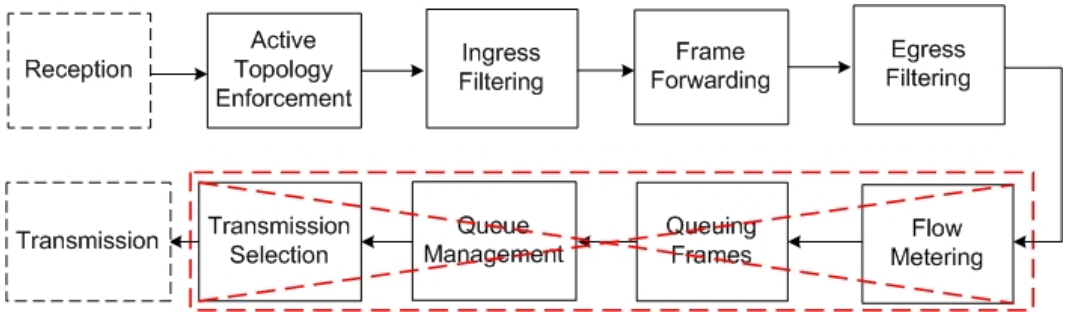


Fig. 2.7: Processing Steps Required by a PBB-TE Frame

Tree. A Spanning Tree tracks paths to other reachable Ethernet bridges and ensures a loop free topology. Since there is a special Spanning Tree reserved specifically for PBB-TE, this step is trivial when only this protocol is being processed (the standard here presupposes that PBB-TE coexists with normal PBB functionality in a switch, hence the requirements for this step).

2. Ingress Filtering rejects frames whose input port is not a member of the frame's VID. A lookup is required to read this information. The MAC address and VID is used as input for this lookup. It must be possible to disable this functionality through a configuration parameter.

3. Frame Forwarding demands another lookup, this time in search of the appropriate Filtering Entry. In this lookup the destination MAC Address and the VID fields are used to identify the entry, which contains a port map, identifying the possible output ports for the frame. Additionally Group Entries can exist, which correspond to group MAC addresses. There is no interdependence between steps 2 and 3 so they could be executed in any order (with the reservation that if step 2 rejects the frame then frame 3 is pointless to execute).

4. The Egress filtering filters all the output ports, which where included in the Filtering Entry but are not in the member set of this VID.

Again similarly to the MPLS forwarding concept, the description here is purely logical and it is expected that the transition to an actual hardware implementation will mean the reshuffling of the order of the operations (as far as this does not impact the forwarding results of course) in order to make processing more efficient.

2.3 Carrier Grade Ethernet Node Architecture

NPs are, like all processor types not stand alone components, but part of a system with which they interact. This section describes the environment in which the NPU100 is to reside. This is necessary to gain the appropriate insight into some fundamental requirements of the NPU100 functionality (e.g. how the node architecture dictates the need for an internal header in the NPU100). Section 2.1 provided an overview of a CG network, and thus here we will delve deeper into what makes up a typical 100 Gbit/s CGE node, how

its components communicate with each other and how all of this influences the NPU100. As the NPU100 was designed as part of the 100GET-E3 NPU100 project, decisions taken within the project permeate many aspects of the design, which are elucidated in further on in this section, as well as in the subsequent chapters.

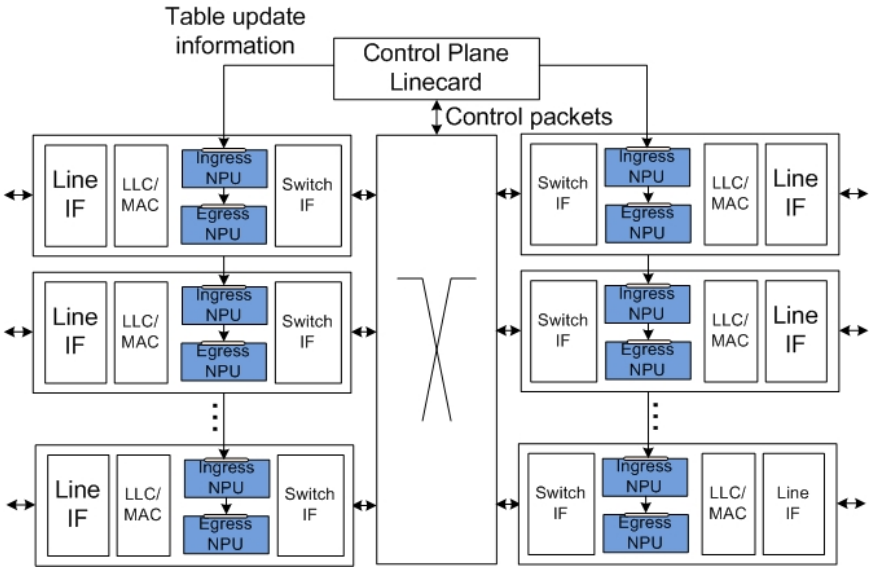


Fig. 2.8: Overview of an Example 100 Gbit/s CGE Node Architecture

The architecture of a 100 Gbit/s CGE node is illustrated in figure 2.8. It consists of several line cards interconnected with each other with a crossbar as found in most modern switches. Each line card communicates with the other line cards over this crossbar. Furthermore, each line card includes only one bi-directional 100 Gbit/s physical interface. Current line cards usually house more than one physical interface, however one of the development goals for 100 Gbit/s Ethernet is to reduce the component count needed on each line card by providing for a faster interface, and thus reducing the amount of ports is in line with that goal. Several NPs reside on this line card, depending on how much traffic each NP can service. Again following the same line of thought, the goal is to reduce this number as much as possible. Thus, each physical interface is in turn served by two NPU100 NPs, each of which handles one direction. Although as will be shown in section 5.1.4 the design is scalable enough to allow for one NPU100 to be used, dimensioning the NPU100 and implementation of the prototype was done with one NP per direction as a guideline and hence this is taken as the basis for its description. Apart from the NPs, each line card has a MAC interface with the respective PHY and LLC circuitry. These are not part of the NPU100 design but are assumed to be able to receive valid frames from the network and transmit them to the NPU100 and are shown on the figure as separate components on the linecard, although in commercial implementations they are typically integrated into the IC together with the NP.

The total number of ports in the node plays an important role in properly dimensioning the NPU100, as it influences the number of output ports that must be addressable and reachable by each NPU via the crossbar. In core networks typical values are a few tens of output ports, since the number of neighboring nodes is relatively small. This is in contrast

to access routers where values up to several hundred ports are not uncommon. In case scaling issues regarding the number of supported ports arise in the future, a hierarchical scheme, which encodes output ports in a group ID, that would be looked up in the switching interface to determine the required output ports, could be utilized.

Each line card is also equipped with a switch interface, which handles communication with the crossbar. Queuing is done in the switch interface for both ingress and egress paths, which means that consequently flow control will be implemented in the switching interface and is not part of the NPU100, which must however provide the appropriate parameters to it, in order for queuing to be performed successfully. The queuing itself is done in two phases, an ingress queuing performed at the input line card's switching interface and egress queuing done at the switching interface of the respective egress line card. These phases correspond to the enforcement of the incoming and outgoing PHB functionality found in MPLS. Thus the NPU100 must not provide any frame buffering functionality apart from any temporary storage, while the frames are being processed.

The control plane is handled by a separate processor (not an NPU100), which is attached to the crossbar and can send and receive control packets to and from the NPs over each line card's switch interface. The control plane line card should have its own dedicated port number for this purpose. Apart from control packets, the control plane processor frequently needs to update various tables in the NPs (e.g. the tables containing forwarding and QoS information). This data can, depending on the occasion, be quite voluminous and thus sending it over the crossbar might disrupt the regular flow of traffic. Thus a distinct interface is used to sent information, which in the case of the NPU100 was selected to be a PCI express interface, which offers abundant bandwidth and has found fairly widespread use in current switches. Sections 3.3 and 4.3.3 provide more information on how the PCI Express link is used to sent configuration information to the NPU100.

2.4 Processing Architectures Overview

Network processor architectures have traditionally followed two main paradigms: the parallel and pipelined [46], as the the various devices presented in the previous section attest to. The former typically includes many RISC like processing elements, which process packets in parallel. A scheduler assigns the packets to a PE, recollects them when processing is done and dispatches them to the output. This model is known as a run-to-completion model [**?**] since a single PE is in charge of completing the processing of an entire packet. Alternative models, like the pool-of-threads, are available to allow for more flexible task allocation to the PEs. The PEs share the routing table memory, any co-processors available in the system as well as the packet buffer memory and the interconnect which binds all the system elements together.

The alternative is a pipelined architecture, which divides the processing into multiple steps, each assigned to a PE. The PEs are not necessarily identical. When each PE is done, the packet is sent to the next one, until the processing is completed. Here the PEs are typically fine tuned to handle their specific tasks (though implementations with more generic PEs do exist), which allows for more efficiency. Furthermore, the memory is no longer shared between the PEs which eliminates unpredictable delays due to resource

contention. Both architectures may include special purpose accelerators depending on the application they target.

The first part of this section focuses on commercially available network processors and after providing a short historical overview goes on to discuss the various architectures on offer, which all neatly fall into the two categories described above. Subsequently, other possibly applicable data processing architectures are considered in order to determine their relevance and possible suitability for packet processing.

2.4.1 Network Processors

When the first network processors started coming onto the market at the end of the 1990s, exuberant predictions were foreseeing their unprecedented expansion into a multi-billion dollar market within a span of a few years. Expectations fell short however, since the bursting of the dot com bubble in the US saw the industry enter an extended slump, which respectively curbed demand for new network services. The initial hyper-optimism however was enough get every major player (and some new ones) to join in a stampede in order to grab a piece of what was predicated to be an increasingly bigger pie.

This led to an impressive amount of designs coming into a market quickly becoming saturated with products on the one side and stagnating of demand on the other. No good could come out of this and thus, the market landscape today looks wildly different. Pretty much every big company has withdrawn from the market and sold its NP division or spun it off to some new venture and many of the smaller start-ups went belly up. And this despite strong, double-digit growth every year in the past decade.

Extracting information about the architectures of commercial NPs has and still is a challenging task. Since this data is judged as sensitive, it is kept under lock and key, and thus the little trickle of information that is publicly released is insufficient to lift the fog of war that covers the design details. This lack of data proves to be a serious impediment is asserting the efficiency of a design, which has important ramifications on this work, since it makes comparing the design propounded herein in with commercial alternative impossible.

Since this work focuses on Carrier Grade Networks, this section also limits its scope to processing architectures designed to cater to these demands. Thus many vendors of network processing designed for various networking applications ranging from security to compression (e.g. Cavium) are not considered here for brevity's sake.

Shortly before the submission of this thesis several vendors announced next-generation NPs, that will come into the market in the coming years (most likely sometime in 2014). These are recited here for completeness, although as with their already on the market counterparts, the available information is sketchy at best.

Intel IXP family and Netronome

Intel's IXP family is a most curious beast, an adjective it earns mostly not due to any architectural traits but due to the fact that although Intel has sold off its NPU business to Netronome since 2006 its NPs remained quite popular for a significant time after that.

That being said, the attention will be turned to Netronome's latest offering the NFP-32xx [13], while providing some information on the last Intel NPU the IXP-28xx [25], its

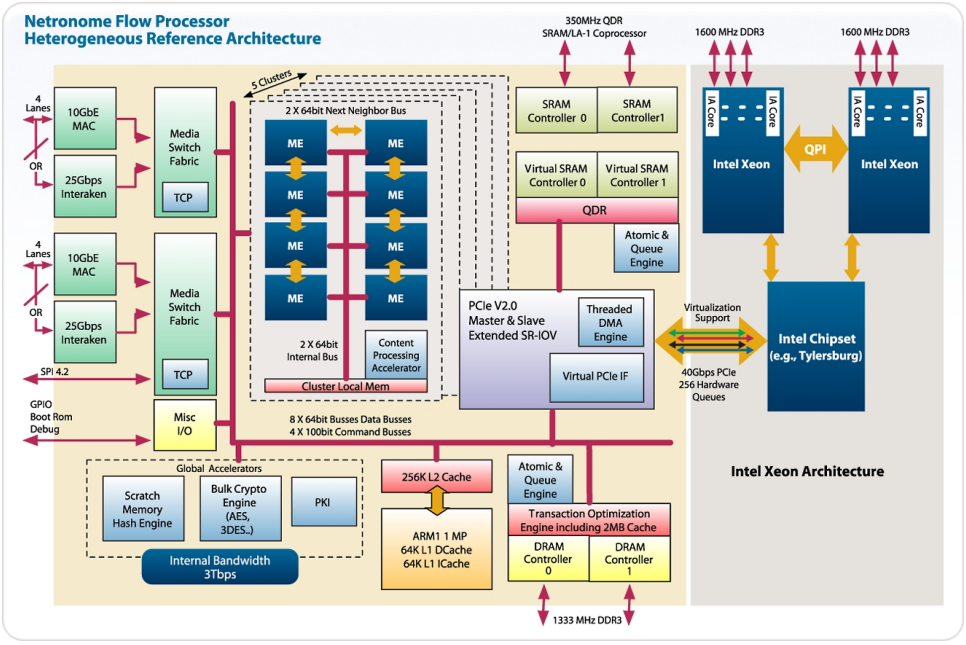


Fig. 2.9: Netronome NFP-32xx Overview [13]

evolutionary precursor. The two share a lot of architectural features, with the Netronome offering updated technologies on almost every front. Hence, the XScale CPU cores have now been replaced by more modern ARM ones and the processing engines have had both their numbers (from 16 to 40) and their properties bumped up (mostly translating to more instruction and data memory). Memory and media interfaces have been updated with DDR3 SDRAM and QDRII SRAM now being used on the memory front and XAUI [8], SGMII [52] and Interlaken [10] serving as media interfaces. A PCI Express [4] interface connects the Netronome with other system elements.

Figure 2.9 illustrates the architecture of the Netronome NFP-32xx NPU. It consists of 5 clusters of 8 micro-engines. Each cluster has its own bus, over which it gains access to a common memory. All the clusters share eight 64 bit data buses and four 100 bit command buses, over which all communication with the remaining modules takes place. These include a pool of various accelerators connected onto the bus. As such the architecture remains vulnerable to typical bus congestion/contention issues, encountered in similar architectures. The NFP-32xx uses either a run-to-completion or a thread-pool programming model, thus it is quite flexible in micro-engine utilization issues.

In the fall of 2012 Netronome announced its new architecture on which the NFP-6xxx NPUs [44] are to be based. In contrast to the NFP-32xx one, the new design contains one cluster of 120 flow processors and one of 96 packet processors. The former are designed for layer 2 to layer 7 tasks, whereas the latter are aimed at layer 2 and 3 processing. It also contains 100 dedicated accelerators which offload crypto and hash functionality (among others). Another deviation from the NFP-32xxx design is an additional level of memory hierarchy called the proximity memory. The NFP-6xxx is aimed at 200 Gbit/s networks and can cater to a variety of applications ranging from Carrier Ethernet routing and switching to intrusion prevention.

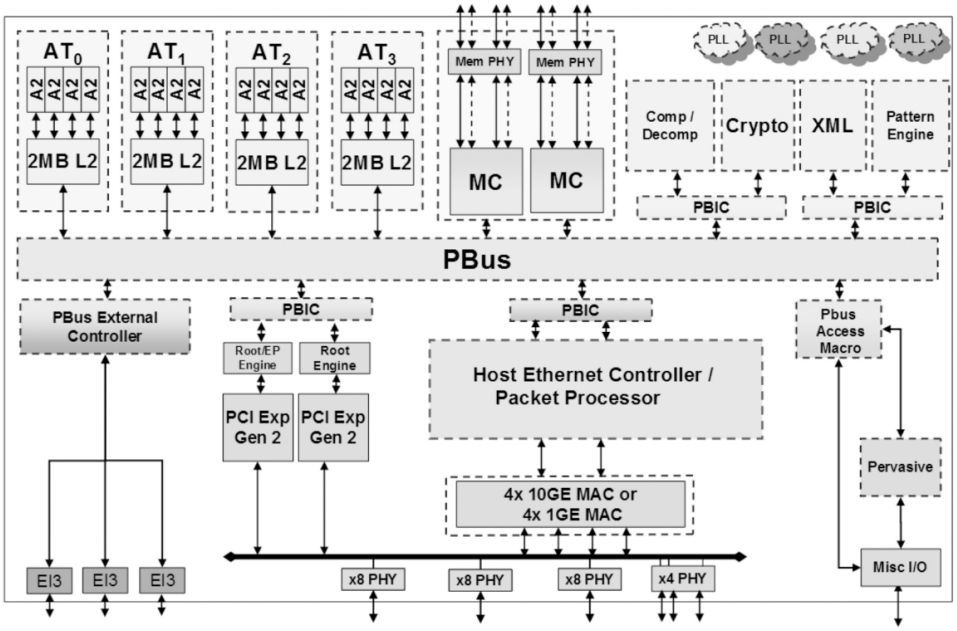


Fig. 2.10: Block Diagram of the New Generation IBM NPU [37]

IBM Network Processors

IBM has been out of the network processor business for a while now. Its last foray in the NP world was with the PowerNP architecture [14], a powerful NP, which followed a typical run-to-completion processing model. A recent paper [37] though shows the company is determined to get back in the game with what it calls a wire-speed processor.

The new chip is shown on figure 2.10 and consists of four throughput nodes, each with four cores. In an architecture not dissimilar to the one described in the previous section, as each throughput node has a memory shared between the cores. Accelerators, memories and external interfaces are accessible over a common bus. According to IBM, this approach can easily scale to 64 cores and 100 Gbit/s and achieves considerable power savings by using lightweight, simple cores, which are highly threaded but do not fare as well in complex, single-threaded tasks.

Freescale Semiconductor

Freescale is an offspring of Motorola, which created the company in 2004. It provides multiple solutions for the automotive and communications market. It offers a variety of NPs, which it dubs communication processors, covering all segments from the access to the core. Of these the higher end models ([49]) present the greatest relevance to this work.

All processors of this series consist of several e500 cores interconnected to the typical NP peripheral set (memory interfaces, external interfaces, specialized modules, such as lookup engines) through a common bus. The important difference with other solutions presented in this chapter, is that here we have a small number of cores, which however offer higher single threaded performance then the RISC cores found in other designs. This makes the Freescale line more suitable for applications, which require more complex processing of a smaller number of packets, which is however the opposite of what one encounters in a core

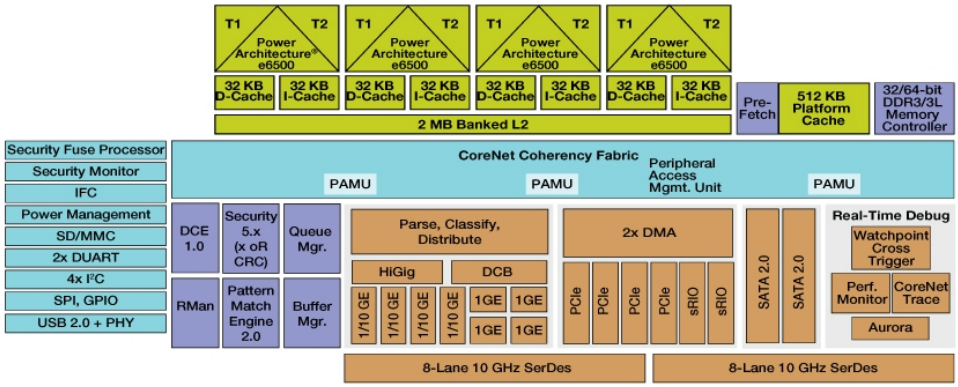


Fig. 2.11: Block Diagram of the Freescale QorIQ AMP Series T2080 Communications Processor

network.

Shortly before the submission of this work, Freescale refreshed its communication processor line with the QorIQ line [32]. The new design is a continuation of the previous philosophy and thus still uses processing cores based on the power ISA from IBM. Four e6500 processors are available, although this time, to offset poor performance at multi-threaded applications because of the complex cores used, Freescale incorporates functionality similar to Intel's Hyperthreading, which allows two threads to share the resources of one physical processor. Release details on the QorIQ series are unclear, though early 2014 sounds a reasonable target launch date.

Cisco QuantumFlow Architecture

Cisco is the networking industry's seemingly perennial juggernaut, dominating most of its aspects one way or the other and as such could not be absent from the NP market, which it entered relatively late in 2004. The fruit of Cisco's extended efforts were the Cisco Quantum Flow architecture [24], on which then several products, such as the ASR1000 Processor shown on figure 2.12, are based. Cisco's peculiarity lies in the fact that in does not sell its NPUs to other vendors but uses them exclusively in its own products. Nevertheless, this detracts nothing from the interest that they pose from an architectural perspective.

There are three principal features, which characterize a QuantumFlow processors. The first is a pool of 40 packet processing cores, which reside on chip and perform all the processing. The second is a flexible queuing system allowing for 5+ levels of queuing hierarchy and more than 100.000 queues. Finally the third component is a software architecture based on C, which allegedly takes most of the burden away from the programmer.

From an architectural perspective, which is of interest for this work, the Cisco NPUs are parallel processing, non-pipelined designs, which use a common shared memory. The Processing Elements (PEs) process packets, which they then forward to the traffic manager, which queues the packets accordingly before sending them to the output. The PEs are 32-bit RISC cores, each of which can handle up to four threads in parallel. Thread assignment is done automatically at packet reception and the PE to which the packet is assigned

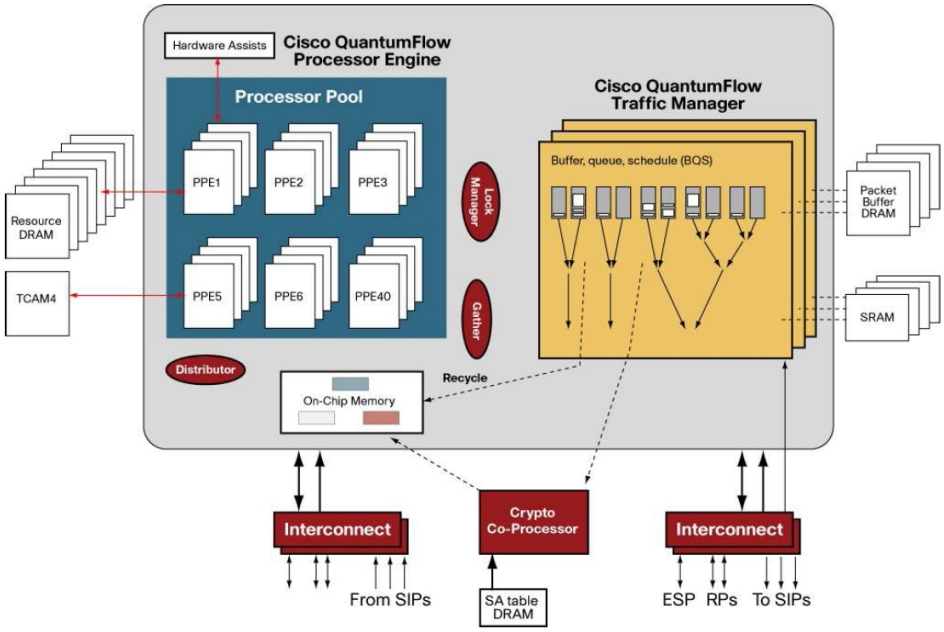


Fig. 2.12: Architectural Overview of the Cisco ASR 1000 Embedded Services Processor [24]

is responsible for it until the end of its lifetime in the NPU. Hence the QuantumFlow is a typical run to completion model, and as such it is susceptible to bottlenecks and unpredictable behaviors plaguing them. Cisco attempts to mitigate this by allowing a second thread to be spawned and assigned to a different PE for complex packets.

All Cisco NPUs have high bandwidth interfaces for connecting with external interfaces and use power hungry TCAMs and SRAMs for state information and packet buffering.

LSI

LSI is a California-based company that produces various ASICs for storage systems and network applications. It was founded in 1981 and entered the network processor market in 2007 when it merged with NP vendor Agere Systems and acquired its product line as a result.

LSI offers a wide range of products, catering to all aspects of network processing. As is the case with most NP vendors there is a basic architecture, which is then tweaked to suit the individual demands. Since this work occupies itself with carrier networks, the respective LSI model, the APP3300 [26] is the most relevant one to this work.

The chip, shown on figure 2.13, is essentially broken down into two parts, the Fast path Packet Processing Engine, which handles the data, and the Services and Control Processor, which realizes the control plane. The former, which is the most interesting for this work is unfortunately very vaguely described, with the only concrete information available being that its modules like the classifier are multi threaded and support up to 128 threads.

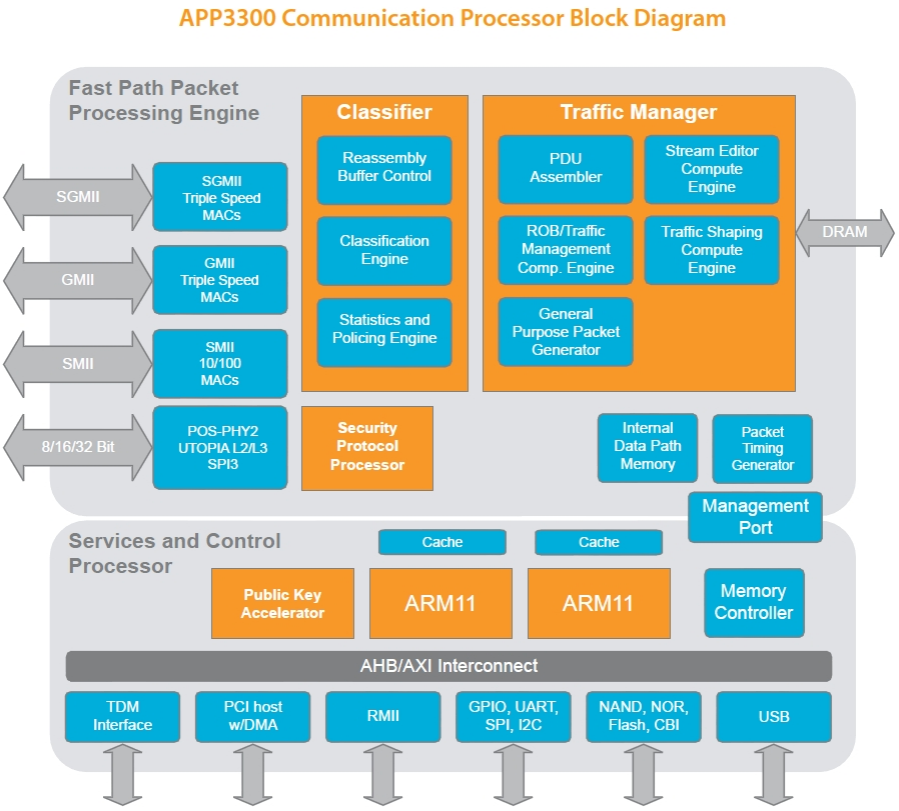


Fig. 2.13: Block Diagram of the LSI APP3300 NP Family [26]

EZChip

EZ Chip has been around since the dawn of NPs in 1999. Although it is a fabless company without backing from a major partner, it has managed in the elapsed decade to make a reputable name for itself in the market as one of the most consistent vendors. It's line-up consists of access and core networks NPUs, which share the same basic design philosophy, with the access ones being scaled down appropriately as needed. As this work targets core networks only, the core NPU line-up is described here as more pertinent.

EZ Chip's NPUs have always been based on the notion of a Task Optimized Processor (TOP). These TOPs have been refined with each successive generation so as to deliver ever higher performance. They are divided into four categories, parse, search, resolve and modify and perform the bulk of the packet processing. There are multiple instances of each TOP, operating in parallel in a super-scalar fashion. Apart from the TOPs, lookup engines are used to perform either switching/routing or Deep Packet Inspection (DPI) lookups. A run to completion model is used to guarantee that even the most complex scenarios can be processed, although the organization of the processing is sufficiently flexible so as to allow pipelines to be created out of the TOPs with each TOP handling part of the processing.

The latest incarnation of this technology, the NP4 uses the TOPs as the basic building block for a 100 Gbit/s (or 50 Gbit/s in full duplex mode) NPU, capable of processing L2 switching traffic, like PBB and PBB-TE (described in section 2.2.2), MPLS (section 2.2.1) or typical IPv4/IPv6 routing. The NP4 uses a multitude of memory interfaces, like DDR

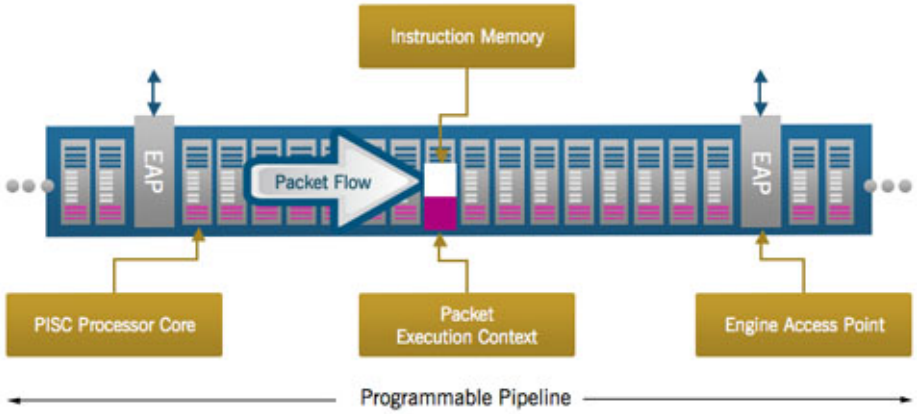


Fig. 2.14: Overview of the DataFlow Architecture used in the Xelerated NPs [62]

SDRAM, RLDRAM and a TCAM and connects to the external world via a multitude of interfaces, e.g. 10 XAUI [8], 3 Interlaken [10] and 1 PCI Express [4] link.

EZ Chip's 200 Gbit/s solution is called NP-5 [23]. However despite having being announced in 2011 and expecting to sample at the end of 2012, details about the NP-5 are virtually non-existent, although the fact that EZ Chip is touting the NP to be a natural evolution of its predecessor, offering 100% backwards compatibility is indicative that an architectural continuity should be expected.

Xelerated and Marvell

Xelerated was another player who has been in the NP market since almost the beginning. Starting out in 2000 in Sweden, it has focused mostly on core networks, but also caters to the metro and access ones. Xelerated was acquired in January of 2012 by the Marvell Technology Group, a leading fabless semiconductor company.

Xelerated processors are based on the so called data flow architecture [62], which is illustrated in figure 2.14. It aims to optimize the flow of packets through the NP and not the flow of instructions. The result is a pipeline of processing elements, the so called Packet Instruction Set Computers (PISCs). An Xelerated NP consists of a multitude of these PEs (e.g. the HX330 NP uses 512), each of which executes one operation in one 64-byte word per clock cycle. The main advantage of this architecture is its predictability, since the time it takes a packet to complete processing is known before hand, which is in stark contrast with run to completion model NPUs, where there is no inherent upper bound to the latency of a packet. As pipelined designs, Xelerated processors exhibit the greatest kinship with the design proposed in this work, which is also based on a pipeline concept.

Apart from the PISCs, the latest Xelerated NPUs utilize internal TCAMs for their lookups and DDR SDRAM memories for packet buffer and traffic management. External interfaces include XAUI, SGMII and Interlaken for the MAC and switch interfaces and PCI Express for communicating with the control plane. The top models are rated for a throughput of up to 100 Gbit/s.

2.4.2 Other Processing Concepts

The various NPs discussed thus far exhibit a significant architecture pluralism. They do not however exhaust the available concepts, which could be used for packet processing and thus should be taken into consideration in this work.

One concept, which has been gaining traction in the academic community and has even seen a product released not too long ago [54] is the use of Graphics Processing Units (GPUs) for packet processing in the data path. GPUs are nowadays essentially arrays of SIMD processing engines using shared memories of various hierarchy levels. Hence this trend should be viewed in the broader scope of attempting to use SIMD architectures for packet processing.

Like NPUs, GPUs are commercial products and thus their architectural details are typically not disclosed. However since traditionally programming a GPU required intimate knowledge of it's architecture and GPUs are much more widely available than NPUs (almost everyone has one at home and can start programming applications for it, whereas almost nobody has an NPU available), information has been that much more forthcoming.

The GPU market is dominated by two players, nVIDIA and AMD (who bought off ATI's business several years ago). Their long standing feud has resulted in very short product cycles and monstrous chips (with transistor counts reaching the 3 billion!), which however pack some impressive processing power. It is not the goal of this thesis to get into the architectural details of each chip (which changes with each generation). More details can be found in [45] for nVidia and in [15] for AMD designs. Instead a short overview of the results found in the academic literature is provided.

Mu et al. [42] investigated IP routing performance with graphics processors. The paper performs IP lookup and pattern matching, as well as DPI operations on the packets. While the results are impressive when compared to a pure software implementation on a CPU (IP lookups are sped up by a factor of 6, while regular expression and Bloom filter-based string matching exhibit a 5x increase in performance), no NPU comparisons are made, which detracts from their value. Comparison of a DPI Bloom Filter based algorithm with an FPGA implementation shows a 50% speed up in the GPU implementation.

Han et al. [33] perform a similar study but offer a comprehensive packet processing framework on a GPU, complete with adaptations on the software level to optimize the packet flow in the system. The tested applications include IPv4 and IPv6 forwarding, operation as an OpenFlow switch and as an IPSec gateway. The results show again significant gains in throughput, particularly in the OpenFlow switch and IPSec gateway applications. In both IP forwarding application a maximum throughput of 40 Gbit/s is reached and held steady independent of packet size and protocol. The perseverance of this ceiling indicates that some other bottleneck exists within the system, which does not allow performance to scale as needed.

While the results show promise for GPUs in network processing, they do not in any way vindicate GPU proponents claiming that GPUs could replace NPs. This is mainly because even if GPUs can in some future generation and with some architectural adaptations offer the performance leap expected of them, one still has to weigh in their considerable power dissipation, which is far higher than any current NP (the EZ Chip NP-4 is rated at 35W, whereas a top-of-the-line GPU can peak out at over 200W). Furthermore, while

programming GPUs is not inherently more difficult than programming NPs, it follows completely different programming paradigms and uses different tools and libraries, which means that a large investment is required to retrain personnel and port existing software.

2.4.3 Useful Conclusions

Section 2.4.1 introduced the Network processor architectures which can currently be found on the markets. Most of these architectures have tried to cope with higher throughput by increasing the number of processing cores available. However multi-processor systems always run into scalability issues, since several resources, like the communication infrastructure and common memory, are shared among the processing elements. In [60] Weng et al. illustrate this point by providing simulation results of the memory access latency in a multiprocessor NP. A linear increase in access latency is observed, which can only be mitigated by increasing the number of memory channels, something that is of course limited by the complexity of the memory controller and the high pin requirements of memory interfaces. In contrast pipelining the NP allows for an increase in operating frequency, which in the optimal case is proportional to the number of stages (this being subject to proper balancing between the pipeline stages), and subsequently an increase in throughput, while alleviating the memory access problem.

Furthermore, multi-core architectures use a Load/Store model to access instructions and data, then perform some operation on it and write the results back to a memory. This approach is effective for instruction intensive application (e.g. higher layer processing) but less so for the layer 2-3 processing typically done in core networks processor. These applications are data intensive, meaning there is a large load of data at any given time to be processed but the processing itself does not require that very complex work is done. This is offered by a pipeline architecture, which allows the packet data to flow through it, while being processed. In summary, pipeline architectures exhibit the following advantages:

- Deterministic performance since the processing path and involved latencies are predictable.
- No complex, power hungry cross bar or interconnect is needed.
- No memory stalls due to data being fetched from memory.
- Instruction and context data are in the immediate proximity of the processing elements.
- No branching delays or perplexing thread synchronization schemes

Due to these factors, pipeline architectures are able to offer more performance for significantly reduced power consumption, die area and pin count. Therefore, pipelined architectures are gaining traction in high-speed packet processing. This work proposes such a pipelined approach, which attempts to take the basic pipeline concept and spin it around in order to optimize its performance for core networks. The resulting architecture improves the flexibility of the basic pipeline approach and reduces the amount of information that needs to flow through the pipeline core, thus significantly reducing the number of bits that

have to toggle with each clock cycle and as a consequence thereof the power consumption of the pipeline.

2.5 Programming Network Processors

NPs have been been plagued by complicated programming concepts during their existence, which reduce the usability of the devices by requiring that the user has intimate knowledge of their architecture and thus make programming them a challenging task. Why this has been the case is complex question, whose answer requires that we initially define what exactly programming an NPU means. This a multi-faceted process, which entails everything from the programming language used, to the compiler and its capabilities and to how and where the data is stored in the NPU. This section examines some of the NPUs, whose architectures have already been described in section 2.4.1 from a programming perspective and tries to answer the above questions. As with their architecture, details on the programming interface of each NPU are scarce and mostly concern the highest level, that is the development environment with which the prospective programmer comes into contact with.

Programming network processors has always been compounded by the need for processing efficiency, the partially custom designs and lack of a homogeneous architecture even at a high level. This has led to a very fragmented market which consist of various solutions for architectures ranging from typical multiprocessor systems with co-processors to more exotic pipeline structures and in turn to very divergent programming concepts, which this section will attempt to sum up.

Many of the NPUs described in section 2.4.1 are based on a multi-core approach, integrating multiple PEs, usually RISC processors but on occasion more custom design (e.g. the EZChip NPUs). Thus from a programming perspective they are Load/Store architecture, which share much in common with typical CPUs. Several architectures fall into this category, such as the Intel and successive Netronome ones, the afore mentioned EZ Chip, and the LSI and IBM offerings. How each of these NPs is programmed varies between a mixture of high- or low-level languages and automatic or manual partitioning. Some also provide software libraries which can be used to speed up the development of low-level code. The common denominator is that they all leave a lot to be desired, since none offers a comprehensive, automated programming flow.

The Xelerated network processors [12], being a pipeline design differ considerably from previous solutions from a programming perspective. Programming these NPs is considerably easier due to the fact that they view the array of resources as one processor and write code which is then automatically assigned to the various PISCs and whose performance is guaranteed. Each instruction takes up one processing core in the pipeline. The downside is that the software has to be written in assembly language, which can be cumbersome, despite giving the programmer full control over the hardware resources. According to the company this approach reduces the code development time dramatically, by shaving off the need for excessive performance optimizations, required in multi-core architectures due to thread contentions. A further advantage of the Xelerated architecture is the proximity of the code to each PISC. Each processing elements stores its own code and thus avoids

costly and unpredictable accesses to for away instruction memories.

The most promising development in the NP programming space was announced by EZ Chip shortly before this thesis was submitted. The EZ Chip NPS [28] is a new series of network processors aimed at core networks and set to debut in 2014. What makes them unique is that despite the fact that they are slated to share the same TOP-based architecture with the NP series, the NPS is going to be programmable using C. The fact that commercial vendors are slowly adopting this approach shows the importance of higher level programmability for NPUs. Unfortunately, at the time of completion of this work there were no further details available on the programming paradigm that the NPS series will follow.

The above examples provide a glimpse into the incoherent landscape of NP programming. As a logical consequence thereof, a lot of effort has been put into improving programmability, focusing on various areas. The three core aspects that a programmer-friendly NPU must posses are: use of a high level programming language, automated task allocation to processing elements and predictable behavior. None of the products explained previously offers all three, though some come close. The scientific community has provided useful insights into each aspect and has come up with interesting concepts for each of them.

Wagner and Leupers investigate in [59] the design of a compiler for an application specific architecture, such as a network processor. The paper provides useful ideas on how to map high level programming constructs to low level functions, such as bit-level access to data registers, as well as how to exploit specific features such as accelerators, without resorting to low level programming or sacrificing code efficiency.

Looking into the problem of task mapping from a different angle, Yu et. al attempt in [36] to improve task partitioning and allocation of tasks into processing modules by using a task graph and applying a divide and conquer approach to it.

A further interesting idea is the use of meta-programming to extract additional efficiency out of a C++ code [39]). Meta-programming is a technique applied to a piece of code, which transforms the code before compilation (doing e.g. loop transformations). This concept can be useful in the NPU setting in order to take the programmer's code and respin it accordingly in order to optimize resource use in the processor.

The NPU100 design presented in this work takes a step back from current approaches and attempts to integrate the programming paradigm into the design phase. Thus how the NPU100 is programmed, and the way the program is stored in the processor is deeply ingrained in the system. This works in tandem with a software side, which simplifies the work of the programmer by allowing him to write high level code which is then automatically assigned to the appropriate NPU100 processing modules and translated into the appropriate programming commands for them. Thus a complete high level programming flow is provided, something missing from current NPs found on the market.

2.6 Hashing

Hashing is such a wide topic, that it readily defies attempts to limit it to an area of electrical engineering. It is ubiquitous to anything from cryptography, image compression,

error detection and of course network processing. Since these fairly incongruous areas present different requirements and thus have given birth to diverse hash approaches, this section does in no way attempt to provide an overview of the topic at hand, but rather attempts to incise and extricate only the elements required for this thesis. First however an overview of the concept of hashing is called for.

Hashing on its highest level involves mapping one set of values to a different set. With how many elements each set is populated is immaterial though typically the destination set is smaller than the original set. Figure 2.15 illustrates the difference between a simple 1:1 mapping scheme (shown on the left side), where each element of the original set corresponds to one element of the destination set and an M:1 scheme (shown on the right), where the number of elements in the destination set is smaller than the ones in the originating set and thus more than one elements of the latter are mapped to the same element of the former.

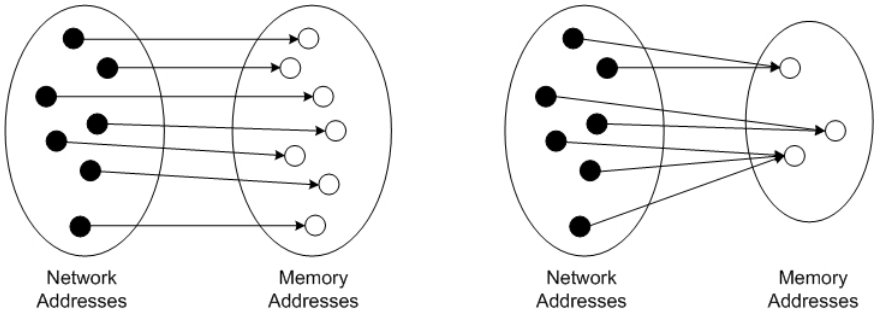


Fig. 2.15: 1:1 and M:1 Hash Mapping Schemes

This phenomenon, termed a collision, is the pariah of all hash approaches. Collisions are a direct consequence of mapping more than one elements of the original set to each element of the destination set. Given that this is usually the case in hashing applications, collisions have been well studied due to the extended applications of hashing in a variety of scenarios. Extensive investigations have, as was to be expected, yielded a variety of results regarding the so called collision resolution. These results again depend on the application and the implementation platform (since e.g. software implementations tend to prefer serial algorithms, while hardware ones are adept at sorting out collisions in parallel).

The mapping is done through a function called the hash function. An input key which identifies the object is fed to the function, which returns then the element of the destination set, which matches that key. The trick is then in finding an appropriate hash function, which minimizes collisions, while correctly mapping elements from the one set to the other. This has consistently been something of a black art, since there are no guidelines to follow and finding a suitable function depends completely on the application that in which it will be used [50].

There is a multitude of reasons to employ hashing depending on application. From a purely algorithmic perspective, it is superior to all search algorithms where speed is concerned and loses out only to the binary search on space use. A good general introduction into hashing, which outlines its attributes and characteristics, is provided by Knuth's seminal work on computer algorithms [40].

Hashing in the NPU100 is used in the Forwarding lookup module (section 4.2.4) to reduce the number of bits of the input key to a number that can then be used to access the forwarding lookup memory and retrieve the required entry. The specific requirements that arise from this application environment as well as specific algorithms that were tested are being discussed in that section.

2.7 Interfaces

This section provides an overview of two standardized interfaces used in the design of the NPU100. The first, PCI Express, is a point-to-point lane based interconnect, which is used for connecting the network processor with the control plane, as well as to transfer the program which is executed on the NPU100, to it. The second interface is Interlaken, a chip-to-chip solution designed mainly with networking in mind, which is used as the NPU100's interconnect with an external MAC/LLC chip as well as with the switch fabric of a hypothetical NPU100 line card, as described in section 2.3. A short overview is provided for both protocols, with references pointing to detailed information.

2.7.1 PCI Express

PCI Express (PCIe) [4] is a high performance interconnect, most typically known for its use in computer peripherals, but one that has found widespread adoption in several other areas. PCI Express was launched in 2004 as a replacement for a host of other interconnect technologies such as PCI and AGP. PCIe's greatest advantage is perhaps it's versatility. Being based on a lane concept, it is scalable and thus can be used to implement any link independent of the bandwidth requirements. Further advantages are a low pin count, as one differential Tx-Rx pair is used per lane and low operating voltage which ensures reduced power consumption. The improvements over the previous PCI technology are immense since a single PCIe lane can transfer 2.5-10 Gbit/s (depending on the PCIe standard version implemented), almost double the maximum PCI bandwidth, while using only one tenth of the pins (Standard PCI used a 32 bit interface). This translates into savings across the board, since the technology requires smaller connectors, which occupy less valuable PCB real estate and the fewer signals that need to be routed result in simpler board designs.

PCI Express's scalability is underpinned by two important features: the lane concept and the exclusivity of each link. The lane concept allows the aggregation of multiple lanes in one link in order to satisfy the given requirements. The standard defines connectors with up to 32 links (termed x32 in PCIe jargon) but does not preclude other configurations. Link exclusivity means that each PCIe channel is a point to point link and not a bus as used to be the case with older technologies. This was a necessary move since connecting many devices on a wide buses presented electrical issues, not to speak of performance suffering due to saturation of the bus' bandwidth.

PCI Express however is more than a physical interface, despite that being an indispensable part of it. It consists of the 5 layers [61] illustrated in figure 2.16, which work harmoniously to ensure efficient data transfer. The two top most layers implement the in-

terface with the Operating System (OS). Here, PCIe inherits the PCI programming model, which it extends accordingly, in order to enable backwards compatibility. It maintains the PCI flat addressing model, which uses loads and stores from that addressing space to write and read information to and from the device and provides additional addressing space (known as the PCIe extended configuration space) to facilitate further functionality not supported by PCI. The industry standard Plug and Play (PnP) model is used to configure a PCIe device on power up without the need for user intervention.

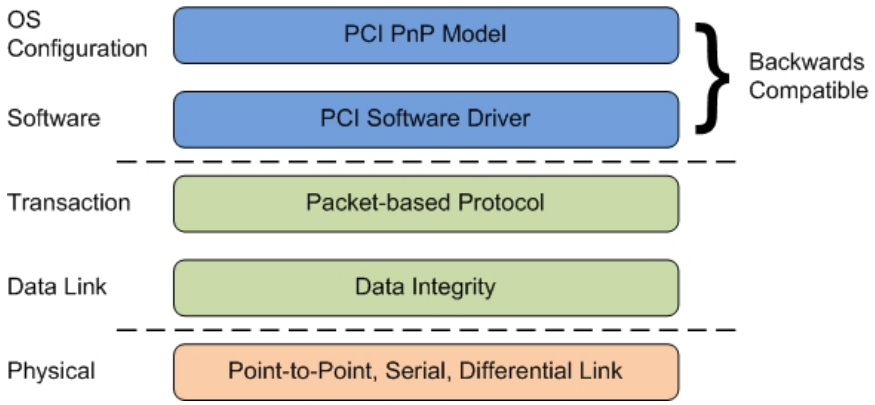


Fig. 2.16: PCI Express Layer Stack

The load and store commands from the software layer are transformed by the the transaction layer into the so called Transaction Level Packets (TLPs). Each TLP is at most 4096 bytes, while typical implementations support only a fraction of that (128 or 256 bytes). Thus, the user data are broken up into several TLPs. TLPs also implement the "virtual wires "functionality, which enables PCIe to transfer all sorts of information (e.g. interrupt handling, power management) through virtual links, over the same physical interface. This enables PCIe to minimize the number of signals used. All this requires a header which is prepended to the user data.

Then the data link layer adds a sequence number and CRC to these packets to create a reliable data transfer channel. It implements flow control between the receiver and the transmitter nodes to avoid overflowing the receiver with data. Erroneous packets are automatically retransmitted without higher layer intervention. Finally, the physical layer consists of a Tx and Rx differential pair per lane. The physical layer also encodes the clock into the signal using an 8b/10b encoding and the TLP overhead consumes some throughput, which brings the bandwidth available to the application to 2-8 Gbit/s per lane. Data is stripped across the available lanes, although the dis- and reassembly process is transparent to the upper layers.

PCIe is used within the confines of this work to transfer data between external devices and the NPU100. Under external devices three are commonly meant: The control plane CPU, which must be able to update to lookup tables of an NP in order to set up or to erase a connection and sets the priorities for packets flows, the management plane which monitors the device operation and its performance and other related OAM tasks, and finally the NPU100 compiler, which generates the programming words for the NPU100. These are then send by a PCIe driver over the PCIe link to the FPGA in order to program

the NPU100. PCIe was chosen because it has been making great strides at becoming an industry workhorse, since it offers a very high throughput point-to-point architecture with a very low pin count, which is advantageous in typical switches, where data has to be sent out to NPs residing on multiple line cards. Line cards and NP platforms from almost all vendors (almost the entire product line of Netronome [43] and Advantech [13] just to name a few), as well as most of the NPUs described in section 2.4.1. utilize PCIe for various purposes, most commonly for communicating with the control plane. Hence it was deemed beneficial to design the NPU100 using an industry standard interface.

2.7.2 Interlaken

Interlaken [10] is a serial multi-lane interface developed by Cortina Systems and Cisco, which is optimized for high bandwidth packet transfers. It is based on SPI4.2 [3] from which it borrows several features. Interlaken was developed to bridge the gap between the two previous dominant protocols in chip-to-chip data transfer, XAUI [8] and SPI4.2 . XAUI offers a narrow 4-lane interface and long range but lacks channelization and flow control, while SPI4.2 offers these, but has a wide interface with limited scalability. Interlaken bridges the gap, by providing a narrow, extensible, channelized packet interface, which is in principle, not dissimilar to the PCI Express described in 2.7.1.

Interlaken is based on a lane concept, with multiple lanes being grouped together to achieve higher throughputs. The user data are segmented into bursts, with each burst being delineated by a control word at the beginning and one at its end. The control words carry information like error detection, etc. Data is striped in 8 byte words to all lanes to achieve high efficiency and scalability. All data, even the control words are 8 byte long. Striping is done sequentially starting from the first lane. It provides up to 256 logical, prioritizable channels over the same interface, which can be used either for data or for other functionalities like in-band control flow signaling, etc.

Its target applications are almost exclusively network oriented, with line card to switch fabric and framer/MAC to NPU interfaces standing out. This is exactly the role it plays in the NPU100.

3 NPU100 Architecture

This chapter introduces the architecture of the NPU100 network processor. Pivotal to this chapter is not simply a blunt description thereof, but a comprehensive narrative, which highlights its advantages and how they were realized by methodically pursuing the goals set in section 1.2, which were to provide a power-efficient, modular, programmer friendly, high-throughput CGE network processing architecture. Devising such an architecture, which fulfills these requirements is, however, far from a trivial issue. The first consideration is that any network processor is not a stand-alone component, but lives as part of an environment, which was described in section 2.3. Section 3.1 introduces the outer layer of the NPU100 architecture, which interfaces with that environment, namely the network node.

After describing the NPU100 operating environment, the focus shifts to the NPU100 processing core architecture itself, which is the task of sections 3.2 and 3.3. These seek to explain the way the NPU100 architecture realizes the purported advantages, the former dealing with the power consumption aspects, while the latter expounding the programmability and modularity facets.

Section 3.2 starts with an introduction of starting point of the architectural exploration, which formed the basis for the architecture proposed further down the road. The conclusions drawn from this account flow into a pool of ideas, which form the clay from which one of the two main contributions of this thesis, the folded architecture is molded. This architecture puts the specific traits of CGE protocols to use in order to improve upon the power consumption of a typical pipeline architecture. The main operational concepts of data buffering in order to minimize data path width and an early exit strategy in order to limit the number of stages each packet header traverses and the methods which enable them are explained. The folded architecture is then compared against two other alternatives. The first is a standard pipeline structure and the second is a design which attempts to minimize power consumption by storing the data to be processed in registers and sending only the part to be processed to the appropriate module, thus eliminating the continuous toggling of pipeline registers. The comparison verifies that the folded pipeline architecture is the most suitable one for CGE networks.

