POLYPUBLIE
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Reconfigurable Programming of Data Plane Communication Networks on FPGA Platforms
Auteur: Author:	Parisa Mashreghi-Moghadam
Date:	2025
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Mashreghi-Moghadam, P. (2025). Reconfigurable Programming of Data Plane Communication Networks on FPGA Platforms [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/71350/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/71350/
Directeurs de recherche: Advisors:	Tarek Ould-Bachir, & Yvon Savaria
Programme: Program:	Génie électrique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Reconfigurable Programming of Data Plane Communication Networks on FPGA Platforms

PARISA MASHREGHI MOGHADAM
Département de génie électrique

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie électrique

Décembre 2025

© Parisa Mashreghi Moghadam, 2025.

POLYTECHNIQUE MONTRÉAL
affiliée à l’Université de Montréal

Cette thèse intitulée :

Reconfigurable Programming of Data Plane Communication Networks on FPGA Platforms

présentée par **Parisa MASHREGHI MOGHADAM**
en vue de l’obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d’examen constitué de :

Guy BOIS, président
Tarek OULD-BACHIR, membre et directeur de recherche
Yvon SAVARIA, membre et codirecteur de recherche
Felipe GOHRING DE MAGALHAES, membre
Fakhreddine GHAFFARI, membre externe

DEDICATION

To my mom and dad,

my sisters,

and my love ♡

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisors, Prof. Yvon Savaria and Prof. Tarek Ould-Bachir, whose guidance has profoundly shaped both this thesis and my academic path. To Prof. Savaria, thank you for opening the door to this journey. Your confidence in my potential gave me the courage to enter a new field and pursue a direction that ultimately defined my future. I feel truly fortunate to have been supervised by you before your retirement. To Prof. Ould-Bachir, thank you for guiding me from my very first steps in digital design. Your dedication was invaluable when everything felt new and overwhelming. The time you invested in teaching and mentoring me laid the foundation for all the work that followed. I am sincerely grateful for your mentorship.

My deepest gratitude belongs to my mom and dad, the two pillars of my life. Your countless sacrifices, quiet strength, and boundless love have carried me farther than any journey ever could. Thank you for enduring the distance, for allowing me to cross oceans and continents in pursuit of my dreams, even when it meant holding your hopes and worries from afar. Everything I strive for has always been driven by the wish to make you proud. This accomplishment is as much yours as it is mine. You are, and will always remain, my greatest blessing. I am also deeply grateful to my sister Parastoo and her husband Reza, whose kindness and support carried me through many difficult moments. The comfort of their presence, and even the light, joyful rustle of Sunny's little paws, made the hardest days gentler and the longest nights easier to bear. To my little sister Sarina, who means the world to me, thank you for standing beside me through every hardship and holding my hand day after day. Your presence softened the heaviest moments and gave me the strength to keep moving forward.

To the love of my life, my husband Amir: You have walked with me through every step of this journey, always believing in me. Your patience, unwavering support, and steady encouragement gave me the strength to continue, even when the path felt overwhelming, and carried me through the most challenging days. Over more than a decade together, we have grown side by side, facing challenges and celebrating milestones that have shaped us in ways I will always treasure. Your presence, your faith in us, and the unconditional love you give so generously have made this journey warmer, steadier, and filled with moments of joy.

A special acknowledgment goes to Amirali and Mehrnoosh, who have been like family to me throughout this journey. Amirali, your constant help and support for both my family and me meant more than words can express. Mehrnoosh, my closest friend, thank you for standing beside me through every difficult moment and for helping me find strength.

Warm appreciation is also owed to Amin, Morteza, and Diba, whose support made my early days in Montréal possible. Their kindness, presence, and willingness to help during moments of uncertainty greatly eased my transition to a new city, and their generosity will always be remembered. Also, thanks to Mina, Javad, and Reza, whose encouragement and unwavering belief in me have been a steady source of strength.

I would also like to thank Jeferson, Thomas, and Thibaut, my peers in the research group, whose discussions and insights helped refine my ideas and strengthen this thesis. Finally, my sincere appreciation goes to my colleagues at National Bank Capital Markets, whose support and encouragement accompanied me throughout this journey.