Finally, section 3.3 briefly introduces the programming concept of the NPU100. It is divided into a hardware and a software part, which work together in a harmoniously to enable the prospective user to leverage the processing capabilities offered by the NPU100, while avoiding resorting to cumbersome techniques which would hinder this goal. The direct high level language-to-hardware approach is unique in the NP world, since no design has thus far offered a similar programming concept. Modularity is preserved by abstracting the task-to-module assignment from the programmer. Thus the NPU100 is endowed with a complete programming tool-chain satisfying both the programmability and modularity requirement.

3.1 NPU100 Outer Layer

This section introduces the architecture of the NPU100 network processor. It moves down into the first level of detail of the NPU100 design to describe the functionality of the NPU100, excluding its pipeline core and programming concept, which are the subject of subsequent sections. This architecture includes elements that support an efficient, power conserving processing core, along with an intuitive programming mechanism, thus providing a complete solution for processing CGE traffic at 100 Gbit/s. Figure 3.1 provides a high level block diagram of the NPU100 system. It consists of a software and a hardware part.

The software part ensures the easy programming of the network processor by automatically generating the necessary commands from a high level C like code and XML files, which define the operating environment. This programming information is then sent to the NPU100 over a PCI Express interface which ensures the efficient transmission of the programming data to the network processor. There the data is being distributed to the appropriate modules. The complete concept is described in more detail in section 3.3.

The hardware side consists of the actual processing core along with any modules required to facilitate the processing. This includes, apart from the already mentioned configuration interface, an interface to the Ethernet MAC which is responsible from transmitting and receiving packets (though the MAC themselves are not part of the NPU100 design) and link to exchange data with the switch interface. The exact nature of these two links depends on the implementation and in this section Interlaken interfaces (introduced in section 2.7.2) are considered. This serial interface is widespread among implementations where high speed data throughput is required.

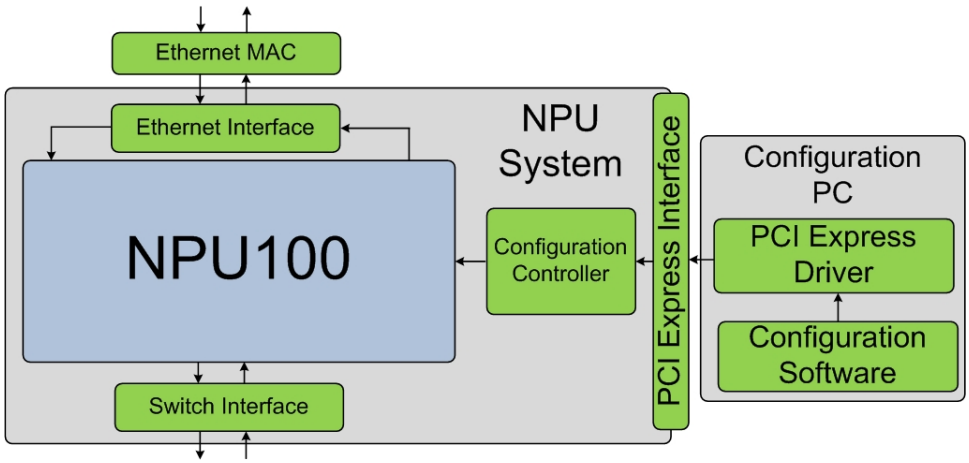


Fig. 3.1: High-level Block Diagram of the NPU100 System

The remainder of this section focuses on describing the functionality of the hardware part of the NPU100 and explaining in detail how it can achieve significant power savings in comparison to other solutions. The description of the NPU100 proceeds in an onion-like fashion, where each layer is peeled to reveal the next one. This process begins with the outer layer described in the subsequent two sections, of which 3.1.1, explains the rationale behind the Header Payload Split concept and how this benefits power saving in the design

by avoiding the sending of entire packets down the pipeline and temporarily buffering them in a memory.

3.1.1 Header-Payload Split Concept

A basic hypothesis which underlies both investigated CGE protocols, is that only the header needs to be processed and thus the payload does not need to be carried through the pipeline. This is the case, since no complex deep packet inspection (DPI) processing is required in core networks. Thus detaching the header from the payload and sending only the former through the processing core will minimize the amount of information that needs to be processed. The data volume to be saved depends on the size of packets that arrive and is in any case limited between a maximum value, encountered when packets of the maximum possible size are used and a minimum which occurs when packets of the minimum possible size arrive at the NP. If we assume the header to be 128 bits wide (something that would correspond to 4 MPLS labels and would also be sufficient for a PBB-TE header), the amount of data saved is calculated using the following equation:

$$DataSaved = (PktSize - HeaderSize) \times PacketsPerSecond \tag{3.1}$$

where number of packets per second can be calculated by considering the minimum (64 Bytes) or maximum (1522 Bytes) Ethernet frame size plus the inter frame gap and preamble:

$$\begin{aligned} MinPacketsPerSecond &= \frac{LinkThroughput}{Preamble + InterframeGap + MaxPktSize} \\ &= 8.1Mpps \\ MaxPacketsPerSecond &= \frac{LinkThroughput}{Preamble + InterframeGap + MinPktSize} \\ &= 148.8Mpps \end{aligned} \tag{3.2}$$

Substituting these values into equation 3.1, the values 57.13 Gbit/s for the minimum and 97.32 Gbit/s for the maximum value of the data to be stored in the memory and avoid being sent down the pipeline aimlessly are calculated. Thus separating the header from the payload makes absolute sense under any circumstances.

The maximum number of packets that can arrive at the NPU100 can additionally be used to determine the required operating frequency of the pipeline core. Since a pipeline is a deterministic structure that can complete its processing within one clock cycle, our system can be viewed as a black box with a given latency (dependent on the number of pipeline stages), in which a packet comes in in every clock cycle and exits after a predictable amount of cycles, thus guaranteeing the completion of processing for one packet per clock cycle. Hence, if one packet can arrive every clock cycle and the maximum number of packets that may arrive on a 100 Gbit/s Ethernet link is 148.8 million, a frequency of 149

MHz will be required.

The Header-Payload separation, called henceforth simply Header-Payload Split is implemented by two modules, the Header-Payload Split and the Packet Reassembly, as figure 3.2 illustrates. The former receives 100 Gbit/s data from the Interlaken interface (labeled as MAC/Switch I/F on the picture since it depends on if the NP is used as an egress or ingress NPU) and copies the part of the data, which was designated as the one containing the header. The entire packet data is then written to the packet memory, which is organized as a simple FIFO, in order to take advantage of the predictable nature of the pipeline's operation. Packets in a pipeline, exit the pipeline in the same order in which they enter it thus a simple FIFO is enough to store them during processing. Packets are written into the FIFO in 128 byte increments and are read out in identical fashion.

After the NPU100 pipeline core is done with the packet header, it arrives at the Packet Reassembly module, where it is being put back together, before transmission. Putting the packet back together the packet reassembly module must consider that some labels (in case of a label based protocol) might be redundant or that new labels have been added, and thus discard any labels erased during processing and add any pushed ones in the right location of the packet.

Between these two units the main processing core of the NP performs the necessary functions to forward the packet. A detailed description of the functionality of the Header Payload Split subsystem and the modules from which it comprises can be found in section 4.1

An important decision in the design of the Header-Payload Split subsystem is whether to make the memory inclusive or exclusive, that is, if the memory should mirror the data in the pipeline and thus contain a complete copy of the packet, including the header, or if it should store only the packet data without the header. The decision to design the memory as an inclusive one was taken for two reasons: The first is that it allows us to simplify the design of the processing core considerably. As will be explained in a subsequent section (3.2.1), the processing core data path is minimized, by only sending the relevant part of each packet header in the pipeline each time. This however could result in data loss since new data generated during processing would need to overwrite existing ones. Thus either a copy of this data must be maintained somewhere for later use or the data path must be widened to provide extra space. Data path width minimization is however one of the pillars on which the NPU100 pipeline core architecture is based and as such must be adhered to. Thus a copy of the data that might be overwritten has to be maintained somewhere else, from where it can be read out and recombined with the processing results. The second, complementary reason for this decision is the limited size of the required additional storage. In section 4.1.4 the size of the buffer memory is calculated to be 94116 bytes used to accommodate 62 packets (62 is the maximum number of packets that can be in the NPU100 in any given time as will be explained in the implementation description in chapter 4). Using the previously made assumption of a 16 byte header the overhead from storing the header in the memory is 1KB, approximately 1% of the total memory utilized. Hence the name Header-Payload Split is not completely justified as the header is not actually separated from the payload, but is simply copied to the pipeline core.

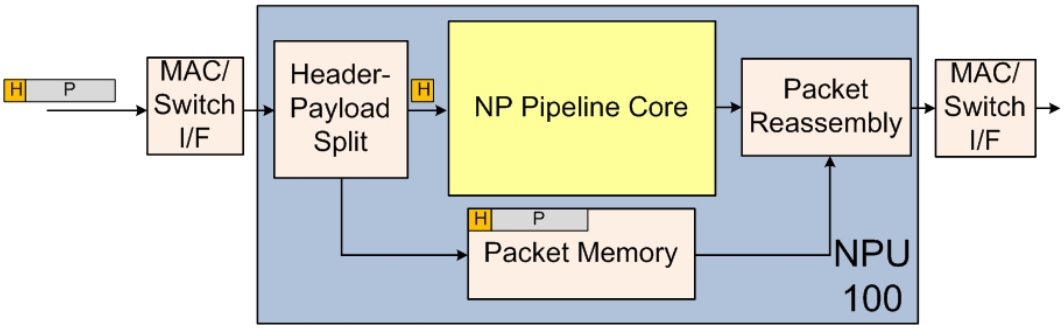


Fig. 3.2: Block Diagram of the NPU100 Outer Layer, which Shows how the Header is Copied and Sent to the Pipeline core while the Complete Packet is Stored in the Packet Memory.

3.1.2 NPU100 Internal Header

As the NPU100 is aimed at linecards, which are to be interconnected into a larger switching system, a method is required to transmit information between the NPUs residing on different linecards (e.g. from the ingress NPU to both the ingress and egress switch interfaces and the egress NP). This information includes the packet length, the input and output ports for this packet, as well as QoS parameters for the ingress and egress queuing systems. To accommodate for this, a switch internal header is used, which the ingress NPU100 creates and fills with the required information during processing and which is removed by the egress NPU100. The format of the internal header is shown in 3.3. It is defined in accordance with the requirements of the MPLS-TP and PBB-TE protocols, however since future protocols may require different structures, the NPU100 must be able to support any changes that may need to be made to it.

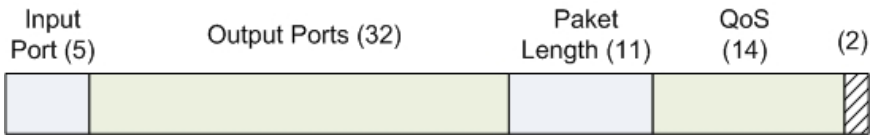


Fig. 3.3: Structure of the NPU100 Internal Header

The currently defined internal header format includes the following fields:

- 5 bits to identify the input port, which by using binary coding allows for up to 32 input ports. Since a packet may have only one input port, binary coding may be used in this case. This number of port is sufficient for a core router, where the total number of ports per node is relatively small in comparison to an access router.
- 32 bits used to designate the output ports for this packet. These are used as a bitmap, since it must be possible to send a packet to more than one output port at the same time.
- 14 bits for specifying the QoS parameters for the packets, further subdivided into 2 sections, one for the input QoS parameters in the ingress path and one for the output QoS parameters in the egress path. Each section contains the following fields:

  - the forwarding class, which defines the throughput and the queuing priority for this packet flow.
  - the drop precedence, which sets the priority of a packet in terms of it being discarded due to network congestion.
  - the drop behavior for this packet, meaning how should a packet drop be handled (e.g. if it should be simply ignored or if the management plane should be notified).

- 11 bits for the packet length. These are sufficient for Ethernet packets, which are the focus of the NPU100. In order to support jumbo packets this field has to be expanded to 14 bits.

The total size of the above fields is 62 bits, which are rounded to a 64 bit internal header. The remaining bits are reserved for future use. It should also be noted that for the currently standardized traffic classes [34], [58], the bits set aside in the internal header suffice, despite the fact that the some protocols (e.g. MPLS) allow for more traffic classes to be used.

The internal header also facilitates the transmission of multicast packets. Such a packet has to be replicated once for every output port and each copy has to be processed separately. This means that if this was to be performed in the ingress NPU, it would have to create copies of the packet equal to the number of output ports to which the packet has to be sent and then process each copy individually. This would put an enormous strain on both the NPU and the switch interface by multiplying the traffic load to be forwarded, since they would then have to queue and transmit all the copies to the egress line cards. To avoid this, an alternative method is used, according to which only a preliminary processing is done in the ingress path. This determines the output ports and the single copy of the packet is then sent to all egress NPUs which complete the processing by performing any label swapping or other required operations.

3.2 A Folded Pipeline Architecture for Low Power Consumption in High Speed Carrier Grade Network Processors

When designing a network processor architecture, three criteria must be typically kept in mind. The first is performance, typically measured in Gigabits of throughput data per second, which the NP can process. Performance in NPs always defines a hard lower bound. Reaching it is critical, but there is no reason to strive for any additional gains in this area after the requirement has been met. Performance evaluation during the architectural exploration phase is tricky, since precise information about the critical paths in a design cannot be known before a prototype implementation has been coded. Nevertheless, estimations based on the organization and functionality of an architecture's functional blocks can provide preliminary insight into the performance that can be attained.

Apart from the performance goal an architecture must minimize the remaining two parameters, namely power consumption and chip area. The two parameters are of course highly interwoven since any reduction in chip area typically means less devices to consume power. The design and implementation of a low power NP architecture was one of the major goals of this work. Determining a suitable method to accomplish this goal is however not a trivial issue, since power consumption is the confluence of different design and technology aspects. In section 1.1 a rough sketch of those was outlined. According to this, static power dissipation lays out of the purview of this work, since it is completely dependent on the process technology used and its characteristics. Thus any improvement must come from focusing on the dynamic power dissipation. This depends on the factors outlined in equation (1.1), from which capacitance (C) and and the operating voltage (V) are also determined by the used fabrication process. The operating frequency (f) was addressed previously in section 3.1.1 through the Header-Payload Split, which allows the use of a low 150 MHz frequency for the pipeline core, which will perform the bulk of the processing. As a result of this, there is little room for improvement on the frequency front and thus the switching factor will play the pivotal role in our strategy.

The switching factor is however an impalpable concept with many constituents and hence it is difficult to determine whence benefits may stem. Our approach was based on attempting to identify if a circuit characteristic exists, which plays a fundamental role in the dynamic power consumption of the circuit. In a scalar, in-order, pipelined NP design this is the pipeline width. Reducing the pipeline width can have a profound impact on the power consumption since it affects the size of each pipeline register. This is a valid assumption only for designs that consist of a large number of pipeline stages and thus employ many such registers. This is clearly the case in a core network NP, in which MPLS protocol requirement make evident that multiple stages have to be used to complete the processing of multiple labels in a single node. This assumption is in line with the Xelerated pipelined design (2.4.1), which uses no less than 512 pipeline stages.

A second facet of the switching factor upon which this work touches and which is particularly evident in long pipelines, is limiting the amount of idle stages a packet has to go through. This typically happens because a pipeline is dimensioned to handle some worst case scenario, which most packets do not adhere to and thus have to go through several pipeline stages without having any processing done to them. This is terribly wasteful of energy since a huge amount of transistors toggle for nothing.

This creative process gave birth to the folded pipeline architecture, which is one of the two main contributions of this work. The remainder of this section explains the key concepts behind this architecture and compares it with a typical conventional pipeline alternative, as well as a second possible architecture called the registered architecture, to determine if it merits the use as the slate upon which the NPU100 implementation is to be based. In light of this, section 3.2.1 elaborates upon the three aforementioned architectures and explains their operating principles and claimed advantages. The last part of this section details the precise pipeline stages that will be used in order to help clarify the comparative analysis that follows in section 3.2.2. There, the three architectures are evaluated, first qualitatively and then quantitatively, with each method revealing different aspects of the each concept.

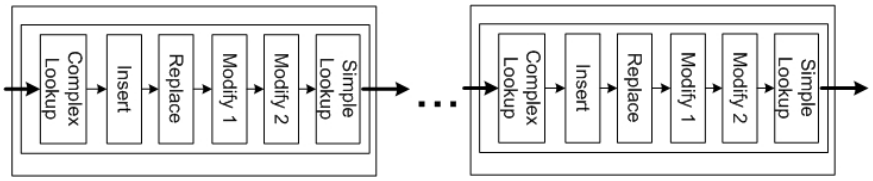


Fig. 3.4: Overview of a Conventional Pipeline Architecture

3.2.1 Architectures Considered

The standard against which the suitability of the proposed folded pipeline, as well as the registered architecture are measured is a conventional pipeline architecture, consisting of multiple stages, set in sequence to one another. Data flows down the stages in each clock cycle until it exits the pipeline after a number of cycles equal to the number of stages. This allows for higher clock frequencies to be reached, since the work that has to be performed in each clock cycle is reduced and also enables deterministic performance since in each clock cycle the data move one stage down the pipeline.

The conventional pipeline structure used for the comparison continues along these lines, as depicted in Figure 3.4. It consists of processing elements connected in series through which packets flow. To allow for parallels to be drawn among the three alternatives, an attempt is made to describe them in similar terms. Thus the basic processing required by any current and most probably future CGE protocol was determined and a basic processing element which can perform these operations was defined. This element is called a mini-pipeline (MP). Each mini-pipeline includes the necessary modules to complete the processing of a single packet header. These modules coincide with the ones required by more or less all current and past core network protocols (e.g. a lookup to fetch next hop data, arithmetic and logic operations on packet header fields, etc). The operations included in the NPU100 pipeline are further elaborated upon in this section. The pipeline is then assumed to be made up of these MPs, which are then connected in series with each other. The MPs are just a logical division here and in no way reflect back to real changes in the design of the MPs. All MPs, as well as the modules that make them up were designed to be identical. The division serves to allow the comparison between the three architectures. A packet is first processed by the first mini-pipeline, then by the second one and then by the subsequent ones. Hence, a packet has to go through all the MPs independently of how much processing it requires. As a result of this, the first stage is always more (or at least equally) as loaded as the following stages, which means that for traffic patterns that have a high share of packets that need to have only one label processed, only the first stage will be used and thus considerable resources remain idle, although the packets still have to flow through the entire pipeline, something which leads to high power consumption, while in addition incurring unneeded latency on these packets. What is more, an overloaded first MP means that power dissipation will be concentrated on that spot, which will overheat. Furthermore, such a structure can only process up to a maximum predetermined number of labels per packet, bounded by the number of mini-pipelines.

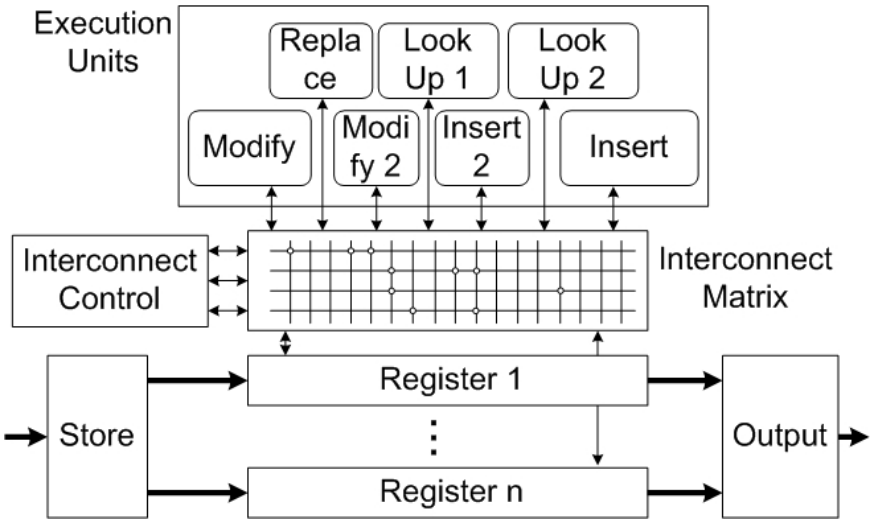


Fig. 3.5: Block Diagram of the Registered Architecture

Registered Architecture

The first of the two architectures proposed in this work is the registered architecture, shown in figure 3.5, which is an innovative approach that attempts achieve a low power consumption by reducing the amount of register toggling that takes place in an NP. In the registered architecture newly arrived packet headers are received by the Store module on the left, which then writes them to the appropriate register using a round robin algorithm. The registers form a register matrix in which the packet headers are stored and are connected to a pool of processing elements (the figure only shows a few for clarity, more will be required in a complete implementation) through an interconnect matrix, which ensures the proper data transfer between them. Then in each clock cycle a part of the data is read out from the register and sent over the interconnect to a PE. This process is controlled by the Interconnect Control module, which is responsible for determining which part of each register is to be sent to which processing engine and where the results should be written back to. Respectively an output module performs the opposite task of the store module and reads out any completely processed packet headers and sends them to the output. The goal of reducing the amount of bits that toggle in every clock cycle ia accomplished by using the register matrix as a packet header storehouse, from which the necessary part is read out and written back into in each clock cycle.

The registered architecture represents a significant departure from a pipelined design, given that packet headers are not flowing down the pipeline stages, but staying put in one register from the duration of the processing. However, the architecture still maintains all the traits of a pipelined design, since the processing is divided into a predictable number stages, which are executed in sequence and thus it maintains all the advantages of a pipeline. One more thing to note is that while at a first glance the architecture is very different from the MP-based concept described previously in this section, this is, however, a mere deception, since the exact number of registers and PEs are required to process the same number of packets, granted though that the organization of these registers and PEs is quite different than the conventional approach.

The major advantage of the registered architecture is that the static storage of the headers in the registers reduces the dynamic power consumption caused by data trickling down a pipeline. Furthermore the design is very flexible, since the order in which the PEs are traversed can be reprogrammed into the Interconnect Control according to each protocols demands. On the down side, connecting several tens of registers with a similar number of PEs over an interconnect consisting of wide buses, is bound to present impediments when implementing the architecture in actual hardware.

Folded Pipeline Architecture

The previous section described the registered architecture, which attempts to reduce power consumption by limiting the amount of data flowing through the circuit. The folded pipeline architecture attempts to do the same, but using a different approach. Here a more traditional pipeline concept is realized, which is however tweaked in order to minimize power dissipation by reducing the width of the pipeline data path. Since in a pipeline data flows to the next stage with each clock cycle, even a moderate decrease in the number of bits can have a significant impact on the resulting power consumption. The second goal was to avoid having data redundantly run through pipeline stages and thus minimize register toggling by not having data words trickle down the pipeline when this is not necessary.

To accomplish this, advantage of the MPLS trait to process more than one labels per node is taken. Assuming that one MP clocked at 150 MHz is able to process one MPLS label per packet in each clock cycle, if more labels have to be processed for each packet header, more MPs have to be made available. Additional passes through successive MPs will then complete the processing of any extra labels. Thus the challenge was how to position these MPs, keeping in the back of one's head, the goal of minimizing the pipeline data path and the in-flight time of the packet headers in the pipelines.

The conventional pipeline concept placed the MPs in sequence, resulting in lower utilization of the subsequent stages and the need to carry all the required processing information down the entire pipeline (meaning, information that would only be required e.g. in the third MP has to be carried needlessly through the first and second MPs). The solution proposed in this section places these MPs in parallel, binds them together through two specialized modules, called the Dispatcher and the Label Reassembly and creates a loop path from the latter to the former. This concept was named the folded pipeline architecture, since it essentially takes a straightforward pipeline, where the MPs would be placed in succession and folds it onto itself to produce the result illustrated in figure 3.6. The folding of the MPs in and of itself however does not endow the architecture with any benefits. The real enablers are the two modules that join the MPs together. These ensure that the data path is kept as narrow as possible and provide an early exit option for packet headers who do not require further processing.

This is done by employing two independent label buffers, one connected to the Dispatcher and one to the Label Reassembly module. These buffers serve as temporary scratch pad memories and allow the two modules to store unneeded header parts, as well as results in them in order to avoid sending them down the MPs.

The Dispatcher and Label Reassembly modules can be programmed to operate in two modes, the label based and the direct forwarding mode. The label based mode is more

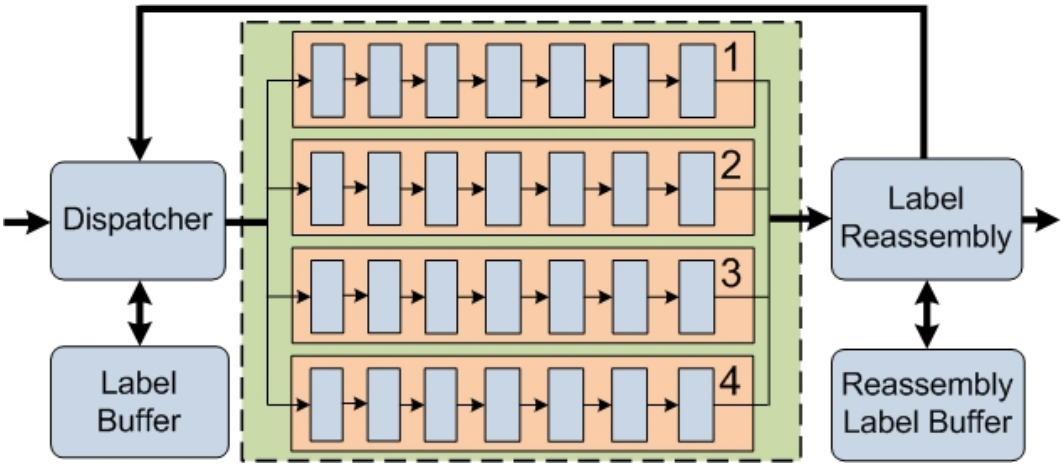


Fig. 3.6: Block Diagram of the Folded Pipeline Architecture

complex and is used for MPLS like protocols, which employ stacks from which labels are popped or to which labels are pushed. The direct forwarding mode simply selects some fields from the packet header for processing. The following description will focus mostly on the label aspects, which leverage the full advantages of the folded architecture, with the occasional digression to highlight how the direct forwarding mode works.

In label based mode the Dispatcher receives a chunk of the packet header, which contains the label stack. It then detaches the top label, which is the first to be processed and stores the remaining three into its label buffer. The first label is then sent to the first MP for processing and after a number of clock cycles equal to the number of MP stages, it comes out at the other side, at the Label Reassembly. There the module checks if the packet header requires additional processing. If it does not, it can be sent to the output, if it does, then the Label Reassembly checks if the packet header needs to store some intermediate results in its label buffer. This is useful in case the space in the pipeline will need to be claimed by some other data. Thus instead of simply making the pipeline wider to accommodate for these data, we simply store the ones who are no longer needed for immediate processing in one of the label buffers. The Label Reassembly determines if data needs to be buffered in its label buffer depending on the operation that was performed on the packet header. If new data have been produced, they have to be stored, lest they be overwritten by the next MP. In MPLS this occurs after a swap or a push operation, where a new label value replaces the existing one. If a pop was performed the label in the packet header is no longer valid and will have to be erased from the stack, thus there is no need to store it.

After any data has been stored the packet header is looped back to the Dispatcher, which has to determine if this packet header needs to have data fetched from the Dispatcher label buffer or if its current data are sufficient for the next MP run. This again depends on the operation that was performed in the previous loop. If the operation erased a label then the next label from the stack must be processed. This next label however, was stored in the Dispatcher Label Buffer when the packet arrived in the pipeline core. Thus it has to be read out and placed in the packet header before it is sent down the MPs again. Push and swap operations do not require new labels to be fetched, as the relevant labels for the

next MP run are the ones already in the pipeline.

This process repeats itself until the processing is complete. In this case the Label Reassembly fetches any data that it has stored in its label buffer, combines them with the latest results from the MPs and sends the processed packet header to the Packet Reassembly, which then reconstructs the complete packet.

While the looping functionality appears at first glance to be MPLS specific, this need not be the case. Other protocols, even non-label based ones, can make use of the loop in order to utilize more MPs if the processing resources offered by a single MP are not sufficient and additional operations need to be performed.

On the surface the folded architecture might seem similar to a ring architecture or a pipeline with an early exit option, in which packets circulate through the ring, passing through the same processing modules on each loop, while the early exit option pertains to the possibility of a packet leaving the ring at specific points. Closer inspection reveals that there is an important difference between these solutions and the one proposed here, since the use of two label buffers, one in front and one after the mini-pipelines, by the folded architecture allows it to reduce the data path width significantly in comparison to these architectures and thus reduce power consumption.

This reduction of the data path width achieves one of the two main benefits of the folded architecture. Since only the data that are needed are sent down the MP and temporary results are stored in the Reassembly Label Buffer, this means that the data path can be made considerably narrower (since this depends on various implementation parameters, the next section (3.2.2) will elaborate on this and provide concrete numbers). The second advantage of the folded architecture is that packet headers can exit the pipeline core when processing is complete and do not have to traverse any unneeded stages. As a consequence we achieve both our goals of reducing the number of transistors that toggle and how often they have to toggle.

Mini-Pipeline Architecture

This section provides a brief introduction into the stages that make-up an MP in order to facilitate the understanding of the architecture comparison that follows. The implementation details for each stage are provided in section 4.2.4. The number of stages in particular plays a crucial role in walking the thin line between required functionality and resource minimization, as it affects the time a packet spends in the system and thus the dimensioning of various system components. These stages have been derived from a detailed study of the MPLS-TP and PBB-TE functional requirements and the discussion conducted here serves to provide an overview of these requirements and how the MP structure was determined from them. It is important to keep in mind that the goal is to make an MP self sufficient in performing basic processing required by typical CGE protocols.

Core network protocols typically call for much simpler processing than their edge network counterparts. This is because a core network node has to be able to handle huge amounts of aggregated traffic at a very high data rate and thus complex processing would be impossible. On the other hand OAM functionality is extremely important for ensuring adherence to SLAs. Hence a core network node is expected upon reception of a packet to determine the output ports for this packet, enforce any traffic management mechanism

(through QoS) that is used, make any changes to the packet header (if any are necessary) and forward the packet. As protocols designed to implement the functionality of a core network node, required MPLS-TP and PPB-TE operations both follow this pattern, with PBB-TE exhibiting exceptional parsimony in the operations it requires.

Operation	MPLS-TP	PBB-TE	Parameters
Field Extraction	Separate the fields requiring processing from the header		Location and width of the extraction
Forwarding lookup	Fetch the data for the next hop		lookup key as input
QoS lookup	lookup the QoS parameters for this flow		lookup Key as input
Delete	Delete a label	Not required	Position and length of the data to delete
Replace	Insert new label	Not required	Position to place the new data and the data itself
Modify	Decrement TTL, set S Bit, modify packet length accordingly	Not required	Fields to be modified, operation to be performed and operands

Tab. 3.1: Required Processing Stages for CGE Protocol Processing

Table 3.1 summarizes the operations called for by the two protocols, as well as in which processing steps they are needed. MPLS-TP is by far the more complex of the two protocols since apart from the forwarding and QoS lookup, which is similar to what PBB-TE requires, it also demands that the packet header is modified in order to add, swap or remove a label. This entails either expanding or contracting the label stack by one label (32 bits) or simply exchanging the current label for a new one. Respectively, if labels are added or removed from the stack the packet length must be adjusted accordingly and in the former case, the S bit has to be set. Finally the TTL field must be processed, which means its value must be checked and if it is not 0 it must be set to it's proper value by either decrementing it in case of a swap or pop or inserting a new value in case of a push. Apart from these operations the in network processors ubiquitous lookups are also required. The forwarding lookup provides information regarding the route the packet is to follow (most commonly the output ports) and the QoS lookup provides the parameter which are used by the queuing algorithm to prioritize the packet according to its traffic class.

The second step in identifying the operations that need to be included in an MP is to allocate these to specific processing stages. Figure 3.7 shows the result of this process, which also mirrors the structure of an NPU100 MP. Some of these stages are further pipelined internally (as e.g. the Forwarding lookup module is divided into three sub-stages), in order to achieve the required performance. These operations were derived from the list in table 3.1, with some minor aberrations to allow for greater adaptability to future protocol needs.

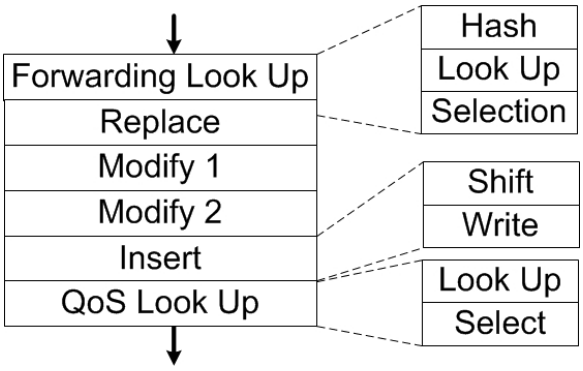


Fig. 3.7: NPU100 Mini-Pipeline Stages Derived from CGE Protocol Requirements

The first module is the Forwarding Lookup module, which produces the necessary information in order to forward the packet. This includes the operation to be performed, a new address value if necessary and so on. It is further subdivided into a hash stage, which transforms the input lookup key appropriately in order to allow for a large pool of entries to be mapped to a smaller one with efficacy, a lookup stage, which performs the actual memory search and access and a selection stage, which picks the most appropriate among many possible results and enforces port filtering policies. Then three processing modules perform various operations on the packet headers. The Replace module can insert a chunk of data in the pipeline word, overwriting the existing data in the process, while the Insert module shifts existing data and then inserts new ones. The latter is again subdivided into two sub stages. The Modify module performs arithmetic and logic operations on the pipeline word. Two Modify modules are included in each mini pipeline ensure the existence of sufficient processing resources for all required processing mandated by the protocols considered. Finally, the QoS Lookup module locates and fetches QoS relevant information, which is then placed in the internal header in order to be used by the switch interface. This module is also divided into two sub stages, one that performs the lookup and a second one, which reads the data from the memory and passes it on.

The sequence of the operations is not rigid, since apart from the forwarding lookup, upon the results of which all other operations are based, the order of the remaining operations is flexible according to MPLS-TP and PBB-TE specifications.

The functionality of each module, as well as the fields on which it operates, is programmable through a microcommand table and can be changed on the fly while the system is operating. Furthermore the architectural concept need not be limited to the number of stages used here, but can readily be adapted as needed since all the pipeline modules share a common I/O interface. For more information on each module and on the complete functional set which it implements refer to section 4.2.4, wherein their implementation is described.

In total the number of stages is 6, however counting the internal pipeline stages of the Forwarding Lookup, the QoS Lookup and the Insert modules, this number climbs to 10. This means that a maximmum of 10 packet headers can be found in each mini-pipeline at any given time. The number of pipeline registers required cumulatively in all the MPs is NxM, where N is the number of mini-pipelines and M the number of stages in a mini-

pipeline. Thus, an issue that still has to be clarified is the number of MPs that need to be supported. As described in section 4.1.1, support for 4 MPLS tunnel layers was deemed sufficient for current and future CGE requirements. Considering that each MP can essentially process one label, which corresponds to one tunnel layer, it follows that 4 MPs are needed to provide the required throughput. As a consequence of this and taking into account the mini-pipeline structure discussed above, up to 40 packet headers will be in the pipelines at any given time. This has direct implications for the sizing of the architectures (e.g. the label buffers, which the folded architecture requires or the switch matrix of the registered architecture).

One final detail to be clarrified is the data that are stored in each of the aforementioned pipeline registers. Each packet that flows into the NPU100 is represented in the processing core by one pipeline word. This pipeline word contains all the information necessary to process the packet and is toggled from one pipeline stage to the other and processed in all pipeline stages as needed. Thus all the necessary information for performing the required operation as well as the space for storing the result must be available in this data word. The actual information contained in the data structure is of course dynamic and depends on the results of the processing stages, as well as on data fetched from the Dispatcher Label Buffer. Figure 3.8 shows the data structure to be used in the pipeline. This structure is compatible with both MPLS-TP and PBB-TE protocols, however the NPU100 is flexible enough so as to be able to use any pipeline word structure that may be defined in the future in order to accommodate other protocols. To this end, the separation of the pipeline word into the three parts described below is purely logical. The length of each of the three parts and the fields that make up each part can be reassigned as needed by the user. The pipeline word consists of the following three areas:

1. the internal header, which carries information needed by the switch interface and the egress NPU. 64 bits are used, as has been described in section 3.1.2.

2. the labels/header. Depending on the protocol it could be either labels (in the case of MPLS-TP), a header, or some fields of the header as needed (as in the case of PBB-TE). Taking into account the required field length of PBB-TE (60 bits for the combined B-VID and B-DA fields) and the length of an MPLS-TP label (32 bits), the length of this structure part is set to 64 bit.

3. the processing context, which is used to transfer data used for intra-pipeline communication. Information which results from one stage and needs to be used by subsequent stages is stored in the processing context. The context is preserved throughout the pipeline core, which means that all temporary results, required by future stages must be placed there. To determine the required length for this field, we have to investigate the functionality of the two candidate protocols. In both cases the major use of such temporary storage will be to accommodate the results from the address/label lookups. These results are used by subsequent stages to determine the packet's destination and treatment. Again MPLS-TP needs more space since a new label field value might have to be stored, accompanied by a new TTL value, QoS parameters and the operation to be performed consuming 42 bits in total. In comparison, PBB-TE requires 32 bits for a port filtering bitmap used to determine the

output ports. In addition to these bits, the NPU100 also requires 9 more bits to keep track of pipeline operations. Thus a total of 52 bits are used in accordance with the processing requirements of MPLS-TP and PBB-TE. A size of 64 was selected for the processing context to allow for a small headroom to be used in possible future functionality.

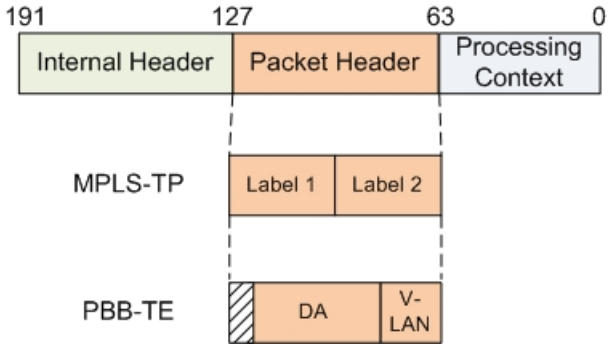


Fig. 3.8: General NPU100 Pipeline Word Structure, along with MPLS-TP and PBB-TE Variants

A common denominator across all pipeline stages and all architectures are the so-called Input and Output Selection modules (ISU and OSU respectively). These provide a common interface to all pipeline modules making them thus easily interchangeable. Each processing stage has one ISU and one OSU. The ISU performs field extraction from the pipeline word and forwards the results to the processing element, which then processes them. Its output is then taken by the OSU and reintegrated into the pipeline word when being written into the next stage register.

3.2.2 Architecture Evaluation

Evaluating the NPU100 architecture was a multi-stage process, conducted with various methods each time in order to get the optimal result. Thus in some cases a qualitative, analytical approach was followed, whereas in others an RTL implementation of key components was preferred. High-level simulations (e.g. using SystemC) on the other hand were neglected, since they would fall short of producing useful results. This is because high level simulations cannot yield precise insight into the power and area requirements of a circuit. Thus in order to get the required information, one has to go the extra mile and delve deeper into lower level simulations, the overhead of which however starts to become comparable to implementing the modules in RTL. Hence the decision was made to omit a simulative analysis from the evaluation of the NPU100 architecture.

This work is centered around the two architectures presented in the previous section, which propound innovative solutions to reign in the challenges of power consumption and complexity in carrier grade network processors. In section 3.2.1 the merits of the two architectures were described and the main tributaries to them were identified. The goal of this section is to elaborate upon these advantages and quantify them in comparison to a conventional pipeline architecture.

The section goes on to first provide a qualitative, analytical evaluation of the three contestants and then moves on to prove that the registered architecture is unfeasible from a technical perspective and finally to compare the folded with the conventional architecture, both analytically and quantitatively in order to prove the former ones supremacy.

Qualitative Results

This section argues the merits of each of the three architectures considered in this work in detail and provides more concrete evidence about the superiority of the folded pipeline architecture as a solution for next generation CGE networks. It functions in a complementary fashion to the next section (3.2.2), which provides quantitative results from the implementation of the three architectures. This implementation does not make the qualitative analysis redundant, since it can help investigate certain architectural traits which are not quantifiable or which the implementation with its broader scope, fails to catch.

In order to provide a better overview, each parameter is discussed in each own subsection and all maintain the same approach of providing an introductory explanation into each parameter discussed and then going on to elaborate upon the specific characteristics of each of the three architectures with regards to this parameter.

It is important to note that the evaluation here corresponds to just the processing core architectures and thus disregards other limitations imposed on them by other modules. Thus, the fact that in the prototype implementation of the NPU100, the main packet buffer is operated in a FIFO queue fashion and thus packets have to exit the processing core in the order in which they came into it, is ignored for the purposes of this comparison. The consequence of this limitation is that the early-exit feature of the folded and registered architectures can't be used to reduce the total packet latency for this packet. Since, however, this is not an inherent drawback of the processing architecture itself, but is imposed on the architecture by design choices made in other components, it is not taken into account when comparing the architectures in this section. This limitation could have been resolved by the use of a more elaborate packet buffering scheme, such as the ones referenced in section 4.1.4.

Datapath Width The data path width refers to how wide the flow of information through the pipeline is, in practical terms, how much information is stored in each pipeline register. This has an impact on both resource use and power consumption and as such it is desired from an implementation perspective to implement as narrow a data path as possible. Thus, an architecture which allows to send fewer information down the pipeline has a clear advantage. When discussing specific numbers in this section we exclude any bits that will be required for the internal header, as well as for the processing context, as they are assumed to be common to all architectures. Thus any numbers given refer to register width required purely to accommodate the packet header data.

In the conventional pipeline architecture, all data that might need to be processed at some point in the pipeline have to available in the pipeline registers from the very beginning. The registers must thus be wide enough to accommodate them. In the concrete case being studied here, the need to fit 4 MPLS-TP labels into the pipeline would require a register

width of 128 bits, all of which would be overwritten with new data in every clock cycle as packet headers flow down the pipeline.

The registered architecture also requires that data that might be necessary for processing are available in the processing core, thus the needed register width remains the same, although the data will mostly remain in that register during the duration of the processing.

In the folded architecture, the possibility to store labels in the Dispatcher and Label Reassembly buffers before and after processing, allows it to send only the part of the data that actually needs processing down the pipeline (one label for MPLS). This permits the use of a considerably narrower data path and is much more scalable, since the data path width will remain fixed independently of the number of mini-pipelines, whereas in the conventional and registered architectures it would grow linearly. In the scenarios considered in this work (namely an MPLS stack depth of 4) this means a shrinking of up to 75% (since only one of those 4 labels will have to be sent down the pipeline each time) in the datapath used for the packet header/labels. As the labels still have to be stored in the label buffer there is no area or resource saving. However power consumption is shrunk since there is no need to constantly transfer these bits from register to register.

Register Power Consumption The differences in the required pipeline width between the architectures have been described previously. This section provides an in depth look into one of the parameters affected by register width, the register power consumption. This depends on how wide the registers are and how often they toggle. The smaller data path and capability to save on the number of processing elements a packet must traverse, allow for extended savings in dynamic power consumption.

To calculate the register power consumption of each architecture we use that of the conventional pipeline architecture as a reference point, and plot the savings offered by the other two architectures for a set of operational scenarios we define. The straightforward architecture is used as a reference as it uses the widest data path and has the highest toggling rate since packet headers must traverse it entirely before exiting.

Several traffic scenarios with different processing requirements in terms of the number of labels to be processed have been used, which are shown in Table 3.2. The scenarios are synthetic workloads, which have been chosen to encompass a wide selection of alternatives, from a more or less balanced number of labels to be processed per packet (scenarios I and II), to more realistic scenarios, where most of the packets will have to have a single label swapped and a minority of labels entering or exiting a tunnel (scenarios III and IV), to the extreme scenarios, where all packet have to be looped one, two, three or four times (in scenario V, VI, VII and VIII respectively).

Figure 3.9 plots the relative savings of the folded architecture in dynamic power consumption, in relation to a traditional pipeline with 4 mini-pipelines. It clearly shows that in even in the worst case scenario for the folded architecture, where all headers have to flow through all the mini-pipelines, a saving of 37,5% can be observed, due to the smaller pipeline width. More impressively the switching power can be reduced by almost 90% when all headers need to traverse the NPU100 only once. This investigation only considers the pipeline register power consumption. Thus, it excludes power consumption in the processing modules themselves as well any modules which are necessary in the one

Tab. 3.2: Percentage of Labels to be Processed for Different Traffic Scenarios

No. of mini-pipeline loops	I (%)	II (%)	III (%)	IV (%)	V (%)	VI (%)	VII (%)	VIII (%)
1	40	25	80	65	100	0	0	0
2	20	25	15	20	0	100	0	0
3	20	25	5	10	0	0	100	0
4	20	25	0	5	0	0	0	100

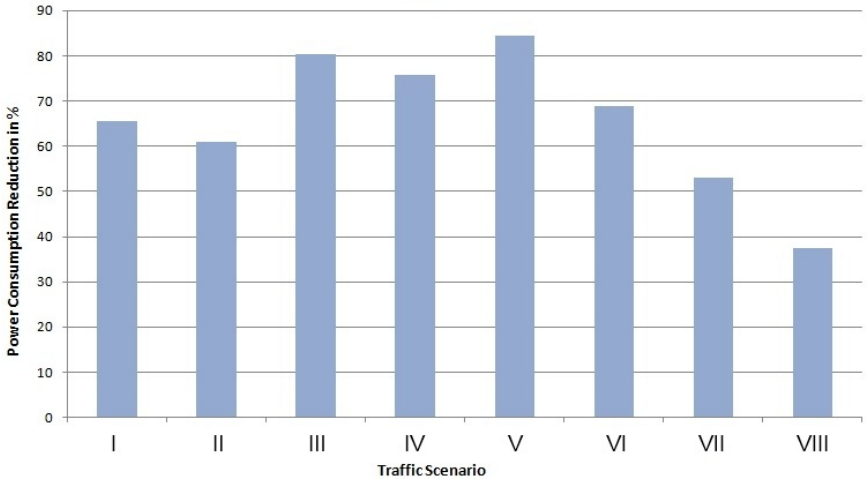


Fig. 3.9: Folded Architecture Dynamic Register Power Consumption Reduction Relative to the Conventional Pipeline

architecture but not in the others. Regarding the former, a narrower datapath will certainly mean lower power consumption in the processing modules as well. This is due to the reduced resource used and complexity, which the reduced datapath width enables. This advantage is however very difficult to quantify with analytical methods. The latter refers to additional modules present in the folder architecture, which make the feedback loop and thus the narrower datapath possible. These modules are the the Dispatcher und Label Reassembly modules and their buffers. Since these modules do not have an equivalent in the traditional pipeline architecture, their power consumption will have an adverse effect on the gains calculated here.

Figure 3.10 provides the same data but for the registered architecture. Explanation of the results here requires some additional information regarding the assumptions made:

- It was assumed that 48 bits toggle in every clock cycle. This number is sufficient to cover CGE protocol requirements and is in line with the more detailed analysis found in section 4.2.4.
- Each packet header can exit the processing core when its processing is done with no further delays.
- Stages which are not required incur no power consumption, since in this case the packet header would simply be kept in the register during these clock cycles in order to maintain synchronization with other packet headers, which might require processing

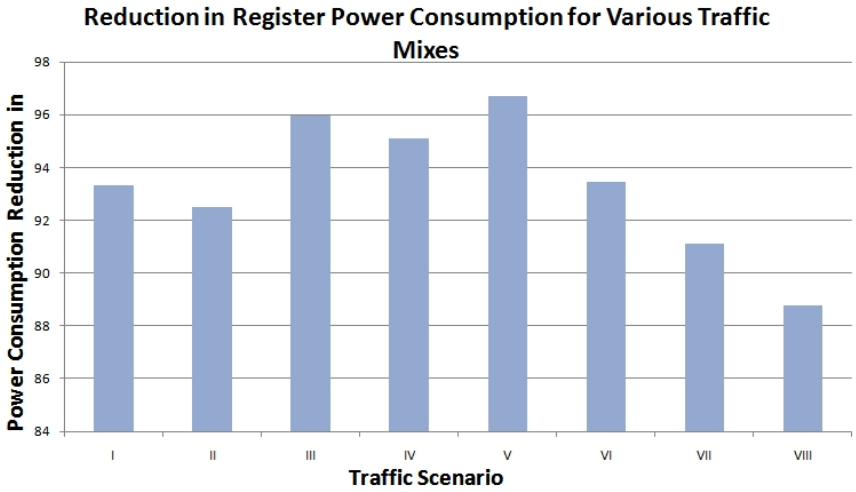


Fig. 3.10: Registered Architecture Dynamic Register Power Consumption Reduction Relative to the Conventional Pipeline

at this stage. An example of this is the QoS lookup in MPLS, which is required only for the first and last labels of a packet.

Since some of the above data is protocol dependent, we use MPLS-TP for this comparison as a reference point.

The results for the registered architecture are even more impressive than the ones for the folded. The minimum gains are more than 88% and in typical scenarios more than 92% registered power consumption is saved. Granted this is only one aspect of how power is dissipated in an NPU, this does not detract from the conclusion, which hints strongly that both the registered has a clear power dissipation advantage over both competing solution and that the folded one offers clear benefits in the same area in comparison to the straightforward pipeline architecture.

Resource Consumption Evaluating resource consumption of the various NPU100 architectures requires us considering some basic architectural characteristics of the design. The first of these is that the processing modules are common in all architectures and thus can be abstracted from the investigation. The same cannot be said for the pipeline registers, which are, as explained in the previous sections, narrower in the folded architecture, something which influences resource use. In order to determine how many registers are saved in a concrete example, the expected difference is calculated analytically in the following equation:

$$NoOfMPs \times StagesPerMP \times (RegWidthConventional - RegWidthFolded) = 2560 \tag{3.3}$$

This calculates the register save between a conventional architecture with a pipeline width of 256 (in order to accommodate all four MPLS labels plus the internal header and the processing context) compared to that of the folded architecture with a width of 192

bits. This is equal to the the difference in pipeline width times the overall number of pipeline stages, which is equal to the number of MPs times the number of MP stages.

The reduced data path width, also has an influence on the complexity of the field extraction and recombination, which is performed in every processing element (the detailed mechanism is described in section 4.2.4). A narrower data path means that there are less bits to choose from and thus the folded architecture should also require simpler ISU and OSU modules in comparison with the other two architectures. The advantage however is difficult to quantify without implementing the modules themselves.

The final resource use element that must be captured are the additional components required by either the folded or the registered architecture in comparison to the conventional one. These are in the former's case the Dispatcher and the Label Reassembly and their respective Label Buffers and in the latter's the switch matrix with it's respective control modules. From these modules, the only ones, whose size can be calculated analytically with any measure of accuracy are the two label buffers. This is given from the following two equations:

$$(NoOfMPs - 1) \times StagesPerMP \times DispCellWidth = 2880 \tag{3.4}$$

$$NoOfMPs \times StagesPerMP \times LRCellWidth = 8000 \tag{3.5}$$

These multiply the number of cells in each matrix with the cell width. The results show that the two label buffers combined require 10880 slice registers, more than what the reduction of the pipeline width saves. This is because cell capacity in the two buffers is more than double what is saved in the MPs. More specifically each buffer needs to hold up to three labels (which is exactly what is shaved of the pipeline word) plus the internal header and part of the processing context for the Label Reassembly Buffer, thus far exceeding the number of bits saved by the narrower data path. It makes up for that deficiency by enabling the use of narrower pipeline as described in the previous section and thus leading to significant power savings.

Packet Latency Packet Latency is the number of clock cycles a packet needs until it is completely processed and has been sent to the output. It is identical for all packets in the conventional pipeline architecture as all packets have to go through all stages of all mini-pipelines. In this concrete implementation this would be 40 clock cycles (4 MPs time 10 stages per MP).

The registered architecture does not suffer from this handicap and it could output a packet directly after its processing was completed, granted the output module has the intelligence to monitor the processing progress of each packet and congestion due to concurrently completed packets is resolved. Furthermore, it could be programmed so, as to skip some stages if they were not required by the current protocol, thus further reducing the latency. Assuming a 7 stage pipeline like the one used in the NPU100 (as introduced in section 3.2.1) and MPLS packet whose label has to be popped, we can determine that in this case the label would only need to pass through the forwarding and QoS lookup modules, as no additional operations need to be performed. This means that this label can

exit the registered pipeline architecture after passing through these two pipeline stages, with a total latency of 5 clock cycles, which is a reduction of 5 clock cycles over the total latency of going through all pipeline stages as is mandatory in the traditional pipeline architecture. If a label needs to be swapped, then the packet needs to go through both lookup modules as well as through the replace and one of the modify ones, which would increase the latency to 7 cycles and reduce the savings to 3.

The folded architecture falls somewhere between the two other architectures, as it has a latency quantized to that of a mini-pipeline. This is so, because a packet can exit the NPU when the processing in the current mini-pipeline is done. In contrast to the registered architecture, however, in the folded one an MP always has to be traversed in its entirety. Thus, an MPLS packet that only needs to have its top label swapped, as described in the previous paragraph, will have a latency of 10 cycles in the folded architecture, in comparison to 7 cycles in the registered one. The packet will then be able to leave the pipeline however, whereas in the traditional architecture it would have to go through all 4 MPs for a total latency of 40 clock cycles.

Flexibility Flexibility is also an important parameter to be considered, which however is difficult to quantify. It refers to the capability of an architecture to adapt itself to future requirements. In this respect, the conventional pipeline is the most rigid one. The number of MPs and stages are fixed, as well as the sequence in which they are traversed. Hence the architecture's adaptability is relatively small, as it can process at most as many labels as the number of mini-pipelines it includes.

The folded architecture on the other hand would be able to process any number of labels in a packet, while being limited by the availability of processing resources and of memory to store the labels in the label buffers. However, even if more space was available in the buffers, the finite number of processing resources available might lead to performance degradation, since packets would have to be queued in the dispatcher until a mini-pipeline becomes available. Furthermore, the order of the processing stages is still fixed, although now there is an early exit option after each MP.

The registered architecture is the clear winner in this category since the stages to be used for processing, the order in which this is to take place and how many times each will be used, can all be changed by programming the switch matrix control appropriately. There are, however, still the limitations found in the folded architecture, regarding availability of register storage space for packet headers and processing elements for performing operations on them, which have to be taken into account in any such design.

Summary To summarize the findings of the previous section, table 3.3 provides an overview of the results of the qualitative investigation. This helps identify some trends regarding a possible winner among the three contestants and provides some pointers about the issues that need to be investigated further during the course of the quantitative comparison and the implementation of the system.

The qualitative analysis provides clear indications on the benefits of both the folded and the registered architectures, but also leaves several areas with a lot of unresolved questions. These mostly focus in obtaining more data about the additional modules required by the

Tab. 3.3: Summary of the Qualitative Analysis's Findings

	Conventional	Registered	Folded
Datapath Width	256 bits	256 bits	192 bits
Register Power Consumption	X	3-11% of X	17-62% of X
Resource Consumption	Wide pipeline registers, no additional requirements	Wide pipeline registers, additional modules of unknown sizes required	Narrow pipeline register, Considerable additional modules required
Packet Latency	Fixed at 4 x no. of MPs	Minimal	Quantized to the no. of MPs
Flexibility	None	Extensive	Good

registered and folded architectures and of determining if these modules induce a very high implementation penalty, high enough to render the respective architecture impossible to implement with contemporary IC technologies. In this regard, the registered architecture switch matrix remains an unknown factor, as no evaluation of its resource requirements and power consumption can be made without implementing it. This can negate the clear advantages that the registered architecture exhibits in all areas. These questions can only be answered unequivocally by a comparison of the implementation of the three architectures, which will provide additional data regarding the power consumption and complexity of the each concept.

Quantitative Results

The quantitative analysis attempts to rectify some of the deficiencies of the qualitative analysis that preceded it. There, the clear edge for the registered architecture in most investigated parameters was shown. Caution was however dictated for two reasons: The complexity of a switch matrix of the order required by the number of PEs in the design could very well prove impossible to implement with contemporary IC technology and even if this was not the case, the power consumption of the switch matrix might negate any advantages.

To determine the validity of this suspicion, an implementation of the registered architecture for different numbers of PEs was performed in an effort to disprove both of these possibilities. Unfortunately, this was to no avail, as the suspicions were confirmed and the registered architecture proved overly complex to implement. Thus the remaining two architectures, the conventional and the folded one, were implemented and compared in order to test the hinted superiority of the folded pipeline concept.

Registered Architecture The registered architecture held great promise on reducing power consumption, since it aspired to minimize the number of bits that are sent to the processing modules and back, thus reducing the number of bits that toggle in every clock cycle. It was however identified from the very beginning that the complicated interconnect matrix required to interconnect the PEs with the registers holding the packet headers could

very well be the Achilles' heel of this concept. To this end we implemented an interim version of this concept as a stopgap approach to evaluate its feasibility. Figure 3.11 shows how it was implemented. The PEs shown are indicative and were multiplied according to the needs of each test implementation. It uses the same modules as the implementation of the folded and conventional pipelines described subsequently in this section, but rearranged so as to form the new architecture. Hence, the registers normally found in between the MP stages are now located in a register file, in which a new header is written in each clock cycle. From this register file data is selected by a module called the Input Selection Unit (ISU) and sent over the interconnect matrix to the required PE. The ISU is a programmable module which ensures that the processing modules have access to the appropriate packet header register bits. A similar module, the Output Selection Unit (OSU) performs the opposite task, and merges the results of each PE back with the original packet header data. Each ISU and OSU is programmed using a microcommand table, which directs the bit selection process. After processing the results are sent back to the register where they are placed in the right location by an OSU. A multiplexer between each OSU and register selects between the output of the OSU and a new packet header coming from the input. The interconnect control module is the brain, which directs the flow of packet header data to each PE and back. The approach used here is one of configurable assignment, which means that the user can configure the order in which packet header data are sent to the PEs and this can be changed over the configuration interface of the NPU. The assignment is, however, identical for all packets.

One deviation in the functionality between the registered architecture implementation on the one hand and those of the conventional and folded architecture implementations on the other is in the programming flexibility provided by the microcommand tables. In the folded and conventional architectures, each stage has it own microcommand table, whereas here each register has its own microcommand table. This means that in the former case, each stage can have up to a specific number of operations that are performed (this number is eight in our implementation, since there are eight entries in the table), and that this specific number of operations is common to all packets (since all packets flowing through this stage use this microcommand table), whereas in the latter case each packet header has a number of specific operations equal to the number of entries in the ISU/OSU microcommand table for its entire lifetime in the NPU. Thus all operations to be performed on this packet have to be controlled by the entries contained in that microcommand table and this could lead to the need for a table containing more entries to accommodate for all possible processing combinations. The up side in this case however is that each packet header can have a completely different set of operations performed on it. In praxis, the benefit of this is questionable since in core network large number of packets belonging to one protocol are processed, thus this kind of additional flexibility would be of no avail.

In order to determine if the concept has merit, we implemented the architecture for an ever increasing number of stages. A generic-based VHDL implementation was used for this purpose, so as to allow for a change in stage number, with minimal alterations in the code itself. Figures 3.12a and 3.12b show the results of the implementation with regards to device resource utilization and achieved synthesis frequency. While the latter experiences a respectable but tolerable drop, the real show stopper is the resource consumption, which

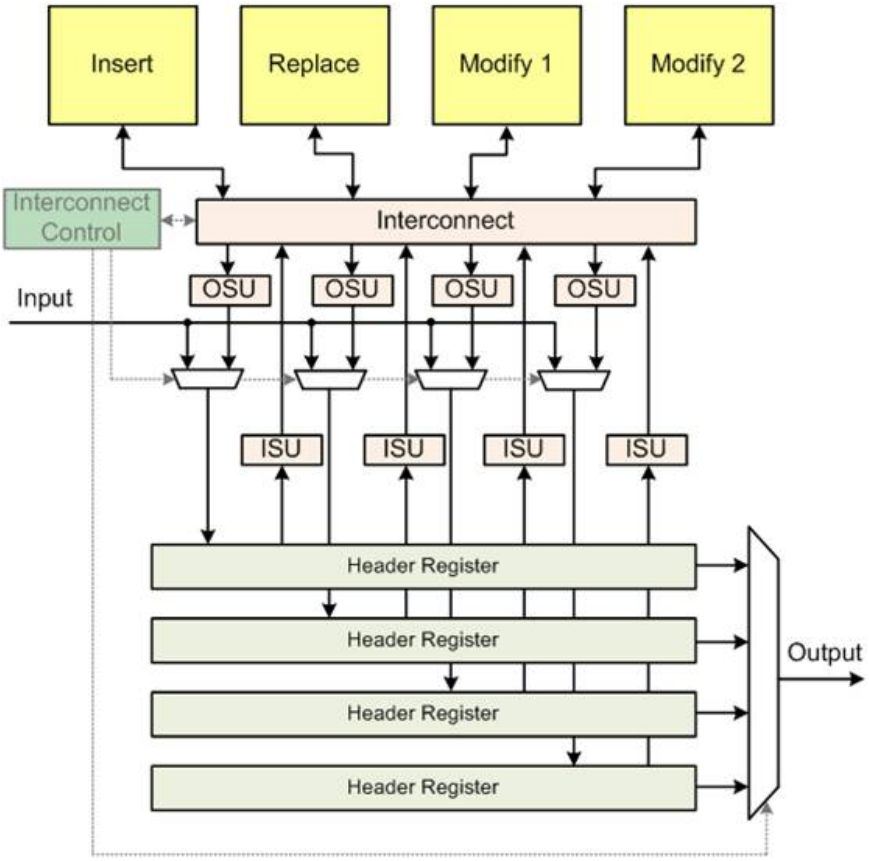
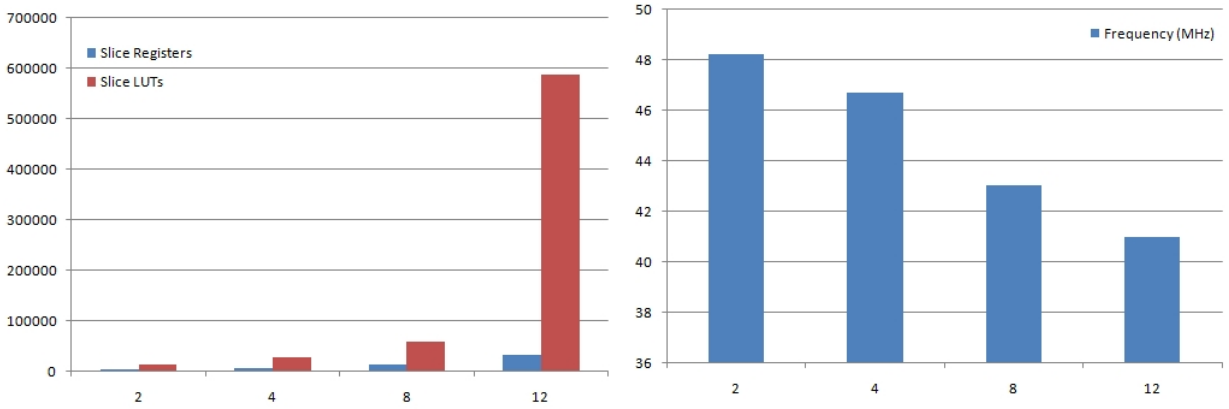


Fig. 3.11: Implementation of a Registered Architecture Prototype

skyrockets as the number of PEs increases. Keeping in mind that the Virtex 5 LX330T (the largest in the Virtex 5 families), for which these results where synthesized, offers 207360 slice registers and an equal number of LUTs, we can see that a variant with up to 8 stages is the maximum that could be accommodated in this device, and this at a significant resource cost. Since the complete NPU100 with all four MPs consists of in excess of 40 pipeline stages, it is easy to see that this architecture is not a realistic choice for a 100 Gbit/s CGE design. This is not meant to invalidate the whole concept, as an implementation using a small number of processing stages and registers is feasible, but makes clear that it is precocious at best, given the current state of semiconductor technology. An alternative use of this concept, that was not further examined within the confines of this thesis, would be to utilize this architecture for only a small part of each MP (say, for the four processing modules - insert, replace and the two modify modules), so as to provide some extra flexibility, as in this case it would be possible to change the order in which these operations are performed.

Conventional and Folded Architectures Comparison This section compares the implementations of the folded and conventional architectures in order to verify the advantages bestowed upon the former in the qualitative analysis. It compares the three main performance characteristics of any circuit, resource consumption, performance and power dissipation, starting with the latter as it has been touted to be the area where the folded ar-



(a) Resource consumption of registered architecture for various PE numbers

(b) Frequency of the registered architecture for various PE numbers

Fig. 3.12: Registered Architecture Resource Consumption and Frequency for Various PE Numbers

chitecture exhibits the greater advantage. This section only compares the processing cores of the two architectures, meaning modules like the Header-Payload Split and the Packet Reassembly are excluded, as the focus here is in comparing the architectures, which consist of the two processing cores and not entire systems based on these architectures.

Power Consumption Estimation Evaluating the power consumption of a circuit before actually implementing it is a difficult problem. To this end, the Xilinx XPower tool [**?**], which allows the estimation of power consumption on a FPGA, was used. Its results are directly applicable only to FPGA-based systems, but they are also applicable to digital circuits in general, as the basic building blocks are to some extent common. It uses place and route data of the implemented design as well as design constraints and considers various parameters such as the switching activity in the circuit. The tool can be fed with specific operational scenarios in order to extract accurate results.

In order to get a fairly accurate estimation of the power consumption of the two architectures, information regarding the activity of each module had to first be provided to the tool. This data was determined by classifying the modules into 4 levels of switching activity with each level having it's own switching factor. In order to set the proper value for each module, the resources used by each was analyzed. An overview of these for the entire NPU100 testbed is provided in table 5.1, which also includes the modules relevant in this part of the study. Then a qualitative estimation of the activity of each module was performed. This was based on the work performed by each module.

Table 3.4 provides detailed data about this classification, as well as about the number of flip flops (FFs) contained in each module of the NP, which also influences the switch factor. Figure 3.13 provides an overview of the folded architecture pipeline modeled to facilitate comprehension of the modules listed in the table. Internal module pipeline register and configuration LUTs are not shown on the figure. Modules not listed in the table include no FFs. The ones deemed to have minimal switching activity have the switching factor set to 0. These are FFs which hold configuration information which are expected to change value

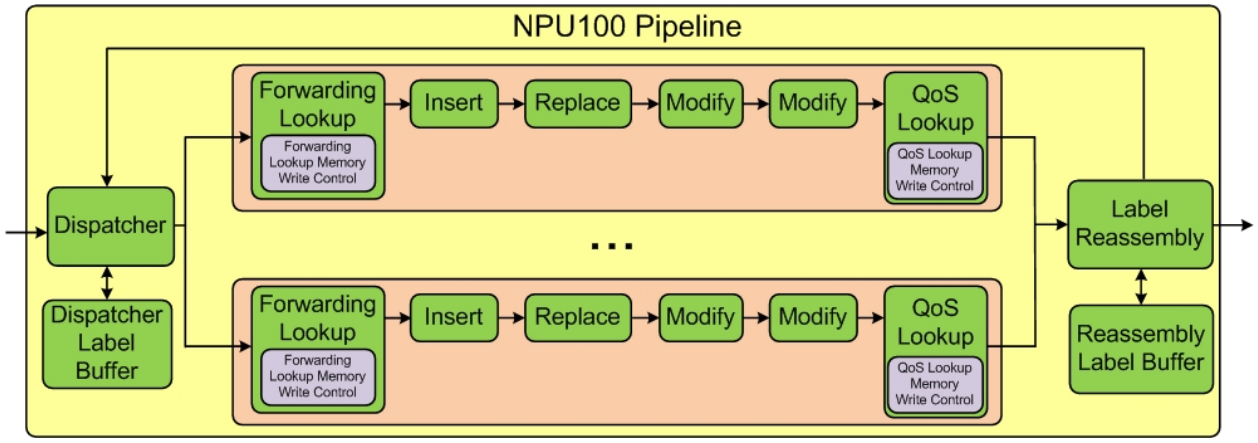


Fig. 3.13: Folded Pipeline Architecture Modelled in xPower

only when the NP gets reprogrammed, something should only happen rarely. Thus for all intents and purposes this toggling rate is considered to be infinitesimal and is ignored. Small switching activity (factor 0.03 on the table) is attributed to those FFs which are rewritten every several tens of cycles, like the ones in the dispatcher and label reassembly buffer, which are rewritten once NxM clock cycles. Medium switching (factor 0.25 on the table) activity is assigned to FFs which toggle every small number of cycles. These can be internal registers of the dispatcher and label reassembly modules which buffer data coming to and from the mini-pipelines, the buffers and the input or output. Finally FFs which toggle with every clock cycle are classified as high switching activity ones. We assume that in every clock cycle in which a FF is rewritten there is a 50% probability that its value is going to change. Using this assumption we calculate the number of FFs which toggle per module and cycle, which we then use to derive the switching factor for the scenarios listed in table 3.5. The table also includes the values used in the XPower tool in each scenario and provides the results for the power consumption for the two architectures, plus the isolated power consumption for a single mini pipeline in both architectures,are presented.

Comparing the results from the XPower tool with our theoretical analysis in section 3.2.2 we can see that for a single pipeline the folded architecture shows an 12% reduction in dynamic power consumption. In scenario VIII of section 3.2.2, which considers all the minipipelines stressed to the maximum and thus it is directly comparable with the results derived here from the tool, the total calculated savings were 37.5% for all minipipelines, thus approximately 9.4% for each minipipeline, which is quite close to the XPower value, if one considers that our analysis only took into account the registers, while the XPower considers all the modules, including the wider ISUs and OSUs in the conventional architecture. Thus, the reduced power consumption of the folded architecture is verified. Both the single mini pipeline and the complete folded architecture demonstrate significantly reduced power consumption in comparison to their conventional counterparts. One has to note, that the advantage of the complete folded pipeline is dimished in comparison to that of a single MP. This is reasonable since the complete folded pipeline core also include the Dispatcher and Label Reassembly modules which contribute to the power consumption respectively.

Tab. 3.4: Estimation of the Activity Factor for the NPU100 Modules

Module	No. of modules	No. of FFs per module	Activity Estimation	Toggles per cycle
Forwarding Lookup Memory Write Control	1	20075	0	0
Forwarding Lookup Module	4	1983	0.25	496
QoS Lookup Memory Write Control	4	651	0	0
QoS Lookup Module	4	690	0	0
Modify Module	8	1376	0	0
Replace Module	4	552	0	0
Dispatcher	1	4649	0.25	1193
Dispatcher Label Buffer	1	2058	0.02	187
Label Reassembly	1	1320	0.25	336
Label Reassembly Label Buffer	1	5740	0.02	346
Configuration LUT	28	640	0	0
Conventional Architecture Pipeline Registers	40	256	0.5	5120
Folded Architecture Pipeline Registers	40	192	0.5	3840

Tab. 3.5: Power Consumption Data for Conventional and Folded Pipeline Architectures

	Switching Factor	Static Power (W)	Dynamic Power (W)	Total Power (W)
Conventional Architecture	0.10	9.32	17.67	26.99
Folded Architecture	0.08	8.98	14.63	23.61
Conventional Mini Pipeline	0.08	4.98	5.36	10.34
Folded Mini Pipeline	0.07	3.76	4.13	7.89

Resource Consumption Comparing the resource usage of the two architectures is interesting, since a deeper investigation reveals additional information which is not evident during a superficial first look. To this end, table 3.6 provides synthesis results for the conventional and folded architecture and MPs respectively. Furthermore, the Dispatcher and Label Reassembly modules are shown separately from their respective buffers so as to be able to decouple the impact of each on the total circuit size. For the synthesis of the design the Xilinx ISE v13.1 XST [**?**] synthesizer and the Virtex 5 LX330T were used. This FPGA offers 207360 slice LUTs and an equal number of registers.

On a high level, the folded architecture shows somewhat higher slice register consumption (by approximately 14000 registers), but at the same time reduces slice LUT usage by about 34000. Both of these variations are explainable. The increase in slice registers is a result of the extra modules demanded by the folded architectures and most importantly of the two label buffers. Because of the high number of required concurrent inputs and outputs, these buffers were implemented as register matrices and thus they consume slice

Tab. 3.6: Resource Consumption for Conventional and Folded Pipeline Architectures

	No. of Slice Registers	No. of Slice LUTs
Conventional Architecture	40312	174119
Conventional Mini Pipeline	10104	43991
Folded Architecture	54800	140018
Folded Architecture Mini Pipeline	7732	31983
Dispatcher/Label Reassembly (w/o Buffer)	5969	4366
Dispatcher/Label Reassembly Buffers	16298	4291

registers. Instantiating and using Block RAMs (BRAMs) was not advisable as they are also in short supply, due to them being used as lookup memories. Furthermore, instantiating BRAMs appropriately to fulfill the requirements would lead to a very complicated design, and very probably one that would waste precious BRAM capacity (due to BRAM aggregation in order to get the appropriate cell width and matrix organization right). In an ASIC implementation a custom memory with the appropriate characteristics would solve this problem. Even so, one might have expected the number of registers to be roughly equal, based on the notion, that the folded architecture reduces the register width in the pipeline and thus saves on registers there, but uses up these registers in the label buffers. This explanation does not take into account the fact that the Label Reassembly Buffer contains the necessary space not only for the labels, but also for the internal header and part of the processing context, as described in 4.2.3. This more than compensates the registers saved in the pipeline. The exact number of registers required in the two buffers is calculated by equations (3.6) and (3.7), which are essentially identical with the equations provided in the qualitative evaluation ((3.4) and (3.5) respectively), with the addition of extra stages for the Dispatcher and Label Reassembly modules (marked DispStages and LRStages in the equations respectively.

$$(NoOfMPs-1)\times(StagesPerMP+DispStages+LRStages)\times DispCellWidth = 4320 \tag{3.6}$$

$$NoOfMPs\times(StagesPerMP+DispStages+LRStages)\times LRCellWidth = 12000 \tag{3.7}$$

Despite the increased slice register usage, slice LUT consumption in the folded architecture is significantly reduced, as a result of the simpler ISUs and OSUs needed to cope with the narrower pipeline. It must be stressed here that the difference is not just in numbers. It might be that significantly more slice LUTs are saved than slice registers are consumed, but one must keep in mind that a LUT table is a much more complex structure than a simple register. In fact, the Virtex 5 FPGA architecture uses 6-input LUTs, whose implementation, although not publicly, available is bound to be more resource consuming than

a single register. This further accentuates the resources saved in the folded architecture.

The number of registers consumed by the Dispatcher and Label Reassembly buffers confirm the calculations performed in equations 3.4 and 3.5. Furthermore it shows that both modules together with their label buffers consume less than 8% of the resources of a modern FPGA device. In general, the total resource consumption of the folded architecture equals the sum of four folded architecture minipipelines plus the Dispatcher and Label Reassembly modules and their buffers. Similarly, the total resource consumption of the conventional architecture equals that of four single mini pipelines.

Circuit Performance Higher performance was never one of the claimed benefits of the folded pipeline architecture. Quite the opposite, one of the most important concerns, during the design of the the system was whether some module would pose insurmountable problems in the form of a performance bottleneck. This was refuted, since the additional modules of the folded architecture easily overcame the set target frequencies.

The upper bound of the frequency limitation in both the folded and the conventional pipeline designs stems from the long critical path in the Modify module, which is formed from the ISU, PE and OSU module chain. Section 5.1.3 provides more details on this, as the current section's main goal is to compare the two architectures. Despite the fact that the critical path is identical, the folded architecture exhibits slightly higher performance in comparison to the conventional one (approximately 112 to 97 MHz respectively).

Tab. 3.7: Impact of the Pipeline Width on the ISU/OSU Latency

	192 bits	256 bits
ISU	2.151 ns	2.661 ns
OSU	5.477 ns	5.945 ns

The underlying cause is the simplification of the ISU and OSU modules due to the smaller pipeline word width. To prove the validity of this assumption, latency data for both modules is provided with a pipeline word width of 192 and 256. The results, shown on table 3.7 make evident, that increasing the width has an adverse effect on the latency of the system. Thus the folded pipeline architecture provides a small and secondary, albeit tangible, benefit in the performance area as well.

3.3 A Distributed Programming Paradigm for Pipelined Network Processors

One of the more important pitfalls, which had to be avoided in the NPU100 design was ending up with a powerful but difficult to use processor, a calamity which has been constantly present in the NP space. To this end, the development of the programming paradigm for the NPU100 was integrated into the design process from the very start and was not squeezed into it belatedly after the bulk of the design work had already been accomplished. This section introduces this programming paradigm and the advantages it offers.

The objective of the NPU100 programming concept was simple: Provide a comprehensive, easy-to-use programming procedure in order to allow the user to program the NPU100 with minimum overhead. Overhead in this case consists of two distinct areas. First of all the user must not need to know about the internal workings of the device and the programming interface and second he must not need to learn new complex programming languages in order to write the required code. Thus, the aspiration was to create a system that can be programmed with a high-level language by writing simple code using familiar syntax and furthermore one that then takes this code and writes it into the NPU100 without the need for further programmer intervention.

Accomplishing the above stated goals clearly meant that the system would have to comprise a hardware and a software side, which will have to work harmoniously to ensure the programmability of the processor. Figure 3.14 illustrates the concept behind the programming mechanism.

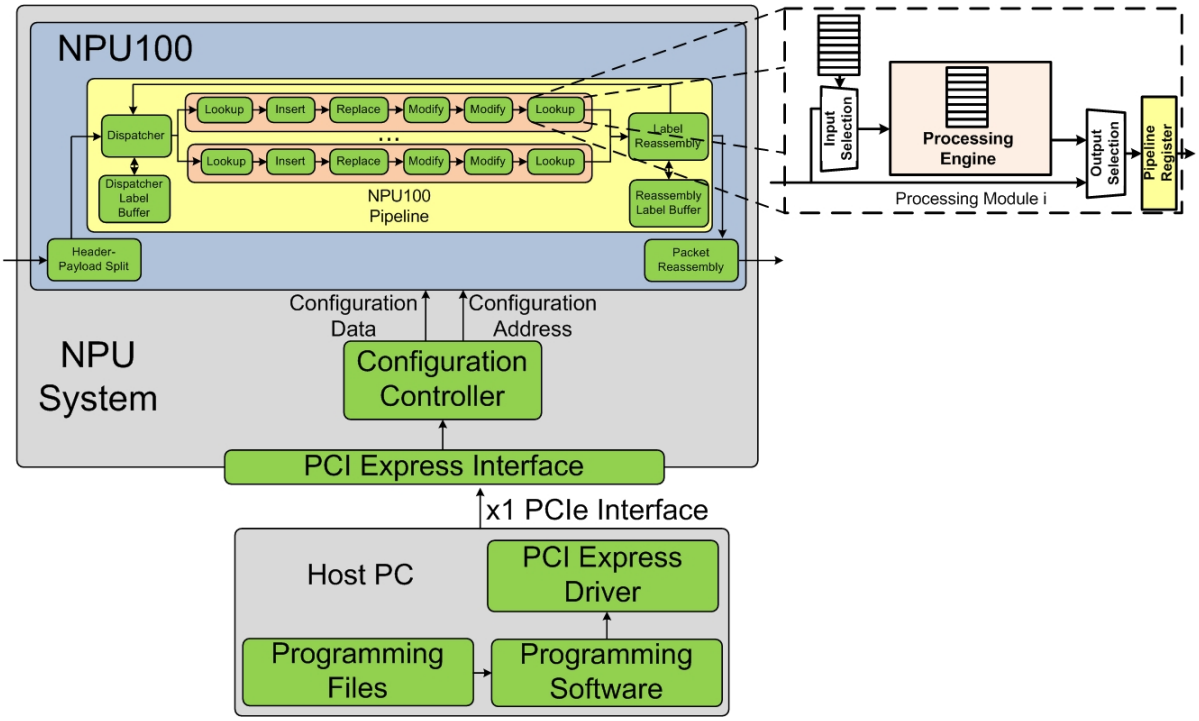


Fig. 3.14: Block Diagram of the NPU100 Configuration Architecture

The hardware side consists of the NPU100 itself, in which programming is implemented through microcommand tables located in each pipeline stage as shown in the excerpt on the right. In each processing module an Input Selection Unit (ISU) and an Output Selection Unit (OSU) are used to extract fields from the pipeline word (done by the ISU) and weave the results back into the pipeline word before the next stage pipeline register (the job of the OSU). There is one common microcommand table for the ISU/OSU modules, which determines which fields are extracted and send to the pipeline word and which not, as well as where the results are placed in the pipeline word. Depending on the processing module (not all processing modules require one), a separate microcommand table may be found in it. This table directs the processing performed in the PE and contains the operations

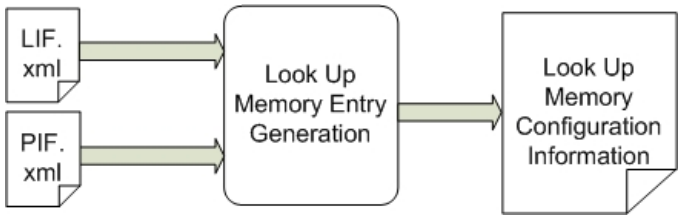


Fig. 3.15: Forwarding and QoS Table Entry Generation Process

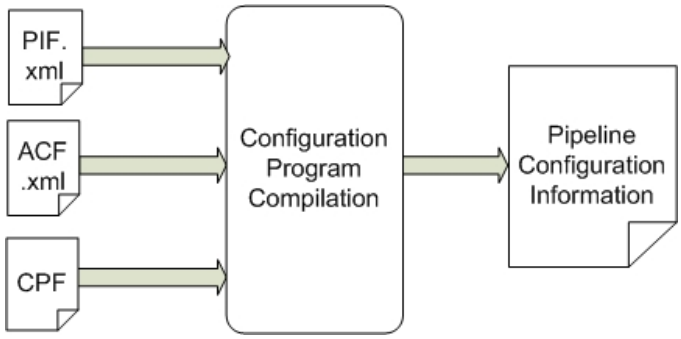


Fig. 3.16: Microcommand Programming Information Generation Process

to be performed and the operands to be used in them. This keeps the commands to be executed in close proximity to the modules, which execute them. This reduces the path that the commands have to traverse each time, with the accompanying advantages in circuit complexity and power dissipation reduction. Furthermore, it eliminates any congestions caused by multiple processing cores accessing a common instruction memory at once, a problem common in multi-core designs.

The software side consists of a compiler and a lookup entry generator, which read several input data files and produce the necessary programming files. Figures 3.15 and 3.16 illustrate how this is performed. There are two separate strands. The first shown on figure 3.15 generates the data for the forwarding and QoS lookup tables of the NPU100. A lookup entry generator creates a binary file which specifies which entries have to added to or erased from the two lookup memories. The second (figure 3.16) generates the programming file that is to be sent to the NPU100 over the PCIe and which determines the functionality of each processing module. This file is produced by the compiler, which interprets a file containing the program to be executed on the NPU100 described in a high level language and automatically identifies the required processing modules, as well as performs the necessary task assignment, before creating the binary file containing the microcomands.

In order to accomplish these tasks, the compiler and the lookup entry generator use four different files as input. Three of those are XML files:

- Architecture Configuration File (ACF): Provides information about the current NPU100 implementation (number of available MPs and modules in each MP). The compiler uses this information to perform the resource allocation.

- Protocol Information File (PIF): Defines the various protocol fields (e.g. label, TTL, Cos in MPLS-TP) and their location in the pipeline word and their length.

- lookup Information File (LIF): Specifies the entry data for the forwarding and QoS lookup tables.

The fourth and final file used is the Configuration Program File (CPF), which contains a mix of compiler directives and C-like language constructs used to define the functionality of the NPU100 modules in an easy-to-understand, algorithmic manner. How the two programs parse the input file and distill from them the information to sent to the NPU100 is explained in detail in section 4.3.3.

The programming language used by the NPU100 uses a C-like syntax, with which most engineers are already familiar. Developing a program for the NPU100 consists of defining values for a set of constants, which are then used to set various parameters of the design. Two examples of this are shown below, one setting the value for the label size to be processed in the NPU and the other for the number of header bits that the Header-Payload Split module must then copy and forward to the pipeline core. The keyword #parameters tells the compiler that the following definitions refer to NPU100 programming parameters. The names the user has to use (labelSize and headerSize in the example below) when defining the values are fixed and the values are constrained between values that the architecture supports (16-64 bit label size and up to 256 bits header size in the example). The compiler reads the values of all parameters and generates the appropriate programming words. Some configuration words are only relevant for one module (e.g. headerSize only refers to the Header-Payload Split), whereas others (e.g. labelSize) are used by multiple modules within the NPU100.

```
#parameters
#define labelSize 32
#define headerSize 192
```

The main program then follows demarcated by the #program directive. The command set allowed is fairly restricted. This is because the NPU100 is not a complete software-programmable CPU, but a network processor with weakly programmable processing elements. Thus the functionality and the flexibility of those elements is limited in comparison to a general-purpose CPU, but is still sufficient to process packets within a core network.

Thus the NPU100 compiler supports variable assignment statements, where the variables are aliases which must have previously been defined in the PIF. The signal assignment statements can be placed in conditional statements (either *if-else* or *switch* statements) or *for* loops. The compiler allows for one level of nesting, granted that heterogeneous statements are nested. This means that a for loop nested in an if-else or switch statement and an if-else or switch statement in a for loop are both permissible, but not a loop statement nested within another loop statement or any two conditional statements nested in each other. The compiler reads through the statements and tries to optimally match the statements in the file to the resources specified in the ACF. This is a complicated process, which depends on the conditions and iterations, which influence the execution of each statement as well as the availability of the PEs in the NPU100 implementation for which the CPF is being compiled. This process, the result of which is a programming file with all the necessary commands, is described in detail in section 4.3.3.

Binding the software and hardware sides together is a PCI Express link. The binary

files resulting from the compiler and the lookup table entry generator are forwarded over this link to the NPU100. The software side utilizes a PCI Express driver to interface with the the PCI Express link. The link on the hardware side includes a Xilinx PCI Express Endpoint block core which receives the programming data and stores them temporarily into a buffer and a programming controller which then forwards the programming words to the appropriate modules. The communication between the programming controller and the various NPU100 modules is performed through an address and a data bus. When the correct address is put on the former, the respective module stores the data in the latter in the appropriate location.

Summarizing the above, the NPU100 programming paradigm improves current know how by combining a streamlined approach on the hardware side and thus minimizing complexity, power dissipation and congestions, while simplifying code writing by allowing the programmer to input high level code, which is then automatically partitioned and assigned to the processing modules, thus making life for the prospective NPU100 user that much easier.

4 NPU100 System Implementation

This chapter provides a detailed description of the implementation of the various NPU100 modules. This includes the entire NPU100, with its outer layer, the pipeline core and the programming subsystem. Implementation results are generally provided in chapter 5.1, except in cases where a specific design decision has to be undergirded by them.

The first section (4.1) introduces the outer layer modules, namely the Header-Payload Split and the Packet Reassembly. Then the focus moves into to the processing core (section 4.2) and first to the Dispatcher and Label Reassembly Modules and their buffers. The principle behind the efficient operation of these two modules is revealed and the way the looping functionality is realized in order to minimize the data path width and the number of loops each packet header has to perform is explained. Following this description, the implementation of each MP module is provided. Last but not least the software and hardware sides of the programming interface are described in section 4.3.

4.1 Outer Layer Implementation

This section provides an overview of the NPU100's outer layer. This includes the Header-Payload Split (4.1.2), which stores the packet in the buffer memory, while copying the header to the pipeline core, the buffer memory itself (4.1.4), which holds packets in temporary storage during processing, and the packet reassembly module (4.1.3), which rejoins the packet data from the buffer with the processing results and forwards the packet to the output.

4.1.1 Outer Layer Module overview

The dimensioning of the NPU100 and all the included modules was done according to assumptions met with Nokia Siemens Networks (NSN). This allowed the perspective of an equipment vendor to be brought in. Furthermore in the course of the NPU100 project discussions with other partners (e.g. Deutsche Telekom) also provided the point of view of a service provider. Consequently, the NPU100 was designed with the requirements of key players in the core network in mind.

One of the key parameters that needed to be set early in the design phase of the project was the number of MPLS tunnels that must be supported by an NPU100. As discussed in section 2.2.1, the MPLS stack can theoretically extend to an infinite number of levels, which means that if theoretically all tunnels ended at the same node, that node would have to process a infinite number of labels before being able to forward the packet. Since processing resources however are not and can never become infinite a middle ground has to be found, which bridges the gap between theoretical possibility and practical application.

To answer this question, one has to consider the possible MPLS scenarios, that can take place in a core network. The standard case is when a packet enters the network and is assigned to an LSP, which represents one tunneling layer. Two additional tunneling layers are required to provide backup paths in case the primary or the secondary path fail or encounters congestion. These act as safeguards, allowing the packets to be swiftly rerouted over the alternate path with minimal delay. Finally, a fourth tunneling layer is required to cater to the management packets, which in the worst case will be at the bottom of the stack. The aforementioned limitation extends to any other label based protocol.

This means for such protocols the design of the NPU100 can guarantee fault-free 100 Gbit/s performance, when up to 4 labels per packet must be processed. As will be shown, the architecture of the NPU100 is flexible enough so as to allow the processing of higher throughput when processing a smaller number of labels or of a larger number of labels at lower throughput.

Apart from its core functionality of processing packets in the pipeline core, the NPU100 has to receive information from an outside source (MAC module or switch I/F, depending on if the NPU is used as an ingress or egress NPU), to process the data and to make them available to the NPU100 pipeline core, where the packet processing takes place. Respectively, additional operations have to be performed on the data after the pipeline processing in order to send a packet to the switch I/F or the MAC module. These operations are handled by the outer layer of the NPU100 and are described in this section. Figure 4.1 illustrates the structure of the NPU100's outer layer. It consists of Interlaken interfaces for data input and output and serial to parallel and parallel to serial converters, which convert the data from the Interlaken format to the NPU100 one and vice versa. A Header-Payload Split module, which reads the parallelized data, separates the header section from the rest of the packet, forwards the former to the pipeline and stores the entire packet to the main memory, and a Packet Reassembly module, which reads the processed headers from the pipeline, fetches the payload from the main memory and recombines them before sending them to the output. The preceding section 3.1.1 provided a detailed explanation on the advantages this method offers.

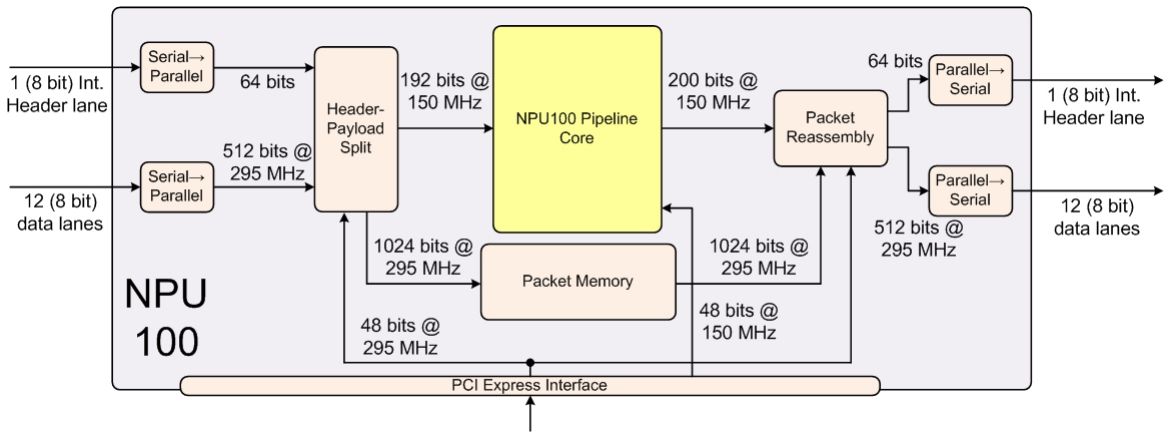


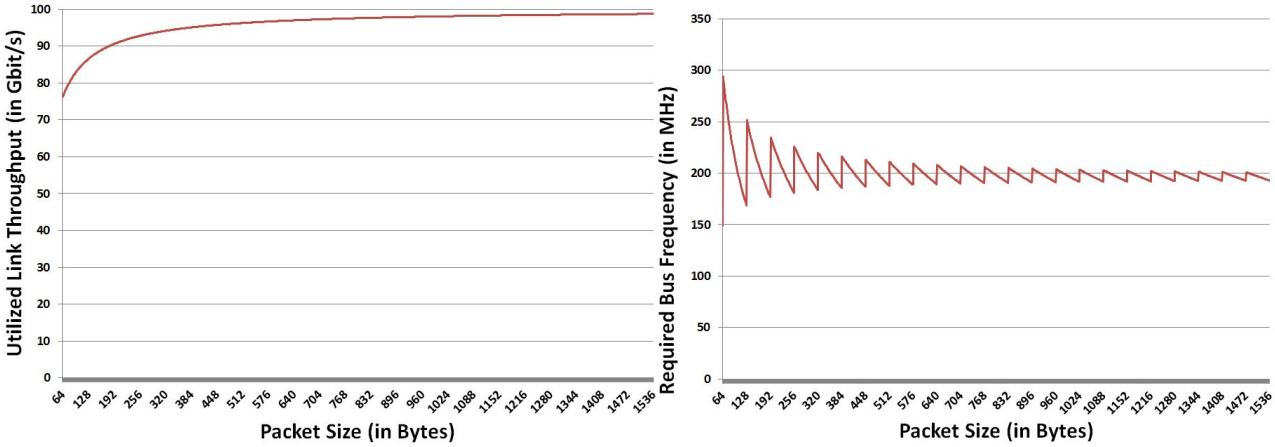
Fig. 4.1: Overview of the Architecture of the NPU100 Outer Layer with Bus Widths and Operating Frequencies

Data input and output is assumed to be done through Interlaken interfaces. These consist of 8 bit wide lanes which are used to transfer data at a high rate as described in section 2.7.2. To achieve the needed throughput of 100 Gbit/s with this 8 bit lane technology and given Interlaken working frequencies, 12 lanes working at 1.6525 GHz are required. The overprovisioning here is done to accommodate for the Interlaken protocol overhead. As described in section 2.7.2, the Interlaken protocols works in bursts, delimited by a control word. A maximum and minimum burst parameter determines the permissible length of each burst, with 32 bytes being a minimum and no inherent maximum. Assuming a worst case scenario of continuous 64 byte packets and that each packet uses an own burst, 16 bytes or 25% of overhead will be incurred for each packet. In the previous section we calculated the number of 64 byte packets possible to be transmitted through a 100 Gbit/s Ethernet link to 148.8 million, which means that excluding Ethernet protocol overhead (such as the interframe gap) but including the overhead incurred by Interlaken itself, the interlaken interface will need to transfer a total 95.2 Gbit/s, which requires 12 interlaken lanes to send through.

An additional Interlaken lane is used to transfer the internal header, which is used for intra-switch communication (and is described in section 3.1.2). Since this lane is required at either the NPU100 input (when the NPU is used in the egress path) or at the NPU100 output (when the NPU is used in the ingress path), it must be possible to deactivate the unneeded lane as appropriate. The data is then converted into a parallel form, which is better suited for processing in a contemporary integrated circuit. There are to be separate units which parallelize the 12 data lanes and the internal header lane (and vice versa for the output).

The buses that are used to transfer data between the various modules must be sized accordingly in order to accommodate for the high throughput required. The most straightforward solution for this would be to utilize a 512 bit wide bus at 200 MHz. However, an important issue arises in the handling of large streams of packets with a size marginally larger than an integer multiple of the 512 bit signal width (e.g. 65 byte packets, 129 byte packets and so on). In the 65 byte packet size case (which is the most extreme of all) two clock cycles would be needed to completely transfer the packet, but in the second cycle almost all of the signal transfer capacity would be wasted, and thus the NPU would not be able to achieve the targeted throughput. At 200 MHz, a 512 bit signal would be able to transfer a maximum of 102.4 Gbit/s in the optimal case of full, consecutive link utilization, however assuming a constant stream of 65 byte packets, the effective throughput would be 52 Gbit/s. To overcome this obstacle the frequency of the outer layer has to be increased to 295 MHz respectively. Calculating this is less than trivial, since we have to take into account the actual amount of bits that are received over the 100Gbit/s link. The maximum amount of utilized bits of the 100Gbit/s Ethernet link per packet size, as calculated using the equations of section 3.1.1, are illustrated in figure 4.2a. Important however is how many data words have to be transferred on a 512 bit bus in each case, taking into account the not fully utilized cycles per packet size. This is shown on figure 4.2b, whence we can extract that the maximum required frequency for the bus, in order to cover the worst case traffic load scenario, is 295 MHz. While this frequency is difficult to reach by modern FPGAs when implementing relatively complex circuits, ASICs can handle it with

ease. In the prototype described in chapter 5 the outer layer modules run at a frequency of 200 MHz, thus allowing the processing to attain its maximum processing throughput (performance is then limited by other modules in the design).



(a) Effective 100Gbit/s link utilization per packet size **(b)** Required frequency of the 512 bit bus in order to transfer the received effective bps rate per packet size

Fig. 4.2: Utilized Link Throughput and Required NPU100 Internal Bus Frequency per Packet Size

Thus, the Header Payload split receives data from the serial to parallel converters using a 512 bit bus operating at 295 MHz. At the same time a separate 64 bit signal transfers the internal header as needed. The internal header is transfered in the same cycle as the first packet word. The interface with the main memory has a width of 1024 bits and operates at a frequency of 295 MHz. The doubled bus width here is used to facilitate the packet reconstruction process at the packet reassembly. The exact reasoning behind this will be explained together with the implementation of this module in section 4.1.3. The data structure sent to the NPU pipeline is 192 bits wide and operates at a frequency of 150 MHz, the operating frequency of the pipeline core. The clock domain crossing is implemented by a FIFO queue inside the Header Payload Split (see section 4.1.2). The Packet Reassembly receives the processed header data from a 200 bit wide 150 MHz bus, performs the reverse clock domain crossing (again using a FIFO queue) and creates the processed packet by fetching the packet data from the buffer memory using another 1024 bit wide bus at 295 MHz and combining with the processed packet header as required. The output data along with the internal header is then sent to the parallel to serial converter, which interfaces with the Interlaken transceiver. Again packet data is sent via a 512 bit bus operating at 295 MHz and the internal header over a 64 bit bus in the first data transmission cycle. The converters output an identical number of lanes as the input ones to the Interlaken interface.

4.1.2 Header-Payload Split

The Header Payload Split is responsible for receiving packet data from the input interfaces, storing them in a temporary buffer memory for the duration of processing and copying a

part of the packet header, which it then forwards to the processing core. As figure 4.3 shows, it consists of the Header-Payload Split core modules, which is an FSM, that performs the previously described tasks, as well as an additional FIFO queue, the HP Split Buffer, which serves as a bridge between the outer layer and the pipeline core of the NPU100. The task of this FIFO queue is the synchronization between the two clock domains (295 MHz for the Outer Layer and 150 MHz for the processing core).

In order to determine the size of the queue we have to consider the worst case scenario, that is when the Header Payload Split feeds the FIFO queue with the maximum amount of packets. This is when only 64 byte packets arrive on the link, however, important in this case is not only the total number of usable bits arriving at the Header and Payload Split, but also the inter-arrival times between new packets. In order to calculate this, one has to take into account that every Ethernet packet has 20 additional bytes used for the interframe gap and the preamble. Practically this means that every 6.72 ns a new packet arrives and thus using a 295MHz bus, the Header-Payload Split data input will be empty every other cycle. Since the pipeline core operates at half the frequency of the outer layer, this means that the pipeline core will read the split packet header every other Header-Payload Split cycle. Thus, the FIFO queue only needs to be one entry in size in order to allow for the crossing between the two clock domains (from the outer layer to the pipeline core).

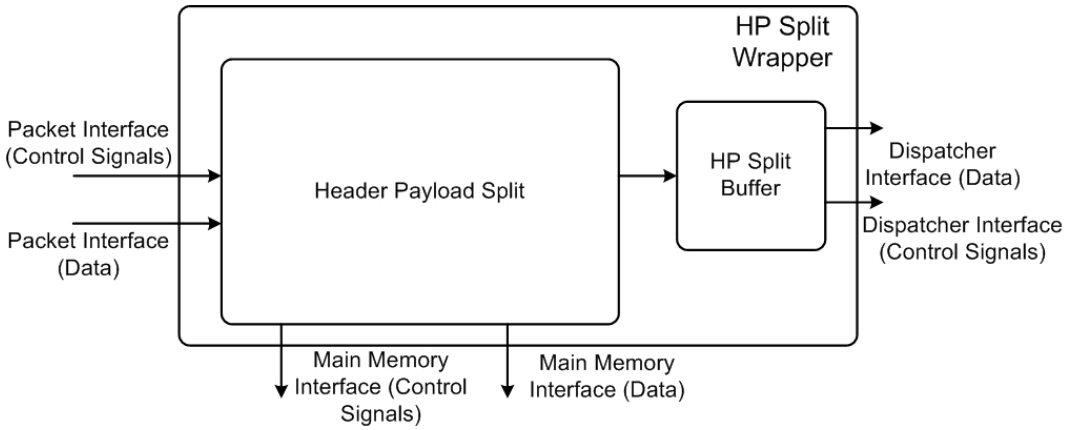


Fig. 4.3: Overview of the Header Payload Split Submodule

The Header-Payload Split must accomplish the following series of tasks:

- Read data from the Serial-to-Parallel Converters at every cycle in which new valid data is present. This is to be marked by a strobe signal, which is asserted one cycle before the first word of the packet data is available at the input.
- When the first word of a data packet is received, a part of this is replicated. The length of this part can be different per protocol and thus must be programmable. Values up to 256 bits are sufficient for current protocols. For both MPLS-TP and PBB-TE this value is to be set 128 Bits, enough to include 4 MPLS labels or the B-DA and B-VID fields of the PBB-TE header. This value does not affect the width of the pipeline, which is fixed at 192 bits as described in section 3.2.1, although how

many of those 192 bits are used to accommodate the packet header is up to user. The Dispatcher receives the replicated packet header data and then processes them accordingly, storing parts in the pipeline word and buffering others in the Dispatcher Label Buffer according to how it was programmed.

- If the NPU is an ingress NPU it must create the internal header and write the packet length and input port number fields. If it is an egress NPU the internal header will be received from the appropriate Converter as shown on figure 4.1. Since any NPU must be able to function as either this functionality must be configurable.

- When both packet header and internal header are ready, they are forwarded to the pipeline. A strobe signal is used to synchronize the process.

- Simultaneously the packet is being written to the main memory. If the NPU is an ingress NPU, the length field of the internal header must be filled, which means the entire packet must be received first in order for the packet length to be calculated. In this case, the H-P Split module can proceed with the storage of the packet data words in the main memory as they arrive in the module, to avoid having to buffer the entire packet in the H-P Split module.

To perform these operations, the module requires the following interfaces:

- Input from Data Serial to Parallel Converter. One 512 bit signal is needed, plus new packet indication signal (1 bit).

- Input from Internal Header Serial to Parallel Converter. One 64 bit signal plus any required control signals.

- Output to the Main Memory interface. One 1024 bit signal, plus any additional signals needed for addressing or controlling the memory (e.g. FIFO Full, etc signals since the main packet memory is implemented as a FIFO in this design).