RÉSUMÉ

Les réseaux définis par logiciel (*Software-Defined Networking* — SDN) et le langage P4 ont introduit la programmabilité dans les architectures de plan de données en découplant le comportement de transfert de son implémentation matérielle. Les réseaux prédiffusés programmables (*Field-Programmable Gate Array* — FPGA) se sont imposés comme une plateforme de choix pour la mise en œuvre de pipelines de traitement de paquets programmables en P4, combinant reconfigurabilité et parallélisme massif.

Cependant, les solutions existantes de compilation de P4 vers FPGA reposent largement sur une spécialisation effectuée au moment de la synthèse, où les architectures d'analyseur et de désassembleur de paquets sont régénérées pour chaque programme P4. Ce modèle statique empêche l'évolution de la logique protocolaire à l'exécution, accroît la latence de déploiement et limite la réutilisation des FPGA dans des environnements dynamiques. Les tentatives visant à améliorer la flexibilité s'appuient souvent sur la reconfiguration de machines à états ou sur une logique de commutation dynamique, mais ces approches entraînent une surcharge de contrôle qui réduit le débit. Par ailleurs, les conceptions basées sur des chemins de données à matrices de commutation larges permettent le réordonnancement des en-têtes, mais entraînent un coût matériel quadratique ainsi qu'une faible évolutivité temporelle sur les FPGA.

Cette thèse vise à permettre l'analyse et la reconstruction de paquets programmables à l'exécution sur FPGA, tout en préservant des performances élevées et une utilisation efficace des ressources matérielles. Pour atteindre cet objectif, cette thèse propose un cadre matériel-logiciel cohérent permettant de dissocier la sémantique du protocole de l'architecture du chemin de données. Ce cadre introduit deux modes de reconfigurabilité : des architectures à modèles configurables au moment de la synthèse via des génériques VHDL, et des architectures de superposition permettant une reconfiguration en cours d'exécution grâce à des tables de configuration compactes générées automatiquement à partir de descriptions P4.

Cette thèse propose des architectures reconfigurables pour les phases d'analyse et de reconstruction de paquets dans les plans de données basés sur P4. Une conception d'analyse à deux niveaux est introduite, dans laquelle une architecture à modèles expose les paramètres protocolaires sous forme de génériques à la synthèse pour permettre la réutilisation de la conception, tandis qu'une extension en superposition prend en charge les mises à jour à l'exécution en externalisant les transitions de protocole et les paramètres d'extraction dans des tables de configuration. Pour la reconstruction des paquets, une architecture de déparsing sans matrice de commutation est développée, basée sur un modèle de composition par découpage et décalage, qui élimine les réseaux de permutation larges tout en prenant en charge la variabilité des protocoles et l'alignement des en-têtes en mode flux. Les superpositions d'analyse et de déparsing partagent un modèle de contrôle unifié piloté par des tables de configuration automatiquement générées, permettant l'évolution du protocole sans modifier ni resynthétiser le RTL.

Ensemble, ces contributions établissent une approche évolutive du traitement P4 sur FPGA, conciliant programmabilité et performance. Contrairement aux architectures spécialisées classiques générées à la compilation, les conceptions à modèles et en superposition proposées permettent un déploiement progressif, une évolution des protocoles sur le terrain et une réutilisation à long terme des FPGA, étendant ainsi l'applicabilité pratique du matériel reconfigurable dans les réseaux programmables.

ABSTRACT

Software-Defined Networking (SDN) and the P4 language introduced programmability into data-plane architectures by decoupling forwarding behavior from hardware implementation. Field-Programmable Gate Arrays (FPGAs) have emerged as a compelling platform for implementing P4-programmable packet processing pipelines, combining reconfigurability with high parallelism.