- Output to the NPU100 Pipeline (Dispatcher). One 192 bit signal which includes the internal header (64 higher bits) and the packet header (128 bits), plus a strobe to notify the dispatcher that new data is available.

The Header Payload Split Module has been implemented in VHDL as an FSM. The challenge in the implementation of the modules was ensuring that no clock cycles are wasted during the process. This complicates the state transitions considerably since there are several alternative paths that need to be considered at once, in order to ensure the seamless reading and forwarding of information. In a typical case, a state has to check if the memory is full, then if this is the last word in the packet and if so if there will be a new packet available at the next cycle, conditions which all trigger different transitions. How this is reflected on the state machine implementing the Header-Payload Split module is shown on figure 4.4.

The module starts in Idle state and remains in it, as long as no new packets are to arrive. A new packet strobe signal is received notifying the FSM that a packet will arrive in the next clock cycle. This moves the state machine to state init, during which it buffers the first 512 bit data word of the packet along with other control information such as the packet length. It also stores or creates the internal header depending on the configuration of the NPU100 as egress or ingress NPU100 respectively. Subsequently, depending on if there are other data words for this packet pending, the system either goes to the Write Data state or back to the Idle or Init states. The latter choice depends on if there is a new packet arriving back to back with the current one in the next clock cycle. The new packet strobe signal is used to determined this. If it is set, the system moves to the Init state and starts reading a new packet if not it goes back to the Idle state and resumes its wait. In the Write Data state the FSM reads the next 512 bit data word from the input. Here it packs the two data words (the one read during the Init state) and the one read in this state together into one 1024 bit data word and writes it into the main memory. Then it either moves to the Write Data 2 state or to the Idle or to the Init state depending on the criteria described previously. From that point on the FSM enters a loop and changes from the Write Data 2 to the Write Data state as long as there are still packet data to read. In this loop a 1024 bit data word to be written to the memory is build using one 512 bit data word read during the Write Data 2 state and one read during the Write Data state, during which the 1024 bit word is written into the memory. Whenever the packet is finished the FSM moves to either the Idle or the Init states depending on if a packet is to be received in the next clock cycle, as described previously. If this happens during the Write Data 2 state, where only one 512 bit data word is available, then only this is written to the memory and the rest of the 1024 bits is padded with zeros.

4.1.3 Packet Reassembly

The Packet Reassembly module recombines the input packet data with the processing results forwarded to it by the Label Reassembly module. This seemingly simple task is in reality considerably more complex than the reverse procedure performed by the Header-Payload Split. The Packet Reassembly must be able to perform the following actions:

- Receive processed packet headers from the Label Reassembly module. The Label Reassembly unit produces a strobe when a packet data structure is available.
- When the data structure is fetched, the packet header must be reassembled. To this end the original packet data from the main memory must be available to the Packet Reassembly module.
- Reassembling the packet header is a completely different process in the MPLS-TP and PBB-TE protocols (and respective similar protocols).
  - MPLS-TP: The 8 least significant bits of the Label Reassembly output must be used to decode the sequence of operations done in the pipeline. Table 4.1 in the Label Ressembly description section (4.2.2) provides an explanation for these bits. By using these bits the module must determine which labels must

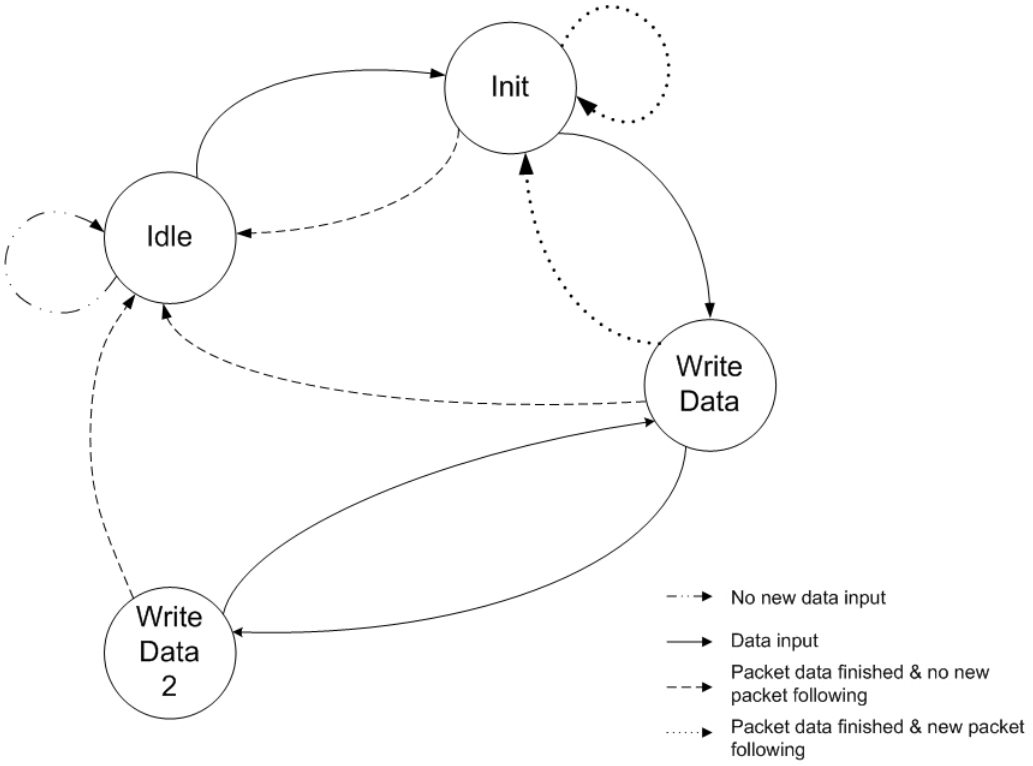


Fig. 4.4: State Diagram of the Header-Payload Split Module. Solid Lines Show Transitions During Data Input, Dashed Lines Transitions when a Packet is Finished and no New Packet is to Start in the Following Clock Cycle and Dotted Lines transitions when a Packet is Finished and is Trailed by a New Packet in the Next Clock Cycle.

be retained from the ones in the main memory copy of the packet and which should be replaced by the ones in the pipeline word.

- PBB-TE: Since nothing in the packet header needs to be changed, only the internal header needs to be prepended to the packet data from the main memory if this NPU is an ingress NPU or the internal header needs to be removed in the opposite case.

- Then the module can begin transmission to the output. The internal header and the data are sent over different signals.

- The internal header is only transmitted to the output for ingress NPUs. In egress NPUs the Packet Reassembly only sends the packet data onwards and strips off the internal header.

- When the last word of a packet is transmitted to the output, any gap at the end of this word must be filled with zeros.

To accomplish the above tasks the module requires a multitude of interfaces to receive and send information:

- A 200 bit interface with the Label Reassembly unit from which the completely processed packet data words are received. Appropriate control signaling must be used, so that the Packet Reassembly can detect in which clock cycle a valid packet header is available.
- a 1024 bit interface with the main memory from which packet data are read. Any additional signals for addressing and control (e.g. a FIFO empty signal, etc) are to implemented as needed.
- 64 bit output for the internal header.
- 512 bit output for the packet data. Must be accompanied by a 6 bit signal denoting the bits holding valid data, as well as by a signal marking the start of a new packet.

The system designed to meet these requirements is shown in figure 4.5. It consists of three submodules. The Packet Reassembly Buffer is a FIFO queue, whose main function is to perform the bridging between the two frequency domains. Thus the Label Reassembly writes to the FIFO using the pipeline core frequency and the Packet Reassembly reads from it using the outer layer frequency. As in the Header-Payload Split, the FIFO does not have to buffer multiple packet headers and thus its size is the minimum possible. This is because, adherence to the Main Memory FIFO (described in section 4.1.4) principle means that packet headers arrive from the Label Reassembly in the order the arrive at the Dispatcher. This allows for any packet bursts to be smoothed out in the Label Reassembly Buffer.

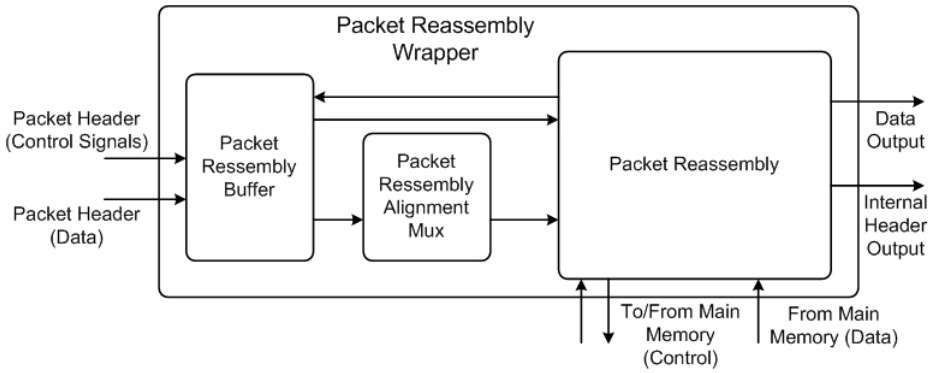


Fig. 4.5: Packet Reassembly Submodule Block Diagram

In between the Packet Reassembly buffer and the Packet Reassembly module, an alignment multiplexer is used to eliminate any gaps in the data word, before it is passed on to the Packet Reassembly. This is necessary because the data word, which the Label Reassembly outputs, consists of placeholders for a specified (albeit configurable through generics) maximum label size. The NP however allows for processing labels of different sizes, with a simple reprogramming of the processing modules. As a result of this it is possible and likely, that the maximum label size and the actual label size will differ (e.g. up to 64 bit labels will be supported, but 32 bit labels will currently be processed by the NPU100). Consequently, the pipeline data path will be sized for the maximum label size, which will

result in gaps in the pipeline word, from which they have to be stripped before the Packet Reassembly can rejoin the original packet data and the processing results. Figure 4.6 illustrates how this is performed on an example using the label sizes previously mentioned. The data word coming from the Label Reassembly consists of the internal header, one 64 bit placeholders for each label (64 bits is the maximum supported label size) and 8 bits (2 for each operation conducted in each MP) from the processing context, which are necessary for packet reconstruction (marked PC on the figure). The alignment process removes any gaps that are present due to the use of smaller size labels than the maximum that the architecture can support. This allows for the simplification of the processing in the FSM.

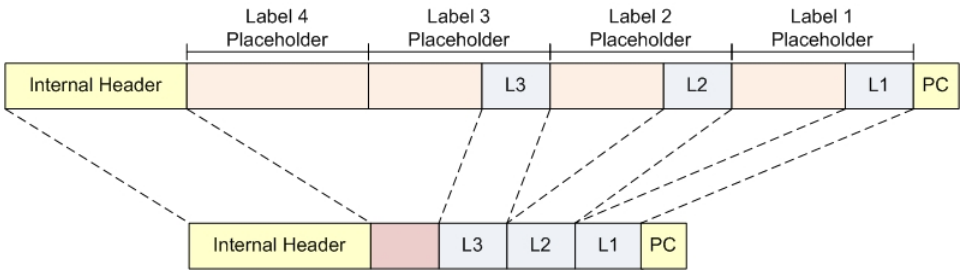


Fig. 4.6: Header Alignment Process Example in the Packet Reassembly Module

The core of the work is done by the Packet Reassembly module, which is a state machine, that takes the aligned header and reconstructs the packet using the packet data fetched from the buffer memory. To do this, it must determine which labels from the original packet need to be maintained and which overwritten (if any in both cases). This is accomplished by parsing the 8 processing context bits and determining the operations that were performed on the labels. From this, the module can extract how many labels in the original packet need to be replaced and how many and which of the labels in the new packet header need to be inserted into the original packet data.

Figure 4.7 describes the reasoning behind the reassembly process. Two parameters are fundamental to it:

- The number of pipeline labels to insert in the reassembled packet
- The insertion offset, that is, the location in which these labels are to be inserted.

It should be kept in mind, that the packet version which is fetched from the main memory contains all labels, including the ones that were sent down the pipeline and as a result of that, might have been changed. Thus what these two parameters essentially accomplish, is to help determine how many labels are to be replaced and from which label in the label stack this replacement is to start. Figure 4.7 illustrates these parameters for each of operation combination, which makes operational sense in a label based protocol, as explained in section 2.2.1.

- Each pop increases the insertion offset by 1. This means that the new labels will be inserted one label to the right, practically overwriting one preexisting label. This is reasonable, since this label has been popped thus no longer exists in the stack.

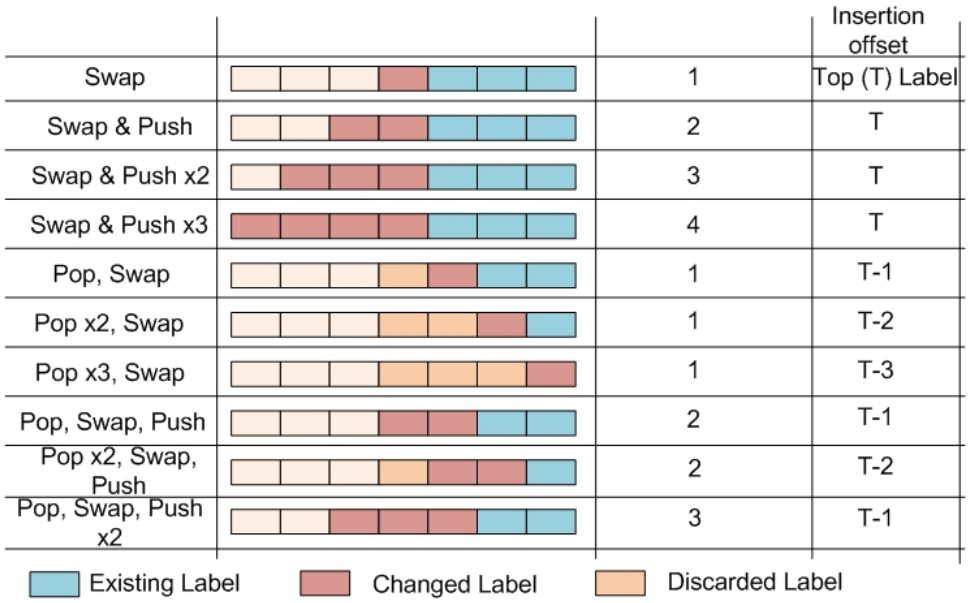


Fig. 4.7: Process Determining the Label Composition of the Reassembled Packet Header Depending on the Sequence of Operations Performed in the Pipeline Core

- Each Swap or Push Increases the Number of Pipeline Labels to Overwrite by 1 (Since a New Label Has Been Added which Was not in the Initial Packet).

Applying this algorithm to any MPLS scenario yields the correct parameter values, which allow the insertion of any labels at the correct location in the label stack. To exemplify, in a scenario where a packet enters two tunnels, the top label is swapped and then two labels are added on top of that. All three of these labels were the results of the processing in the NPU100 pipeline core and are not included in the original packet fetched from the data. As figure 4.9 illustrates (assuming an original packet with a stack depth of 2), the label which was swapped -the top label in the original stack- must be replaced by the label produced by the first pipeline loop (shown in red on the figure). An LSP in which this scenario is applicable is shown on figure 4.8, where packets flow from node 1 to node 11 and have to enter two tunnels at node 5. This means that in node 5 the existing label has to be swapped, followed by two labels which have to be pushed on top of the swapped label. The packet reassembly module thus must place three labels right after the top label in the original packet's label stack. This is in line with the calculated parameters, which would be zero for the insertion offset and 3 for the number of labels to be added. As a result of this the label stack grows to 4 labels.

A further example is show in figure 4.10 where a more complex scenario involving an exit from two consecutive tunnels and an entry into another one at the same node is shown. This means that in a packet which arrives at the node with a three label stack, the top two labels have to be discarded, the third label from the top of the stack has to be replaced with the result of the swap and the pushed label has been added on top of that. This means that after two pops, a swap and a push, the insertion offset is 2 and the number of labels to add is also two, leading to the depicted reassembled packet.

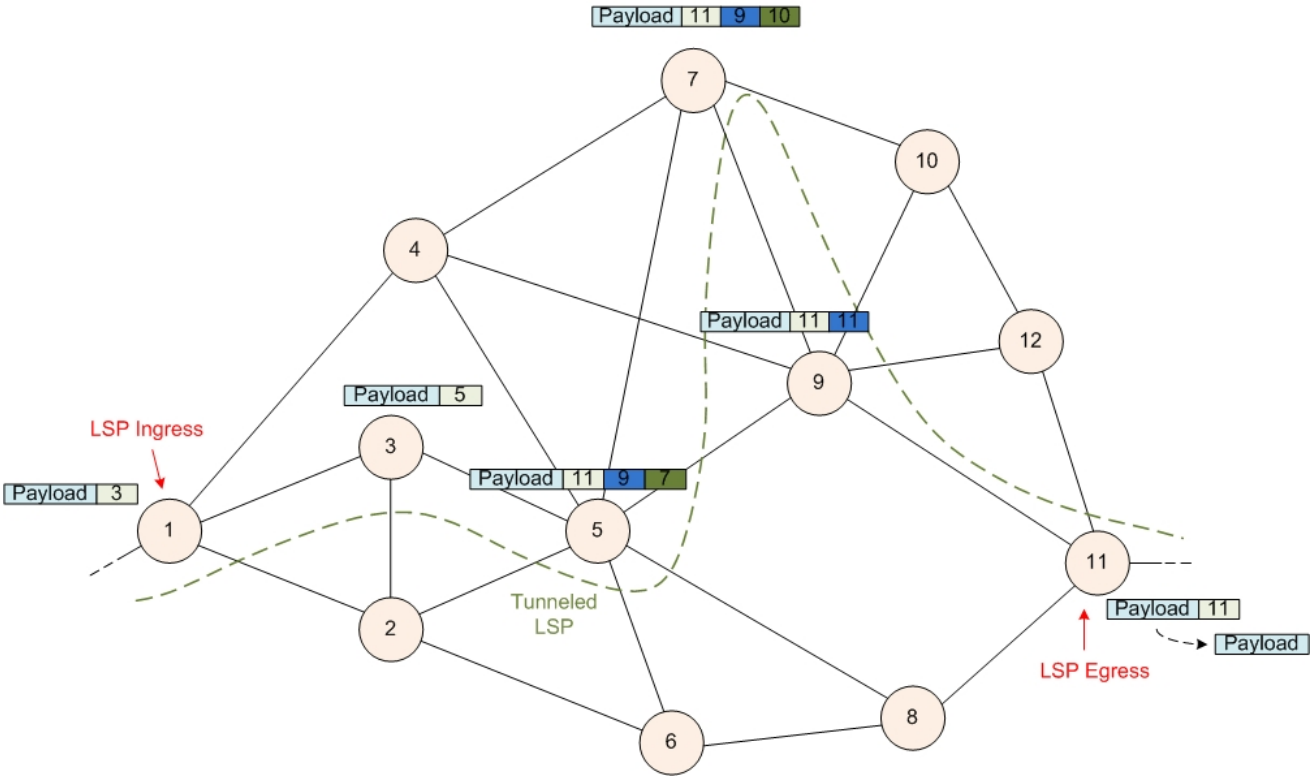


Fig. 4.8: Example of an LSP in which the packets enter a double tunnel at node 5

After determining the number of labels to add and the location in which to add them, the state machine must forward the packet to the output. This presents considerable difficulties, since the packet size changes in the pipeline and thus the data words read from the memory need to be realigned accordingly. Figure 4.11 provides examples for the two possible alternatives that might occur and how these are handled:

- The size of the packet is now bigger than the initial packet (illustrated by the top case in figure 4.11). This means that a part of the first data word read from the memory must be shifted to the next data word, and that all following ones must be subjected to the same process. This occurs when:

$$(NoOfLabelsAdded - NoOfLabelsRemoved) > 0 \tag{4.1}$$

- The size of the packet is now smaller than the initial packet (illustrated by the bottom case in figure 4.11). This means that the initial data word must be filled with data from the second data word and all following ones have to be treated accordingly. Respectively for this to happen:

$$(NoOfLabelsAdded - NoOfLabelsRemoved) \leq 0 \tag{4.2}$$

Equations (4.1) and (4.2) are used to calculate the number of bytes to be added or removed to the packet, a parameter useful in the data word realignment process.

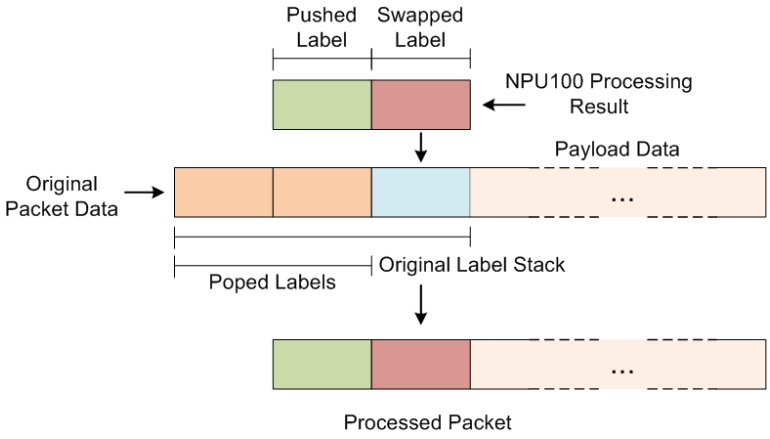


Fig. 4.9: Example Packet Reassembly in a Pop, Pop, Swap and Push Scenario

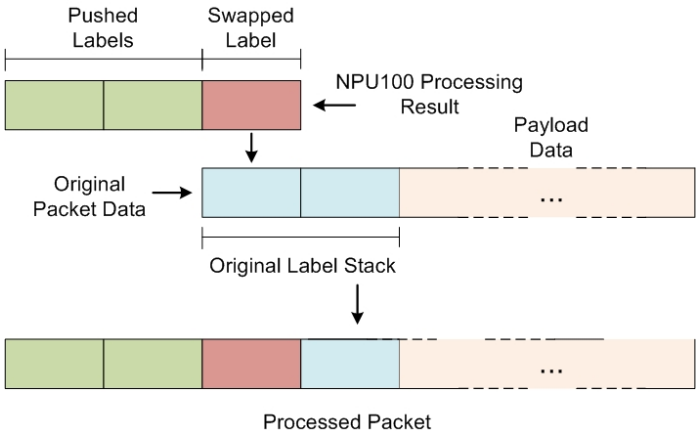


Fig. 4.10: Example Packet Reassembly in a Swap, Push, Push Scenario

These requirements pose considerable challenges to the design of the state machine. Given the high throughput demands, the luxury of sparing several clock cycles to fetch data from the main memory, then recombine the packet and provide the data at the output is simply not there. It must be possible to perform all of these tasks seamlessly and provide one valid data word to the output in every clock cycle. Accomplishing this requires actively prefetching data words from the packet data memory. Thus when a packet header arrives from the Label Reassembly the first data word will already have been buffered in the Packet Reassembly. To ensure the absence of gaps when packets are transmitted back to back, this must be done in parallel to the transmission of the last word of the previous packet. Further complicating the process is that when the packet is actually shortened, the FSM needs more data to fill the resulting void. This is the reason for having a 1024 bit wide buffer memory interface. Thus a 1K data word is prefetched from the memory, from which 512 bit words are then produced as needed, by shifting the necessary parts around while eliminating what is not needed. If a 512 bit interface had been used, the FSM would have to prefetch two data words before being able to reassemble the packet, something that would lead to more complex logic.

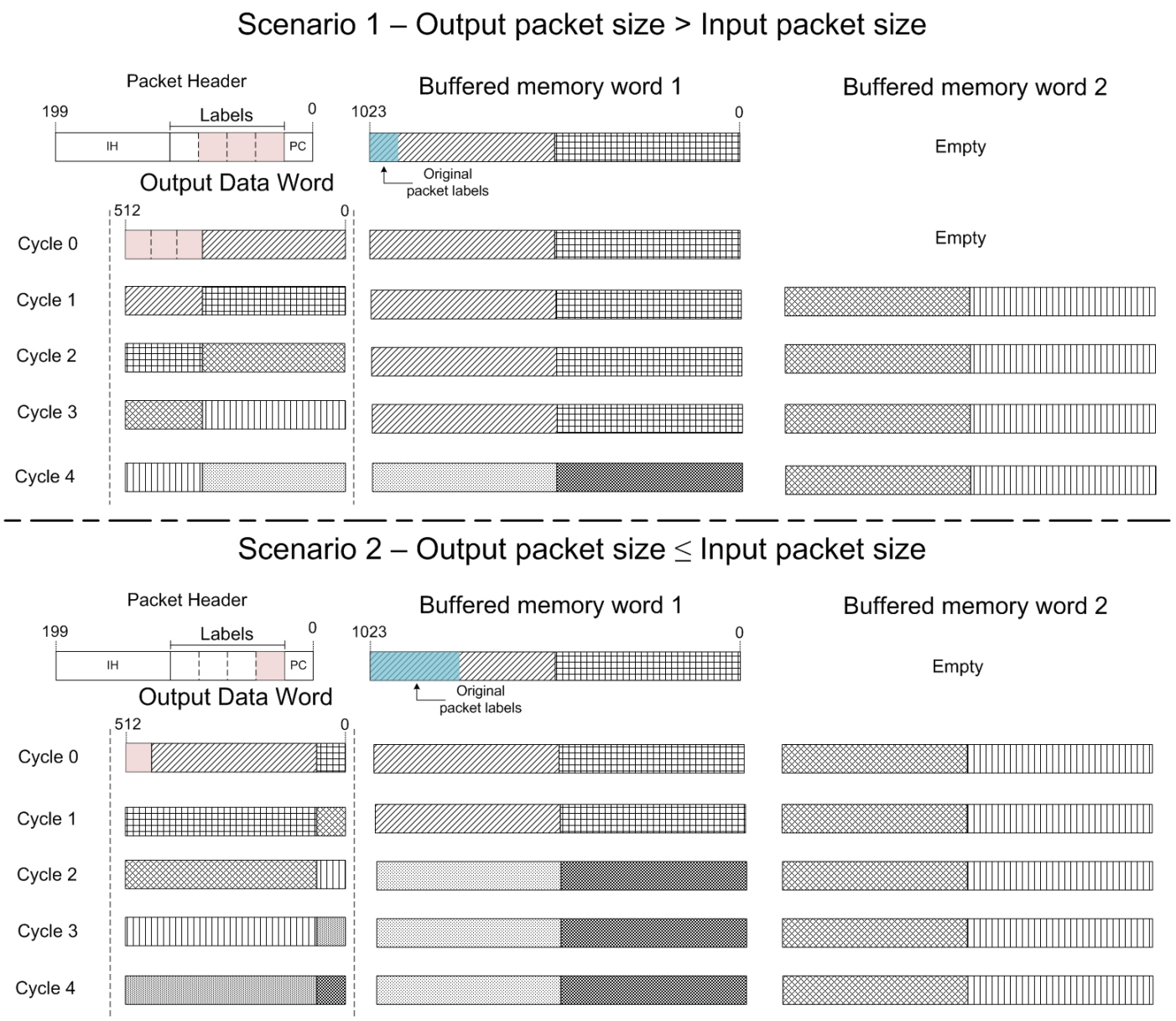


Fig. 4.11: Packet Reassembly Process Illustrating Two Cases, the Top One where the Output Packet Grows in Comparison to the Input One, and the Bottom One where it Shrinks

How both of these factor are weighted in the implementation of the state machine is shown in the figure 4.11 for both scenarios. It shows the packet header which results from the processing in the NPU100 pipeline, as well as the two 1024 bit data words, which the Packet Reassembly FSM internally buffers so as to reassemble the packet. Each of these words is made up of two 512 bits parts which are used to rebuild the output. These 512 bit parts are each marked using a different pattern to differentiate between them.

In the top scenario, the first case, in which the final packet size is bigger than the initial packet size is illustrated. Here the packet grows by two labels, as the packet included one label upon arrival at the NPU100, but has three labels when it is being forwarded to the next node. Since the packet grew in size the first 512 part of the first 1024 data word cannot fit into the first 512 bit data output word at cycle 0 and it spills thus over to the next output data word cycle 1. There, this left over part is joined with the appropriate part from the second 512 part of the first data word. Meanwhile the FSM must fetch a second data word from the packet buffer memory, since in the next cycle (number 2) it will need to output the remainder of the second part of the first 1024 data word and fill the rest of the space with data from the next packet buffer memory data word. The process then continues until the packet length is exhausted.

The bottom scenario pertains to the possibility of a packet shrinking during processing, as is the case when a packet exits subsequent tunnels at the same node. Here the first 512 bit section of the first 1024 bit data word is not enough to fill the output word and thus it must be padded with part from the second 512 bit part. This padding promulgates through the output process for the entire packet, while the remaining data words are fetched from the packet buffer memory and sent to the output.

Finally, when processing protocols like PBB-TE, where there is no change to the header and thus the packet copy from the buffer memory must be forwarded as is, are processed, the processing context bits indicate just that, meaning that no change is to be done to the packet data read from the memory.

4.1.4 Main Memory

The Main Memory temporarily buffers packets that are in flight in the processing pipeline of the NPU100. Thus it needs to be sufficiently large to hold the appropriate number of packets, which lies at 62 (4 mini pipelines times 10 stages per mini pipelines, plus 12 packets in the dispatcher, 8 in the label reassembly module, 1 being written to the memory from the H-P split unit and one being read from the memory from the packet reassembly module). Taking into account the maximum Ethernet frame size of 1518 bytes for normal packets (Jumbo frames are excluded) 94116 bytes of memory must be available in the NPU100 for packet buffering (for comparison the largest Virtex 5 TXT FPGA offers approximately 1.458.000 MB of Block RAM memory). It will be operated like a normal FIFO, written to and read from in 1024 bit words. Thus two input words are packed into one memory word with any required padding performed by the HP Split module. Respectively the Packet Reassembly module reads each 1024 bit word and converts it into two 512 bit words before sending them to the output.

The packet data FIFO consists of 1500 1Kb entries and offers a total of 192 KB of memory. The over provisioning is done to counter the 65 byte problem described in section

4.1.1. According to this, if the NP had to process only 65 byte packets, the 1K bus to the main memory would be 50.78% utilized, which practically means that in this case we would need twice the memory, to accomodate for both the packet data and the considerable padding that would have been performed by the Header-Payload Split. This storage space waste can be avoided by implementing a more complex smart segmentation scheme, such as the one described in previous work performed at the Institute for Integrated Systems (see [67], [66], [68] and [64]), which however remains outside the scope of this thesis.

4.2 NPU100 Processing Pipeline

The NPU100 folded pipeline architecture forms the core and main contribution of this work, since it provides the basis for the significant advantages provided by the NPU100. The architecture it self was described in detail in section 3.2.1. This section describes the implementation of the modules, which make up the architecture and realize the required functionality. They are essentially broken down into the Dispatcher and Label Reassembly modules, each with its own Label Buffer (sections 4.2.1, 4.2.2 and 4.2.3 respectively) and the modules that make up each MP (4.2.4).

4.2.1 Dispatcher

The Dispatcher implements the first half of the loop functionality of the folded architecture, by buffering and fetching labels as required from its buffer before the packet headers are sent to the MPs for processing. Its functionality consists of several tasks that must be completed in parallel:

1. New packets must be fetched from the Header-Payload Split module. Since the clock domain crossing between the two modules is implemented using a FIFO queue the Dispatcher must read from this FIFO queue, whenever data are available.

2. This data consists of a packet header and the internal header. The processing to be done on the packet header depends on the protocol which is currently being executed:

   a) MPLS-TP headers consist of multiple labels. The number of labels that are sent to the pipeline core is programmed in the Header Payload Split. From these, the Dispatcher extracts the top label and assigns it to an MP, while storing the rest in the Dispatcher Label Buffer.

   b) In PBB-TE the B-DA and B-VID fields need to be forwarded to the MPs. These fields are not found in contiguous locations in the packet header and thus the Dispatcher must locate and fetch them. The Label Buffer is not used at all in this case.

3. The processing context has to be added to the new packet header. The context is initialized at this point (all bits are set to zero except bit 8, which denotes that the current pipeline word is valid).

4. Up to three packets may come through the loop for additional processing in one clock cycle. It must be examined if any of these packets requires the fetching of new labels from the label buffer. If so the labels must be fetched.

5. All the packet headers must be assigned to a new MP.

The Dispatcher module must be sufficiently flexible to implement various protocols. This is supported through different configurations on two levels. First of all, the design includes several generics in the VHDL code, which can be set to adapt various design parameters at design time and on a second level, several design features can also be reprogrammed dynamically at run-time through the PCI Express interface. A change in the generics means of course that the system has to be reimplemented. The generics available for configuration are:

- Number of MPs - Designates how many MPs will be instantiated in the design. Affects the buses coming to and leaving from the Dispatcher, as well as the number of packet headers the module has to process in parallel.

- Number of stages per MPs - Specifies how many stages are to be found in each MP. It impacts the structure of the Dispatcher Label Buffer (described in the next section 4.2.3) and thus the synchronization between the two modules.

- Maximum label size - Defines the maximum label size that the Dispatcher can support. This can be different than the label size actually being processed at any given time. This affects the size of the buses and the memory cells of the Dispatcher Label Buffer. Labels of a smaller size can still be processed and this is specified programatically through the PCI Express link.

A whole different set of options, operating in a complementary fashion to the generics described previously, sets the behavior of the NPU100 at run time:

- Direct forwarding - Defines if the protocol used is a label based protocol or not. If not the Dispatcher Label Buffer is not used and the label storage process is deactivated. Instead two fields from the header are extracted according to the parameters defined below.

- Position and length of the first and second fields - Start locations in the packet header and length of the fields to be sent to the MP, when the Dispatcher is in direct forwarding mode.

- Label size - Size of the label, which the NPU100 currently processes. Must be smaller than the maximum label size specified in the corresponding generic.

The Dispatcher includes several interfaces, whose width depends on the configuration of the generics described previously:

1. Packet header input from the Header-Payload split module, whose width must be compatible with the respective value set in the Header-Payload Split module.

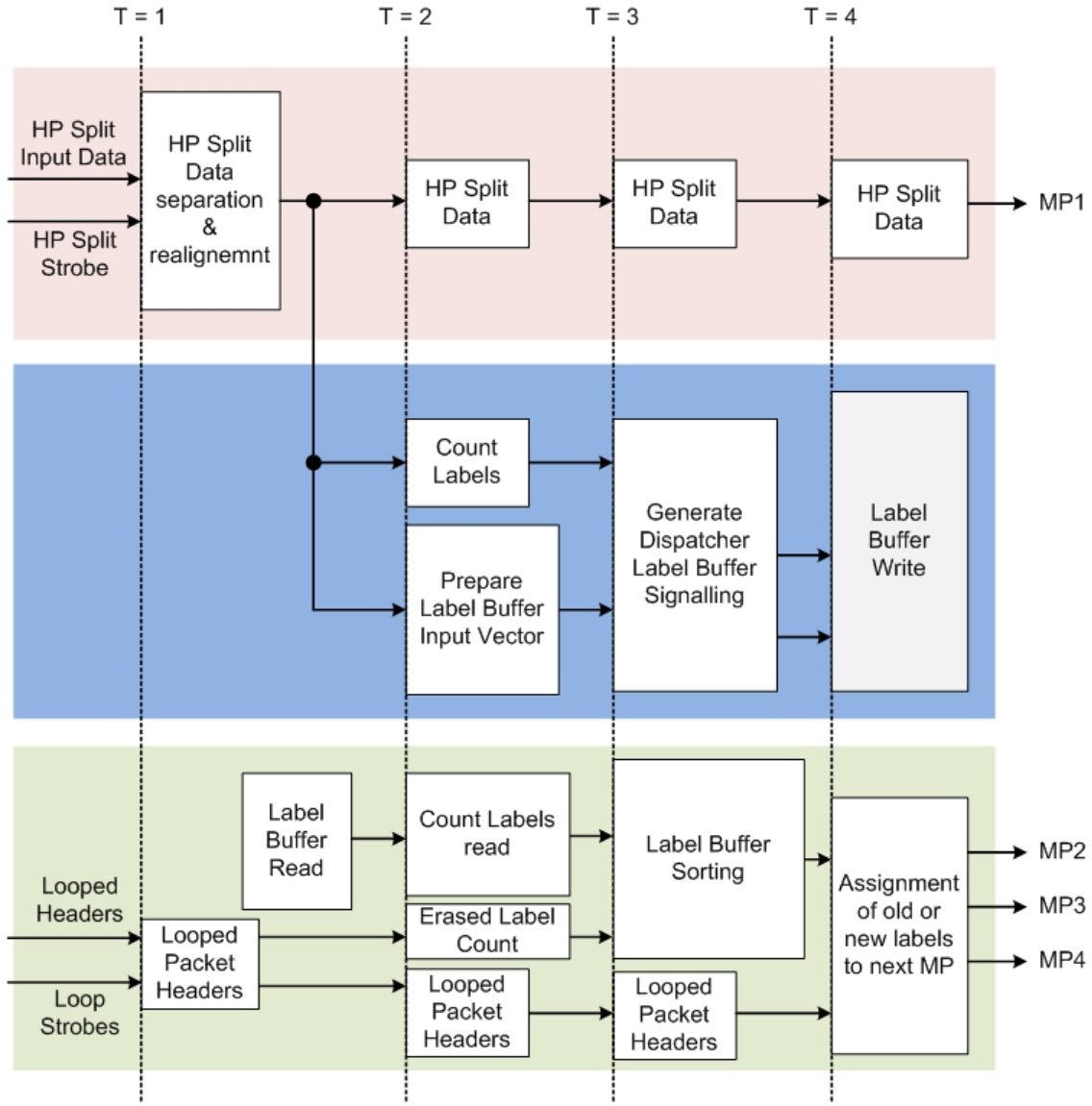


Fig. 4.12: Overview of the Dispatcher Design, Showing the Three Parallel Pipelines which Make up the Module

2. Looped packets input from the Reassembly module. N-1 interfaces are needed, each 192 bits wide in order to fit one pipeline word each.

3. Output to the Dispatcher Label Buffer, whose width depends on the maximum label size generic and the number of MPs in the system. Since all but the first labels have to be stored, the length is calculated by the formula MaximumLabelSize x (NoOfMPs - 1).

4. Output to the mini-pipelines in order to send the Dispatcher output to the MPs for processing. One entry per MP is needed, each one pipeline word wide.

The Dispatcher has been implemented as three parallel executing pipelines, which are shown in figure 4.12. Each pipeline is color coded. The complete processing in the Dis-

patcher stretches over four clock cycles. This is necessary, since there are several processing steps to be completed in sequence and in order to reach the required throughput it is mandatory to spread them over more than one processing stages. Each clock cycle is separated by a vertical dashed line and named appropriately.

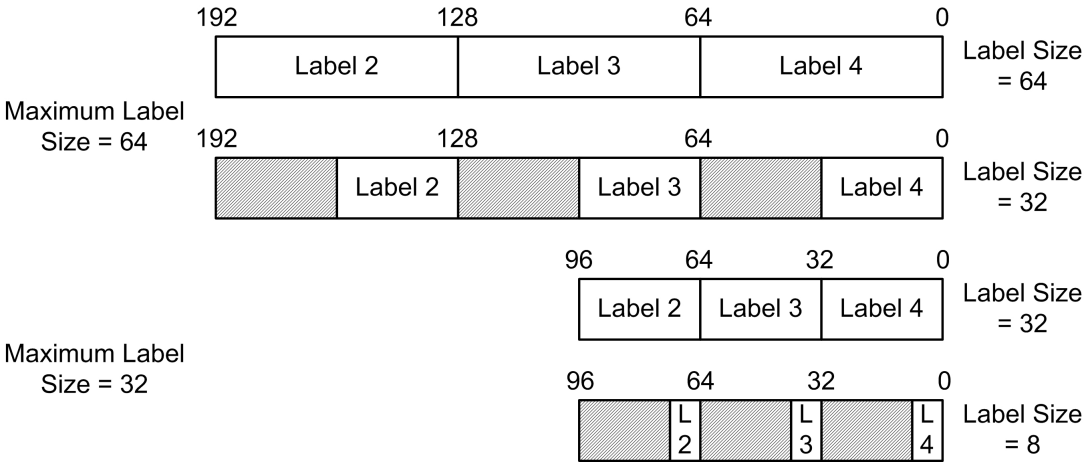


Fig. 4.13: Dispatcher Label Buffer Input Vector for Various Values of the Maximum Label Size Generic and the Actual Label Size Currently Being Processed in the Pipeline

The top domain in the Dispatcher (marked with red on the figure) handles the reception of data from the Header Payload Split. From there it branches off to the second, blue colored domain, where the write to the label buffer is executed. Finally, the third independent part receives the packets from the Label Reassembly via the loop, fetches any labels as needed and reassigns the packets to the MPs.

After being notified of new data by the Header Payload Split module through the appropriate strobe signal, the top domain latches the data and depending on the configuration mode, either splits the first label and sends it to its next stage, while forwarding the rest to the middle domain or extracts the two fields specified in the configuration parameters and passes them on to the next top domain stage. Before forwarding, the data are properly aligned into the domain's internal 128 bit wide bus. This means that the label or the extracted fields (which might be smaller than the bus) are properly placed on it and the rest of the bus is filled with zeroes. The 64 MSBs are occupied by the internal header. The two subsequent clock cycles simply serve the synchronization of the three pipeline domains, while in the final clock cycle the processing context is added and so that the final 192 bit pipeline word is created.

The middle domain is only used in label mode. It receives the detached labels from top domain and writes them into the label buffer. In order to accomplish that a series of tasks have to be completed. The first stage of this pipeline (in T=2) counts the labels, which is necessary for knowing how many will be written to the Label Buffer. This is in turn required so as to correctly signal the Label Buffer and write entries only in the needed locations. In parallel to the label count, the input data is brought to the format, in which it is to be stored in the Label Buffer. This is necessitated by the possible discrepancy between the maximum and the currently used label size. If the latter is smaller than the

former there will be gaps in the entry, as shown in figure 4.13, where several scenarios with maximum label size of 64 and 32 bits and a label size of 64, 32, and 8 bits are illustrated.

Pipeline stage T=3 receives then the data and generates the appropriate signaling to write the required label buffer entry. This entails maintaining a write pointer that tracks the packets, which are in-flight in the pipeline and access their respective data when they arrive at the Dispatcher. The use of the pointer to write the required Label Buffer entries is explained in section 4.2.3, where the exact process is described in the respective Dispatcher Label Buffer subsection.

The final, bottom Dispatcher pipeline domain (green on the figure) is the one which receives looped packet headers and replaces redundant labels with new ones from the Label Buffer. A strobe signal for each packet header notifies the Dispatcher if a packet header is incoming from that MP. In the first clock cycle T=1 the data is fetched from the Label Buffer by using a read pointer, which the Dispatcher maintains. This pointer is incremented along similar lines to the respective write pointer. In cycle T=2, the erased labels in each looped packet header are counted, to determine if new ones from the Label Buffer are required. Notice that the read from the Buffer has already been proactively performed and the required labels are sorted out belatedly. Thus the internal organization of the Dispatcher Label Buffer (as described in 4.2.3) is taken advantage of, to shave off cycles from the processing. In parallel to the erased label count, the labels fetched from the Label Buffer are also counted. In the third clock cycle this information is used to reorder the labels according to the loop in which the packet header is in. The Label Buffer entry which corresponds to a packet header remains fixed, independently of the MP run in which the packet header is in. Thus the results of the buffer read have to be reshuffled accordingly. Finally in cycle T=4 the sorted labels are assigned to the correct packet headers, overwriting the old contents.

4.2.2 Label Reassembly

The second indispensable ingredient of the folded architecture, which makes up the second knot, that binds the mini pipelines together, is the Label Reassembly module. The Label Reassembly module has to perform several tasks, which involve the management of the pipeline data words that arrive from the mini-pipelines. These are as follows:

1. Check if the incoming packet headers require further processing. A maximum number of packet headers equal to the number of mini pipelines may arrive in each clock cycle. In this case, at least one will be a fully processed packet header and the remaining ones may be either completely processed or require further processing.

2. Incompletely processed packet headers are sent through the loop to the Dispatcher. Before this is done, the operation that was performed in this MP run must be marked in the processing context.

3. Temporary results (e.g. labels newly added to the stack) must be saved in the Reassembly Label Buffer, lest they are overwritten in the next MP run. In this case, only the label data must be written, which does not correspond to an entire Reassembly Label Buffer entry (see section 4.2.3 for more details on this).

4. Completed packet headers may have to be temporarily buffered in the Label Reassembly Buffer before being sent to the output. This is because of two reasons:
   - Multiple packet headers from different MPs might arrive at the module at the same time. Only one will be sent to the output in that clock cycle and the rest will have to be stored in buffer in order to compensate for this burst.
   - The main packet memory is a FIFO, which means that packets are read on a first come, first serve basis. Hence, packet headers completing fewer MP loops and thus overtaking other packet headers, which need to run a longer course, have to have their transmission postponed in order to avoid violating the FIFO principle.
5. When a packet is to be transmitted, the data word consisting of the internal header, all labels resulting from the processing plus the 8 processing context bits must be put together by reading the needed data from the Label Buffer and sent to the output.

As with the Dispatcher, the Label Reassembly is also configurable through generics at design time and programmable over the PCI Express interface at run time. The generics used are identical to the ones in the Dispatcher, as are the programming parameters, without the starting location and length of the fields to be extracted in case of a non label based protocol.

The interfaces of the Label Reassembly enable its connectivity with the neighboring modules. Their exact length depends on the configuration of it's generics:

- A number of 192 bit pipeline word inputs coming from the MPs. The number is equal to the number of MPs.
- Number of MPs - 1 pipeline word signals looped back to the Dispatcher plus a strobe signal for each of these signals.
- Interface with the Label Reassembly Buffer. For the structure of the entry to be written and the control signals used, refer to the buffer's description in section 4.2.3.
- Output to the Packet Reassembly. Its width must be equal to the length of the internal header plus four times the maximum label size plus the eight processing context bits used for packet reassembly.

In realizing the Label Reassembly module, a parallel pipelined approach similar to that of the Dispatcher was used. Again it consists of three pipeline domains, as illustrated by figure 4.14, however here only two pipeline stages are used. All domains use the packet headers output by the MPs as inputs. The top domain, marked in red, checks if packet headers have to be send back to the Dispatcher for further processing and forwards them appropriately. The blue colored middle domain reads the MP output, determines which data, if any, are to be saved to the Label Reassembly Buffer and performs the writing accordingly. The green, bottom pipeline checks which packet header must be sent to the output, reads the appropriate data out of the Label Buffer and sends them to the output.

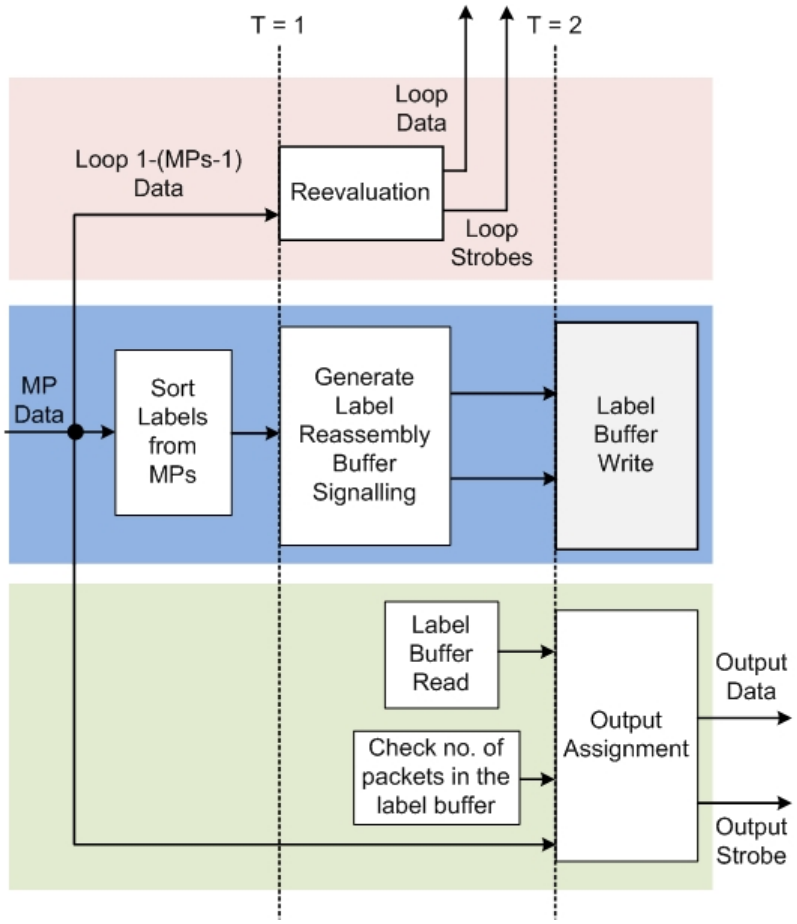


Fig. 4.14: Overview of the Label Reassembly Design with the Three Parallel Pipelines Demarcated

The top domain task involves checking each packet header except the one on its last loop for the operation performed on it in the previous run. This is marked by three processing context bits, whose significance is explained in table 4.1. The first column of the table denotes the stack operation that was performed. Three alternatives, swap, push and pop are available, with two sub categories each: one which indicates the packet header requires further processing and is termed "‘l̈oopänd one which marks the processing as complete, called forward. The operation code for each one is given in column four. In each case, the LSB differentiates the loop from the forward variant, whereas the remaining two MSBs denote the operation. The module checks the afore mentioned LSB and determines if the packet header should be looped back to the Dispatcher or not and marks the operation performed (which essentially means copying the two operation code MSBs) in the processing context to enable the Packet Reassembly to rebuild the packet after processing completes. Strobe signals to the Dispatcher are produced for looped packet headers. Since looping is only required for label based protocols, enabling the direct forwarding parameter sidesteps this domain.

The middle domain is tasked with writing needed data into the Label Reassembly buffer. In the first clock cycle the packet headers are investigated and the part (if any) of each

Operation	When	Further processing required	Op ID
Erase label & forward	Previous to last node in a tunnel with PHP enabled	No	000
Erase & loop	Exit from two tunnels in the same node	Yes	001
Swap & forward	Typical forwarding	No	010
Swap & loop	Tunnel Entry	Yes	011
Push & forward	Tunnel Entry (last tunnel in the stack)	No	100
Push & loop	Tunnel Entry (more tunnels follow)	Yes	101

Tab. 4.1: Supported MPLS-TP Operations and Required Label Reassembly Action

packet header which should be written into the Label Buffer is determined. Immediately afterwards but still in the same clock cycle the data to be written is extracted from the packet header and formatted according to the Label Buffer inputs. In the next clock cycle the signaling required to write only the appropriate parts in the needed area of the correct buffer entries is generated and in the next clock cycle the data is written into the Buffer. A pointer maintained in the Label Reassembly assures that the correct packet header data ends up in the entry where it belongs. The signaling required and the pointer concept are described in section 4.2.3. As with the top domain, this one also works only for label based protocols and is deactivated if the direct forwarding parameter is enabled.

The final bottom domain outputs the data to the Packet Reassembly. The process is different depending on whether a label based protocol is being processed or not. In the latter case the incoming packet exiting its first loop is forwarded directly to the output. The 8 processing context bits are set to 0. For a label based protocol, a read pointer is kept and incremented with every clock cycle, which ensures that the sequence in which the packets are read out is the same as the one the came into the NPU100 processing core. This allows the Label Reassembly to adhere to the FIFO principle used in the memory. This way the module reads and forwards the appropriate Label Buffer entry or if the Buffer is empty forwards the MP output on its highest run directly.

4.2.3 Label Buffers

The Label Buffers of the Dispatcher and the Label Reassembly modules are two fundamental pieces in realizing the NPU100 folded pipeline architecture. They serve as scratch pad memories for labels not currently required for processing (in the case of the Dispatcher Label Buffer) or for results that need to be saved to avoid being overwritten (as in the Reassembly Label Buffer).

The delicate point in the design of the two label buffers is that they need to be kept as simple as possible, while at the same time adhering to strict requirements regarding the accesses they support. The optimal solution was deemed to be to design the label buffers as register files. This is due to the large number of concurrent inputs and outputs that need to access them and which would make it very difficult to implement with a memory

device.

Furthermore, the predictable nature in which packets flow through the MPs invalidates the need for random access to any memory location in the Label Buffers, since the location of each packet in the buffer in relation to its position in the pipeline word is known at any given time. Since the Dispatcher and the Label Reassembly modules need to access their buffers to store data only for packets, which are in that clock cycle in each respective module, pointers in each of them can be used to track the relevant memory cells.

The resulting design is a register array, which uses a time slot scheme to manage the data stored in it. Figure 4.15 illustrates the basic concepts behind the operation of both buffers. Though differences in the implementation details (which will be highlighted in the subsequent sections, dedicated to each of them) do exist between the two modules, the core functionality remains almost identical. The assumption here (met for clarity's sake) is that we have a system with four MPs and 5 stages per MP. This is illustrated in the middle column with a matrix consisting of 4 rows (one for each MP) and 5 columns (one for each MP stage). On the left hand side the Dispatcher Label Buffer consists of three rows and 5 columns. Each of the 5 columns corresponds to one MP processing stage whereas there is one row less than the number of MPs. This is because a packet header which is in its last loop (that is in the fourth MP in this case) cannot have any more data stored in the Dispatcher Label Buffer, since it will inevitably exit the pipeline core at the end of this run. On the right hand side the Label Reassembly Buffer is shown. It consists of a register matrix with 4 rows and 5 columns (that is, equal to the number of MPs and the number of stages per MP). Here, one row per MP is required in order to conform with the FIFO nature of the main packet memory (section 4.1.4). The corollary of this is that the first packet to enter the pipeline must also be the first packet to leave the pipeline, which means that it is possible that the Reassembly Label buffer will have to store multiple other packet headers, before being able to sent the first one to the output.

The figure attempts to highlight all of this with a thorough example, showing the status of the label buffer in several clock cycles denoted by T. In each Label Buffer the red box denotes the location of the write pointer and the green box the location of the read pointer. One thing to notice is that when reading from the Dispatcher Label Buffer or when writing to the Reassembly Label Buffer the pointer points to an entire column instead of a single cell. This is because we proactively read or write data for all packet headers that come into the module in a given clock cycle. The respective module takes care of sorting out the required part and discards the rest. This is done to reduce the complexity of the Label Buffer cells. Extracting the required packet header from each Label Buffer cell would result in a much more complicated circuit, which would have had difficulty meeting the stringent timing requirements.

The example starts with packet header P1 coming into the pipeline at which point the Dispatcher stores its redundant labels in the aptly marked position of the Dispatcher Label Buffer and sends the packet to the first MP. In the next clock cycle a new packet header P2 arrives and receives the same treatment. Now all the pointers in both buffers have moved one location to the right. Thus the pointers follow the progress of the packets down the pipeline. When a pointer reaches the end of a row (clock T=5) it folds back to the beginning. If it points to a single cell (as the write pointer in the Dispatcher Label Buffer

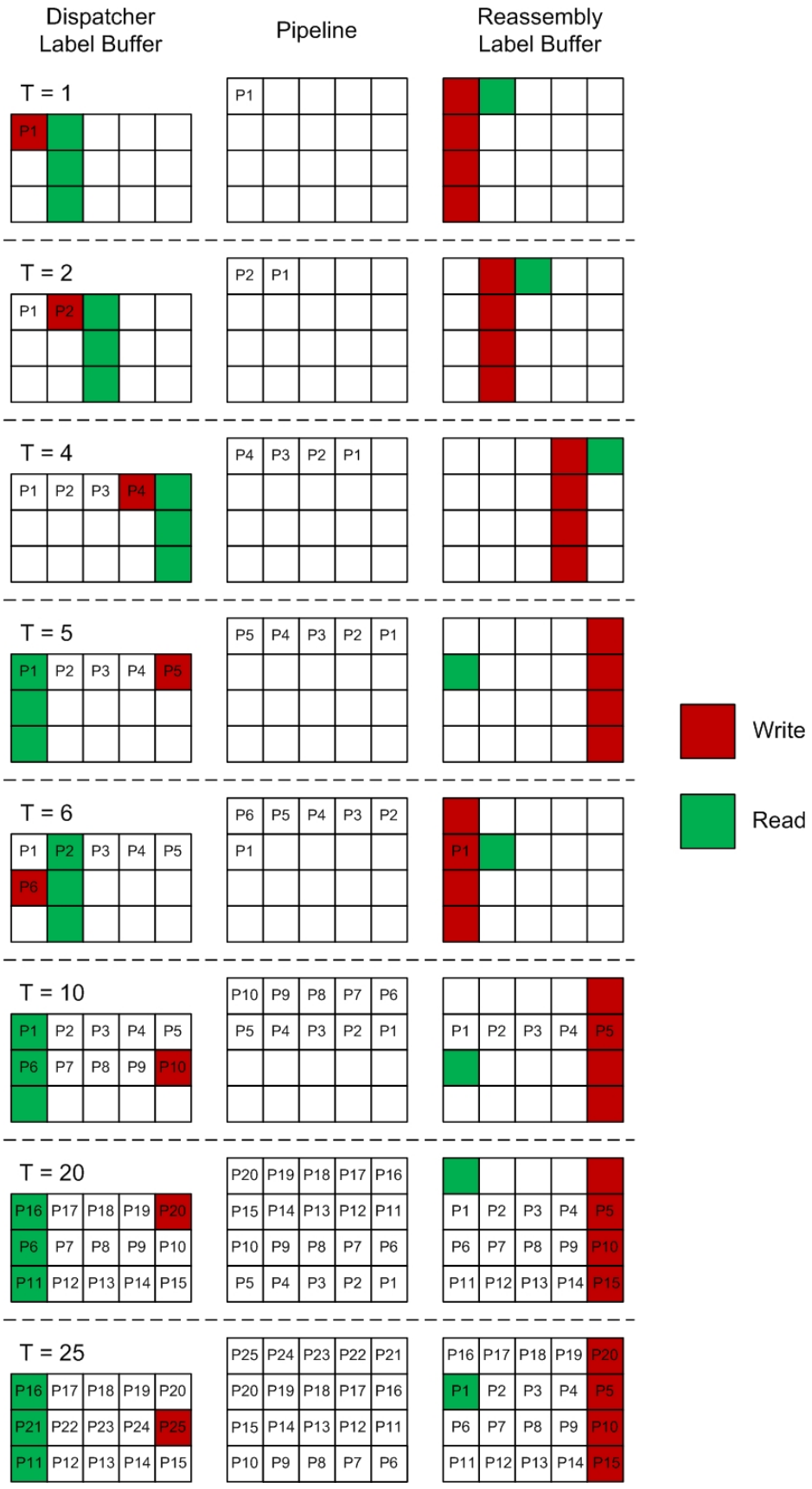


Fig. 4.15: Overview of the Label Buffer Time Slot Concept

and the read pointer in the Reassembly Label Buffer, then it also proceeds to the next row.

In clock cycle T=5 the Dispatcher Label Buffer read pointer has folded back to the beginning, so as to be able to read the stored labels, which would then be available in the next clock cycle, in which the packet header P1 will be sent into the next MP, as is shown it T=6. In that same clock cycle the intermediate results from packet P1 have been stored into the Reassembly Label Buffer memory cell, to which the respective write pointer was pointing. As the pointer points to an entire column, the Label Reassembly generates the appropriate signaling to write only to the required cells. On a side note, it must be stressed that in real operation, it will be impossible to encounter a scenario in which all packets require constant use of both buffers. This is because (using MPLS parlance for simplicity's sake) swaps and pushes require use of the Reassembly Label Buffer, while pops require use of the Dispatcher Label Buffer. The assumption is made here simply to illustrate the functionality of the modules.

This process repeats itself, with the pointers being incremented with each clock cycle to track the packets in the MPs. When a pointer has run through the entire buffer it starts over again from the beginning, thus overwriting the data in these entries. Hence the need to properly size the buffers so that the data that are overwritten are now redundant, since they belong to packet header which no longer has any use for them. Clock cycles T=20 and T=25 demonstrate this. In the former, the first row of the Dispatcher Label Buffer has been overwritten, since the pointer exhausted the available memory locations and folded to the beginning, while in T=25 the second row has also being overwritten. In T=25 the Reassembly Label Buffer read pointer has completed one run through all the memory cells and now coincides with the P1 packet header, which will be read out and sent to the output.

The example described above deviates slightly from the actual implementation in that it does not take into account the extra clock cycles that are required in the Dispatcher and Label Reassembly modules. These are incorporated as additional columns for all rows. Furthermore some MP stages are internally pipelined, which means that the amount of columns for that MP stage is equal to the number of internal pipeline stages it has. The following section describes the specific details for the Dispatcher Label Buffer and the Reassembly Label Buffer.

Dispatcher Label Buffer

The previous section described the operating concept behind the two label buffers. The current section builds upon that knowledge and adds details necessary for understanding the exact structure and implementation of the Dispatcher Label Buffer. The tasks, that the Dispatcher Label Buffer is called upon to fulfill, are as follows:

1. It provides memory cells for storing the currently redundant labels of each packet. M times N-1 cells are required, where M is the number of stages per MP and N the number of MPs.

2. Each cell must provide storage space for N-1 labels. The maximum label size that the Label Buffer can store is determined through a generic, while the label size currently

processed is programmable at run time.

3. The module must be able to read one entry per row in the same clock cycle, thus with the current working assumption of four MPs, 3 reads per cycle are needed.

4. Labels from the newly arrived packet header must be able to be written to the buffer in each clock cycle.

A key part of the design is that the input and output operations must be completely independent of each other. Thus the Dispatcher must be able to write an entry X, while it reads the entries belonging to a column Y. Although, as the previous section explained X and Y are predictable, the design of the Dispatcher Label Buffer does not preclude the writing to and reading from arbitrary cells or columns. This is because the pointers used to select the appropriate cells are maintained in the Dispatcher and not in the Label Buffer, thus if the Dispatcher wanted to access different cells, it would only need to address the Label Buffer accordingly.

With every read, an entire entry is returned by the Label Buffer and thus it is up to the Dispatcher to determine which labels are necessary to use when packets are looped and the respective entries are read from the Label Buffer. As such, it must posses some means which indicate this. Thus, each entry is made up of two parts, the one storing the label data and the other storing 2 valid bits, which are used to denote if the label data is empty, and if not, how many labels it contains. The Dispatcher counts the number of labels in the label stack of a newly arrived packet header. In MPLS-TP the last label is demarcated by the S bit, which is set to 1 for the last label in the stack. Then it sets the valid bits accordingly when storing labels into the buffer and evaluates them when reading labels from it.

The internal structure of the Label Buffer is shown in figure 4.16. It provides an overview of the buffer with multiple columns and rows. Three rows are shown, with the third row also providing a glimpse of its internal structure, in which each column comprises of the entry which holds the labels and the valid bits (marked with a V). A common data input signal is fed to all entries and the decoder shown on the top right side of the figure is used to determine exactly which entry will be written to. A common signal is used for both the data and the valid bits as they are written at the same time, upon arrival of a new packet. Furthermore, a write strobe signal is used to enable the write process, since there will be clock cycles in which no new packet will be present at the input and thus nothing will have to be written to the buffer.

Accessing the buffer involves selecting the appropriate value from two multiplexers, one for the data and one for the valid bits. Both multiplexers are driven by the same signal, as the data and valid bits for an entry are always accessed together. Since the same column is read from all rows, all multiplexers are in the buffer are driven by the same signal. The multiplexers allow the label buffers to provide data on the output with minimal delay as the access procedure is purely combinatorial. This saves clock cycles, which would otherwise be needed to access the required cell.

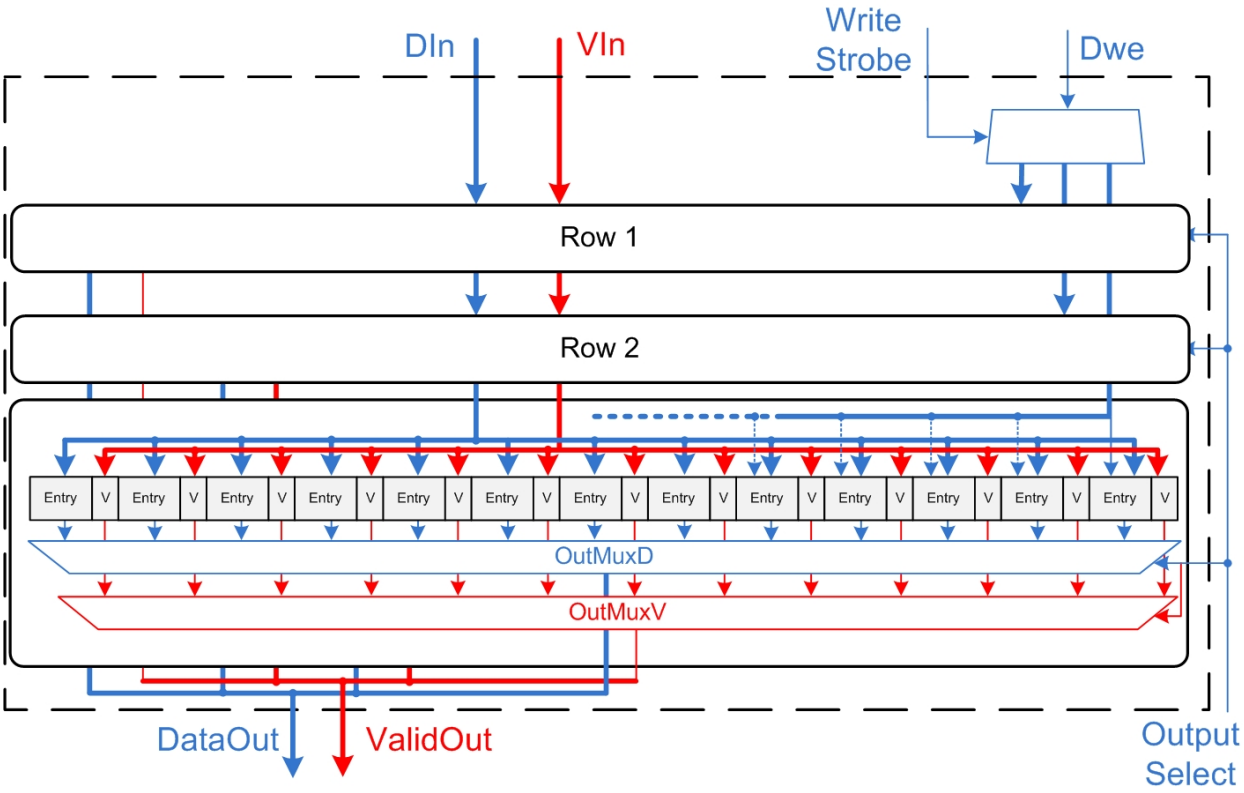


Fig. 4.16: Overview of the Buffer Design for Three Mini Pipelines and Thirteen Mini Pipeline Stages

Reassembly Label Buffer

The Reassembly Label Buffer shares the basic time slot functional concept with the Dispatcher Label Buffer. A corollary of this is that it is based on the same register matrix architecture, as described earlier. The similarities end there however and thus the requirements for the Reassembly Label Buffer are quite different than those of the Dispatcher Label Buffer. These can be summed up as follows:

1. More than one labels must be written per cycle (instead of read, which was the case in the Dispatcher).

2. The write operations need to be able to write only a specific part of an entry. In the Dispatcher Label Buffer only entire entries were written.

3. One read operation per cycle is required to fetch the packet header to be sent to the Packet Reassembly.

4. The entire entry is read and the Label Reassembly then discards any invalid data.

5. More entries are present, since one entry for each MP stage times the number of MPs is used.

6. Entries are larger, due to the demand to store all the labels plus the internal header and a small part of the processing context temporarily until they can be sent to the output.

As in the Dispatcher Label Buffer, an entry can store data belonging to one packet header. Writing into the buffer must be done once every time a swapped or pushed label will be overwritten in the next MP loop. This means that one single label will have to be written in the entry separately in the required location without affecting any data already present in the entry. Hence each Reassembly buffer cell is subdivided into five parts, as shown in figure 4.17:

1. IH + PC - a 72 bit part which holds the 64 bit internal header and the 8 bits of the processing context, which the Packet Reassembly uses to put the packet back together. These have to be buffered, since it is possible that in some cases the output will not be readily available to sent a new packet (e.g. when packet headers, whose processing has been completed, come out from more than one MP in the same clock cycle).

2. Labels 1 to 4 - These are four placeholders, one for each label that might need to be buffered. The fields are populated sequentially, which means that in case loop 1 does not write a label into the buffer but loop 2 does, that label will be written into the Label 1 placeholder.

3. Valid - The valid bits show which parts of the entry are currently occupied. They are written to by the Label Reassembly accordingly.

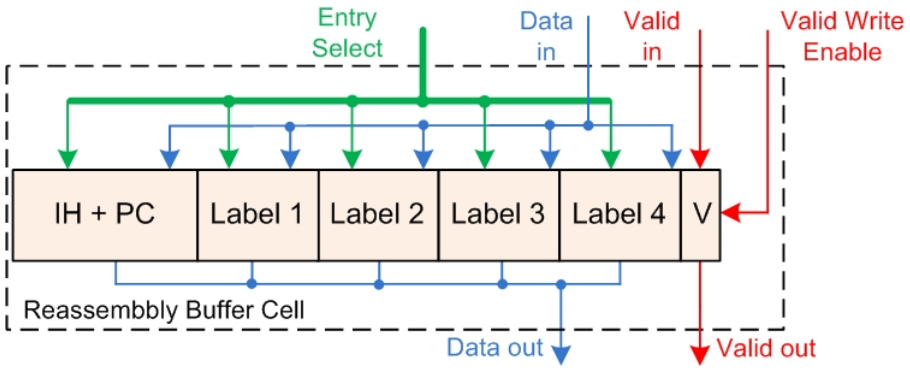


Fig. 4.17: Reassembly Buffer Cell Block Diagram

The complex writing requirements are translated into intricate signaling needed to fulfill them. Essentially it must be possible to write any field independent of the other, as it is be possible that when writing a label after a loop, the IH+PC fields will also have to be filled, since processing would have been completed. Thus each field has its own write enable, which is driven independently of those of the other fields. The same applies to the valid bits, which means that each valid bit is written separately from the others, without affecting their values.

Similarly to the Dispatcher Label Buffer, the cells to which it is written to must be completely independent of the cell from which it is read from. Thus in similar fashion the process is directed by the Label Reassembly, which implements the pointers and controls the Label Buffer accordingly.

Configuring the implementation of the Reassembly Label Buffer also resembles that of the Dispatcher one, with the only difference being the existence of a configurable Internal Header length, which affects the width of the IH+PC field. Apart from that, the number of MPs and stages per MP can be set to dimension the register matrix appropriately, as well as the maximum label size, which defines the space provided in the label placeholders.

Apart from any differences in the organization of each entry cell, the Reassembly Label Buffer is in essence an inverted Dispatcher Label Buffer. This is illustrated in figure 4.18. For clarity's sake only two rows (corresponding thus to two MPs) are shown, with 5 columns per row (thus a 5 stage MP is assumed). Writing into a buffer cell is done through a decoder, which sets the write enable signal for the appropriate cell when strobed. This implicitly includes selecting the appropriate entry fields as described previously. The Data on the Data in and Valid in buses are then transferred into the selected matrix entry.

Reading from the buffer is again performed through multiplexers which select one entry per row. The same entry is selected for all rows, thus one entire column is read out. The difference in comparison to the Dispatcher Label Buffer is that here, only one entry needs to be read, since only one can be sent to the output at any given time. Thus a second multiplexer stage is used (the one shown on the right hand side of the figure). This stage receives the entries from the column and outputs only the one needed. As in the Dispatcher Label Buffer, the output path is purely combinatorial and thus no precious clock cycles are lost, while reading out the data from the buffer.

4.2.4 Pipeline Modules

This section elaborates on the implementation of the modules that make up each MP of the NPU100. These operations and the requirements from which they stem were described in detail in section 3.2.1.

This section starts by describing the ISU and the OSU, which provide programmable access to specific fields of the pipeline word. In this sense, they are not processing engines themselves, but are enablers, residing in each processing engine and endowing it with a common interface from and to the pipeline word. This realizes the goal of making the NPU100 modular and contributes significantly to its programmability concept.

The remainder of the section, follows the sequence in which the PEs are found in an MP, with the forwarding lookup at the very beginning, followed by the QoS parameter lookup and finally the insert, replace and modify modules.

Processing Module Input/Output

One of the main design goals for the NPU100 was modularity. This translates to having a circuit that consists of building blocks which can be easily exchanged for other ones and to which components can be added or removed with minimum effort. One major step in this direction is endowing all modules with a common interface. This allows for predictable

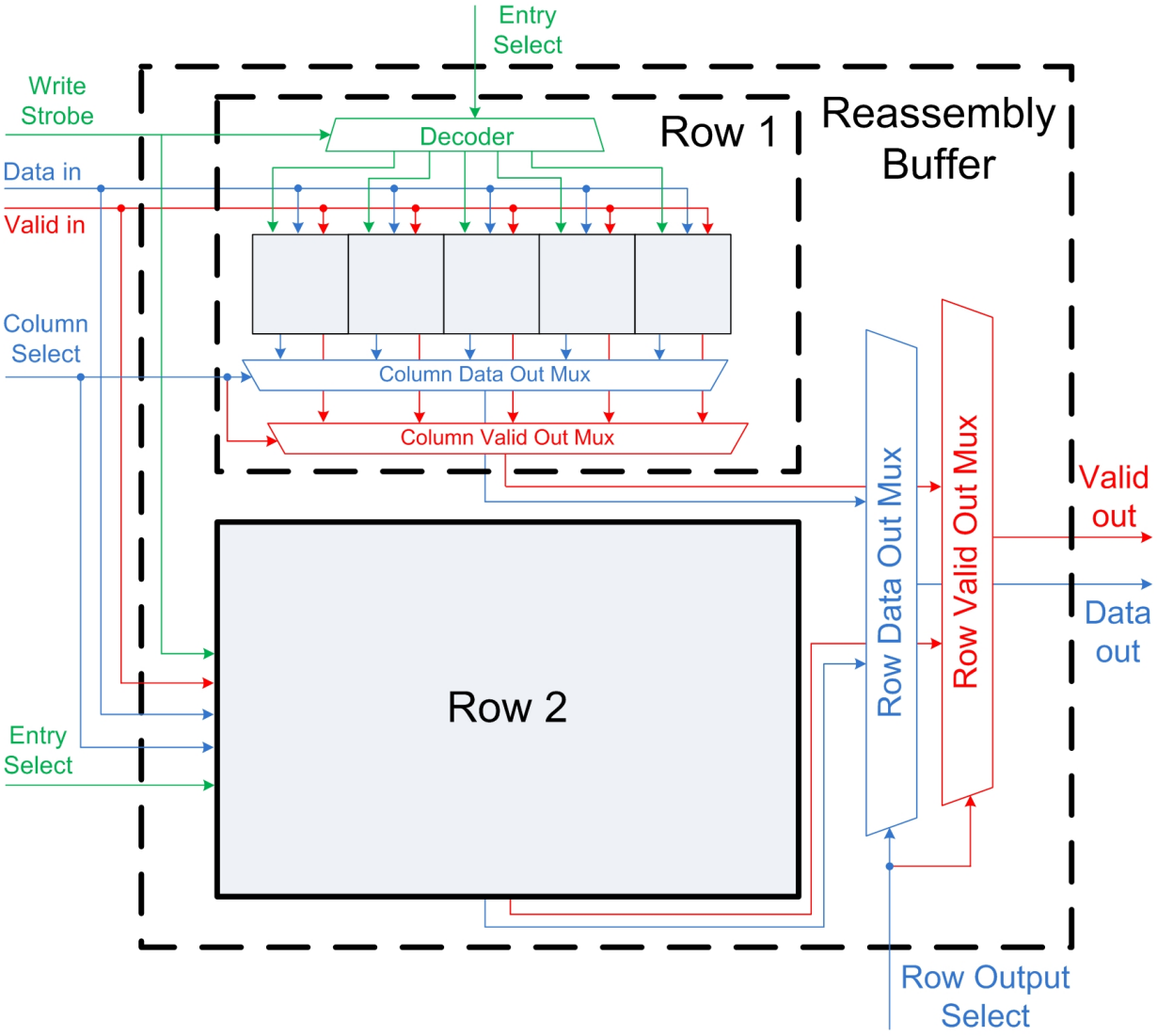


Fig. 4.18: Overview of the Reassembly Buffer Structure for Two Minipipelines and Five Stages per Mini Pipeline

and modular behavior, since in case of a redesign any module can be taken in or out of an MP at will.

A further design goal was to have a programmable, flexible design, which can be adapted dynamically to future protocols. This means that the data in the pipeline registers, which contain the pipeline word (which in turn consists of the packet header, the processing context and the internal header), should provide the execution units with access to the bits necessary for processing and allow them to store their results back to the next stage register. With this in mind, the Input and Output Selection Units (ISU and OSU respectively) for the NPU100's MPs were designed.

Figure 4.19 illustrates the concept behind the pipeline word accesses. A part of the register bits of pipeline stage i-1 are used to index a table, which contains the microcommands, that direct the operation performed by the ISU and OSU. This microcommand

table is found in each ISU/OSU. This keeps the code in close proximity to its PE, thus avoiding lengthy decode circuits and complex paths to distant instruction memories. The bits that are used for the indexing are user programmable via the PCI express interface.

At the same time, the bits of register i-1 are sent to the Input Selection unit, which extracts the bits to be used in the processing module and forwards them to it. The input from the microcommand table determines which bits are to be extracted. The PE then performs the required operations on the data. The same index bits used in the ISU/OSU microcommand table are used here to determine the command to be executed from a similar table found inside the PE (though this is not the case for every PE). The results are then sent to the Output Selection Unit which recombines them appropriately with the data from the previous pipeline word, so as to allow for their storage. If the processing module is further pipelined internally, then there must be additional registers in the processing module to maintain synchronization of the pipeline word with the results delivered by the PE. These internal registers do not require input and output selection modules and only buffer the pipeline word temporarily.

As described in section 3.2.1, it has been established that a sensible length for the entire pipeline word is 192 bits. It would however be unnecessary, as well as detrimental to the design of the unit, to forward all 192 bits to it. The PE itself would then be left to decode a wide data word, which would contain different fields in each PE. This would complicate development of the PEs and reduce the modularity of the design. In spite of this, it must be ensured that the PEs can potentially have access to any part of the pipeline word and that this can be easily reprogrammed by the end user.

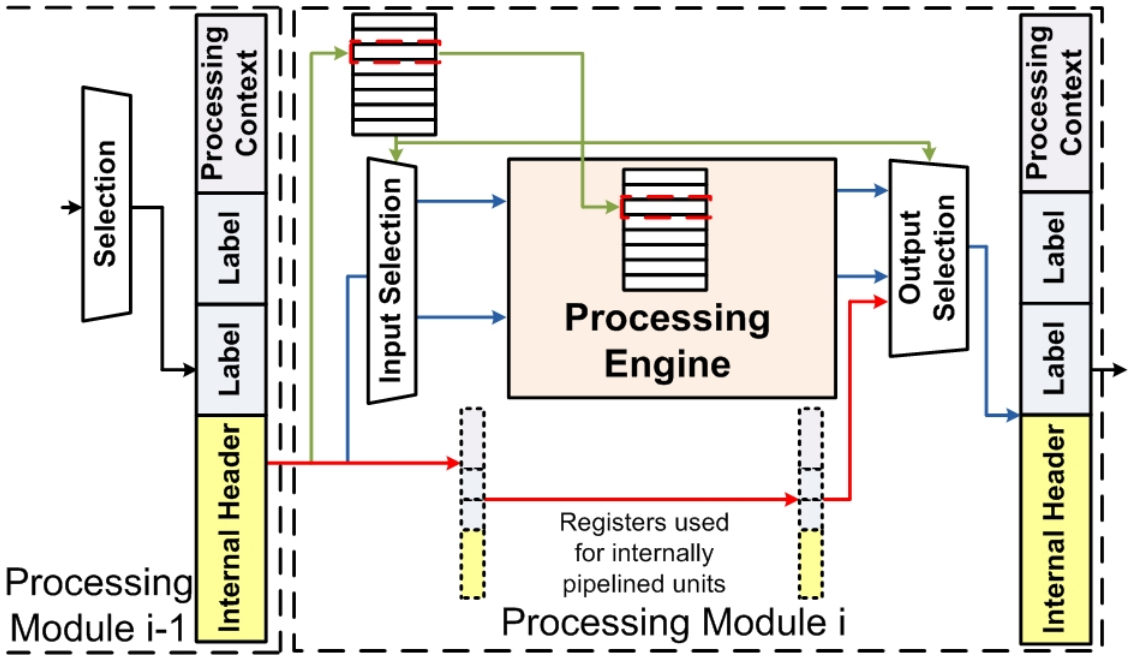


Fig. 4.19: ISU bit Selection and Field Extraction, Followed by the Processing of Data in the PE and Recombination of Processing Results and Original Register Data in the OSU

To this end, there are two critical parameters which determine the complexity of the ISU and OSU:

- the number of separate fields to be read from or written to the pipeline word. Separate is the operative word here, since a contiguous piece of data read to or written from the data structure is counted as one field, even if it can typically consist of more than one protocol field (e.g. the label and CoS fields in an MPLS label can be treated as one field from the ISU/OSU point of view). This shifts some of work to the programmer since he has to define the pipeline word fields in such a way, so as to fulfill this requirement (see section 4.3.3). How this works, is illustrated in figure 4.20, where the writing of the results of the forwarding lookup (see subsequent section) to the pipeline word by the OSU is shown. There are two input fields coming into the OSU from the PE, the first of which includes one 32 bit field (the output ports) and the second which includes several contiguous fields of a total length of 39 bits. In each input there is a part which is unused. The OSU trims this part and writes the two fields in the required location, handling the 39 bit field aggregate in input number 2 as a single field.

- granularity of a write/read. This refers to how fine-grained the definition of the size of the read/write, as well as its location within the pipeline word are when accessing them. In practical terms a granularity of one bit means that each location can be addressed separately, whereas a granularity of two bits would mean that every other bit can be addressed and a granularity of three bits that every fourth location would be addresable. Thus the smaller the value, the better the precision of each access.

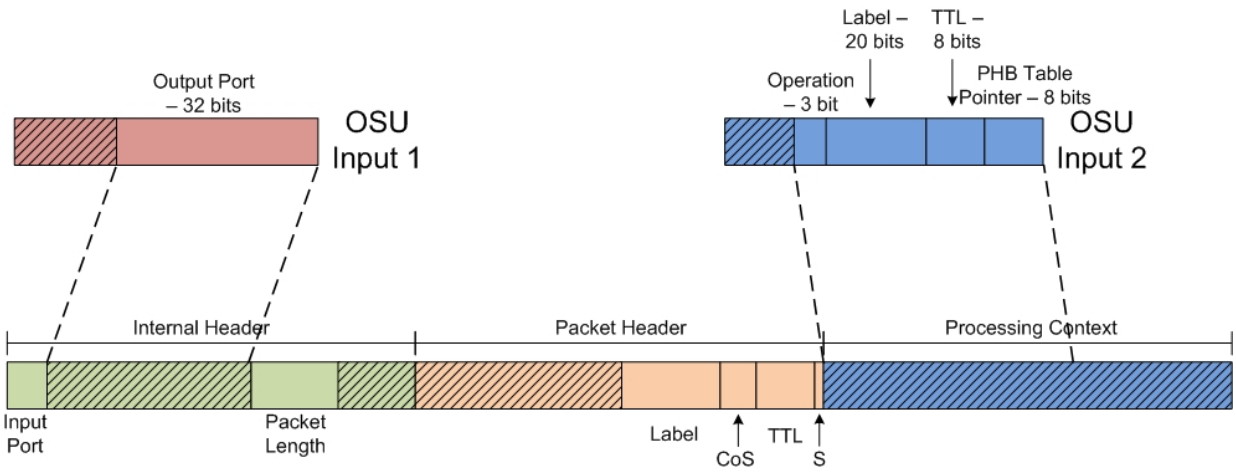


Fig. 4.20: Example of the Processing Results Being Reinserted into the Pipeline Word by the OSU

To size these two parameters appropriately, we need to investigate the requirements of the MPLS-TP and PBB-TE protocols and then perform any necessary generalizations, while at the same time balancing this with the need to maintain adequate circuit performance.

The widest data used as input in any MPLS-TP operation are the 20 bits that make up the label field, which is used as an input to the forwarding lookup, whereas in PBB-TE it

is 64-bit long (the 32 bit output port field and the 32 bit port map field resulting from the filtering entry lookup), while the 48 bit destination MAC field is the longest single field that has to be accessed. Thus 64 bits must be able to be read from the register and send to the processing module.

The output must be wider since a lot of information is stored into the pipeline word as a result of the various lookups. The longest MPLS-TP chunk of data that must be written is the result of the forwarding lookup, which is 71 bits long and contains the fields shown in the example of figure 4.20. The longest PBB-TE data segment is the 64 bits result of the forwarding lookup, which consists of the output port map and a filtering map, both of which are 32 bits wide.

The most fragmented writes and reads for both MPLS-TP and PBB-TE protocols consist of two data chunks. Thus no more than two contiguous pieces of data, along the lines described in the preceding example, need to be accessed when writing to or reading from a register.

Sizing the ISU, OSU and microcommand table represents a critical design decision which must balance between resource consumption and flexibility. Providing for excessively large microcommand tables for example will provide more functional alternatives but will require more memory to store, as well as more power and more complex wiring. Respectively, the number and width of different fields to be extracted from and reintegrated into the pipeline word have an influence on the complexity of the ISU and OSU as described previously. To this end, various combinations regarding the number of inputs or outputs (for ISU and OSU respectively), their width and the granularity of the operation were implemented and compared against the protocols requirements specified previously in order to carefully balance between the requirements on the one hand and the need for flexibility on the other. In the implementation a Virtex 5 LX330T FPGA was used as a synthesis target. This was done to derive numbers as a basis for a quantitative comparison, without detracting from the feasibility to implement the NPU100 architecture both as an FPGA and an ASIC.

Table 4.2 provides the resource consumption and path latency implementation results for various ISU combinations. Granularity values of 2 and 8 bits were also tested but are omitted from the tables for brevity. Their values demonstrate a more or less linear behavior with both FPGA LUT number and the length of the combinatorial path decreasing as the granularity increases. A first observation is that the maximum combinatorial path does not vary significantly, as the biggest deviation between any two implementations is 1.4 ns. It is actually somewhat reduced as the size of the input fields increases, which is reasonable since the result of the synthesis is one multiplexer per input and the width of this multiplexer is inversely proportional to the size of the input. This is because the more bits have to be selected, the less possible combinations are there to select from (e.g. when 3 bits have to be selected from a 6 bit signal, there are 4 possible alternatives, whereas when 2 bits have to be selected there are 5).

Hence resource consumption proves to be the deciding factor here. One must take into account that an ISU and an OSU module will be found in any pipeline stage, which means that even a small difference can have a significant impact in the big picture. We settled on a two 48 bit input with a granularity of one for ISU, since this satisfies MPLS-TP and PBB-TE requirements, while keeping resource consumption near the minimum possible (in

No. Of Inputs	Input Width	Granularity	No. of LUTs	Combinatorial Path(ns)
1	48	1	484	2.067
1	48	4	300	1.850
1	64	1	581	2.150
1	64	4	420	1.962
1	96	1	627	2.024
1	96	4	427	1.625
2	48	1	689	2.151
2	48	4	597	1.650
2	64	1	1152	2.150
2	64	4	840	1.937
2	96	1	1252	1.849
2	96	4	796	1.623
3	48	1	1330	3.053
3	48	4	912	1.950
3	64	1	1738	2.103
3	64	4	1260	1.862
3	96	1	1883	1.949
3	96	4	1223	1.623
4	48	1	1754	2.141
4	48	4	1224	1.930
4	64	1	2310	2.286
4	64	4	1680	2.062
4	96	1	2512	2.024
4	96	4	1650	1.623

Tab. 4.2: LUT Use and Combinatorial Path for Various number of Inputs, Input Width and Granularity Combinations in the ISU

No. Of Inputs	Input Width	Granularity	No. of LUTs	Combinatorial Path(ns)
1	48	1	1868	4.026
1	64	1	1966	4.261
1	96	1	2022	4.334
2	48	1	2783	5.155
2	64	1	2977	5.085
2	96	1	3083	5.425
3	48	1	3706	7.793
3	64	1	3986	7.899
3	96	1	4155	8.301
4	48	1	4641	11.558
4	64	1	5012	13.012
4	96	1	5240	13.492

Tab. 4.3: LUT Use and Combinatorial Path for Various Number of Inputs, Input Width and Granularity Combinations in the OSU

implementations with three or more inputs, it tends to grow out of control) and allowing for enough flexibility by keeping the single bit granularity. This essentially allows the module to probe any 48 bit part of the pipeline word.

A similar investigation was performed for the OSU, the results of which are summarized in table 4.3. Granularity was here always set to one due to protocol limitations (e.g. the need to set or reset the S bit in the MPLS-TP label accordingly requires that every bit location is accessible separately and masking this into the PE itself would complicate the PE and its programming respectively). Here, we can see that both LUT consumption and the path length are highly dependent on the number of inputs, whereas the length of the inputs has little effect on both. Taking into account the protocol requirements, which dictate that at least two different 48 bit inputs need to be available as described previously and the need to minimize resource consumption and latency, this is the number of choice for the OSU.

The table, which holds the microcommands, must also be considered. An 8 entry table is used, with 44 bits per entry, which are used to control both the ISU and the OSU operations. Eight entries are used since they are enough to support all possible MPLS-TP operations (as described in table 4.1). PBB-TE requires no operation differentiation, as the same sequence of processing steps is performed for each frame. Figure 4.21 shows the structure of a single entry, in which 16 bits are used for identifying the start locations of the two 48 bit fields extracted by the ISU and the remaining 28 bits are used for identifying where the OSU places the processing results, by specifying the location in which each field is inserted in the pipeline data word, as well as the number of bits that are to be inserted. Each microcommand table, along with the circuitry for writing into it and selecting the appropriate entry consumes 296 slice LUTs and 360 slice registers.

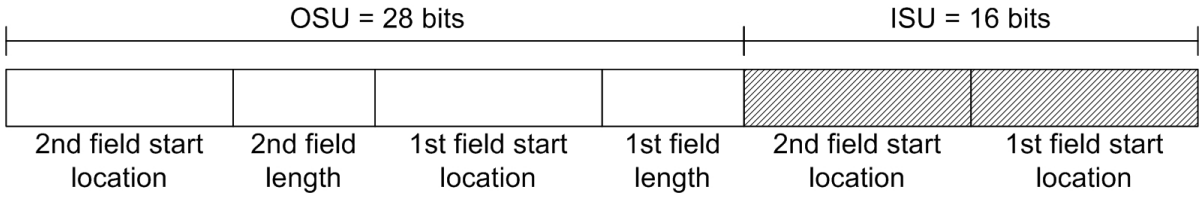


Fig. 4.21: Microcommand Table Entry Structure for the ISU and OSU

Forwarding Lookup Module

The forwarding lookup is without a doubt the cornerstone operation in a core network. In its simplest form it performs an access to a memory from which it fetches data corresponding to the network address of each incoming packet. This data contains most of the information that is necessary for further processing and forwarding of this packet. There is however more to the design of an efficient lookup module for network processors than meets the eye.

The first step in designing the Forwarding lookup module for the NPU100 was to have a detailed look on the requirements that it must satisfy. Indeed the yield from such an investigation, provides some interesting conclusions.

Perhaps the CGE protocol characteristic with the greatest impact on the forwarding lookup is the flat addressing space. This is in strong contrast to IP addressing (see [7], [5] and [57] for IPv4 and [47] for IPv6 Addressing), which is hierarchical and thus creates the need to perform longest prefix matching, that is to find the address in the memory with the most matching bits (and respectively the one with the least don't care bits). The complexity of the problem is made evident by the literature it has spawned. The literally thousands of solutions that have been proposed will not be recited here, as their relevance to the NPU100 is minimal.

The flat address space combined with its large size (up to 2^{60} entries for PBB-TE) creates an ideal environment for applying hash algorithms to map the complete address pool to the ones available in memory. Again the literature is rife with very complex hashing schemes, designed to squeeze maximum efficiency from the memory used. In this design such complex contraptions are eschewed and the focus falls on designing a simple, streamlined hashing scheme, which is proven to be more than enough for the system's requirements.

Directly coupled with the above issue, is the small required routing table size. This is a core concept of CGE protocols, which were especially designed with simplifying the address structure in mind, in order to reduce the number of addresses each node has to be aware of. Different protocols go about this with varying mechanisms. MPLS-TP uses label values of local significance. Thus a node only needs to know the label values of the LSPs belonging to its own neighbors. Since especially in core networks the number of neighbor nodes is relatively small, this number will typically be in the order of some hundreds of labels. PBB-TE on the other hand encapsulates the client header in a core network one, thus reducing the number of addresses the core network nodes need to keep track off.

One final feature that needs to be realized in the NPU100 is support for multiple entries

per packet network address. All of the matching entries should be returned by the lookup module, and a decision is to be made depending on various parameters. This enables fast redirection, by allowing the administrators to preemptively provision alternative paths for a packet flow, which is then called upon when the standard path fails without the need for management plane intervention. This allows for almost seamless path restoration.

For the NPU100 we designed a lookup architecture which tries to satisfy all these requirements. The resulting lookup module is illustrated in figure 4.22. It is based on the FlashLook system presented by Bando et al. in [16], with some important modifications to adapt the rather more complex implementation described therein to the CGE environment for which the NPU100 is targeted. Given that the Flashlook architecture was designed for IP lookups, which are considerably more demanding due to the hierarchical nature of IP addressing and the longest prefix matching employed therein, our approach simplifies the Flashlook concept by using one hash function and discarding IP specific functionality like the Verify Bit Aggregation.

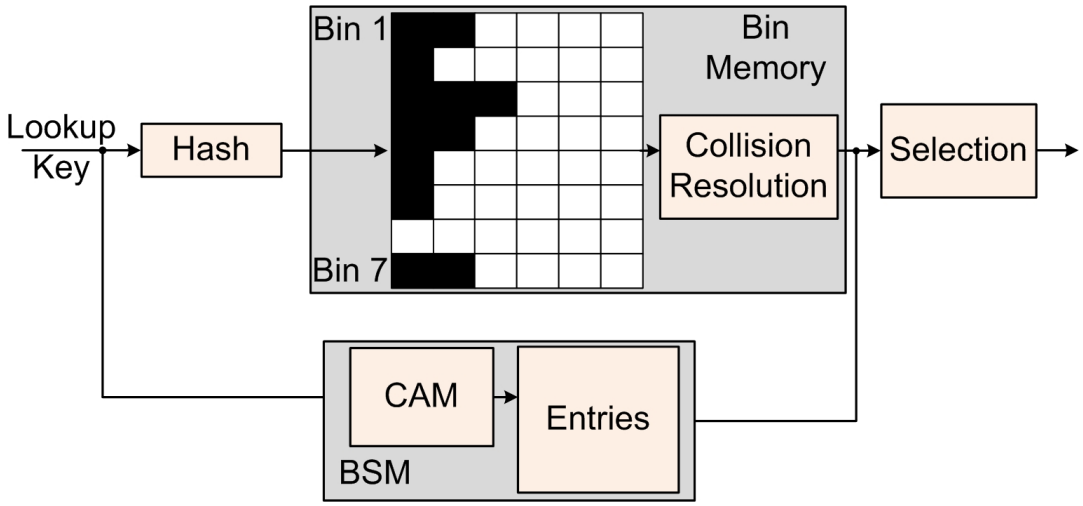


Fig. 4.22: Block Diagram of the Forwarding Lookup Module

It consists of two memories, the Bin Memory and the Black Sheep Memory (BSM), which operate in a complementary fashion to maximize efficiency. The Bin Memory contains all the bins in which lookup entries are written depending the bin to which they hash. The BSM is a CAM, which is used to store any entries which can't fit in the CAM memory due to hash collisions. Both memories contain 156 bit wide entries, whose structure is user defined through a configuration file (described in section 4.3.3). The only limitation is that the 60 MSBs make up the network address to which this entry corresponds. The remaining 96 bits can be used to store the forwarding data.

The forwarding lookup process is performed as follows:

- The lookup key is sent in parallel to the hash module and to the BSM.
- The hash module reduces the lookup key according to its hash algorithm. Hash as a concept was detailed in 2.6, while the specific NPU100 implementation is explained further along in this section.

- The result of the hash points to the bin to be accessed. The matching entries from that bin are sent to the collision resolution module.
- This module then sorts out the entries, which belong the input lookup key (since MPLS-TP supports having multiple entries for the same network address) from the ones which were the result of hash collisions.
- Concurrently to this process, the BSM is looked up and returns one entry matching the input key provided at the Forwarding lookup module's input.
- The Bin and BSM memory lookup results are aggregated and send to the Selection block which then uses a variety of parameters to pick out one entry and present it to the output.

The Forwarding lookup module is internally pipelined for performance reasons. As shown in figure 3.7, it is divided into three stages. The first includes the hash module (and ISU at the modules input of course), the second the memory accesses for the bin and BSM memories respectively, and the last, the selection stage and the OSU to write the data into the pipeline register.

Hashing Hashing was introduced in section 2.6. This paragraph elaborates upon the specific circumstances found in the design of the NPU100 and accentuates aspects of hashing, which are especially pertinent to it. What this translates to in practical terms is the selection of an appropriate algorithm for the NPU100's Forwarding lookup Module and subsequently the determination of an appropriate memory organization for it. All of these parameters are tightly interlocked, since the performance of one hash algorithm affects the number of bins and the number of entries per bin that optimize performance as well as the size of the BSM. At an even more fundamental level, more complex schemes could come into play, using for example multiple hash functions in parallel to optimize memory efficiency (see [21] for an example of this method).

At this point, it is necessary to define what is to be understood under performance of the hashing algorithm. There is no golden rule here, since the extreme diversity of applications in which hashing is used, has given birth to an almost equal number of metrics that are used to judge its performance. Complexity is always a factor and the function selected must be able to be implemented in such a way, that it can produce a new hash result in every clock cycle. One other key parameter that must be weighted in is the memory efficiency of the algorithm, meaning how many collisions it generates for a given data set. This is important since resolving collisions is costly either in time or in resources depending on the collision resolution algorithm used.

When studying hash algorithms it is important to identify any intrinsic properties of the data one has to work with. The data in our case are the network addresses, which form the input of the hash algorithm. Network addresses typically have a large range (e.g. 4 billion in IPv4 or 1 million in MPLS) but only a much smaller number of these addresses is being ever used in parallel at any given time. In core networks in particular this number is even smaller, since one of the core design goals of these types of networks is the minimization of the number of addresses used in each node through various techniques, as described in

section 2.2. Furthermore, memory is limited and usually much smaller than the available pool of network addresses. Hence an M:1 mapping must be used in this environment.

Given the diverse fields where hashing is applied it is no wonder that the are literally thousands (and this is perhaps a gross understatement) of hash functions. The vast majority of those is not of interest for our investigation since they cannot be applied to the data sets used in networks processing (e.g. cryptographic hash functions, which are optimized for variable length inputs and perfect hash functions, which are best suited to static data sets). Furthermore, a hash algorithm's performance is unequivocally tied to the architecture in which it is used. To investigate these intricacies, a C++ program was developed, which simulated the assignment of entries to a bin via various hash algorithms. The investigated case was simple: random addresses within a specific range were generated and then written into the memory, the memory assigned the entry to a location either in the bin memory or in the BSM, imitating the write process used by the Forwarding Memory Write Control module described later on in this section. Using random addresses as the input stimuli imitates the behavior in a realistic manner since network addresses in networks are distributed randomly. This was repeated until the memory overflowed, that means until a write attempt was done and both the required bin and the BSM were full. The occupancy percentage of the memory at this point was the parameter of interest. Both the size of the bin memory and the BSM were taken into consideration when calculating this. To mitigate the effect of the random address selection, each variation and hash algorithm was run 256 times and the results were then averaged in order to smooth out any one-time effects that might present themselves.

Three typical hash algorithms were simulated:

- Modulo hashing, the simplest but still widely used algorithm, which simply performs a modulo operation, with the remainder as the result.
- ELF algorithm, a 32-bit variant of the well know JPW algorithm described in [11].
- Jenkins one-at-a-time algorithm [35], which exhibits good, avalanche behavior for random, variable size input keys.

The most interesting from the results shown on figure 4.23 is that there is a common trend in the behavior of the hash algorithms, which is not affected by the memory organization selected. The memory organization itself will be discussed in the next section. This trend shows that there are marginal if any differences in memory efficiency between the modulo and ELF algorithms, while the Jenkins algorithms always trails somewhat behind. As a result of this the most simple function, the modulo one, is selected and implemented in the NPU100. This functions practically truncates the lowest part of the input key so that it matches the width of the output key. In NPU100 this means that the 48 lowest bits of the 60 bit wide PBB-TE key are be discarded and the 12 highest bits are used as the network address. Similarly for MPLS-TP, the 12 highest of 20 bits of the label field are used.

Memory Organization Apart from determining hash algorithm performance, the simulations performed offer valuable clues pertaining to the optimal memory organization,

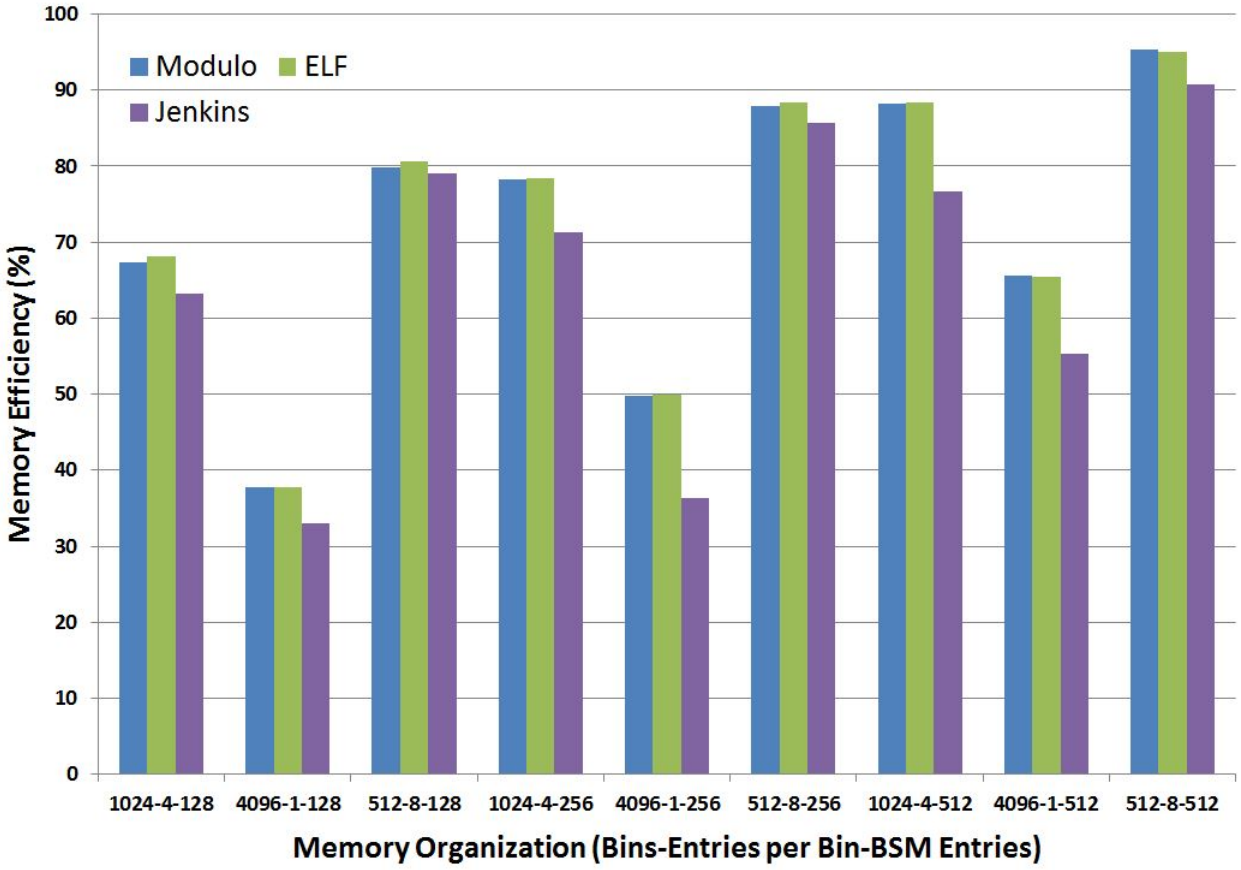


Fig. 4.23: Performance of Three Prominent Hash Algorithms for Various Bin and CAM Sizes

which should be used in our implementation, that is, the number of bins, as well as the number of entries per bin. It should be noted that the starting point were combinations of the previously mentioned parameters, which were feasible to implement in a contemporary FPGA. The trends shown however should have wider applicability. In the tests performed three configurations were investigated. All of them are able to accommodate a total of 4096 entries in the bin memory, plus an additional number of entries in the BSM. 4096 entries were selected, since this number was judged to be sufficient for a core network router (see chapter 2 on a discussion on why this makes sense in CGE protocols). The bin memory was distributed in three different configurations, namely 4096 bins with one entry per bin, 1024 bins with 4 entries per bin and finally 512 bins with 8 entries per bin. This nomenclature is depicted as a combination of three numbers on the X axis of figure 4.23. The first number shows the number of bins, the second number the entries per bin and the final, third number the number of entries that can be stored in the BSM.