However, existing P4-to-FPGA solutions rely heavily on compile-time specialization, where parser and deparser architectures are regenerated for each P4 program. This static design model prevents runtime evolution of protocol logic, increases deployment latency, and limits FPGA reuse in dynamic environments. Attempts to improve flexibility often rely on state-machine reconfiguration or dynamic switching logic, but these approaches introduce control overhead that degrades throughput. Meanwhile, designs based on wide crossbar datapaths support flexible header reordering but incur quadratic hardware cost and exhibit poor timing scalability on FPGAs as datapath widths and protocol complexity grow.

This thesis aims to enable runtime-programmable packet parsing and deparsing on FPGAs while preserving high performance and hardware efficiency. To achieve this goal, it introduces a unified hardware–software framework that decouples protocol behavior from datapath structure. The framework provides two levels of reconfigurability: templated architectures, which allow compile-time configuration through VHDL generics, and overlay architectures, which extend this model by enabling runtime reconfiguration through compact memory tables automatically derived from P4 descriptions.

This thesis proposes reconfigurable architectures for both the parser and deparser stages of P4-based data planes. A two-level parser design is introduced in which a templated architecture exposes protocol parameters as synthesis-time generics for design reuse. At the same time, an overlay extension supports runtime updates by externalizing protocol transitions and extraction parameters in configuration tables. For packet reconstruction, a crossbar-free deparser architecture based on a slice-and-shift composition model is developed that eliminates wide permutation networks while supporting protocol variability and header alignment in streaming mode. Both parser and deparser overlays share a unified control model driven by automatically generated configuration tables, enabling protocol evolution without modifying or resynthesizing RTL.

These contributions establish a scalable approach to FPGA-based P4 processing that reconciles programmability with performance. Unlike conventional compile-time-specialized architectures, the proposed templated and overlay designs enable incremental deployment, in-field protocol evolution, and long-term FPGA reuse, extending the practical applicability of reconfigurable hardware in programmable networks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

AGF	Access Gateway Function
API	Application Programming Interface
ARPA	Advanced Research Projects Agency
ARPANET	Advanced Research Projects Agency Network
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BMv2	Behavioral Model version 2
BRAM	Block Random-Access Memory
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DAG	Directed Acyclic Graph
DDR	Double Data Rate memory
DMA	Direct Memory Access
DPDK	Data Plane Development Kit
DSL	Domain-Specific Language
eBPF	Extended Berkeley Packet Filter
FF	Flip-Flop
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
Gb/s	Gigabits per second
HA	Header Analysis (unit/block)
HDL	Hardware Description Language
HLS	High-Level Synthesis
ICMP	Internet Control Message Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IP	Intellectual Property (core)
JSON	JavaScript Object Notation
LUT	Look-Up Table
MAC	Media Access Control
MD	Memory Data (parser configuration word)
MHz	Megahertz
MPLS	Multiprotocol Label Switching

MTU	Maximum Transmission Unit
NIC	Network Interface Card
NPU	Network Processing Unit
ONF	Open Networking Foundation
P4	Programming Protocol-Independent Packet Processors
PCIe	Peripheral Component Interconnect Express
PFQ	Packet-Filters Queue (PFQ framework)
PHV	Packet Header Vector
PID	Protocol Identifier
PISA	Protocol-Independent Switch Architecture
PIV	Protocol Identification Vector
POF	Protocol-Oblivious Forwarding
QoS	Quality of Service
RAM	Random-Access Memory
RMT	Reconfigurable Match Tables
RTL	Register-Transfer Level
SDN	Software-Defined Networking
SFP+	Small Form-factor Pluggable Plus
SPI	Serial Peripheral Interface
SRAM	Static Random-Access Memory
Tb/s	Terabits per second
TCP	Transmission Control Protocol
TCAM	Ternary Content Addressable Memory
UDP	User Datagram Protocol
UVM	Universal Verification Methodology
VHDL	VHSIC Hardware Description Language
VLAN	Virtual Local Area Network
XDP	Express Data Path

CHAPTER 1 INTRODUCTION

1.1 Context and Background

The development of the Internet was not a singular event but rather the outcome of numerous technological advancements accumulated over decades. Its foundation can be traced to two pivotal efforts that emerged in the 1950s and 1960s. The first was the conceptual development of packet-switched communication, a method that allowed data to be broken into packets and transmitted over shared infrastructure without requiring dedicated circuits. The second was the establishment of the Advanced Research Projects Agency (ARPA). This U.S. government organization played a vital role in funding and coordinating high-risk, high-reward technological projects. These two trajectories converged in the mid-1960s, resulting in the creation of ARPANET, a pioneering network widely regarded as the origin of the modern Internet [1,2]. Initially deployed among four research institutions —UCLA, Stanford Research Institute, UC Santa Barbara, and the University of Utah —ARPANET validated the feasibility of distributed, packet-based communication. Since that time, the Internet has undergone exponential growth, ultimately evolving into a global-scale infrastructure that connected over 5 billion users and more than 25 billion devices by 2022 [3,4].

As the Internet expanded, the design of networking systems evolved to address new demands related to scale, availability, and performance. Early network hardware was designed around simple best-effort forwarding. These devices were stateless, and the network operated under the assumption that complexity would be handled at the endpoints. This design philosophy aligned with the end-to-end argument, which suggested that most features such as reliability and congestion control were best implemented at the application or transport layer [5]. However, the proliferation of real-time applications, such as video streaming, voice-over-IP, cloud services, and mobile banking, introduced new requirements. These services depend not only on connectivity but also on low latency, high throughput, and robust in-network security features [6].

1.2 Motivation and Problem Context

To support these requirements, networking equipment had to evolve. Switches and routers began to incorporate more intelligence into the data plane, allowing them to monitor traffic, enforce access policies, manage queues, and perform load balancing. However, these new capabilities were often implemented in custom silicon and hardcoded logic. The resulting systems were fast but inflexible. Many commercial switches relied on fixed-function application-specific integrated circuits (ASICs) that offered high performance but lacked programmability. Even modest changes, such as supporting a new protocol or adjusting match-action behavior, required costly and time-consuming hardware upgrades. This phenomenon, often referred to as "network ossification," created a significant barrier to innovation in network architecture and protocol design [7, 8]. These limitations in fixed-function infrastructures motivated a search for flexible architectures, ultimately leading to SDN.

1.3 Evolution Toward Programmable Networks

1.3.1 Software-Defined Networking

To address these limitations, the networking community began to explore architectures that could decouple network control logic from data forwarding. This effort led to the development of SDN (Software-Defined Networking), a paradigm that introduces a clean separation between the control plane, responsible for global decision-making and network policy, and the data plane, which performs packet forwarding based on installed rules. Centralized controllers implement the network intelligence in software, dynamically reprogramming the data plane according to the current network state. This architectural shift enabled operators to manage infrastructure more flexibly, improving visibility, automation, and cost efficiency [9, 10].

1.3.2 From OpenFlow to Fully Programmable Data Planes

The OpenFlow protocol emerged as the first practical realization of the SDN model [11]. It provided a standardized interface through which controllers could install flow rules directly into switch hardware. OpenFlow gained traction quickly, becoming a key enabler for research in campus networks, testbeds, and early datacenter deployments. However, its reliance on fixed protocol fields and versioned extensions introduced rigidity. Each enhancement to support a new feature or protocol required updates to both the OpenFlow specification and the underlying switch firmware. As a result, the standard struggled to accommodate emerging protocols and novel use cases, making it less suitable for long-term, production-grade deployments.

1.4 P4 and the PISA Architecture

1.4.1 Protocol-Independent Switch Architecture (PISA)

Recognizing these limitations, researchers introduced a new architecture known as Reconfigurable Match Tables (RMT) [12]. RMT enabled high-performance switches to support user-defined packet processing pipelines. In contrast to fixed-function designs, RMT architectures allowed developers to configure how packets are parsed, matched against tables, and transformed via associated actions. While not as general-purpose as Network Processing Units (NPUs), which resemble programmable central processing units (CPUs) specifically designed for networking, RMT architectures offer a more favorable tradeoff between speed and programmability. Their structure enabled them to achieve throughput comparable to ASICs, while still allowing for limited forms of stateful processing using local memory within pipeline stages.