The foremost conclusion from figure 4.23 is that the number of entries per bin plays a pivotal role in the number of entries that can be accommodated. This is independent of the other parameters, though the difference between the 4096-1 and 1024-4 combinations is much bigger than that of the 1024-4 and 512-8 ones. Thus more entries per bin allow for better bin memory utilization. The only thing that can act detrimentally to an 8 entry bin memory is the required increase in the complexity of the collision resolution block in order

to process the extra entries. Synthesizing the two variants, one can see that the increase is less than 350 slice LUTs in a Virtex 5 FPGA (368 for the 4 entry collision resolution block against 704 for the 8 entry one). Thus the 512 bin and 8 entries per bin scheme is selected as the most appropriate one for the NPU100

Despite the memory being configured in a specific way following the rationale described in this section, generics allow this configuration to be changed. Thus both the number of bins and the entries per bin are configurable, and thus the size and organization of the memory can be adapted accordingly to each design's needs. How the memory is distributed affects the number of entries that will be able to be accommodated into it. More bins with less entries per bin allow for accommodating more entries when these entries all hash to different values, wheres the opposite organization allows to store more colliding entries. This can only depend on each specific use-case however. This investigation considered a general case, which might not apply to all scenarios, thus the capability to fine-tune the design is provided. Furthermore, implementing the design for different technology types (e.g. ASIC libraries or different FPGAs offering more/less on-chip memory) might require that a different bin/entry per bin mixture is selected to optimize performance.

Black Sheep Memory The Black Sheep Memory is a small Content Addressable Memory (CAM), whose purpose is to store entries which belong to overflown bins, thus mitigating the effect of hash collisions. Though it is evident from the hash algorithm comparison, whose results were shown on figure 4.23, that a larger BSM will increase performance, other characteristics, like the resources that the Black Sheep Memory consumes come into play and force us to investigate where the optimal trade off between these two parameters lies.

The BSM block consists of two submodules, the Address Mapping Module and the Entry Memory, as shown on figure 4.24. Both modules consist of two generated Xilinx IP Cores, however configuring this core correctly to balance between performance, optimal memory size and resource consumption was tricky, as CAMs come in several different sorts with different feature sets, which has an impact on all these factors. Keeping the CAM memory size and power consumption under control was critical, since CAMs have been known to be extremely power hungry and exhibit a low bit to area density.

The most important characteristic, which sets CAMs apart, is whether they are ternary or binary CAMs. The former can take up three possible values (0, 1, and X called a don't care value), while the latter can only use the typical 0 and 1 values. To accommodate for the extra value, ternary CAMs use two memory cells for each bit, thus the memory area occupied by them is significantly higher than binary CAMs. Ternary CAMs are commonly used in IP lookup applications, where don't care bits are used to match specific address ranges and thus enable functions like longest prefix matching. In CGE networks, address mapping does not require such functionality and hence a simply binary CAM is used. A further feature that has to be defined is how to react to multiple matches. CAMs offer several different alternatives, of which the Xilinx IP core used in this implementation only supports selecting the lowest or the highest address that matches. The lowest variant is used here.

Thus one of the salient issues to be clarified in the implementation of the Forwarding

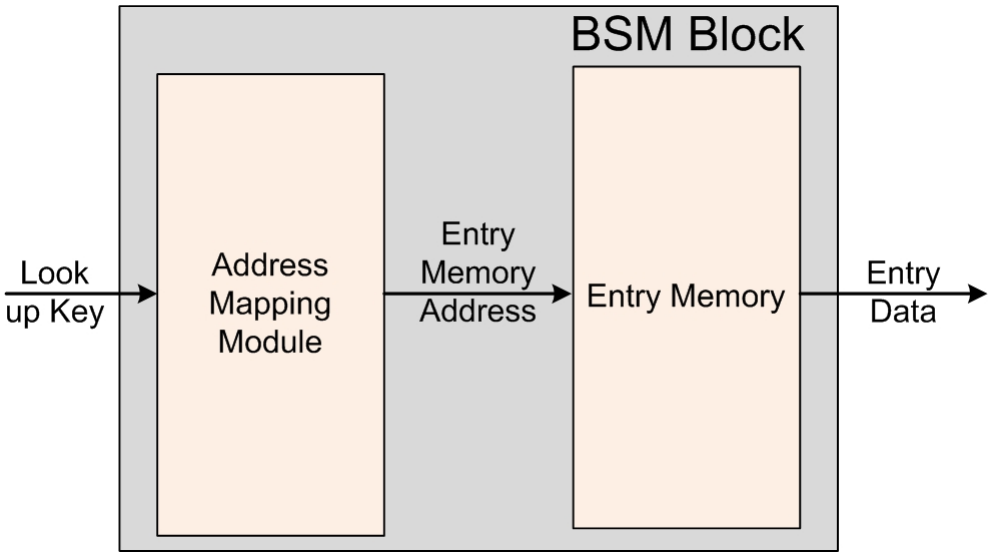
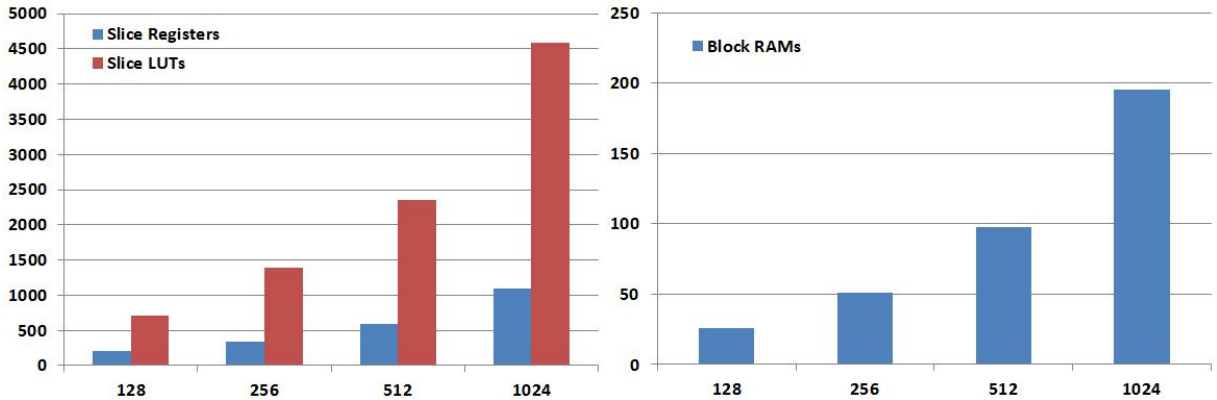


Fig. 4.24: Block Diagramm of the Black Sheep Memory Block

Lookup module is how big of a CAM can be accommodated in an FPGA and ASIC respectively and how many NPU100 forwarding lookup entries can fit in such a CAM. Figures 4.25a and 4.25b provide information on the resource consumption and performance of various CAM sizes. The CAM block implemented in the NPU100 includes a CAM with a 60 bit lookup key (derived by the combination of destination MAC and VID as found in the PBB-TE header), which leads to a memory, each entry of which is 96 bit wide (wide enough to accommodate all the data that need to be retrieved from the lookup table. More details on the structure of this entry are provided in section 4.3.3. This stems from MPLS requirements and includes a new MPLS label, a new TTL value, a pointer to the PHB table, as well as a port map showing the output ports to which this packet must be sent) . The number of entries varies in each scenario.



(a) Slice Register and Slice LUT Resources Consumption for Various CAM Sizes

(b) Block RAM Resource Consumption for various CAM Sizes

Fig. 4.25: FPGA Resource Consumption for Various CAM Sizes

While, as was to expected, the values of FPGA resources increase with the size of the

CAM, it is clear that the critical parameter is the number of Block RAMs consumed. Considering that the largest Virtex 5 FPGA, the LX330T, which is the one used for the NPU100 prototype, contains 207360 slice registers and slice LUTs and 324 Block RAMs, one can see that the bottleneck in accommodating multiple CAM modules in the device is the number of available of Block RAMs. In an ASIC design there is not hard limit on the number of available resources however similar limitation exist, since cheap real estate is in practical scenarios limited by cost. More specifically only a CAM block of up to 256 entries can successfully fit four times (as many as the MPs) into the device and even in this case 200 Block RAMs (61.7% of the total in the device) would be used leaving 124 empty for other uses. The performance achieved by each CAM is also dependant on it's size, although the impact is marginal. Hence the smallest CAM (128 entries) achieves a frequency 128.8 Mhz while for the largest one (1024 entries) this drops to 103 MHz., both satisfactory for an FPGA implementation.

Next, it is important to determine the impact of the size of the CAM block on the number of entries that can be accommodated in the forwarding lookup module. To do this we use the data from hash algorithm investigation in the previous section. In figure 4.23, one can clearly identify the impact of the size of the CAM block to the number of entries that can be stored in the memory. This trend is quite independent from the algorithm used, as well as the structure of the main memory in regard to the number of bins and entries per bin. What the figure also illustrates, is that 256 entry CAM blocks provide tangible benefits in comparison to 128 ones, but as is made obvious from the resource usage requirements it is impossible to fit into a contemporary FPGA (taking into account that there are other modules requiring significant amount of Block RAMs). Thus a 128 entry CAM block was used for the NPU100 test bed implementation. Of course a future ASIC implementation may well increase this number according to the available resources. The NPU100 can be easily adapted to this, as the size of the BSM is configurable at design-time through a generic. Since however larger CAMs sizes gobble up resources, this feature should be increased with caution. If an ASIC is to be used for the implementation than many more entries can be accommodated, as ASICs typically exhibit 20-40 times higher density than FPGAs [41]. Even in this case the resource requirements for a 4096 times would exceed anything that can be fabricated economically with contemporary technologies.

Selection Stage Path Protection, among other OAM functionality is a cornerstone functionality of carrier grade protocols. Path protection means that in case of a link failure in the path, the protocol must provide a mechanism to swiftly redirect packets over an alternative path. This requirement, coupled with several additional ones, such as management plane output filtering (allowing the management to disable and enable specific output ports through the management interface), creates the need for a selection stage, which allows the realization of this functionality based on a variety of criteria. It receives the forwarding lookup results, which may consist of more than one entry, which matches the packet's address, from the memory and selects the appropriate one according to a set of parameters. Each of the different entries represents an alternate path for packets belonging to this aggregate flow. The parameters used during the selection process fall into a set of 6 configurable criteria, which are broadly classified into two types:

- Availability-based criteria, which are only meant to show if a port is a viable output alternative for this packet. This can be used for port filtering as done in PBB-TE or to demonstrate port availability. This is achieved by providing a port map with the valid output ports for this packet and comparing it with the output ports read from the lookup memory in a previous step in this module. Two different types of sets are provided against which to compare the output ports, a pair of sets which are meant to be programmed by the management plane and another pair of sets which are sampled from external pins of the device. The former can be used by the management plane to route a packet flow to specific ports and to avoid a specific routing path, while the latter can be used as input from the lower layers to indicate port availability, so as to automatically avoid severed routing paths which may temporarily be out of service.
- Priority-based criteria, which can be used to determine the relative importance of the paths in comparison to each other and use this information to decide which path to sent the packet down to. There are two different types of priority-based criteria:
  - Entry-based path weighting. Each forwarding lookup entry returned in the lookup step, includes a priority indicator. These are compared with each other and the entry with the largest one is selected.
  - Port-based weighting. The selection submodule includes a table which holds the current relative priority weights of each port. This is programmed by the management plane over the PCIe interface. The weights for the destination ports in each lookup entry are summed up and compared with those of the other entries. The entry with the largest total value is selected.

Each of the sets can be activated or deactivated independently of the other, either on an NPU or on an entry basis. Thus the sub module itself contains a table, which determines which criteria are active and each lookup entry includes a similar table which is only relevant for this entry. The former supersedes the latter and furthermore for a criteria to be taken into account in a specific selection process all entries pertaining to that process must have this criteria marked as active.

All criteria are evaluated in parallel and then these results are evaluated again in order for an entry to be selected. This is done in a complex series, which mirrors the purpose of criteria. The availability based criteria are evaluated first, starting with the ones based on external stimuli. In this case, entries in which all output ports are available are kept in the race. If none such entry exists, all entries are maintained and considered for the management plane defined criteria. The first stage operates in a similar fashion as the external stimuli based ones. Entries who have all ports qualified by the comparison are kept, unless none does, in which case all are maintained. The second stage is used as a filtering one, thus essentially a logical and operation is performed between the two port maps, so that the output ports that remain active after the operation are the ones desired by the management plane.

The priority based criteria are considered only for packets which survive the previous four steps. This is logical since the previous stages are based either on physical layer limitations or on explicit commands from the management plane. The two priority based stages are used to whittle down the remaining entries and select one according to the

scheme defined previously. If both priority based criteria are active then the entry-based weighting is preferred over the port-based one.

Forwarding Lookup Write Control The forwarding lookup functionality is indispensable in any NP and the system described thus far in this section fulfills the requirements of CGE protocols in this area. The last, missing piece of the puzzle is providing the means to write and delete entries from this memory. The required software part of this process, which allows the user to create the entries and send them to the NPU100 is described in section 4.3.3. This section describes the hardware module that complements this procedure on the FPGA side. It receives a request to write or delete an entry and performs the task accordingly.

Writing and deleting entries entails keeping track of the memory's contents and accessing them appropriately. The whole process must be abstracted from the control plane, which only supplies the network address to which this data belongs, without knowing how the forwarding memory is internally organized. Thus it is up to the write control module to determine if the bin memory or the BSM is to be used and in any case, where exactly the entry must be written to. The tasks that must be performed are:

- Check if the required bin has a free entry, and if so, write the new data into the bin and internally note that this entry is now occupied.
- If the required bin is full then write the data into the BSM and record this internally.
- If the BSM is full as well, return an exception.

Deleting an entry involves a similar process:

- Determine the location in the memory, in which the entry to be deleted resides. This can be either in the bin memory or in the BSM.
- Internally mark this entry as empty, and thus as available for reuse.
- Erase the data from the entry in the memory.
- Return an exception if the entry does not exist in the forwarding lookup memory.

It is obvious from the above description that the write control module needs to maintain state information regarding the occupancy of the memory, lest it queries the memory itself for the required information. This would however reduce the availability of the forwarding lookup memory for serving packet header queries and thus could result in performance loss. Thus the write control module needs to internally maintain the following information regarding the current status of the forwarding lookup memory:

- A table in which the status of all bins is contained. Detailed information is not required, just which entries in each bin are occupied and their addresses.
- A table which tracks if the BSM has free entries and can return the next available entry.

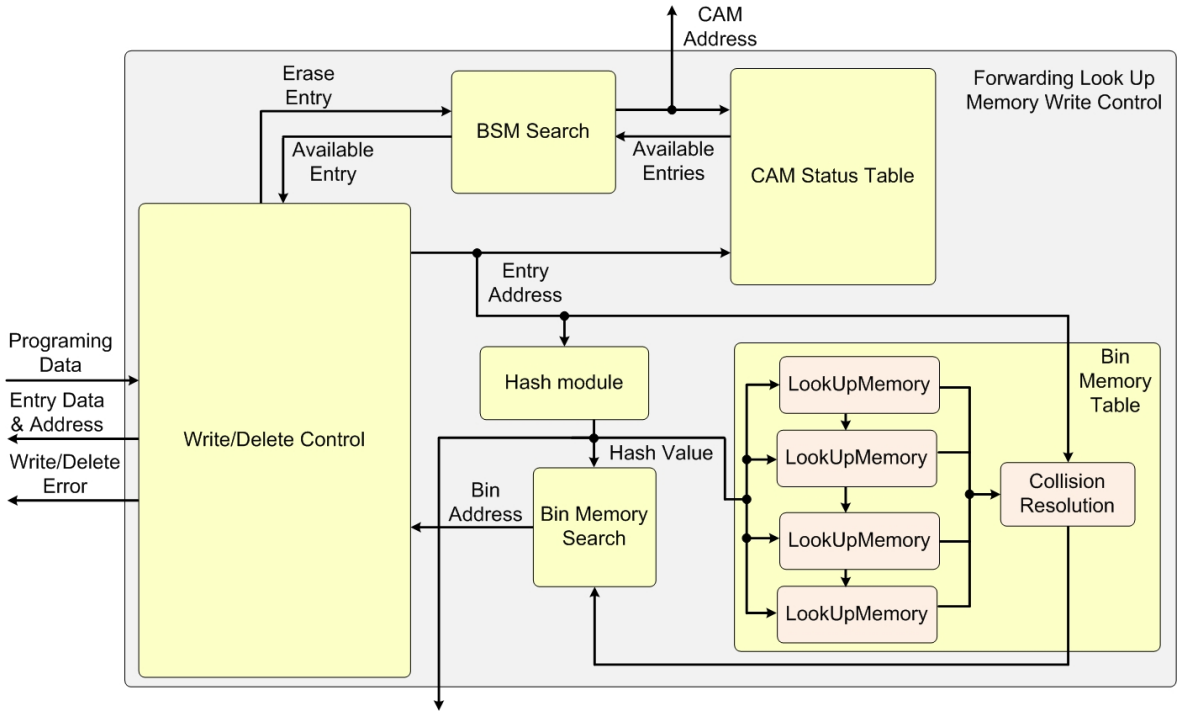


Fig. 4.26: Block Diagram of the Forwarding Lookup Write Control Architecture

Both above mentioned tables do not contain the entire entry, but simply the corresponding network address, which is used to match the entry with the query input and determine in which location in the memory this network address has been stored. The bin memory table is two dimensional, with each row corresponding to a bin and the number of columns to the number of entries in each bin.

Apart from the bitmap the write control submodule must also be equipped with a hash input in order to determine to which bin an entry belongs. Since this must be done in parallel to any normal hash operation taking place in order to process packets, a dedicated hash module, identical to the one used for processing is used.

The structure of the Forwarding lookup Write Control module is shown in figure 4.26. It consists of several blocks, whose interplay ensures the correct functionality. The Write/Delete Control block receives the programming data from the PCIe interface and directs the write or delete process.

In order to write an entry into the forwarding lookup memory a complex series of steps has to be performed:

1. First of all, the data must be written through the programming data signal to the Write/Delete Control module. This takes 5 cycles in total. In the first two the network address of the entry is written and in the next three the entry data.

2. After the network address is received, two parallel queries are launched, one to the BSM Search in order to check for a free space in the BSM and one to the Bin Memory search via the Hash module. This returns possible entries in the bin memory.

3. The Hash module produces the address of the bin to which the entry has to be written to. This hash value is the fed to the bitmap, which tracks the occupancy of the bins and their entries. A four entry per bin memory is shown on figure 4.26 for clarity's sake. The results bypass the Collision Resolution block and are sent to the Bin Memory Search, which checks how many entries are there in the bin and thus deduces if there are any available ones.

4. At the same time the BSM Search block checks if there are free entries in the CAM Status table.

5. Both results, along with any required control information (e.g. the available bin address) are sent to the Write/Delete Control.

6. The Write/Delete Control evaluates these results and determines where the entry must be written to. Preference is given to the bin memory if available space is found. If not the entry is to be write into the BSM. Finally if the BSM is also full, the Error signal is asserted, to denote that the request cannot be served at this time.

7. The Write/Delete Control block then asserts the necessary signals to write the data to the appropriate entry in the bin memory or the BSM and concurrently marks that entry as occupied in the respective internal table. The process for writing these tables is identical to the forwarding lookup memory write process, the data are simply ignored.

Deleting an entry from the memory is done in the following manner:

1. The first task, as in writing an entry, reading the necessary data from the configuration interface. The process simply takes for 2 cycles, as only the address to be erased has to be transmitted.

2. Both BSM and Bin Memory Search modules are then queried at the same time to determine if the entry is stored in any of these memories and if so into which.

3. Again the address is hashed and the hash value used as input to the bin lookup as in the write process. Divergent however from that process is the fact that now the Collision Resolution block is used to return the exact entry that was looked up.

4. In parallel the BSM Status Table is checked and a possible entry is returned.

5. The results are returned to the Write/Delete Control, which then proceeds to remove the entry from the memory. Only one entry corresponding to the address is removed. If more of them are present in the memory, one delete operation for each has to take place.

6. Entries in the BSM are deleted first in a LIFO fashion. If no entry is found in the BSM, the last found in a bin is deleted.

7. If no entry is found at all, the Error signal is asserted.

8. Actually deleting the entry, requires removing the address from the BSM or bin memory and the doing the same for the internal tables. The Write/Delete Control block orchestrates this process accordingly.

The NPU100 implements one Forwarding lookup Write Control module per lookup module, which means that every forwarding lookup memory in each of the four MPs can contain a different set of entries. This is necessary to support the appropriate number of forwarding lookup entries (4096). The reason for this is not intuitively clear. Assume a packet belongs to an LSP which requires it to swap the top label and push a new one on the stack at this node. In this case, the forwarding lookup table of the first MP will contain the label to be swapped and that of the second label the label that is to be pushed onto the stack. Thus one entry each of those two table is needed for accommodating one LSP. Extrapolating for the worst-case scenario where all MPs have to be used all the time, it is evident that in order to store all the labels for all possible LSPs, one entry per lookup table is necessary.

QoS lookup Module

Lookup Module The QoS lookup module is aimed at implementing the memory accesses related to the QoS parameters of each packet in the NPU100. The architecture of this module has been influenced to a great extent by the respective MPLS-TP requirements for incoming and outgoing PHB lookup. According to the MPLS-TP protocol standard, each forwarding lookup entry might have a different CoS to PHB mapping. This means that a CoS field value which for one label corresponds to the PHB class X might for another label correspond to the PHB class Z. In order to support this functionality, the memory of each QoS lookup module is divided into subtables. Each of these subtables contains one specific CoS to PHB mapping. In order to determine which subtable has to be used for each packet a pointer is used. This pointer is part of the forwarding lookup entry, which has been read out of the memory in the first stage of each MP. It is 8 bits wide and contains the address of the CoS to PHB subtable, which must be used for this label value. Essentially it represents an offset to the QoS lookup memory. Using this offset, the appropriate subtable is read from the lookup memory as shown in figure 4.27 and then the appropriate field from the pipeline word (designated incoming CoS on the figure) is used as an index to extract the appropriate entry. The figure shows an eight entry subtable for illustrative purposes though sixteen entries are used in the actual implementation.

It should be noted that while the above description reflects the MPLS-TP process, the module is built with generality in mind. Thus in the real system, any part of the input register may be used as a memory offset, which returns a chunk of memory, which can then be accessed using a four bit vector from the input register. In order to implement PBB-TE's QoS lookup for example one has to take into account that PBB-TE uses the B-VID, a 12 bit field of the header, as the key for the lookup, but does not share MPLS-TP's separate mapping per address (or label) requirement. This means that the entire table can be viewed as one entry. Thus from the 12 bits available, eight can be used to select the subtable and then the remaining four to index the subtable and read out the required entry. ,

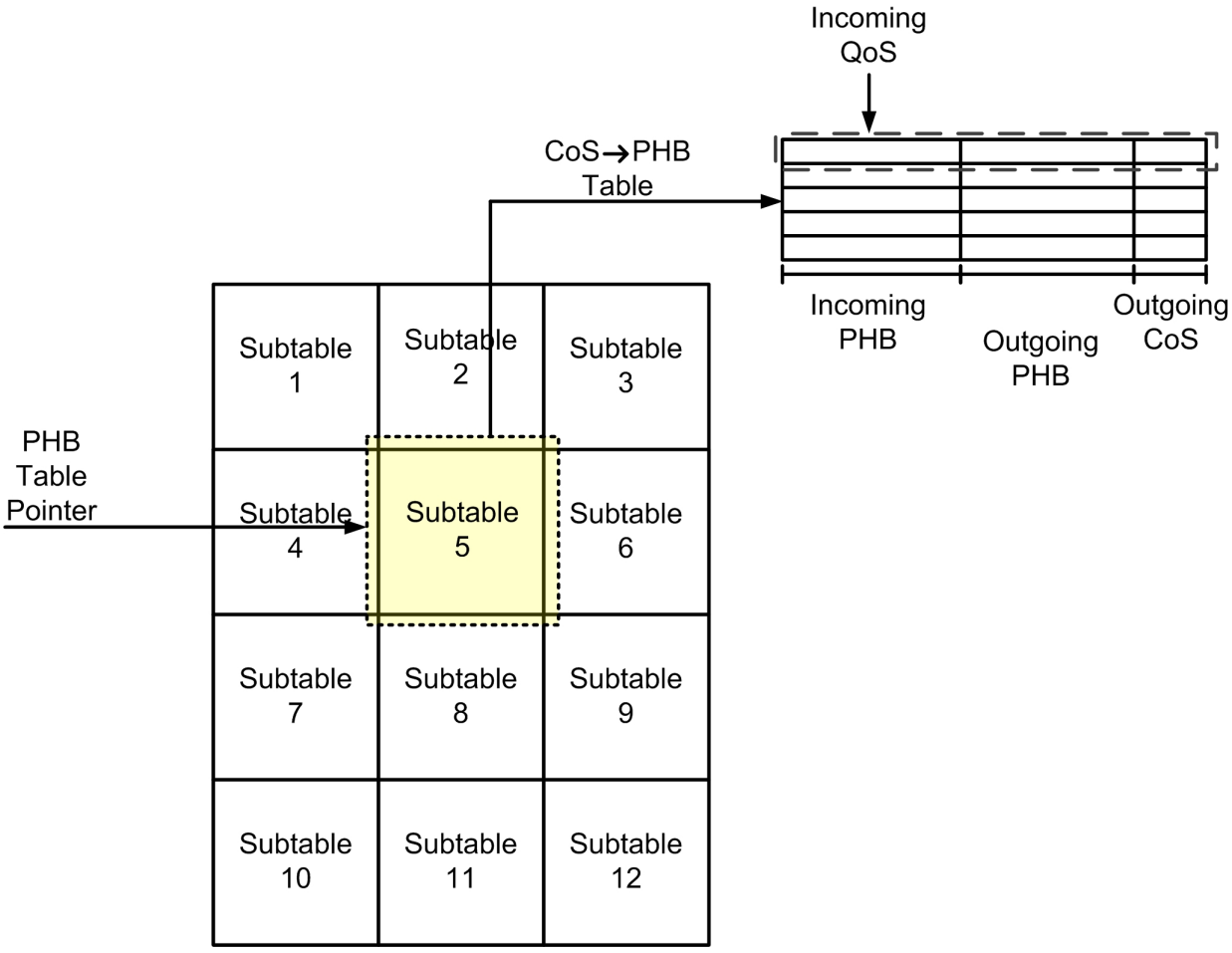


Fig. 4.27: Overview of the QoS Parameter Lookup Operation

The table sizes of 256 CoS to PHB subtables with 16 entries per table can be configured at design time if different requirements call for their expansion (or shrinking respectively). The tables are dynamically programmable as described in the following section.

QoS lookup Write Control The QoS Lookup Memory Write Control module takes care of the writing of the lookup information to the table. The information is transmitted to the module over the PCI express interface. As in the Forwarding Lookup Write Control, there is one module per table. The module is a state machine which buffers the received information when the control address matches that of the module and then writes it to the memory in one stroke. An entire subtable is the minimum amount of information, which can be written, and each subtable consists of eighth 19 bit entries, thus a total of 152 bits must be written over the 32 bit programming bus. The data transfer takes for five cycles in order to transfer the data and one additional cycle in order for the subtable address to be send. The appropriate programming address must remain set at the programming address bus during the entire process.

Replace

The replace operation writes data already available in the pipeline word in an already existing field. This field might be occupied by existing data or be empty. In any case the specific area is overwritten with the new data. In this respect, replace operations are very simple in nature. The input data is simply placed at a different part of the data structure. Thus, upon closer inspection of the tasks for this module it is clear that no further functionality is required apart from configuring the input and output selection units appropriately in order to move data from the one part of the data word to the other. In the implemented module two fields can be moved in one stroke, since the ISU and OSU can process up to two different fields independently of one another in one clock cycle.

Required MPLS-TP functionality is limited to replacing a label field during a swap operation. The new label, which was read out of the forwarding lookup memory and stored temporarily in the processing context, must take the place of the original label in the packer header. PBB-TE requires no such functionality.

Insert

While conceptually similar to the replace operation, the insert operation involves the shifting of the data previously occupying the target area in the data structure as shown in figure 4.28. The data to be inserted are to be provided at the input and the location and width of the data to insert need to be specified as configuration parameters. This data must already be stored in the pipeline data word. Additional required configuration parameters include the location and width of the data to shift, the length of the shift, as well as its direction. To realize this, the operation will be done in two stages, which essentially breaks the insert operation down into two consecutive replace operations, which should be implemented as described in the previous section, using two register selection modules in sequence. The first set of ISU/OSU performs the shift operation and the second inserts the target data into the specified location. Thus the data which the last operation overwrites have previously been moved to another part of the data structure. Practically the two sets of ISU/OSU's can be programmed to perform two consecutive replace operations. There is no inherent limitation in the way the unit can be used.

Modify

The Modify module performs arithmetic and logical operations in order to manipulate the various fields of the pipeline word. A whole array of operations is available, making the Modify module practically an Arithmetic Logic Unit(ALU). Logical operations include AND, NAND, OR, NOR and XOR, and arithmetic operations addition and subtraction. There are two modes of operation supported by the module:

- Operations between the two input fields.
- Operations between the two input fields and fixed operands. In this case the fixed operands are stored in the module's microcommand table.

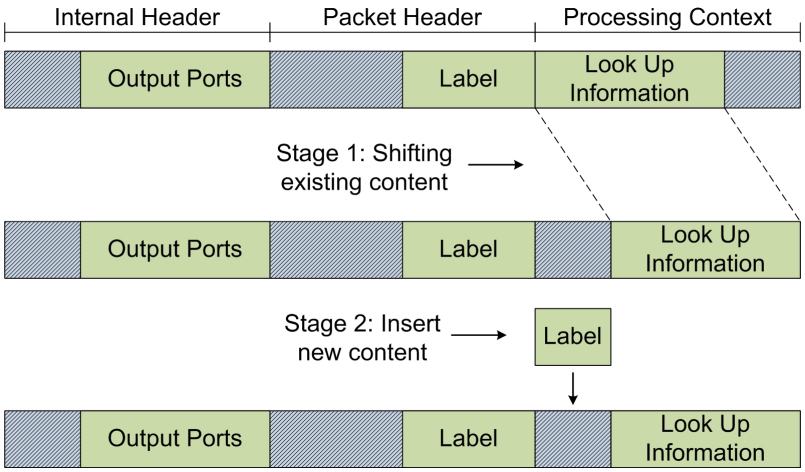


Fig. 4.28: Overview of the Insert Operation

In MPLS-TP the only required processing is the decrement of the TTL field, the recalculation of the packet length when a header is added or removed and the setting or unsetting of the S bit in case the current label becomes/ceizes to be the last in the stack. Both operations involve a pipeline word field and a fixed operand. In PBB-TE the only modify operation is the comparison between the Filtering Map and the Output Ports in order to determine the output ports, which is a simple bitwise AND operation between two pipeline word fields. We decided to allow for two operations to be executed in parallel. This provides significant additional flexibility, while sacrificing little additional resources.

Logically since there are only two input fields to the PE, the user can select either to execute one operation between the two input fields and one between one of the two and a fixed operand, or perform one operation between each of the input fields and fixed operands. There is no limitation on the combination of operation that can be executed in either mode.

4.3 NPU100 Programming Concept

One of the main contributions of this work was in programming the resulting architecture. Special attention was given to this task in order to avoid one of the typical pitfalls of NP design, that is ending up with a very potent but extremely programmer unfriendly piece of hardware. This commonly happens because most attention is devoted to the hardware side, while neglecting the software one and relying on serendipity to avoid ending up with a complex, user-unfriendly programming scheme. The approach followed was based on tightly integrating the hardware design with the development of the programming concept, meaning that the modules, their functionality and the way they were going to be programmed proceeded hand in hand from the very beginning of the design of the NPU100.

A great part of the hardware side of the programming concept behind the NPU100, resides with the ISU and OSU modules described in section 4.2.4. These modules use custom programming words, stored into microcommand tables in order to direct the pro-

cessing. The missing link is thus the means to generate the required programming words, to transfer them to the NP and to write each of them into the appropriate microcommand table.

Furthermore, the NPU100 includes Forwarding and QoS lookup tables which are used to forward the incoming packets. These tables include entries which are defined by the user of the NP (typically an administrator with an ISP or core network firm) and contain all the relevant information. Since connections are constantly being set up and torn down again all the time during the lifetime of an NP, it must be possible to write and delete entries into these tables. An appropriate interface must provided to enable the transfer of this information to the NPU.

This section provides a detailed description of the programming architecture of the NPU100, first introduced in section 3.3, which aims at providing a solution to allow for the easy programming of both the NP microcommands, as well as the Forwarding and QoS lookup tables, all while using a user friendly software, an efficient, industry standard interface to get the data to the NPU100 and finally an comprehensive system to internally distribute the data to the appropriate modules.

4.3.1 Addressing Scheme and Programming Table Access

With the one end of the programming concept (the microcommand tables) having been already introduced, the remainder of it is explained here. The first element is the addressing scheme used in the NPU100. Configuring the NPU100 involves writing programming information over the PCI Express bus to the microcommand tables which hold the program and guide the operations performed by each module. To ensure the correct flow of programming data to the appropriate unit each time, an addressing scheme has to be defined. Figure 4.29 illustrates the address structure devised to accomplish this.

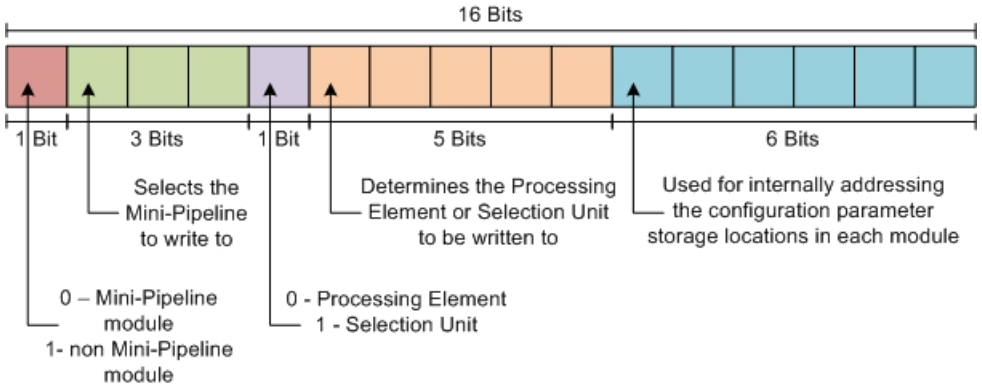


Fig. 4.29: NPU100 Configuration Addressing

The address length is 16 bits, which allows for sufficient addresses to accommodate all of the NPU100 modules and leave plenty of room for expansion (e.g. addition of pipeline stages). The MSB identifies if the module belongs to an MP or is a non MP module. Examples of non-MP modules are the Dispatcher or the Packet Reassembly module, while the Modify or Insert modules are examples of MP modules. The following three bits select

the MP whose module is to be addressed and should be set to 0 for non-MP modules. The next bit identifies if the data are to be written to the microcommand table of an ISU/OSU as described in section 4.2.4 or to that of a processing engine. The following 5 bits identify the module itself (thus 32 modules per MP and 32 non MP modules can be addressed) and the LSBs are used to identify the internal structure of the tables as described in 4.2.4 for the both ISU/OSU microcommand tables and the modify module.

This template is used to derive the addresses for each NPU100 module, as shown in Table 4.4. The table also provides the name of the generic in the VHDL code, which sets the address for each respective module. Some addresses contain Xs where a mini pipeline identifier is to be placed. In the instantiation these Xs will be replaced with a corresponding number of bits, identifying the mini pipeline to which this unit belongs. Where only 10 bits are given the remainder are used to assign values to module internal parameters.

Module	Generic Name	Address Value	Description
Forwarding Memory Write Control	RDAddress	1000000000000000	Erases an entry from the forwarding lookup Memory
	WRAddress	1000000001000000	Writes an entry into the forwarding lookup memory
Packet Reassembly	PktReAddress	1000000010000000	Configures label size and ingress or egress mode
HP Split	HPSplitAddress	1000000011000000	Sets ingress or egress mode, as well as the length of the packet header data to be copied
Dispatcher	Disp_LL_Address	1000000100000000	Sets label based or direct forwarding mode and the label size for label mode
	Disp_PBB_Address	1000000100000001	Sets label based on direct forwarding mode and extraction field location and sizes for the latter
Label Reassembly	LRAddress	1000000110000000	Configure label based or direct forwarding mode and label sizes
Forwarding lookup module	SelectAddress	0XXX000000	Determines the criteria, with which an entry is selected from several valid ones
	MergeAddress	0XXX000001	Defines how addresses are merged between results returned by the BSM and the bin memory
	NHLFEAddressSU	0XXX100000	Configures the ISU and OSU of the module
Insert Module	I_Address_SU1	0XXX100001	Sets value for the first ISU and OSU of the insert modules
	I_Address_SU2	0XXX100010	Similar for the second ISU/OSU pair
Replace Module	R_Address_SU	0XXX100011	Programs values for the ISU/OSU
Modify Module 1	M1_Address_SU	0XXX100100	Sets the ISU and OSU parameters of this module
	M1_Address_PE	0XXX000011	Sets the operation and the operands for the PE of the first modify module
Modify Module 2	M2_Address_SU	0XXX100101	Configures the ISU and OSU parameters of this module
	M2_Address_PE	0XXX000100	Sets the operation and the operands for the PE of the second modify module
QoS lookup module	PHB_Address_SU	0XXX100110	Writes the ISU and OSU parameters
	PHBWrAddress	0XXX000110	Writes an entire subtable to the QoS lookup memory

Tab. 4.4: NPU100 Configuration Address Assignment

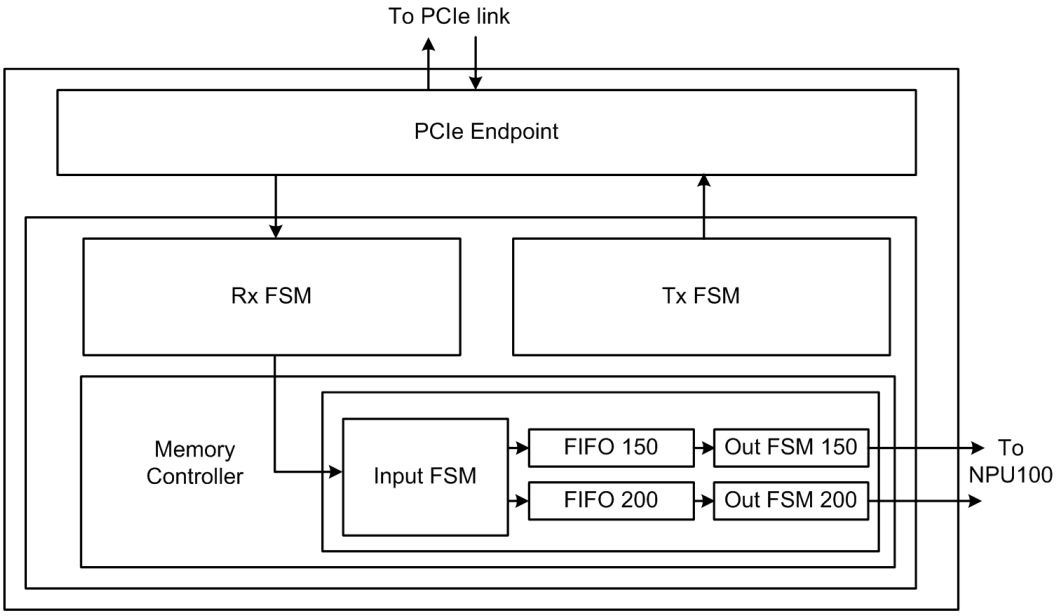


Fig. 4.30: Block Diagram of Hardware Modules Making Up the PCI Express Interface

4.3.2 Hardware Modules

The hardware side of the NPU100 programming interface consists of the Xilinx Endpoint Block Plus for PCI Express which realizes the functionality required to receive and send information over the PCI Express link and several accompanying modules which receive this data and interface with the NP modules, as shown in figure 4.30. This is accomplished by receiving the data from the PCI Express core with a state machine which then writes the data into two separate FIFO queues. The FIFO queues have a different write and read clocks, which allows us to change clock domains from the one driven by the PCI Express core clock to the clock domains used in the NPU100. As the NP uses two different clocks, one at 150 MHz and one at 295 MHz, two FIFO queues are used, one for each clock domain. Another state machine then reads the data from each of the FIFOs and forwards them to the programming buses.

The PCIe interface state machines feeds a pair of 32-bit buses, which are used to transfer the data from the PCIe block to the processing modules with each bus being connected to all the modules that belong to that particular clock domain. Each bus transfers the maximum amount of configuration data it can, without however breaking a configuration parameter into two transfers, unless the transfered parameter is longer and must spawn multiple 32-bit words. To exemplify the above, assume a scenario where all in all 55 Bits must be written, nine 6 bit fields and one 1 bit field. This should be done in two data transfers, the first of which includes five 6 bit fields and the second one the remaining four, along with the 1 bit field. The module must then internally handle the data write to the appropriate locations. The NPU100 software described in 4.3.3 takes care of this task for the programmer automatically.

Apart from the data buses, the modules of each clock domain are connected to an address bus which contains the address (as described in section 4.3.1) of the module to

which the data word on the data bus has to be written to in each clock cycle.

The PCI Express block does not offer the capability to read from the module microcommand tables or the forwarding and QoS lookup tables. This was deemed unnecessary since the information are written into the NPU by the user in the first place and thus he can keep track of them at his end. The Tx part of the PCI Express interface is still implemented however, since it is necessary to sent confirmation messages (called Transaction Level Packets - TLPs in PCI Express slang) to the PCI Bridge upon data transmission.

4.3.3 Programming the NPU100

The compiler and the lookup entry generator comprise the heart of the software side of the NPU100. Their goal is to ensure that the user is able to leverage the resources made available to him in the NP. As described in section 2.5, NPs have a long tradition of providing the prospective programmer with difficult to use programming paradigms. The NPU100 could very well have shared this deficiency, since the programming information that are required for it to perform its task, would have been very difficult to generate by hand. This is exactly the problem addressed by the NPU100 compiler, which offers a straightforward method to generate the required programming information before they are transferred to the NPU. In summary the compiler fulfills the following tasks:

- Reads the configuration program, which is written in high level code.
- Maps the functionality described in the code automatically to mini pipelines and pipeline stages.
- Converts the high level code into the microcommand format to be fed into the NPU100.

The lookup entry generator's sole task is to create the forwarding and/or QoS lookup table entries found in each MP.

Configuration Files

The first step in utilizing the compiler and the lookup table entry generator to create the NPU100 configuration information is to provide it with the data that facilitates this task. The format in which this information is to be provided is important, since it must balance between providing sufficient flexibility so as to put the NPU100s capabilities to the best possible use and simplicity so as not to confuse the potential user with a convoluted interface.

In the NPU100 this is accomplished by the use of the four configuration files which were introduced in section 3.3, each of which fulfills a different task. This section provides details on how the files are structured and how they cumulatively enable the programming of the NPU100. Three of the files are XML files and the fourth uses a custom but very C like, high level code format. XML (eXtensible Markup Language) has been selected, because it allows for the structuring of the information in a meaningful way, through an easy to understand, tag-based format, which can then be easily accessed programatically

through off-the-shelf libraries in almost any programming language. The following four sections each provide an overview of each file and how it factors into the programming process.

Architecture Configuration File (ACF) The ACF is the lowest level file, which defines the architectural parameters, which the compiler will use to perform the task mapping from the code to the NPU PEs. As such, this file will need to be changed only when a different NPU100 design is targeted (that is, an NPU100 with different processing modules or a different number of MPs). Thus, for each NPU100 design there is only one corresponding ACF file. The file provides information about the number of mini pipelines and the order of the PEs in the mini pipelines. Without this information the compiler cannot proceed with the task assignment as it cannot determine the resources, which it has available to assign tasks to. An ACF file always begins with an ACF tag, which identifies the type of the file. Then the number of MPs in the NPU100 for which the code is to be generated is given, followed by an enumeration of the number of stages in the order in which they appear in the MP. The following code provides a sample of the ACF file created for the NPU100 version used throughout this thesis. As can be deducted from the code, the structure supports only identical MP structures, although this could be easily extended in the future, to include less homogeneous ones.

```
<ACF>
<number_of_minipipeline>4</number_of_minipipeline>
<stage>ForwardingLookup</stage>
<stage>Insert</stage>
<stage>Replace</stage>
<stage>Modify</stage>
<stage>Modify</stage>
<stage>QoSLookup</stage>
</ACF>
```

Protocol Information File (PIF) The Protocol Information File defines aliases for areas of the data words used throughout the NPU100. Its goal is to ease the software development for the NP by hiding the internal complexity of the NP's inner workings. As was explained in section 4.2.4, the PE I/O modules identify which fields to extract from the pipeline data word by a start location-length tuple found in the microcommand table. To avoid demanding from the user to specify each data field using this tuple, the PIF is used to provide a moniker for the fields that are to be used when writing the actual code.

There are three different data words used within the NPU, whose fields can be aliased in the PIF file:

- The pipeline word, described in section 3.2.1.
- The forwarding lookup table entry.
- The QoS lookup table entry.

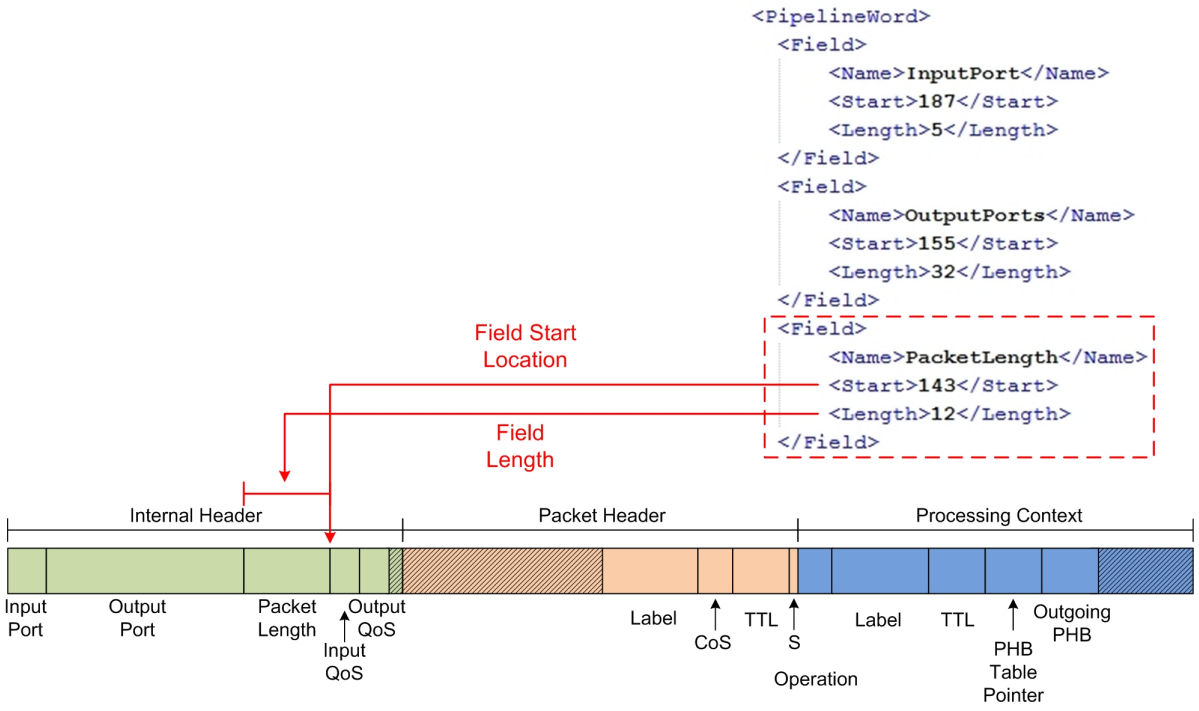


Fig. 4.31: Example of the Definition of a Field in the PIF

The respective sections define the structures that are used when programming the NPU100 for MPLS-TP and PBB-TE use. Any other format is acceptable as well, as the PIF is very flexible in defining field names and locations. One area can have multiple aliases and various aliases can overlap. This will generate an informational message from the compiler, but is permitted if done intentionally.

Figure 4.31 provides an example of how the PIF is used in order to assign an alias to a specific portion of the pipeline data word. The upper right corner of the figure shows a code snippet from the file, which defines several fields. At the top of the snippet the keyword PipelineWord denotes that the entries that follow specify fields that belong to the pipeline word. Respectively the user must define similar areas for the forwarding and QoS lookup tables by using similar XML identifier tags. The lower part of the image is dominated by the pipeline word with all the various fields used during processing. The arrows then demonstrate how an alias for one field, in this case the packet length is specified. First the name, to which this pipeline word area is to be mapped, is being provided, followed by the bit at which this area starts and finally by the length of this area, all enclosed in the appropriate tags. The compiler uses these the determine the location of each data and thus how to program the NPU100 PEs to access the required data and perform the needed operations on them each time. Other fields are similarly defined.

lookup Information File (LIF) The LIF facilitates the creation of QoS and forwarding lookup table entries as well as their deletion. Creating and deleting entries is a quintessential process in an NP, as every entry corresponds to a connection in the network. What the LIF does, is to use the aliases defined in the PIF and assign values to these fields.

The fields correspond each time to one entry in one of the lookup memories. Which entry is being currently specified must also be provided. Any alias used in the LIF must correspond to one defined previously in the PIF, otherwise an error occurs. LIF processing occurs independently of the configuration program compilation. This is necessary since adding and removing connections and thus lookup table entries is commonly done during run time in NPs and without having to reprogram the device.

The file can contain up to two separate areas, one specifying forwarding lookup table entries (marked with a Forwarding tag) and one containing QoS table entries (denoted by a QoS tag). The following code provides an example of the former for the MPLS-TP protocol. The Entry tag delineates the beginning of a specific entry, whose fields are then specified in succession. If a field is omitted a warning is given out by the lookup generator and its value is automatically set to 0. The Action tag specifies if the entry is to be added or deleted. When deleting an entry, there is no need to provide values for any fields apart from the address and any provided are ignored. The entry generation software does not maintain status information regarding the memory and thus does not and cannot check if the entry that the user attempts to delete exists in memory. Hence a deletion configuration word would be generated and sent to the NPU100 in any case, without however any negative consequence on the functionality of the NP.

```
<Forwarding>
    <Entry>
        <Action>Add</Action>
        <NetworkAddress>0</NetworkAddress>
        <OutputPorts>5</OutputPorts>
        <Operation>0</Operation>
        <Label2>192</Label2>
        <TTL2>45</TTL2>
        <PHBTblPtr>32</PHBTblPtr>
        <OPHB>71</OPHB>
    </Entry>
</Forwarding>
```

Figure 4.32 illustrates the most complex of the two cases encountered in the LIF, namely the one where an entry of the QoS lookup table is specified. This is because this memory is divided in subtables, which must be specified separately. The Subtable tag, thus, demarcates exactly such a subtable, whose entries are then described one by one as in the case of the forwarding lookup table. In this case however the entries are written to the table in the order they are present in the LIF (the first entry to the first address, etc). Entries for which values are not specified are automatically filled with zeroes. Before the entries and right after the Subtable tag, the address for this subtable is specified. The address value within the tag is user specified and must be provided in order for the subtable to be addressable.