Building upon the RMT architecture, the Protocol Independent Switch Architecture (PISA) was proposed to provide a standardized model for programmable switches. PISA formalizes the data plane as a pipeline that includes three main components: a configurable parser that identifies protocol headers, a set of match-action tables that apply transformation and forwarding logic, and a deparser that reconstructs the outgoing packet. This abstraction is designed to be protocol-independent and target-agnostic, enabling switch behavior to be redefined through software rather than fixed hardware logic. The flexibility of this model supports a wide variety of advanced use cases, including service chaining, custom tunneling formats, fine-grained telemetry, and adaptive traffic engineering.

1.4.2 P4: A Language for Programmable Packet Processing

To enable programming of PISA-based hardware, the P4 language was introduced as a domain-specific language for packet-processing pipelines [13]. P4 allows developers to define header formats, parse graphs, match-action tables, metadata, and control flow, all in a way that abstracts away hardware-specific details. It also enables control-plane integration by allowing dynamic insertion or modification of forwarding rules at runtime. Unlike OpenFlow, which is restricted to a predefined set of fields and actions, P4 is fully extensible. This means that developers can introduce entirely new protocols or modify existing ones without needing to change the underlying hardware design. This extensibility has made P4 the de facto language for programming modern data plane devices, from ASICs to Field-Programmable Gate Arrays (FPGAs) and even software switches.

1.5 Relevance of FPGA Platforms for P4

Since its introduction, the PISA/P4 ecosystem has expanded rapidly. Major hardware vendors such as Barefoot Networks (acquired by Intel) [14] have released programmable ASICs like Tofino [15], which implement the PISA model in silicon. In parallel, FPGA vendors and researchers have proposed PISA-compatible architectures on reconfigurable platforms to offer greater customization and support for research and prototyping. These advances have also led to the development of compilers, intermediate representations, and toolchains that translate P4 programs into configurations deployable across diverse hardware targets.

Together, PISA and P4 represent a transformative shift in how networks are designed, implemented, and operated. They bring the benefits of software-defined control to the data plane, enabling unprecedented flexibility without entirely sacrificing performance. In contrast to earlier approaches like OpenFlow, which offered only incremental improvement, the PISA model introduces a foundational redesign that aligns hardware, software, and protocol design in a unified, programmable framework. This thesis builds upon this foundation, exploring how PISA and P4 can be efficiently mapped onto FPGA platforms to enable reconfigurable, high-speed packet processing.

1.6 Research Problem and Gaps

The functionality of traditional networks has long been dictated by fixed-function hardware designed around standardized protocols, leaving little room for user-driven innovation. As network infrastructures evolved to support cloud computing, virtualization, and a rapidly expanding set of applications, this rigidity became a significant bottleneck. The emergence of SDN addressed this limitation by decoupling the control plane from the data plane, thereby enabling centralized management and programmable control of network behavior [11, 12]. This architectural shift empowered operators to modify routing and policy logic dynamically through software. However, despite its architectural elegance, achieving both flexibility and high performance within the SDN data plane remains an ongoing challenge, as demonstrated by several FPGA-based implementations that trade programmability for throughput [16,17].

1.6.1 Need for Flexible High-Performance Hardware

ASICs continue to dominate the high-performance networking domain due to their optimized throughput and power efficiency. Nevertheless, their design and fabrication processes are inherently rigid and time-consuming, often requiring months or even years to accommodate new protocols or features [18]. This slow development cycle contrasts sharply with the rapid evolution of modern network services, where frequent updates and protocol innovations are essential. At the opposite end of the spectrum, general-purpose processors such as CPUs and NPUs provide high programmability but cannot deliver the multi-gigabit data rates demanded by contemporary data centers and carrier networks [19]. The widening gap between the flexibility of software-based solutions and the performance of ASIC hardware highlights the need for a platform that can reconcile both worlds.