Configuration Program File (CPF) The CPF is by far the most complex file of the four. This section will provide an overview of the syntax it uses, however the bulk of the file's features will be explained along with the compiler itself in the following section, since

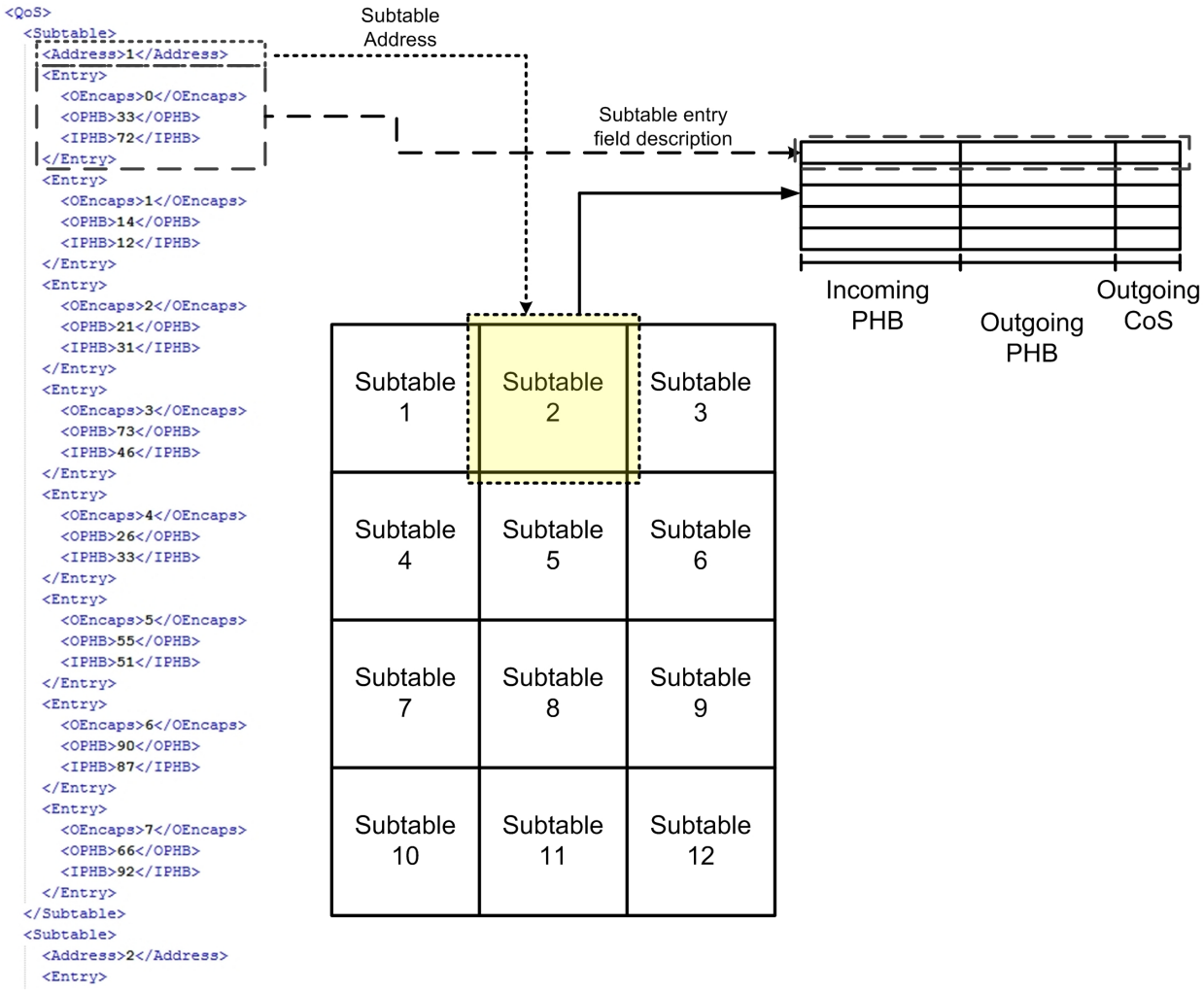


Fig. 4.32: Example of a QoS Lookup Table Entry Definition in the LIF

the structure of the code it self and the mapping of the functionalities to the NP PEs are closely intertwined.

In essence the CPF file, as with any source code file, is a plain text file, which contains the code to be compiled. It consists of two sections. The first section starts with the "#parameters"directive and contains compiler directives and functions, which write programming information to the non-processing modules of the NPU100 (e.g. the Header Payload Split). The second section is denoted by a "#program"directive and includes the program to be compiled and loaded into the MPs. This is the most complex section as it defines all the functionality that is to be mapped to the processing modules.

How code is written in this second section of the CPF depends to a great extend on the programming architecture of the NPU100. The microcommand tables described in section 4.2.4 and the way they are indexed through a programmable part of the pipeline word, dictate a specific code structure that results in a meaningful description of the program's functionality and which is described in the next section.

NPU100 Programming Software

As was hinted at in the programming file's description, the NPU100 programming software performs two distinct tasks. The first task involves reading the ACF, PIF and CPF and creating the programming data words for the microcommand tables in the NP's processing modules. This is performed by the NPU100 compiler. The second task takes the PIF and the LIF and generates the appropriate forwarding and QoS lookup table entries. This is done by the lookup entry generator software. The three following subsections shed light onto how these processes are executed, with the first section providing a short introduction into how the files are parsed.

Parsing the XML Files Independently of which process is executed, the initial, indispensable part is the parsing of the XML configuration files. Since there are several C++ libraries that perform this task, this is fairly easy to accomplish. We use TinyXML [6] in our software.

If the forwarding and QoS memory is to be programmed, then the PIF is parsed first and then the LIF. If the microcommands for the PEs are to be generated then the LIF is omitted and the ACF is compiled first, followed by the PIF and finally the CPF.

Parsing each file begins by matching the appropriate tag at the beginning of the file. The PIF file, identified by the PIF tag, begins by designating which of the three sections is to be defined. Then the name, start location and length for each alias are provided, each with their appropriate tags. Each alias is stored in an entry in a vector, which holds the respective fields. Furthermore the parser checks whether there is any overlap between the various fields defined in the PIF and throws an informational message in this case.

ACF parsing is similar, identified at the beginning by the ACF tag. Then the number of MPs are stored in a separate variable, as well as the MP stages defined in the file in a vector in the order in which they appear in the file.

Parsing the LIF is more complex, since it involves cross referencing the read data with those found in the vector in which the PIF results were stored to verify that values in the LIF are assigned to valid fields. An inconsistency between the two leads to an error message.

Upon parsing the file two vectors, the LIFforwardingEntry and the LIFQoSEntry vectors are populated. The structure of each vector is different and mirrors the requirements of each entry type, ergo the LISforwardingEntry includes an action variable, denoting if this is an entry add or remove operation and the LISQoSEntry includes an sub table address variable and a sub vector assigning values to each sub table entry. The parser returns a warning for fields to which no value has been assigned and sets their values to zero.

Compiling the CPF The basis for compiling the NPU100 microcommands is the CPF, which was briefly introduced previously in this section. Here, however, we delve deeper into its structure, and how this translates into meaningful code for the NPU100.

Figure 4.33 attempts to elucidate the process of parsing the CPF file, compiling it and generating the microcommands for each module in an NPU100 implementation. The first step the compiler performs is to strip the code of any comments. Then the remaining commands are checked for syntax errors, unknown aliases and out of place statements (#program statements in the #parameters section and vice versa). Any violation results in an error. Whitespace trimming is also performed at this point. The final check is ensuring that only appropriate nesting structures are included in the code. The compiler allows only one level of nesting and that only in case of a conditional statement nested in a loop statement or the other way around. Nesting more than one conditional or loop statements is not allowed, as it would not correspond to a valid operation allocation in the NPU100 pipeline. The conditional and loop statements are also checked for including any impermissible aliases or values. Finally, loop statements are checked for bound upper limits, which fall within the resources of the NPU100 implementation for which the file is compiled. Possible violations also result in errors being thrown by the compiler.

The next step is the parsing of the #parameters section. This is fairly straightforward and follows the process described in section 3.3. After this step is completed, the compiler uses the data from the ACF file to generate a resource vector. This data structure contains an entry for each module of each MP, including information to identify its position in the NPU100 (meaning in which MP it belongs and where in this MP it is placed) as well as the microcommand entries which have been assigned to that module. The compilation process then essentially parses the commands in the CPF's #program section and assigns each command to a module. When parsing the program section the compiler creates another data structure, the command vector, and populates this with all the commands found in the CPF file. The command vector is the result of successfully parsing the CPF file. Each vector entry has several fields, which detail everything the compiler needs to know about the command, from which module it requires to be executed to the operands it will use. The compilation is thus the result of the correlation of two vectors, the command and the resource one.

The compiler then goes through the command vector sequentially and for each command checks the resource vector to determine if the resource required for the command exists (that is, if one or more appropriate modules have been instantiated in this NPU100 version) , if it is available (meaning determining which module can be used for this command and if the microcommand table of at least one appropriate module has space to accommodate this command) and if all the parameters for the operation are provided in the correct form as

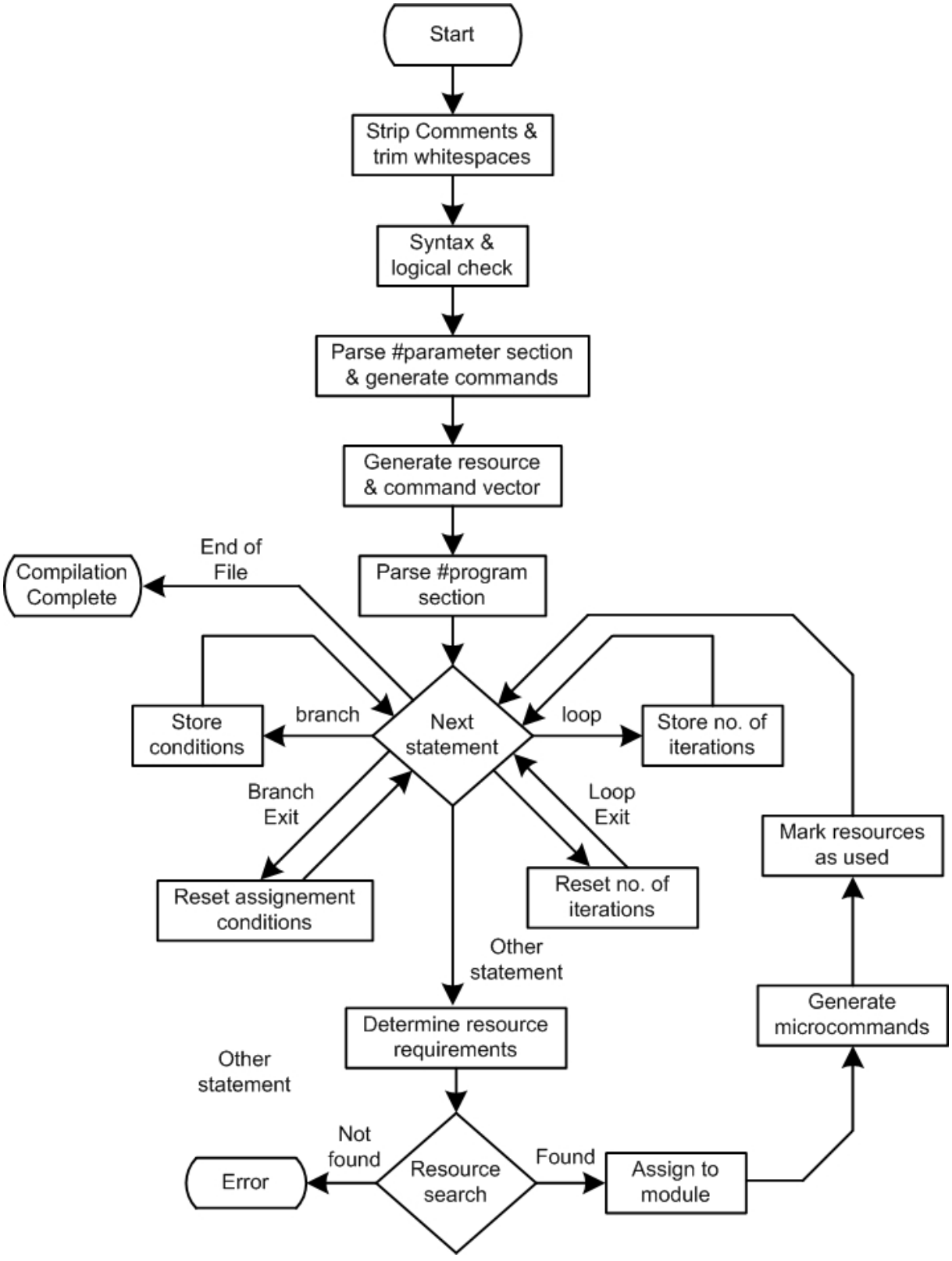


Fig. 4.33: Flow Diagram of the CPF Compilation Process

demanded by the module (this entails the verification of all the parameters against the PIF file in order to check that they have all been defined and that their lengths are compatible

with what the module expects). This seemingly innocuous endeavor is rather complex, since conditional statements and loops can have a profound effect on how statements are assigned to processing modules.

This process is illustrated in the lower part of figure 4.33. When the compiler parses a new statement it is encountered with a multitude of alternatives. A statement in the NPU100 programming language can be either a conditional one, a loop one or an assignment one. Each of those is handled different by the compiler. A conditional or loop statement causes the compiler to mark an internal state for all subsequent commands until the loop/conditional branch is exited. Thus in this example:

```
for(int i=0;i<2;i++) {
        S = 0;
}
```

the compiler reads the for statement and notes that all subsequent commands (in this example the statement setting the S field to zero) until the loop is exited have to executed twice. Similarly a branch statement causes the compiler to mark the condition which apply to all subsequent statements until the branch is exited. When this state determination is complete the compiler proceeds to the next statement in the command vector. When an assignment statement is encountered by the compiler, it checks the state to find out under which conditions and how many times the statement is to be executed. Then it looks up the necessary resources in the NPU100 pipeline. This entails looking up the required number of modules and checking if the availabe ones have the appropriate microcommand table entries free to accommodate the statement. If an appropriate number of modules are found then the commands are generated and stored into the microcommand entries of those modules in the resource vector.

To help clarify this process further, figure 4.34 shows a series of simple examples of how the structure of the code can influence the allocation of tasks into the entries of a microcommand table. All the code samples illustrate a decrement of a field called TTL. Since this corresponds to a subtract operation, it will be assigned to the modify module in all four cases. However, in the first case the decrement operation is not included in any conditional statement and thus it occupies all 8 entries of the microcommand table of the lookup table. This is done because the compiler interprets the code in such as way, as to mean that the user wanted to have the operation described in this statement executed, independently of any conditions. This type of statements should be limited as they occupy an entire module.

The second scenario shows how the operation is allocated to the specific entry of the modify module, which corresponds to the value with which the FieldName field is compared to in the conditional statement. Thus the chunk of code is to be executed only if the condition described holds, that is in this case only if the field named fieldName is equal to value. Mapping this operation into the functionality of the ISU/OSU modules of the pipeline cores (as described in section 4.2.4), means that the fieldName is used to index the ISU/OSU microcommand table. The command read from the table is then used to select the TTL field from the pipeline word and pass it on to the module. The same field (fieldName) is used to index the processing element's microcommand table and select the command to be executed, which tells the PE to subtract the value 1 from the input field

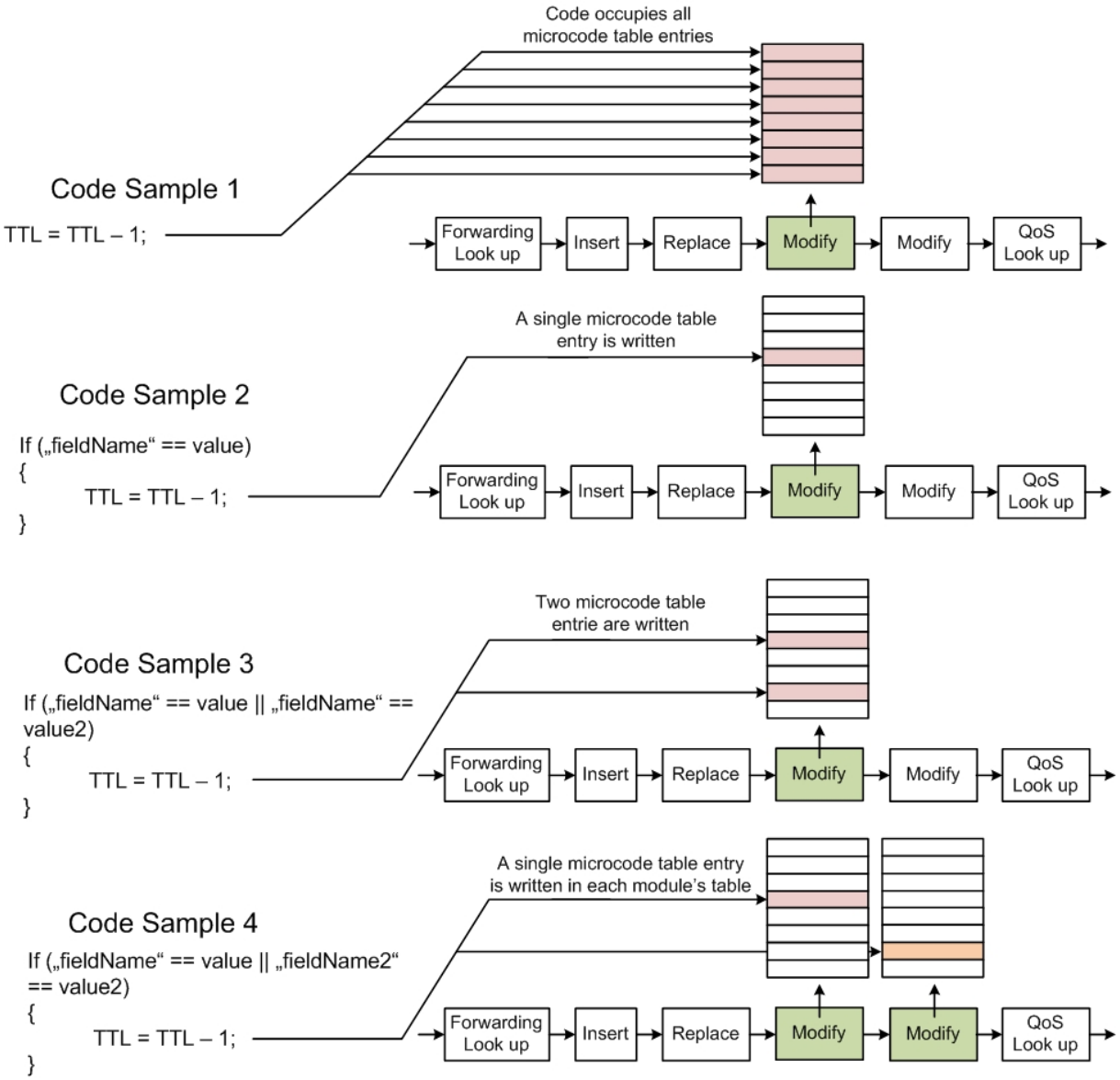


Fig. 4.34: Examples for Different Task Allocation to the PEs

(TTL in this case). The OSU then overwrites the old TTL value in the pipeline word with the new one. It should be noted, that any one of the aliases defined in the pipeline word section of the PIF could have been used on the left hand side of the conditional statement but the right hand operand of the comparison must always be a fixed number, since an alias would not be resolvable by the compiler at compile time and thus it would not be able to determine to which entry to assign the operation.

The third example demonstrates the use of a logical or operator in the conditional statement in order to assign the operations to more than one entries of the same module. In this case the field used to index the ISU/OSU microcommand table must be the same (since only one field can be used for that purpose in each clock cycle) in both statements. The operands on the right hand side of the equal operation must differ between the two conditional statements. This allows the compiler to assign the execution of this command

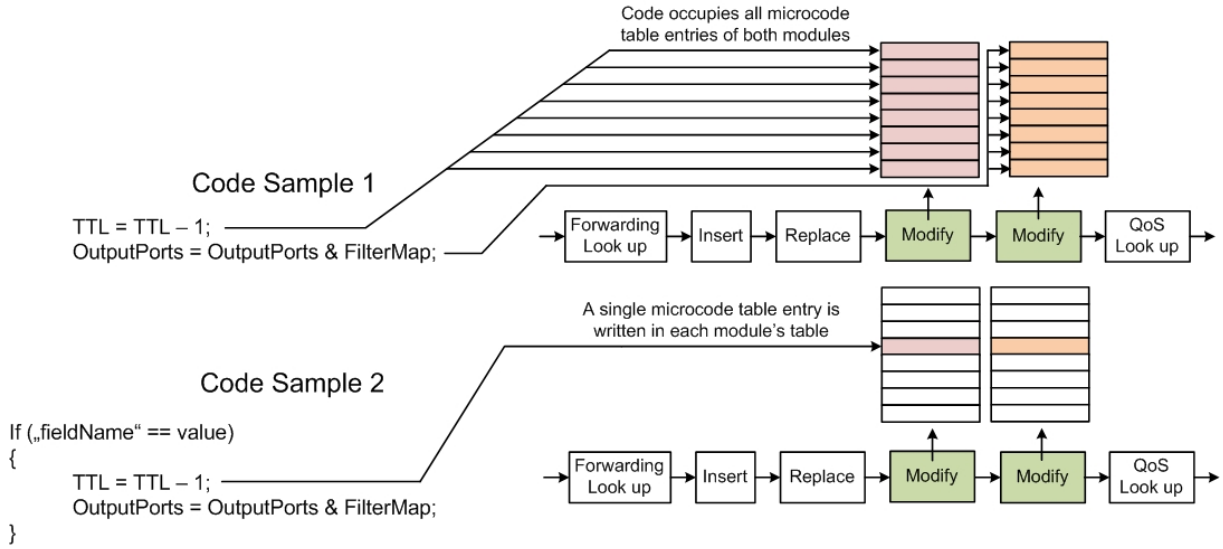


Fig. 4.35: Further Examples of Different Task Allocation to the PEs

to one module. Each part of the logical or statement corresponds to one entry in the microcommand table. Both entries contain the same data, since both commands perform the same functionality, but the fact that they are implemented when the value of the field fieldName is equal to either value1 or value2 forces the compiler to use two entries in the table.

Using different operands at the left hand side of each condition would lead to the scenario demonstrated in the fourth case, where both of the modify modules in an MP were utilized to execute this operation, since the two different aliases (in this case fieldName and fieldName2), cannot be used to address the same microcommand table. Other operations could be assigned to these two modify modules by using the same alias, while comparing it with a different operand as in the previous example.

Figure 4.35 provides further examples on how code is mapped to the various modules. Again the modify module is used, but the procedure applies to any other modules just as well. On the top scenario two lines of code, the TTL decrement and a logical and operation are performed. As in the first scenario of figure 4.34, each statement is assigned to all entries of the microcommand table. This means that since the TTL decrement occupies the entire first modify module, the and operation must use up the entire second modify module. A third statement requiring a modify module would then have to assigned to the subsequent MP.

The second scenario illustrates the case where the two statements are part of the same conditional branch. This means that they each have to be allocated to a different modify module, as the required entry which corresponds to that condition cannot be shared. Thus the TTL decrement takes up the appropriate entry of the first modify module microcommand table and the and operation the same entry of the second modify module's microcommand table. Since the modify module is capable of executing two operations in parallel, allocation to the same module could have been possible for a different set of operations. Here the problem lies in the three inputs that are required (TTL, OutputPorts and Filtermap fields) and cannot be supported in the module's ISU. The following statement:

```
if (fieldName == value){
        TTL = TTL + 1;
        PacketLength = PacketLength - 4;
}
```

would be allocated to one modify module, since only two ISU inputs are required.

The NPU100 compiler also provides limited support for loop statements, using a for loop. This however should be used with caution, since what a loop essentially does is to assign the same code to subsequent modules of the appropriate type. To exemplify, the following code decrements the TTL field three times:

```
for(int i=0;i<3;i++) {
        TTL = TTL-1;
}
```

This code would be assigned to three consecutive modify modules and would occupy all microcommand table entries of each module as discussed previously. Putting the TTL decrement statement in an if conditional statement nested in the for loop, would enable us to assign the statement to a single microcommand table entry as in the second code sample of figure 4.34, while still occupying three consecutive modify modules. The variables used in the for loop statement cannot be aliases, as this would inhibit the compiler in determining the allocation of tasks to the PEs at compile time.

There are two more elements of NPU100 compilation that must be explained. The first is the mapping of different statements to module in relation to each other. This depends on the data dependencies between the various commands. The example below:

```
if (fieldName == value) {
        TTL = TTL-1;
        lookUpRead(Label);
}
```

shows how the assignment between two statements which don't share any data dependencies is performed. In this case, the resulting assignment will be the one shown on the first code example of figure 4.36. In this case the TTL decrement statement is assigned to one microcommand entry of the first available modify module. The lookup read statement which follows however can be assigned to the first pipeline, despite the fact that it is located physically before the modify module to which the preceding command was assigned. The compiler automatically checks data dependencies of both the input and output fields of the operations and decides if this is feasible.

Code example 2 of figure 4.36 shows a case where the aforementioned assignment cannot be performed. In this case the lookup read uses the TTL field as a pointer to the appropriate memory address and thus it depends on the result of the previous operation. Thus it must be assigned to a forwarding lookup module, that is located after the modify module that performs the decrement operation, which in this example happens to be in the next MP. This helps to illustrate the final key element of NPU100, namely how looping is enabled to facilitate the use of the resources found in the remaining MPs. In order to accomplish this a specific flag in the pipeline word needs to be set (as described in the Label Reassembly module description in section 4.2.2). This operation must be performed

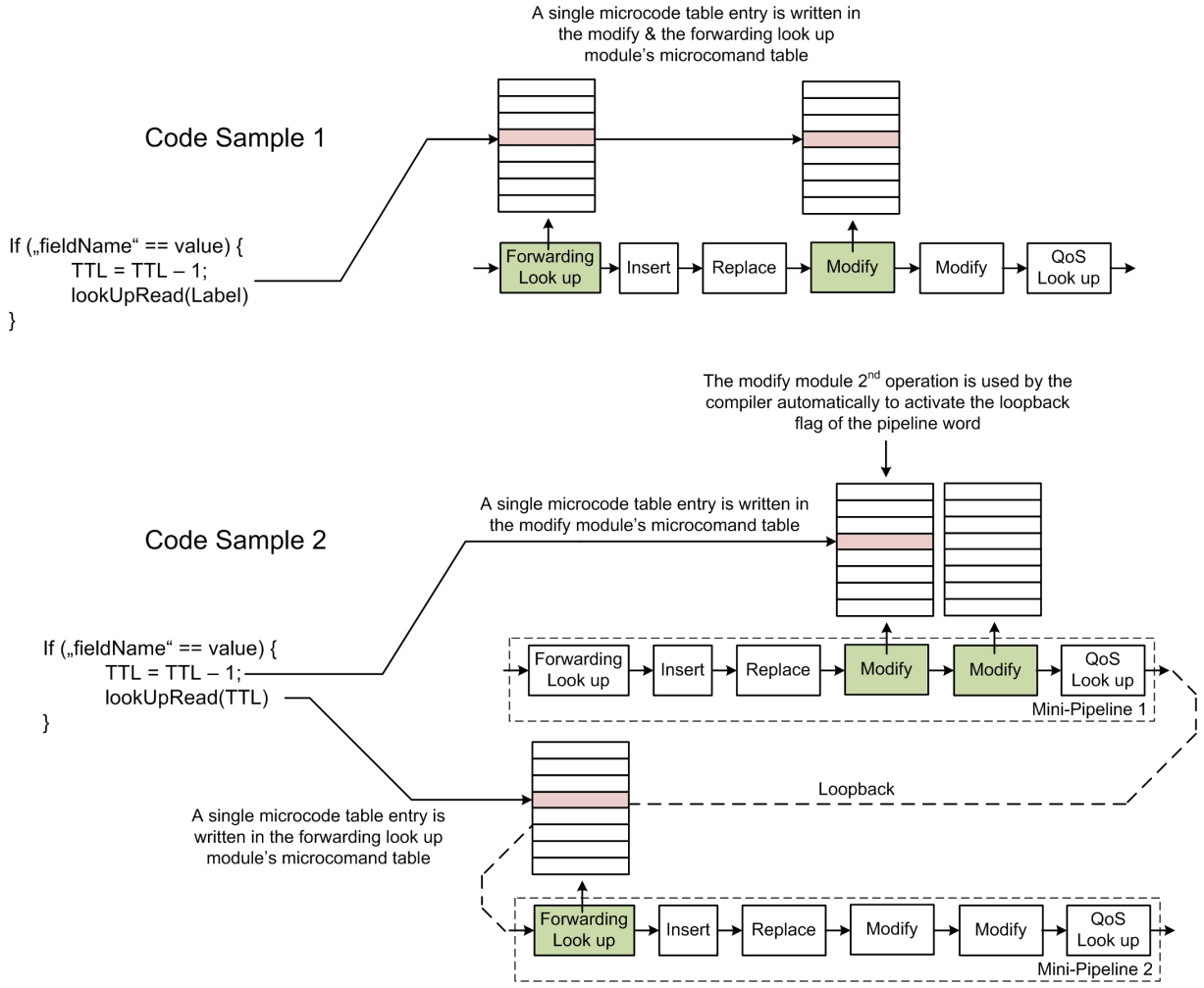


Fig. 4.36: Example of Statement Mapping to PEs with the MP Loop Being Used

by a modify module and since the appropriate entry in the first modify module is occupied with the TTL decrement operation this must be assigned to the second modify module of the first MP.

The examples provided here attempted to shed some light on how the NPU100 compiler maps CPF statements to specific microcommand table entries of the PEs. The outcome of this process is a vector called the fundamental vector, which is structured identically as the pipeline core of the NP. It consists of MPs, which are divided into stages that contain microcommand tables, which are in turn further subdivided into entries. The entries are populated with the aliases and other operands and operations contained in the code. This vector is parsed in the final step of the compilation process in order to create the programming words. This entails determining to what each alias corresponds from the PIF and writing the appropriate data into the programming word for accessing the field to which the alias maps to.

To facilitate the programming of protocols like MPLS-TP in which all MPs need to share the same code, the NPU100 compiler includes the _duplicate directive, which as the name implies, ensures that the code is duplicated appropriately across all the available MPs. The compiler automatically determines how many times to duplicate the code. Hence,

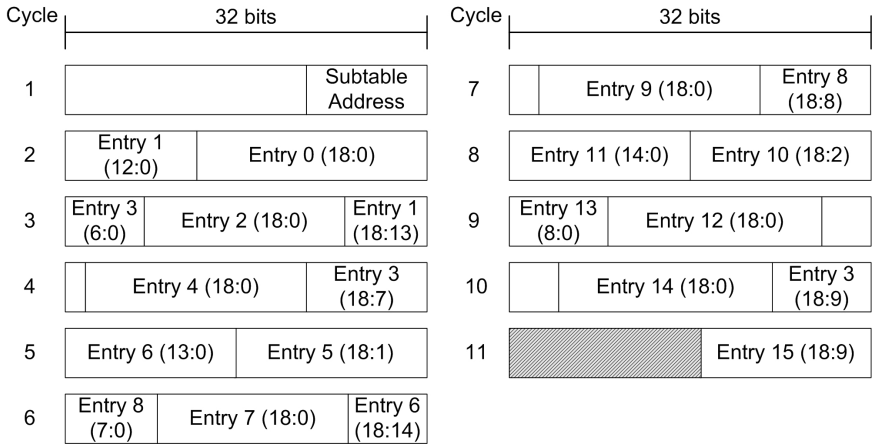


Fig. 4.37: Allocation of QoS Subtable Entry Words to Programming Words

assuming one MP is used by the written code and four MPs are available the code is duplicated four times, whereas if the code took up two MPs it would have been duplicated two times.

After the duplication process, all the 32 bit programming words and the corresponding 16 bit addresses are created and written in a text file. The PCI Express driver reads the text files and transmits them. Each programming word occupies one 32 bit PCIe TLP (where addresses are concerned the 16 MSBs are set to 0) and are always transmitted in pairs (address first, followed then by the data).

Forwarding and QoS Memory Programming One of the two tasks of the NPU100 programming software is to generate the programming words for writing and deleting entries to and from the forwarding and QoS lookup tables of the NPU100. The initial steps to this end were described previously in this section and included parsing the PIF and the LIF and populating the vectors which hold the entry data. Now this data must be converted into a format suitable for writing into the NPU100 memories.

This involves two steps: first the data have to correctly packed into 32 bit data words, in the appropriate way, since the memory write control modules (section 4.2.4 for both the forwarding and QoS lookup modules) of the NPU100 expect them to be served to them in a very specific format in order for them to be written into the memory properly.

Thus each 164 bit forwarding lookup entry is broken down into a 5 cycle transfer, in which the higher order bits are sent in the first cycle until the lowest 32 bits have been transmitted. Whether the entry is to be written to or wiped from the memory is determined by the address placed on the programming bus, which depends on the value of the Action tag, to be found in each entry.

Programming the QoS lookup memory is similar. The memory is made up of 256 individually programmable subtables, each of which must be programmed in its entirety each time (that is one cannot change individual entries in a subtable without reprogramming the entire subtable. The user specifies all 16 entries separately and then the lookup entry generator reads the information and agreggates it into one 304 bit long entry (since each of the sixteen entries is 19 bits wide). A subtable address must also be sent to the NPU100

(this is different than the programming address of the QoS lookup memory write control module). This address is sent in the first configuration cycle and the memory data follows in 10 consecutive cycles. Entries follow one after the other and their length of 19 bits means that they are not aligned with each programming word, as figure 4.37 shows. The lookup entry generator must take this into account and generate the programming words appropriately. While this complicates the software, it saves resources in the QoS memory write control, which, would in an opposite case, where each 32 bit programming contained one subtable entry, have to receive all the programming words, strip them of the padding that would have been necessary and then form one 304 bit word, which it would then write to the memory. Since hardware resources are scarce and software execution time is not, the obvious choice was to shift the load to the software side.

In both lookup table configurations, the software does not check for duplicate entries. It is left up to the user to avoid such a fallacy. The software outputs a script file, which contains the data which the PCI Express driver then reads and sends over the link.

5 NPU100 Implementation Results and Testbed

After meticulously explaining the architecture and implementation of the NPU100 hardware and software side, the focus shifts to providing results about the implemented modules and realizing an actual NPU100 testbed on an FPGA, that proves the viability of the design. The first part of this chapter provides results on the implementation on critical points, such as power dissipation, resource consumption and throughput. The second part describes the test bed developed to verify the functionality of the NPU100, including the pipeline core and the programming interface. The test bed contains some additional modules for debugging the NPU100 as well as some compromises that had to be struck in order to allow for the implementation of the NPU100 on the selected FPGA board. The NPU100 test bed is a complete test platform, offering everything from a software component to generate the NPU100's programming, a connection with a Spirent traffic generator over which complicated traffic patterns can be used to test the design, a link with a Tektronix logic analyzer, which can be used to debug the internal signals of the circuit and finally the NPU100 itself. This test bed has been used to verify the functionality of all aspects of the NPU100 design. All results presented in this chapter were derived using a Xilinx Virtex-5 LX330T FPGA, the largest in the Virtex 5 family, where logic resources are concerned, unless explicitly stated.

5.1 Implementation Results

This section provides a detailed look into the NPU100 design from four perspectives: resource consumption, power dissipation, scalability and performance. Its goal is twofold: first to explain the characteristics of the entire NPU100 system, not just the folded architecture from which its core consists and then to provide a more thorough analysis of many design details in comparison to the discussion in section 3.2.2, whose main aim was the comparison of the conventional architecture and the two proposed alternatives and the selection of the most appropriate one.

5.1.1 Resource Consumption

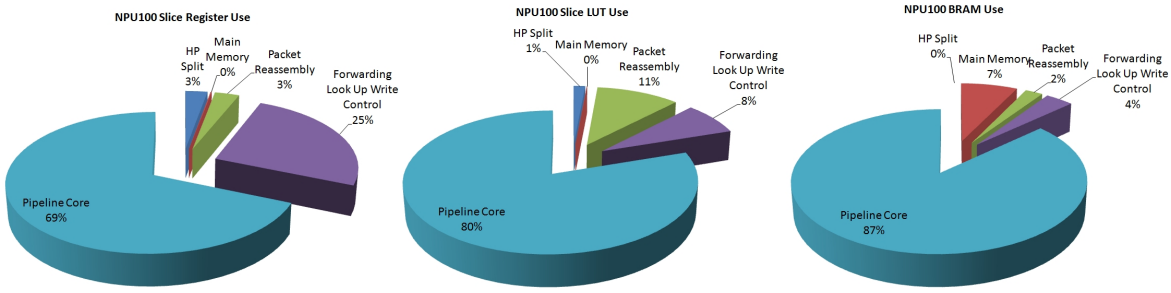
This section includes detailed data regarding the resource consumption of the various NPU100 modules, as well as the NP as a whole. Table 5.1 provides an thorough look at the amount of resources that each module and the design in total consumes. It starts by listing the total resource use of the system and then proceeds to break it down to its constituent parts, the NPU100 and the PCIe core. It then delves deeper into the NPU100, in order to provide detailed information on the resource use for each specific submodules in the design.

Tab. 5.1: Detailed Overview of the Resource Consumption for the Various NPU100 Modules

	No. of Slice Registers	%	No. of Slice LUTs	%	No. of BRAMs	%
System Total	89520	43	176698	85	271	84
PCIe Total	2985	1	2190	1	6	1
PCIe NPU100	430	0	318	0	0	0
NPU100	82563	39	171737	82	221	68
Header Payload Split	2259	1	2589	1	0	0
Packet Memory	126	0	104	0	16	324
Packet Reassembly	2513	1	18714	9	5	1
Forwarding lookup Write Control	20075	9	13367	6	8	2
Pipeline Core	54800	26	140018	67	192	59
Dispatcher	4649	2	2993	1	0	0
Dispatcher Label Buffer	2058	0	929	0	0	0
Label Reassembly	1320	0	1373	0	0	0
Label Reassembly Label Buffer	5740	2	3362	1	0	0
Mini Pipeline	7732	3	31983	15	48	15
Forwarding lookup	1983	0	6521	3	68	20
Insert	1104	0	8194	3	0	0
Replace	552	0	4162	2	0	0
Modify	1376	0	4832	2	0	0
QoS lookup	1341	0	3926	1	5	1
ISU	0	0	816	0	0	0
OSU	0	0	2783	1	0	0

The data shows that the impact of the PCIe core on the resource consumption is minimal in comparison to the total FPGA capacity as each module consumes at the very most 1% of the FPGA resources and that the NPU100 specific additions to the core accounts for only a very small part of that.

Table 5.1 provides a break down of the distribution of resources in the NPU100. On all fronts these are all dominated by the pipeline core, however it is interesting to note how the Forwarding Lookup Write Control module contributes a significant part to the total resource use, particularly when it comes to registers, which are used for the tables indicating the forwarding lookup memory occupancy. This should influence any decision to either increase the size of the memory as this would also increase the size of the Forwarding Lookup Write Control modules. In any of these cases and despite the fact that in absolute number, there are significant unused FPGA register resources, exchanging distributed RAM (and thus slice registers) for BRAMs should be considered to keep resource use under control. An ASIC implementation would eliminate this problem by implementing the memory state tables as a custom size memory block.



(a) NPU100 Slice Register Consumption

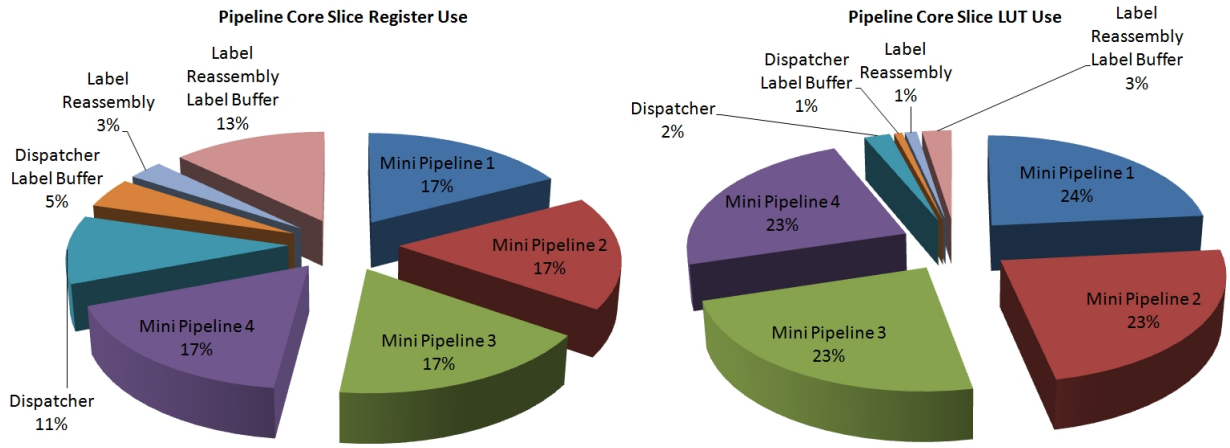
(b) NPU100 Slice LUT Consumption

(c) NPU100 BRAM Consumption

Fig. 5.1: Resource Use Break Down between the Various NPU100 Modules

From the remaining modules only the Packet Reassembly uses up a considerable number of slice LUTs to realize the complex mechanism needed to rejoin packets with the processed headers as described in section 4.1.3. Even so, the total resource use of the Header-Payload Split modules (12% of slice LUTs and 6% of slice registers) is only a small fraciton of what the LX330T FPGA offers. This verifies the merit of the Header Payload Split approach, which eases demands on the processing core throughput, without requiring overwhelming resources to realize.

A similar overview as the one given by figure 5.1 for the NPU100 is given for the pipeline core and its constituent modules on figure 5.2. BRAM data are omitted since they are evenly distributed among the four MPs and the Dispatcher and Label Reassembly, along with their two buffers take up no BRAM resources, as they have been implemented using the registers found in the FPGA's CLBs.



(a) Pipeline Core Slice Register Consumption

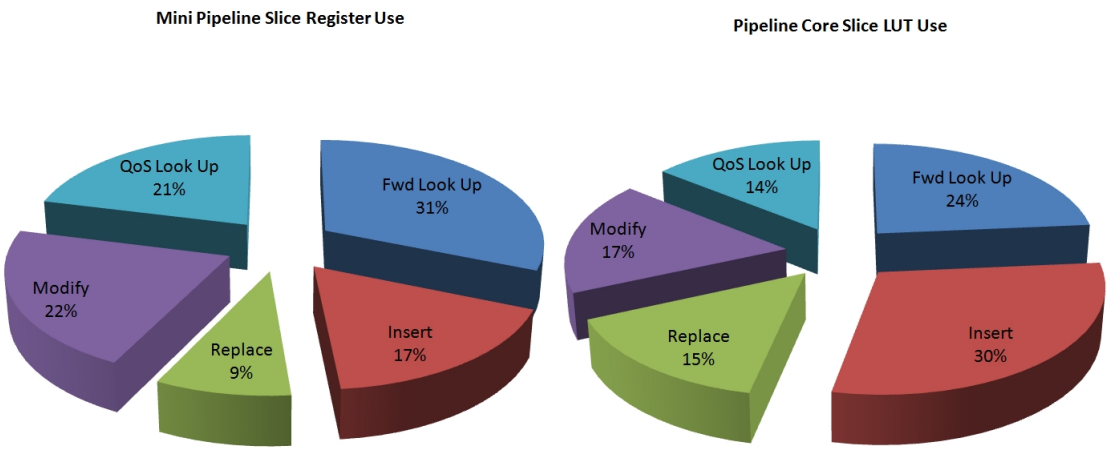
(b) Pipeline Core LUT Consumption

Fig. 5.2: Resource Use Break Down between the Various Pipeline Core Modules

In this case, the lion's share in both slice register and LUTs is used up by the MPs. In particular where slice LUTs are concerned the impact of the Dispatcher, Label Reassembly and their buffers is minimal in comparison to what the MPs take up. Register use paints a slightly different picture since the two Label Buffers do occupy a decent part of the total

number of slice registers used. This is to be expected since these are essentially register matrices. Furthermore the significant difference in size between the two buffers (with the Dispatcher Label Buffer trailing considerable behind) is in line with our analytical calculation in section 4.2.3. The Dispatcher and Label Reassembly also take up a non-trivial number of registers, which are in all likelihood used for the internal pipelining of these modules, with the Dispatcher having a considerably lead, something expected as it consists of four pipeline stages instead of two.

Repeating the exercise for the modules making up an MP results in figure 5.3. Register consumption is more or less evenly spread among the five modules, with some dominating more than others. This depends on various factors: The Replace module is the simplest case, where the only registers used are the microcommand table and the pipeline register, whereas the insert module has two of both and thus uses twice as many registers. Register use increases in the modify module since it also includes an internal microcommand table, which contains the operands for the operations and additional information. The QoS lookup and the Forwarding lookup modules are both internally pipelined and thus use up more registers because of this. The former includes two pipeline stages, whereas the latter four, which accounts for the difference in register use.



(a) Mini Pipeline Slice Register Consumption **(b)** Mini Pipeline LUT Consumption

Fig. 5.3: Resource Use Break Down between the Various Mini Pipeline Modules

In Slice LUT use on the other hand the ISU/OSU resource consumption sets the tone. This is made obvious by the fact that the Replace module, which is essentially only an ISU/OSU pair consumes 15% of the slice LUT resources in an MP. Multiplying that by six (since the Insert module comprises tow ISU/OSU pairs in sequence), means that the 90% of the slice LUT resources in an MP are used up by the ISU/OSU modules. This points both to the importance of efficient register access and represents an area where tangible improvements could be possible.

5.1.2 Power Consumption

A first glimpse into the power consumption advantages that the folded architecture provides, were outlined during the comparison of the three architectures in section 3.2.2. Here

a thorough look into the performance of the folded pipeline architecture is provided. Since a direct measurement of the FGPAs power consumption was impeded by the lack of a shunt resistance on the used FPGA development board, the XPower analyzer was utilized again, this time with more detailed data in order to extract a clear picture about the power dissipation of the system. Delving deeper into an analysis of the power consumption system presupposes a more detailed modeling of its switching behavior. Modelsim and XPower support this through the generation of the SAIF files, which record the activitiy of each circuit node in detail. Modelsim generates the file after each Post Place and Route (PPR) simulation, which is then used as input for the XPower tool in order to generate a very precise evaluation of the power consumption of the device. This methodology was used to further investigate the power consumption of the NPU100, dependent on various traffic scenarios and with different packet sizes. The breadth of scenarios tested includes the ten scenarios described in table 2.1 for various packet sizes, as well as the first four of the traffic mixes included in the qualitative architecture comparison of section 3.2. A summary of these scenarios is provided in tables 5.2 and 5.3 respectively. Furthermore in the remaining four scenarios of table 5.3, the kind of operations to be used in the mix must also be defined. Saying that a packet requires two MP loops does not implicitly imply, which operations will be performed on this packet, which in this case could be either a swap and push or a pop and swap (here only the operation combinations determined to be meaningful in table 2.1 are considered). Since the operations performed differ only minimally between operations requiring the same number of packet header loops, an equal mixture of all of them is used in all cases (e.g. in the example used previously 50% of swap and push and 50% of pop and swap operations). The packet size spectrum used represents ones typically encounterer in NPs.

Operations	No. MP loops required
Swap	1
Swap and Push	2
Swap and Push x 2	3
Swap and Push x 3	4
Pop and Swap	2
Pop x2 and Swap	3
Pop x3 and Swap	4
Pop, Swap and Push	3
Pop x2, Swap and Push	4
Pop, Swap and Push x 2	4

Tab. 5.2: Meaningful MPLS-TP Operation Sequences for which Power Consumption was Calculated

To maintain comparability with the results provided in 3.2.2, only the pipeline core was taken into consideration in this phase. Further on in this section, results for the entire NPU100 testbed design will be provided. In each scenario, the simulation starts with the NPU100 receiving the programming data stream being written to the appropriate modules. The complete NPU100 is programmed in all cases, independently of if it is required or not

Tab. 5.3: Percentage of Labels to be Processed for Different Traffic Scenarios for which Power Consumption was Calculated

No. of mini-pipeline loops	I (%)	II (%)	III (%)	IV (%)
1	40	25	80	65
2	20	25	15	20
3	20	25	5	10
4	20	25	0	5

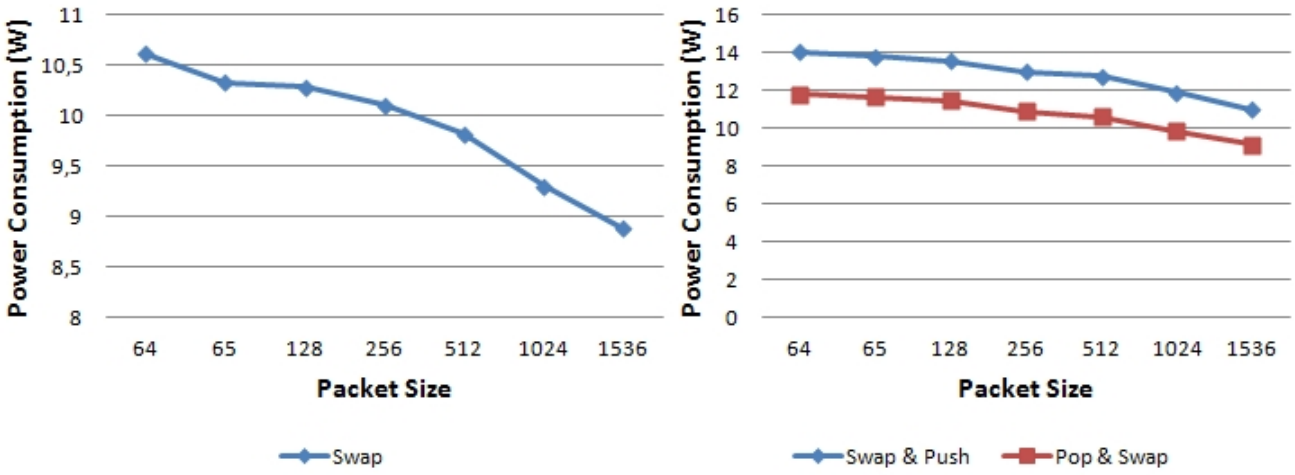


Fig. 5.4: Dynamic Power Consumption per Packet Size for Loop Run Scenarios One and Two

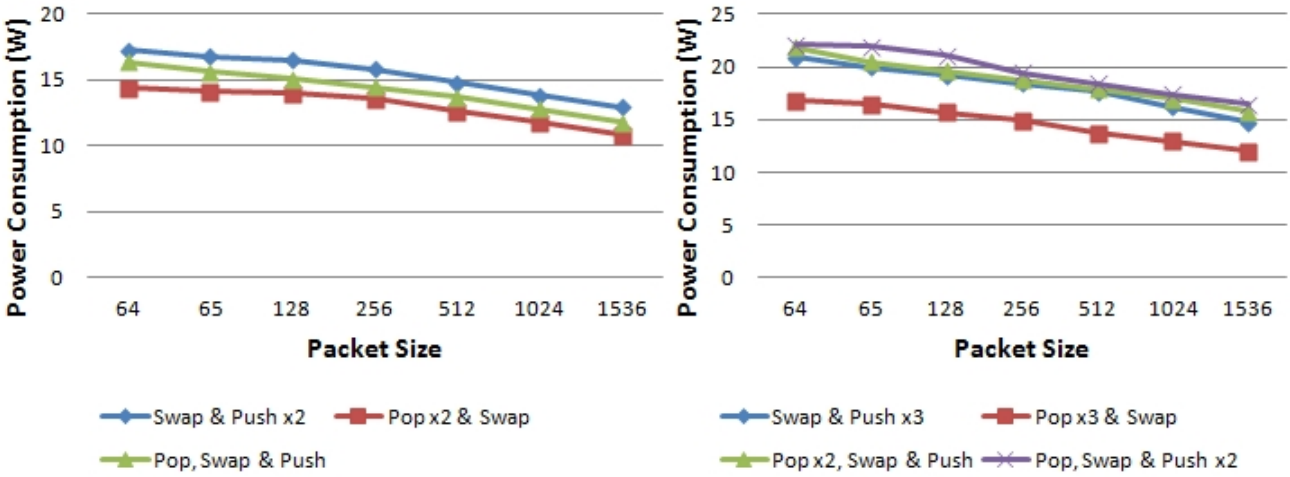


Fig. 5.5: Dynamic Power Consumption per Packet Size for Loop Run Scenarios Three and Four

(e.g. if a simple MPLS-TP label push is being simulated, only one MP is used and thus only this one has to be programmed) in order to maintain uniformity across all scenarios and mitigate the effect of programming the NP. Subsequently, packet headers are being

sent to the dispatcher at a rate emulating the appropriate packet size (when sending 64 byte packets, one header is sent every clock cycle, when sending 128 byte packet one every other clock cycle and so on).

Figures 5.4 and 5.5 illustrate the results for the ten different label stack operation sequences of table 5.2. These are grouped depending on the number of loop runs each sequence of operations requires, in order to enable a direct comparison between them and investigate how the operations performed impact power consumption. There are two trends which are immediately discernible. The first is that packet size has a very big influence on power consumption. This makes sense, since it affects toggling in the pipeline and thus dynamic power consumption. Thus small packet sizes, which translate to more toggling, consume a lot of power. In contrast, large packets only require that one packet header is processed every tens of cycles (e.g. 24 for a 1536 byte packet) and thus the processing modules remain idle for a significant amount of time.

The second trend is that label pops consume considerably less power than swaps and pushes. Again this is reasonable since a pop typically only uses the Forwarding lookup module and then simply flows down the pipeline. There is still, however, some power dissipation there from the pipeline registers toggling. By looking at the numbers for the various scenarios, we can reach the conclusion that a pop saves approximately 2W in comparison to a push or a swap.

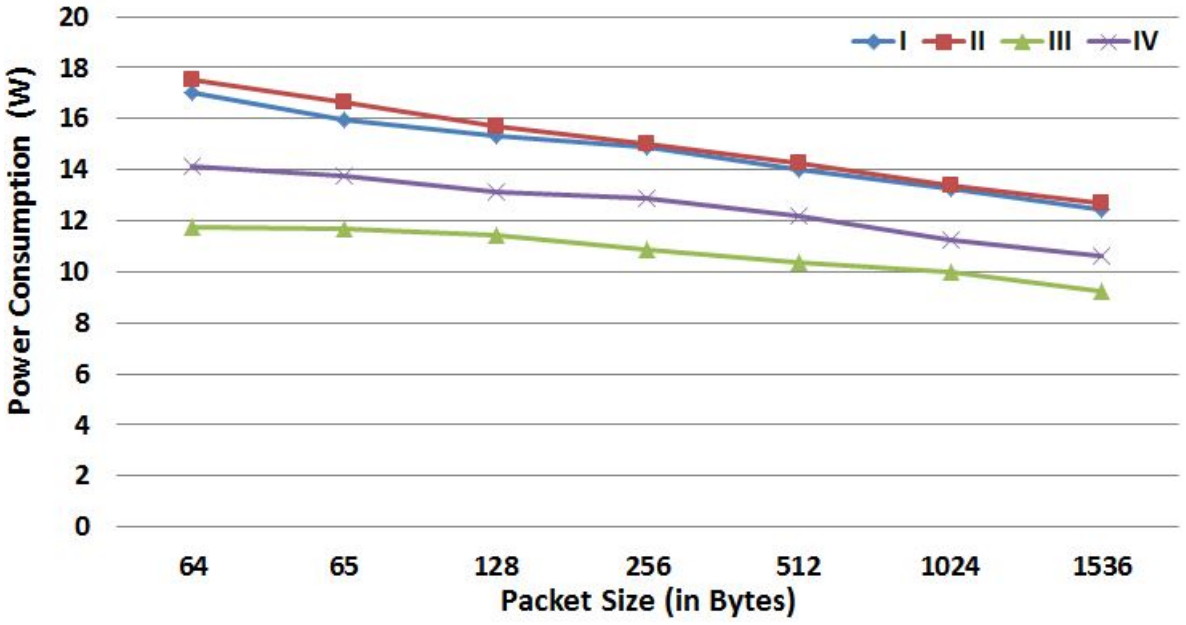


Fig. 5.6: Dynamic Power Consumption per Packet Size for Traffic Scenarios I through IV as Detailed in Table 5.3

Comparing the folded architecture's results of this investigation to the folded architecture results reached in the evaluation performed in section 3.2.2 reveals that they are similar. Pitting those results against the most comparable scenarios from the current set (that is, scenarios with 64 byte packets and 4 minipipeline loops) we can see that the current, more precise results show an even lower power consumption. The difference lies

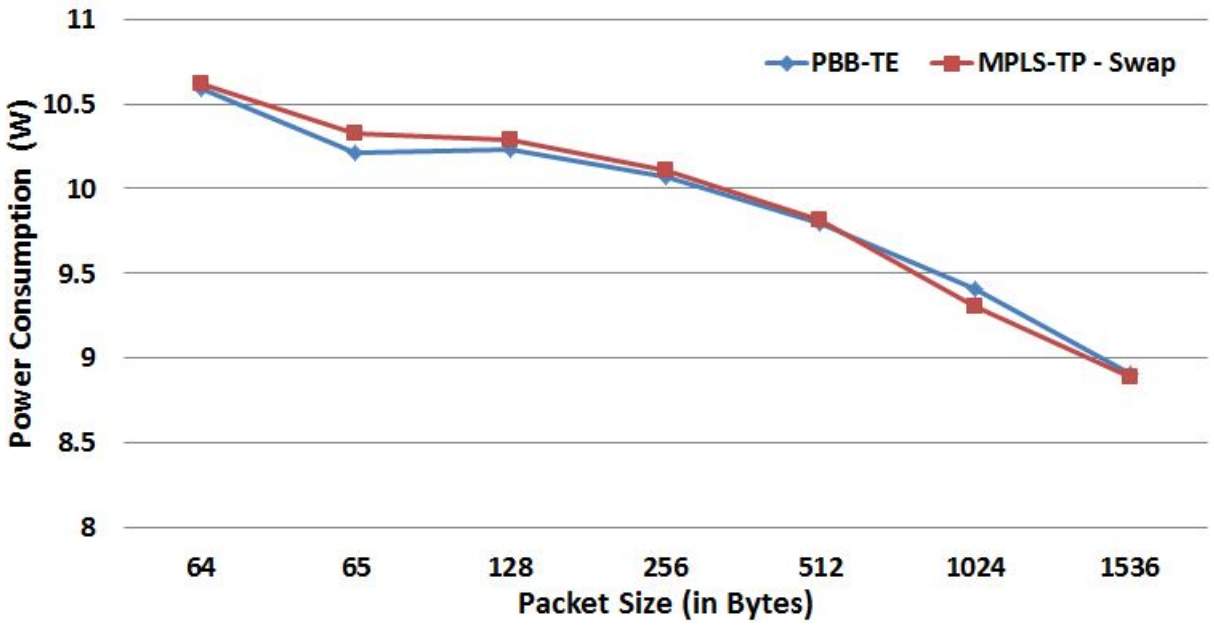


Fig. 5.7: Dynamic Power Consumption per Packet Size for PBB-TE Processing

between 1.5 and 7 Watts, depending on the sequence of operations to be performed. This reaffirms the validity of the evaluation performed during the quantitative comparison and the low power benefits of the folded pipeline architecture.

Figure 5.6 plots the dynamic power consumption of the four scenarios listed in table 5.3. Again the power consumption is inversely proportional to the packet size. Furthermore, it depends directly on the proportion of packets which have to perform multiple loops through the pipeline core. Thus scenarios 3 and 4, which have most packet being looped once or twice exhibit lower power consumption in contrast to scenarios 1 and two.

Finally, figure 5.7 illustrates the power consumption for forwarding PBB-TE frames, which require only one traversal of the pipeline core.Additionaly the power consumption for MPLS-TP processing of a simple swap is also shown as measure of comparison. The decreasing power consumption trend continues here as well. The take-away from this figure is that PBB-TE requires somewhat lower power consumption than MPLS-TP, which is reasonable since its functionality is significantly simpler (e.g. it doesn't use the Dispatcher and Label Reassembly Label Buffers at all).

Apart from the results described previously, worst case results for the entire test bed are provided in table 5.4. This time the PCIe interface is included, as well as results for each specific module. A worst case scenario in which all MPs are utilized in every clock cycle is used. This requires that 64 byte packets arrive constantly at the Header-Payload Split and that all packet headers require 4 MP loops. A pop, pop, swap and push operation was selected to utilize the biggest possible variety of operations in all MPs.

The results yield no surprises in that the pipeline core consumes the bulk of the power in the NPU100 as was expected. The outer layer modules add a considerable amount on top of that (approximately 3.5W in total power consumption, a bit less then an entire mini pipeline). This is reasonable since the modules included therein, and in particular the

Tab. 5.4: Detailed Overview of the Power Dissipation for the Various NPU100 Modules

	Static Power	Dynamic Power	Total Power
System Total	11.26	18.48	29.74
PCIe Total	0.11	0.1	0.22
PCIe NPU100	0.04	0.11	0.14
NPU100	10.9	17.17	28.11
Header Payload Split	0.23	0.38	0.6
Packet Memory	0.08	0.11	0.19
Packet Reassembly	0.53	0.97	1.21
Forwarding lookup Write Control	1.49	0.14	1.61
Pipeline Core	8.19	15.58	23.67
Dispatcher	0.56	0.49	1.05
Dispatcher Label Buffer	0.39	0.62	1.03
Label Reassembly	0.45	0.52	0.94
Label Reassembly Label Buffer	1.06	1.19	2.31
Mini Pipeline	1.48	3.25	4.62
Forwarding lookup	0.43	1.2	1.61
Insert	0.22	0.04	0.25
Replace	0.12	0.46	0.56
Modify	0.32	0.73	1.03
QoS lookup	0.39	0.78	1.15
ISU	0.04	0.16	0.21
OSU	0.08	0.29	0.35

Packet Reassembly and Forwarding lookup Write Control modules are quite complex. From these two, the Packet Reassembly only contributes considerably to the power dissipation, something expected, since the Forwarding lookup Write Control is used only sparsely during the initial configuration of the forwarding lookup tables, which explains its low dynamic power consumption. In contrast to this, the Packet Reassembly also exhibits continuous switching, due to its constant processing of packets in order to sent them to the output. Finally the impact of the PCIe and Ethernet modules is minimal, since the consume minimal resources and in the former's case have a very low switching factor (as it is to be expected, that information from the control plane or programming data are sent only sporadically to the NPU100).

5.1.3 Performance

In section 3.1.1 it was determined that in order to reach the required throughput of 100 Gbit/s the NPU100 has to be able to output one processed packet per clock cycle at a frequency of 150 MHz. If the design is synthesized, placed and routed using a Xilinx Virtex-5 FPGA, it achieves a maximum frequency of 112 MHz, thus being 38 MHz shy of

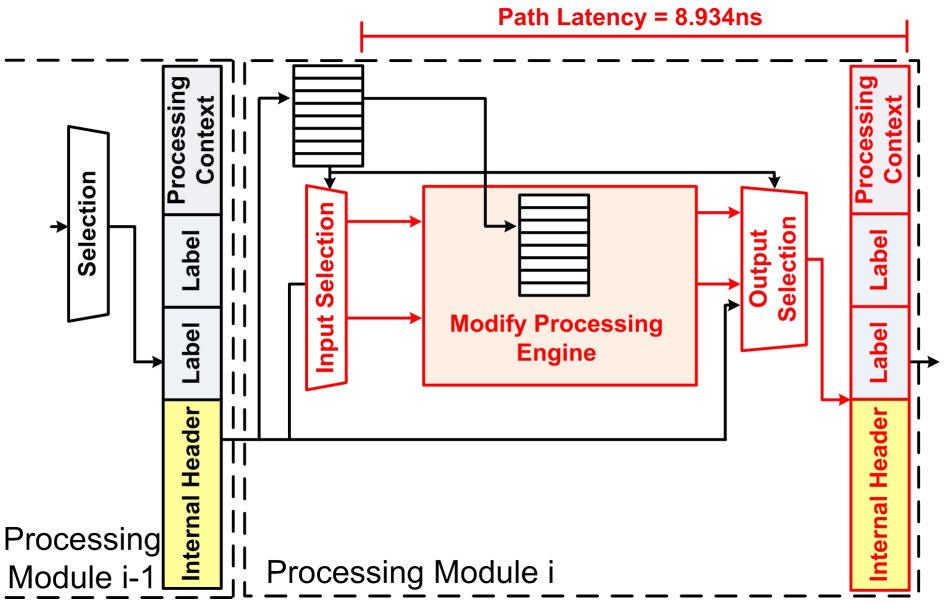


Fig. 5.8: Critical Path Constraining the Performance of the NPU100

the target frequency. Repeating the process with an 65nm ASIC library, however, results into a more than 2x performance improvement as will be described in detail further on in this section.

The NPU's critical path is located in the modify module, starting from the results of the microcommand table and flowing through the ISU, the processing module and the OSU and ending up in the pipeline register as figure 5.8 shows.

Inserting a pipeline stage alleviates the issue at a cost of additional registers. The pipeline stage can be inserted at the output of the processing element and before the OSU. This requires two times a 48 bit register plus a 28 bit register for synchronizing the microcommand table output to the OSU and finally a 192 bit register for the original data from the previous pipeline stage, thus in total 296 register bits are required. In additional to that, the Dispatcher an Label Reassembly buffers have to be expanded accordingly to accommodate two new pipeline stages (since there are two modify modules per MP) for every MP, thus 8 in total. This adds another 2368 register bits for in the label buffers.

Analyzing the critical path in the synthesis results, reveals that inserting a pipeline register can save 2.62 ns by shortening the path considerably as shown in figure 5.9.

In the FPGA implementation this is acceptable from a resource perspective, since as indicated in section 5.1.1, there is a significant number of registers unused, however it was avoided since it exacerbates the difficulties the router encounters when placing the design. This is an especially acute phenomenon in FPGA designs, where the fixed allocation of resources in CLBs can result in complex routing if resources are unevenly distributed (that is if in some CLBs only registers are consumed, whereas in others only slice LUTs are used up). This mitigates any gains from the additional pipeline stage. Nevertheless, an ASIC implementation will not be plagued by similar issues, since in that case the layout of the various functional blocks can be performed optimally. Keeping the above in mind, it was reasonable to expect that a subsequent implementation with an ASIC library is sure

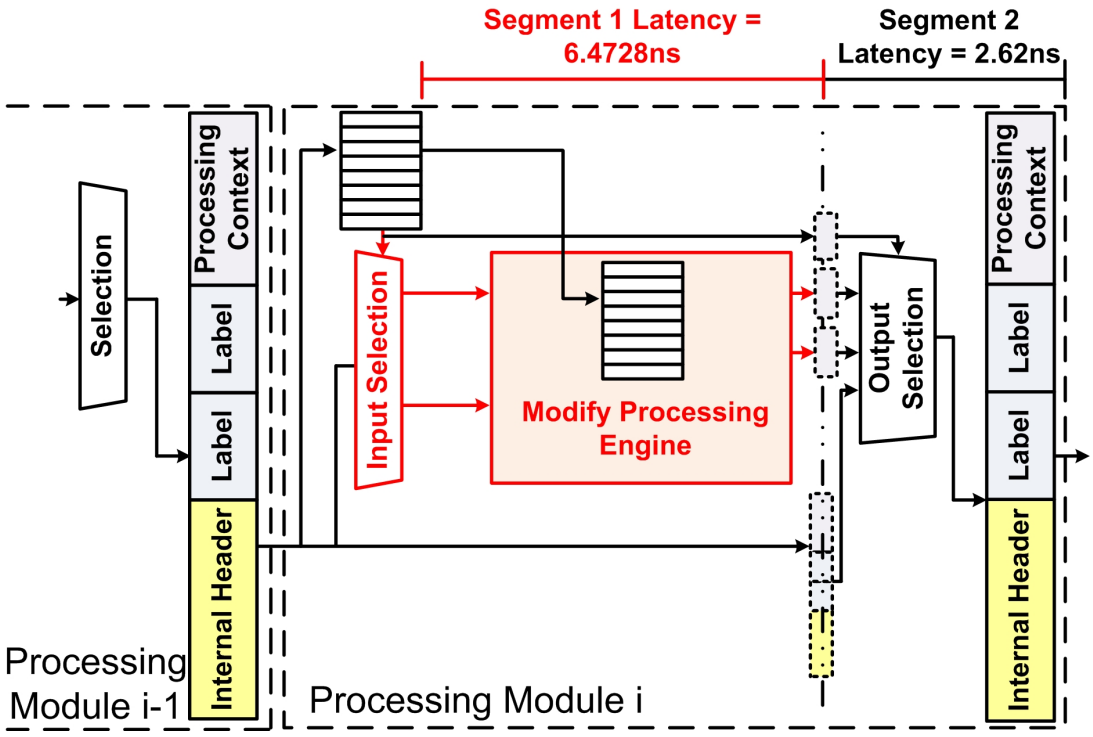


Fig. 5.9: Reduction of the Critical Path by Pipelining

to reach the required throughput, given that ASICs are typically at least 3-4 faster than FPGAs [41]

In order to verify this claim, we implemented the bottleneck module using an ASIC library using Synopsis's Design Compiler and Cadence's Encounter tools. The library used was a TSMC 65nm one, thus allowing room for improvement using newer, finer processes. However, it allows for a more direct comparison with the FPGA implementation, since the Virtex 5 devices are fabricated on a 65nm process as well. Figure 5.10 provides a screenshot of the fully routed modify module, which exhibits a critical path of 3.07ns and thus achieves a frequency of approximately 325Mhz, more than outdoubling the 150Mhz NPU100 requirement. Furthermore it should be stressed that there is significant room for improvement, as the placement and routing process was performed using the Encounter default settings and with very little optimization.

5.1.4 Scalability

While the current design is sufficient for 100 Gbit/s CGE Protocol processing, it is interesting to investigate how scalable the architecture is. Scalability can be seen in light of many different factors and design aspects. The mini-pipeline in itself has all the typical attributes of such a design and thus additional stages can be added, simply increasing the latency but having no impact on the throughput. The number of mini-pipelines itself can be increased, being limited by the available resources and the power budget of course. This depends heavily on integrated circuit fabrication technology, however with current ones there is sufficient headroom for a significant increase in the number of pipelines, par-

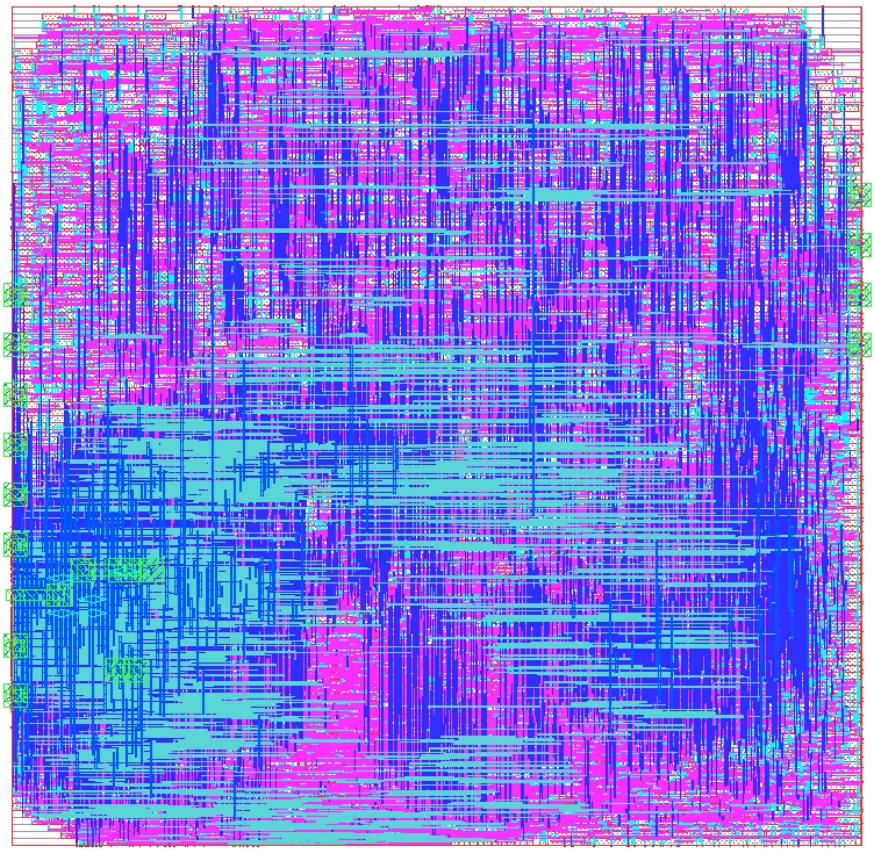


Fig. 5.10: Die Shot of a Fully Routed Modify Module

ticularly in an ASIC implementation. At 12W power consumption per MP for an FPGA and taking into account that an ASIC implementation typically reduces power consumption at least 4 times [41], it could very well be that even 20 MPs in a single ASIC is not be an unrealistic expectation. Increasing the number of mini-pipelines would yield even higher power consumption benefits in comparison to the traditional pipeline since in the latter's case the pipeline data width will grow linearly with the number of mini-pipelines.

The most probable soft spot in the folded architecture are the Dispatcher and Label Reassembly and their respective buffers. Increasing the number of MPs means that the Dispatcher and Label Reassembly and their buffers will have to be expanded accordingly in order to accommodate for the increase in packet headers. Since the design of the modules is modular increasing their size simply involves changing the value of a couple of generics in the VHDL code. The implication of the extra circuitry on its performance should be minimal, since the Dispatcher and Label Reassembly consist of parallel pipelines which process the headers coming from the various NPs. Thus, increasing the number of parallel modules should increase resource consumption without however impacting performance adversely.

To test this, we have synthesized our system with 8 mini-pipelines and using the same Virtex-5 LX330T FPGA as in the previous section. The results for the Dispatcher and Label Reassembly modules as well as their two buffers are provided in figure 5.11, in order

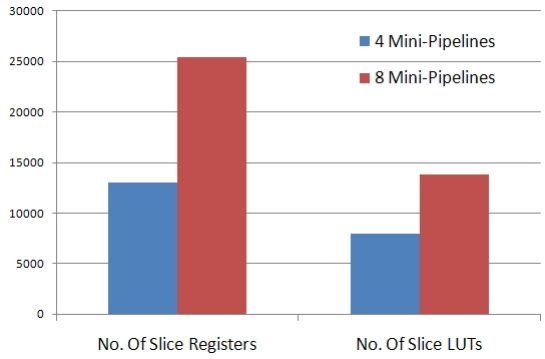


Fig. 5.11: Dispatcher and Label Reassembly Module Resource Consumption for 4 and 8 Mini-Pipelines

to gain insight into the resource consumption, which as expected roughly doubles. The achieved frequency (something important due to the significantly increased complexity of the modules), is somewhat reduced from 300+ MHz for 4 mini-pipelines to approximately 280 for 8, but it still remains above the target frequency, confirming our hypothesis about the scalability of our design.

5.2 NPU100 Test Bed

This section provides an overview of the NPU100 test bed, as well as the design flow that is employed in order to verify the functionality of the NP. It describes of each component and of the functionality it offers.

5.2.1 NPU100 Test Bed Description

The test bed is made up of four components:

1. The NPU100 processing core itself, which is the core of the design and includes the entire NP with its processing engines and any accompanying modules (e.g. for the storage of packet during processing), as described in the previous chapter.

2. The PCI Express interface, which comprises a software and a hardware part and facilitates the generation and transfer of the programming information to the NP, as described in section 4.3.

3. An Ethernet interface, along with a "bucket" system, which makes the interconnection of the NPU100 with the Spirent traffic generator possible.

4. External components, namely an expansion board which allows for the connection of the FPGA board containing the NPU100 to the logic analyzer for debugging purposes, the logic analyzer himself and the Spirent traffic generator used to stimulate the design.

Figure 5.12 illustrates how these components are interconnected. Since the NPU100 architecture, module and programming subsystem implementation itself have been thoroughly described in chapters 3 and 4, thus the following sections will focus on the remaining components.

The FPGA is the heart of the NPU100 test bed and thus a suitable evaluation board, which combined a device offering sufficient capacity with the peripherals, which will be required for the test bed, was necessary. The board ought to have Ethernet ports for transmitting and receiving data to and from the NPU100, as well as a PCI Express link to send the programming data to the device. Taking these into account, one High Tech Global HTG-V5-PCIE-LX330T board (shown in the center of figure 5.12) was used. This board includes one Virtex 5 LX330T FPGA device, which is the largest in the Virtex 5 family, and furthermore incorporates one PCI Express x8 port, two Gigabit Ethernet ports and two Samtec QSE-060-01-F-D-A connectors, which provide the capability to monitor up to 138 different FPGA pins.

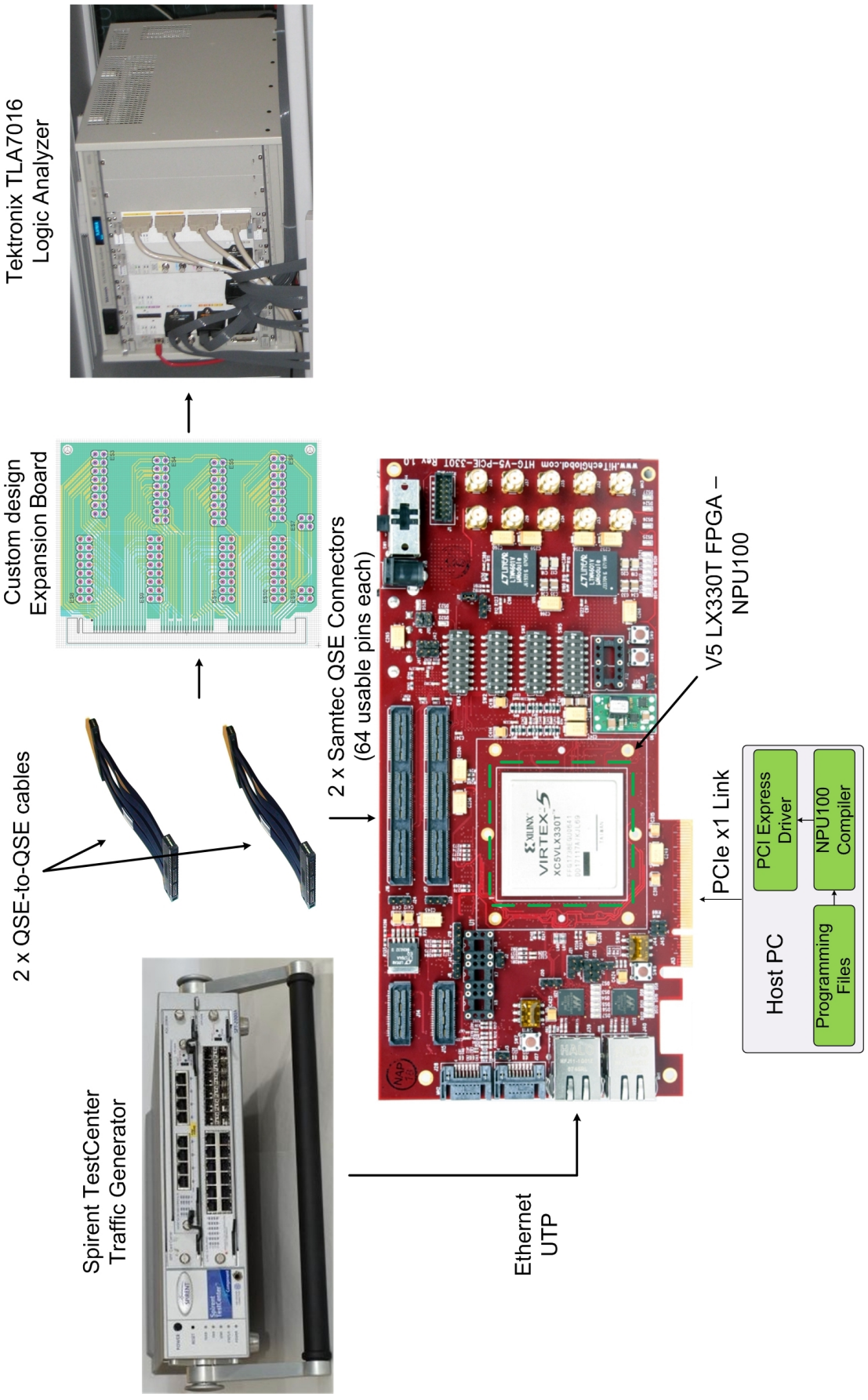


Fig. 5.12: Overview of the NPU100 Testbed Setup

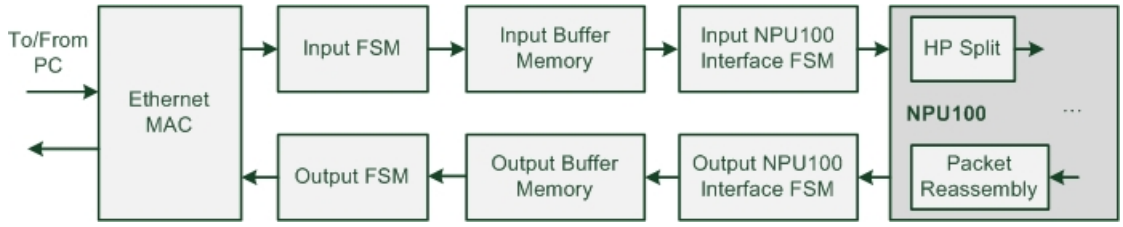


Fig. 5.13: Ethernet Interface Module Chain Block Diagram

The Samtec connectors are used to transfer data to the logic analyzer, which the captures and displays them in order to verify the functionality of the NPU100. Since the Samtec connectors were incompatible with the ones found on the logic analyzer probes, a special expansion board was designed to convert between them. The FPGA board is connected to the expansion board through two QSE-to-QSE cables.

Feeding packet stimuli into the NPU100 is done over a standard Ethernet interface, which provides the means to sent data from the Spirent traffic generator to the NPU100 and vice versa. Since the Spirent traffic generator can produce traffic at rates of up to 1 Gbit/s, and the FPGA board used only support 1 Gbit/s ports, a bucket system is employed to throttle the incoming traffic up to a 100 Gbit/s and respectively throttle outgoing traffic down to 1 Gbit/s so it can be sent back to the traffic generator.

Figure 5.13 illustrates the modules used to achieve this. The basic building block is the Virtex-5 Embedded Tri-Mode Ethernet MAC Wrapper, which connects to the Marvell PHYs on the FPGA board to transmit and receive Ethernet frames.

The Xilinx MAC core provides a specific interface from which data are to be received by the user logic. This interface, apart from being incompatible with the input interface of the NPU100 at the Header Payload Split module, it is also incapable of transferring data at a throughput of 100 Gbit/s. Thus a chain of modules is inserted at the input to bridge this throughput gap and to render the interface between the MAC core and the NP compatible by allowing the packets to be stored in a temporary memory and then to be sent to the NPU100 at a higher throughput. A similar module chain at the output performs the reverse functionality.

At the input the data are initially received by a state machine which fetches the data from the MAC core and stores them into a Block RAM memory on the FPGA. This memory provides enough space for a maximum of 1024 512 Bit words. This decision was dictated by the limited amount of memory on the FPGA and the need to use this memory for other purposes in the NPU100 design. For each packet written into the Block RAM an entry stating its size is written into a parallel FIFO queue. This helps the subsequent module delimit each packet correctly. A second FSM reads the data from the Block RAM and interfaces with the NP in order to deliver the data to it. This FSM implements the receive interface of the Header-Payload Split module and allows the transfer of the packets in the Block RAM at a throughput of 100 Gbit/s. The FSM allows for two different functional modes to be utilized, the toggling of which is performed through the DIP switches found on the FPGA board. In the first the data is accumulated in the memory and released upon triggering the state machine through an input signal. In this mode there are two variants.

The first allows for purging the memory once and sending its content into the NPU100. The second replays the memory contents continuously, thus creating an endless 100 Gbit/s stream of packets, which can be used to test the NPU100 functionality over a long period of time. Another DIP switch is used to select between the two variants. When in the latter mode, the changing the position of the DIP switch stops the memory replay process. The second operating mode reads the data out, immediately after they have been written to the memory, thus essentially sending a 1 Gbit/s stream of packets to the NPU100. This could be used for testing longer run times on the NPU100, albeit at a lower speed.

At the output, the reverse process is performed. An FSM receives processed packets from the Packet Reassembly module of the NPU100 and stores them into a buffer memory. The capture process is activated by a DIP switch. Until the DIP switch is activated the FSM ignores any incoming packets. When the DIP switch is toggled it starts storing packets into the memory, starting with the next complete one. Since the process is the exact reverse of the input one, a space for up to 1024 512 bit data words is provided by this memory as well, along with an extra FIFO that holds the packet length of each packet. A second FSM reads the packets and sends them to the Xilinx Ethernet MAC core.

Table 5.5 provides an overview of the implementation results for the bucket system. The entire Ethernet subsystem, as well as the NPU100 testbed specific modules require only modest resources in comparison to what the LX330T FPGA offers.

Tab. 5.5: Resource Consumption of the NPU100 Testbed Ethernet Block

	No. of Slice Registers	%	No. of Slice LUTs	%	No. of BRAMs	%
Ethernet Total	3972	2	2871	1	44	13
Ethernet NPU100	1894	1	2834	1	44	13

The NPU100 testbed deviates slightly from the implementation developed and described in the previous chapters. This was dictated by several factors, the first and foremost being the unavailability of 100 Gbit/s equipment on which to implement a complete test chain. At the time of implementation no FPGA board with 100 Gbit/s Ethernet PHYs was available, let alone test equipment like traffic generators, which would have been necessary for testing. Thus in order to verify the NPU extensively it had to be coupled to the Spirent traffic generator available at LIS. Neither the traffic generator nor the board used for the prototype support 100 Gbit/s Ethernet ports, thus a compromise was made to use 1 Gbit/s traffic ports, together with the bucket system described in this section. This in turn means that the Interlaken interconnection described in section 4.1.1 was replaced by the required Ethernet MAC cores.

An additional compromise that was struck, was to reduce the number of MPs found in the NPU100 test bed implementation to two instead of four. This reduces the FPGA resource use of the NPU100 considerably and provides the tools with enough headroom in order to satisfy the stringent timing requirements of the Ethernet and PCI Express cores. Furthermore it allows the saving of precious BRAMs, which could then be used for the bucket system described herein. This compromise refers only to the actual functional

validation tests performed with the NPU100 test bed. This reduction detracts nothing from the use of the test bed as a vehicle for the validation of the concepts introduced in this work, since these remain identical regardless of the number of MPs that one chooses to include in the design. In fact, one of the goals of the design was to allow it to operate with a configurable number of MPs without need for additional VHDL code modifications.

5.2.2 NPU100 Verification and Test Bed Results

The NPU100 testbed was used to verify the functionality of the design using the scenarios described throughput this work. Both the compiler and programming module chain, as well as the functionality of the NPU100 modules had to be verified. To test the former MPLS-TP and PBB-TE descriptions were written, including all four required configuration files and were run through the compiler to generate the appropriate programming files. These were then sent to the NPU100 over the PCIe interface and used to program the processing modules. Special attention was paid to verifying the correct writing and deleting of entries in the forwarding and QoS lookup tables. This was done by performing extensive tests, in which series of entries were written to both memories and then some were deleted while the NPU100 was processing. This tests proved that the NPU100 does not forward packets when a suitable forwarding lookup entry is not found, that it starts doing so the moment a suitable entry is written (while the system is in operation) and stops again when the entry is deleted again. Similar tests were conducted for the QoS lookup module. A further test necessary to completely cover additional test cases present only in the forwarding lookup module are those related to the intricacies of the hash algorithm implemented therein as well as the functionality of the entry selection module and the filtering module. Thus, tests using consecutive writes to the same address were performed so as to determine if they are written to the appropriate entry in each bin each time and if the Black Sheep Memory is used after all entries have been exhausted. The logic analyzer was used to verify that entries were correctly written to and read from the bins and that subsequently the correct entries were chosen in the selection module, as well as filtered out in the filtering module depending on their programming. In all of the above cases the programming data was generated automatically by the compiler as described in section 4.3.

The processing functionality of the NPU100 was tested by using both MPLS-TP and PBB-TE traffic. In the former case all scenarios listed in table 2.1 were tested first separately and then in mixed tests utilizing all operations in parallel. For each scenario 4 different classes of service per label were used. The factor limiting the breadth of different packet behaviors that can be tested is the bucket system, that buffers packets from the traffic generator before sending them to the NPU100 at a 100 Gbit/s line rate. Since the packet buffer can fit a finite number of packets (depending on their size), which it then loops endlessly, the number of scenarios that can be tested is limited by the number of different label/Cos combinations that can fit into the bucket. Testing PBB-TE is less complex , since PBB-TE does not requires frames to use more than one MP for processing under any circumstances. The loopback functionality was thus not utilized in these tests. In the PBB-TE tests 10 different flows were used with four different traffic behaviors per flow. The results verify the functionality of the NPU100 as an architecture that can process different Carrier Grade Ethernet Protocols by testing all aspects of the design

comprehensively.

From a performance perspective, the demonstrator exhibits a drop in frequency in comparison to the stand alone version of the NPU100. The drop in frequency is equal to 36 MHz, from 112, which the standalone version reaches, down to 66. The cause for this drop in performance is the placement and routing difficulty with which the tool is confronted due to instantiation of the Ethernet and PCI Express cores. These two cores consume relatively few resources, however they display very strict timing requirements, which leads to a need to place the cores very carefully in the device, in order for these constraints to be met. This means that the place and route tool must work its way around these constraints something that results in the afore mentioned drop in achievable frequency for the remainder of the NPU. This limitations stems however mostly from the combination of FPGA and demonstrator board used and its effect would be mitigated if a custom FPGA board were to be designed.

6 Conclusions

This thesis presented the NPU100 network processor, an NP architecture aiming at processing packets in a Carrier Grade Ethernet network at an appropriate throughput (meaning 100 Gbit/s at the moment this is written) in a single chip solution, which must be flexible, modular and power efficient, while remaining easy to program. With these stated goals in mind, this work makes two main contributions to the state of the art:

- A novel pipeline architecture, called the folded pipeline, which reduces the pipeline width and thus the number of bits that have to toggle in each clock cycle. This leads to significantly lower power consumption in comparison to a conventional pipeline structure.
- A easy-to-use programming paradigm, something conspicuously absent from current NP designs. This paradigm allows the NPU100 user to only describe the functionality he requires of the NP in a high-level language. A compiler then takes care of the task assignment to the pipeline stages and the generation of the commands for each of them.

The folded pipeline architecture is a pipelined approach, which breaks the workload down into smaller increments in order to reach the necessary frequency, while ensuring the determinism needed to achieve a constant flow of over 100 Gbit/s of traffic. This is achieved by dividing the required functionality into elementary processing entities called mini pipelines. Each mini pipeline is made up of various processing modules, which suffice to achieve basic packet processing. The mini pipelines are then parallelized and joined together through two modules, the Dispatcher and the Label Reassembly, thus forming a loop. The two afore mentioned modules control the flow of packets through the loop, making sure that those that need additional processing are sent back to go through an additional mini-pipeline, while forwarding any completed packets to the output. The processing modules that make up each mini pipeline all share a common I/O interface, which means that if need be they can be exchanged for new ones swiftly and effortlessly. This approach ensures that the design is adaptable to future needs as well as expandable.

Apart from providing a modular design, which can be changed to fit new requirements, an NPU100 implementation must itself be flexible enough to process varying protocols without the need to modify the hardware. This is accomplished by providing programmable processing modules and a programmable I/O interface, whose functionality can be set by the user dynamically at run time. Programmability is realized by employing microcommands, which saves on decoding complexity and thus contributes to the NP processing data at the required throughput. However this approach leads unavoidably to reduced instruction density and thus only a small number of instructions per processing module or I/O interface can be supported. These are however still sufficient to cover all needed

network processing operations in a carrier network. The microcommands are generated automatically by the NPU100 compiler, which was also developed as part of this work. The compiler reads XML configuration files and a source file written in a C-like, high-level language and converts the code to microcommands, while allocating the microcommands to the available processing stages. The microcommands are then written into the NPU100 via an industry-standard PCI Express interface.

Due to the folded pipeline architecture, the NPU100 achieves impressive power consumption savings, ranging from 25% to 78%, depending on the application scenario in comparison to a traditional pipeline architecture. This result is reached by reducting the number of bits that flow through the mini pipelines, achieved by storing redundant data momentarily in two buffers found in the Dispatcher and the Label Reassembly modules and an early exit strategy, which avoids sending packet headers on unneeded pipeline loops.

While this work represents a significant advancement in two key areas of NP design in comparison to the state of the art, namely power consumption and programmability, it is surely not the last word in what can be achieved within the confines of the folded architecture proposed. More specifically, two improvements can be used to further harvest the advantages of this design. The first is to use dynamic partial reconfiguration to exchange or even erase mini pipeline modules. This will increase the flexibility of the design by allowing it to change is functionality considerably more than what the current programmability scheme allows it to and will furthermore even contribute to the reduction of static power dissipation by eliminating no longer needed modules from the system.

In parallel, a power gating scheme can be implemented to reduce or even shut down power to specific mini pipelines. This would extend the savings to static power dissipation. What should be noticed in the implementation of such a scheme is that multiple cycles will be needed to restore power to a mini pipeline, which means that the mechanism devised to perform this task must be able to respond timely to incoming packets and reactivate the needed MPs.

Bibliography

[1] 802.1P - Traffic class expediting and dynamic multicast filtering.

[2] IEEE standard for local and metropolitan area networks - virtual bridged local area networks - amendment: Provider backbone bridge traffic engineering.

[3] OIF-SPI2-02.1 - System Packet Interface level 4 (SPI-4) phase 2 revision 1: OC-192 system interface for physical and link layer devices.

[4] PCIe base 3.0 specification.

[5] RFC796 - address mappings, 1981.

[6] Tiny XML - http://sourceforge.net/projects/tinyxml/.

[7] RFC791 - Internet Protocol specification, September 1981.

[8] IEEE: 802.2AE-2002 - amendment: Media Access Control (MAC) parameters, physical layers, and management parameters for 10 Gbps operation, 2002.

[9] IEEE standard for local and metropolitan area networks - virtual bridged local area networks, 2005.

[10] Interlaken protocol definition, October 2008.

[11] Aho A., Sethi R., and Ullman J. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[12] Xelerated AB. HX network processor family, http://www.xelerated.com.

[13] Netronome. Netronome NFP-3200 network processor - http://www.netronome.com.

[14] J. R. Allen, B. M. Bass, C. Basso, R. H. Boivie, J. L. Calvignac, G. T. Davis, L. Frelechoux, M. Heddes, A. Herkersdorf, A. Kind, J. F. Logan, M. Peyravian, M. A. Rinaldi, R. K. Sabhikhi, M. S. Siegel, and M. Waldvogel. IBM PowerNP network processor: Hardware, software, and applications. *IBM Journal of Research and Development*, 47(2.3):177 –193, march 2003.

[15] AMD. AMD: Unleashing the power of parallel computing. In *Proceedings of the SIGGRAPH Asia 2009 Conference*, 2009.

[16] Masanori Bando, N. Sertac Artan, and H. Jonathan Chao. Flashlook: 100-Gbps hash-tuned route lookup architecture. In *Workshop on High Performance Switching and Routing (HPSR 2009)*, Paris, France, June 2009.

[17] S. Blake, D. Black, M. Carlson, E.Davies, Z. Wang, and W. Weiss. RFC2475 - an architecture for differentiated services, December 1998.

[18] M. Bocci, S. Bryant, and L. Levrau. A framework for MPLS in transport networks, November 2008.

[19] M. Bocci, M. Vigoureux, and S. Bryant. RFC5586 - MPLS generic associated channel, June 2009.

[20] R. Bolla, R. Bruschi, F. Davoli, and F. Cucchietti. Energy efficiency in the future internet: A survey of existing approaches and trends in energy-aware fixed network infrastructures. *IEEE Communications Surveys and Tutorials*, 13:223–244, 2011.

[21] A. Broder and M. Mitzenmacher. Using multiple hash functions to improve IP lookups. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1454 –1463 vol.3, 2001.

[22] S. Bryant and P. Pate. Pseudo wire emulation edge-to-edge (PWE3) architecture, March 2005.

[23] EZ Chip. NP-5 network processor family, http://www.ezchip.com/pr_110524.htm. 2011.

[24] Cisco. The Cisco Quantumflow processor: Cisco's next generation network processor. 2008.

[25] Intel Corp. Intel IXP2800 network processor: For OC-192/10 Gbps network edge and core applications.

[26] LSI Corp. APP3300 communication processor product brief.

[27] Schwartz Eitan. Mobile backhaul: Challenges and opportunities. *White Paper*.

[28] EZ. EZchip introduces NPS c-programmable NPU. *http://www.ezchip.com/pr_120905.htm*, 2012.

[29] F. Le Faucher, L. Wu, B. Davie, S. Davari, P. Vaananen, R. Krishnan, P. Cheval, and J. Heinanen. RFC3270 - Multi-Protocol Label Switching (MPLS) support of differentiated services, May 2002.

[30] D. Fedyk and D. Allan. Ethernet data plane evolution for provider networks. *IEEE Communications Magazine*, 46:84–89, 2008.

[31] Kerim Fouli and Martin Maier. The road to carrier-grade ethernet. *Comm. Mag.*, 47:30–38, March 2009.

[32] Freescale. QorIQ amp series t2080 communications processor fact sheet. 2012.

[33] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a GPU-accelerated software router. *SIGCOMM Comput. Commun. Rev.*, 40:195–206, August 2010.

[34] Heinanen J., Baker F., W. Weiss, and J. Wroclawski. Assured forwarding PHB group, June 1999.

[35] Bob Jenkins. Hash functions. *Dr. Dobbs Journal*, 1997.

[36] Jia Yu Jignan, Yao Bhuyan, and L. Jun Sang. Program mapping onto network processors by recursive bipartitioning and refining. In *Proceedings of the 44th ACM/IEEE Design Automation Conference (DAC)*, 2007.

[37] C. Johnson, D.H. Allen, J. Brown, S. Vanderwiel, R. Hoover, H. Achilles, C.-Y. Cher, G.A. May, H. Franke, J. Xenedis, and C. Basso. A wire-speed Power processor: 2.3GHz 45nm SoI with 16 cores and 64 threads. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 104 –105, feb. 2010.

[38] A. Kirstädter, C. Gruber, J. Riedl, and T. Bauschert. Carrier-grade ethernet for packet core networks. volume 6354, page 635414. SPIE, 2006.

[39] Reimund Klemm and Gerhard Fettweis. Bitstream processing for embedded systems using C++ metaprogramming. In *Design, Automation, and Test in Europe*, pages 909–913, 2010.

[40] Donald Knuth. *The Art of Computer Programming Vol.3 - Sorting and Searching*. Addison-Wesley, 1998.

[41] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. In *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 21–30, New York, NY, USA, 2006. ACM.

[42] Shuai Mu, Xinya Zhang, Nairen Zhang, Jiaxin Lu, Y.S. Deng, and Shu Zhang. IP routing processing with graphic processors. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 93 –98, march 2010.

[43] Netronome. Netronome accelerator cards overview: http://www.netronome.com/pages/acceleration-cards. 2010.

[44] Netronome. Netronome NFP-6xxx 200 Gbps programmable flow processors product brief. 2012.

[45] nVIDIA. nVidia's next generation CUDA compute architecture: Fermi.

[46] Mohammad Peyravian and Jean Calvignac. Fundamental architectural considerations for network processors. *Comput. Netw.*, 41(5):587–600, 2003.

[47] Hinden R. and Deering S. RFC2373 - IP version 6 addressing architecture, July 1998.

[48] E. Rosen, A. Viswanathan, and R. Callon. RFC3031 - Multi Protocol Label Switching architecture, January 2001.

[49] Freescale Semiconductor. MPC8572e Powerquicc III integrated processor hardware specifications.

[50] M. Singh and D. Garg. Choosing best hashing strategies and hash functions. In *Advance Computing Conference, 2009. IACC 2009. IEEE International*, pages 50 – 55, march 2009.

[51] Cisco Systems. Cisco carrier routing system - http://www.cisco.com.

[52] Cisco Systems. Serial-GMII specification.

[53] Cisco Systems. Cisco green research symposium - introduction and overview. 2008.

[54] Ngarua Technologies. GAPP - GPU accelerated packet processing: http://ngarua.com/index.php.

[55] D. Thaler and c. Hopps. RFC2991 - multipath issues in unicast and multicast next-hop selection, November 2000.

[56] M. Uhm. Multi-mode basestation common platform software dened radio: What's in a name? In *Proceedings of the Software Digital Radio,TechnicalConference (SDR08)*, 2008.

[57] Fuller V., Li T., Yu. J, and Varadhan K. RFC1519 - classless inter-domain routing (CIDR): An address assignment and aggregation strategy, September 1993.

[58] Jacobson V., Nichols K., and Poduri K. An expedited forwarding PHB, June 1999.

[59] J. Wagner and R. Leupers. C compiler design for a network processor. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(11):1302 –1308, nov 2001.

[60] Ning Weng and Tilman Wolf. Pipelining vs. multiprocessors - choosing the right network processor system topology. In *in Proc. of Advanced Networking and Communications Hardware Workshop (ANCHOR 2004) in conjunction with The 31st Annual International Symposium on Computer Architecture (ISCA 2004*, 2004.

[61] Adam Willen, Justin Schade, and Ron Thornburg. *Introduction to PCI Express: A Hardware and Software Developers Guide.* Intel Press, 2003.

[62] Xelerated. Data flow architecture: http://www.xelerated.com/en/dataflow-architecture/.

Own related publications

[63] S. Hauger, T. Wild, A. Mutter, A. Kirstädter, K. Karras, R. Ohlendorf, F. Feller, and J. Scharf. Packet processing at 100Gbps and beyond - challenges and perspectives. In *Proceedings of the 10. ITG-Fachtagung Photonische Netze*, 2009.

[64] K. Karras, D. Llorente, T. Wild, and A. Herkersdorf. Improving memory subsystem performance in network processors with smart packet segmentation. In *Embedded Computer Systems: Architectures, Modeling, and Simulation, 2008. SAMOS 2008. International Conference on*, pages 210–217, July.

[65] Kimon Karras, Thomas Wild, and Andreas Herkersdorf. A folded pipeline network processor architecture for 100 Gbit/s networks. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '10, pages 2:1–2:11, New York, NY, USA, 2010. ACM.

[66] D. Llorente, K. Karras, M. Meitinger, H. Rauchfuss, T. Wild, and A. Herkersdorf. Accelerating packet buffering and administration in network processors. In *Integrated Circuits, 2007. ISIC '07. International Symposium on*, pages 373–377, Sept.

[67] D. Llorente, K. Karras, T. Wild, and A. Herkersdorf. Advanced packet segmentation and buffering algorithms in network processors. *HiPEAC - Transaction on High Performance Embedded Architectures and Compilers*, 4, 2009.

[68] D. Llorente, K. Karras, T. Wild, and A. Herkersdorf. Buffer allocation for advanced packet segmentation in network processors. In *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*, pages 221–226, July.