FPGAs have therefore emerged as an ideal middle ground, combining reconfigurability with hardware-level parallelism. They enable developers to construct packet-processing pipelines that can adapt to new network protocols while maintaining wire-speed performance. The introduction of the Programming Protocol-Independent Packet Processors (P4) language further strengthened this paradigm by allowing developers to define parsing, matching, and forwarding logic in a protocol-agnostic manner. As a result, the integration of P4 programmability with FPGA reconfigurability has become a promising pathway toward realizing fully programmable, high-speed data planes [16–18, 20].

1.6.2 Research Gap

Despite this progress, current P4-to-FPGA tool flows exhibit several limitations. Each modification to a P4 program requires lengthy synthesis, place-and-route, and bitstream generation steps. This process hinders rapid experimentation and prevents the runtime reconfiguration expected in SDN environments. Moreover, although high-level synthesis tools have reduced the barrier to FPGA development, designing efficient packet-processing hardware still requires deep expertise in hardware description languages, digital design, and FPGA architecture.

A practical solution to these challenges involves introducing a layer of abstraction between the static hardware fabric and the P4 program, namely, an overlay architecture. In such a model, the hardware implements a generic pipeline whose behavior is defined by configuration data stored in embedded memories. Updating these memories alters the packet-processing behavior without requiring the FPGA to be resynthesized. Two key principles emerge from this approach: (1) programmability through memory updates, and (2) reuse of a single, parameterized hardware instance instead of regenerating multiple specialized datapaths.

1.7 Proposed Research Direction

The first principle, programmability through memory, enables an end-to-end framework that allows users with minimal or no hardware knowledge to deploy new P4 programs on real hardware. The memory contents act as configuration tables generated automatically from the P4 code and can be reloaded within a few clock cycles. Consequently, external buffering requirements are minimized, since reprogramming the embedded memory incurs negligible downtime compared to reprogramming the entire FPGA. The second principle, hardware reuse, supports scalability and isolation: multiple instances of the overlay can operate concurrently, each assigned to a specific interface or network tenant. Under the concepts of network slicing and multi-tenant virtualization, this enables independent customization of network shares while preserving global performance guarantees [21, 22].

The limitations of current P4-to-FPGA compilers and the absence of efficient runtime-reconfigurable architectures underscore the need for a unified programmable hardware platform. Such a framework must provide throughput comparable to ASIC designs, maintain FPGA-level flexibility, and support memory-based reconfiguration to dynamically accommodate new protocols and applications. Addressing this problem constitutes the foundation of the present research. Therefore, the key research problem is to design an FPGA architecture that sustains ASIC-level throughput while enabling runtime P4 reconfiguration without resynthesis.

1.8 Research Objectives

The overarching goal of this thesis is to develop FPGA-based P4 architectures that harmonize throughput, programmability, and runtime reconfigurability. While ASICs deliver unmatched throughput, they are inherently rigid and slow to evolve. Conversely, software-based solutions offer flexibility but struggle to maintain line-rate performance. FPGAs occupy a promising middle ground, offering hardware-level parallelism and reconfigurability, yet fully exploiting their potential requires specialized architectural and methodological innovations. This research bridges that gap by proposing FPGA design frameworks that integrate P4 programmability with runtime reconfigurability, addressing both the parser and deparser stages of the PISA pipeline.

The objectives of this research are summarized as follows:

1. Investigate the design space of FPGA-based P4 data planes: Analyze existing P4-to-FPGA compilation frameworks and identify architectural bottlenecks that limit runtime flexibility, scalability, and design automation. Particular attention is paid to how the parser and deparser stages can be abstracted to support protocol-agnostic packet processing.

2. Formulate a methodology for memory-driven reconfigurable architectures: Develop an architectural abstraction layer that decouples static FPGA hardware from the P4 control specification. The objective is to enable rapid functional updates through configuration memories, eliminating the need for full resynthesis while maintaining deterministic performance.

3. Establish a consistent reconfiguration model for parser and deparser architectures: Develop parallel design methodologies for both the parser and deparser, applying common memory-driven control principles without requiring hardware integration. This enables each stage to support dynamic protocol graphs and runtime updates while maintaining predictable latency.

4. Validate scalability and performance across multiple configurations: Evaluate the proposed frameworks under varying bus widths, protocol complexities, and reconfiguration scenarios. The goal is to demonstrate that high-throughput, protocol-independent, and runtime-programmable packet processing can be achieved within realistic FPGA resource and timing constraints.

By fulfilling these objectives, this thesis seeks to advance the state of FPGA-based packet processing through a cohesive architectural and methodological foundation. The resulting frameworks enable high-speed, memory-programmable, and protocol-agnostic network pipelines that evolve in step with emerging SDN and P4 data-plane applications.

1.9 Research Contributions

This thesis investigates programmable packet processing architectures on FPGAs, addressing the need for high-throughput, flexible, and reconfigurable data-plane designs within P4-programmable SDN infrastructures. It examines fundamental challenges in hardware-based packet parsing, including the rigidity of traditional RTL implementations, the synthesis time overhead of P4-to-hardware compilation, and scalability limits when targeting complex protocol graphs. The overall objective is to propose architectures that remain efficient across a wide range of protocol complexities and throughput requirements while enabling reprogrammability without full hardware re-synthesis.

To this end, the work introduces a framework named PrismParser, which integrates three architectural models: base, overlay, and pipeline, each designed to explore specific trade-offs between performance, area efficiency, and flexibility. The framework automates hardware generation from P4 descriptions through software abstraction that extract parser graphs, generate control tables, and produce memory initialization data for FPGA configuration.

This research has resulted in four publications:

1. Parisa Mashreghi-Moghadam, Tarek Ould-Bachir, and Yvon Savaria.

   "A Templated VHDL Architecture for Terabit/s P4-programmable FPGA-based Packet Parsing." Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), Austin, USA, 2022.
   This contribution presents a templated pipeline parser architecture implemented in VHDL. A Python-based compiler extracts the directed acyclic graph from P4 descriptions to generate configuration vectors for each stage. The design achieves 1 Tb/s throughput on a Xilinx UltraScale+ FPGA with low latency and moderate area utilization. Each header analysis unit performs protocol transitions through bitmasking, key extraction, and data alignment mechanisms.

2. Parisa Mashreghi-Moghadam, Tarek Ould-Bachir, and Yvon Savaria.

   "An Area-efficient Memory-based Architecture for P4-programmable Streaming Parsers in FPGAs." Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), Monterey, USA, 2023.
   This work introduces a memory-based overlay parser architecture that supports protocol updates through memory reconfiguration rather than RTL modification. The single-stage header analysis unit operates on control data stored in embedded memories and can be dynamically updated via SPI. The design achieves 11.1 Gb/s on a small-footprint Xilinx Virtex-7 device, demonstrating its suitability for resource-constrained systems.

3. Parisa Mashreghi-Moghadam, Tarek Ould-Bachir, and Yvon Savaria.

   "PrismParser: A Framework for Implementing Efficient P4-Programmable Packet Parsers on FPGA." Future Internet, volume 16, number 9, article 307, 2024.
   This publication consolidates the architectural models into a unified framework that integrates software–hardware co-design. The work evaluates scalability across multiple protocol graphs and bus widths, achieving up to 282 Gb/s on a Xilinx Alveo U280 board while supporting parser graphs with up to seven transition nodes. It also demonstrates the framework's multi-tenant compatibility for applications such as firewalls and 5G gateways.

4. Parisa Mashreghi-Moghadam, Tarek Ould-Bachir, and Yvon Savaria.

   "Templated and Overlay HW/SW Co-Optimization for Crossbar-Free P4 Deparser FPGA Architectures" Electronics, 2025 (submitted).
   This work extends the PrismParser methodology to the deparser stage of the PISA pipeline. It introduces a recursive-select overlay architecture that replaces crossbars with slice-and-shift primitives, maintaining runtime reconfigurability while achieving over 200 Gb/s throughput. The design leverages template bounds and memory-based configuration control to achieve predictable scaling across different bus widths and protocol depths.

These contributions establish a comprehensive framework for high-throughput and protocol-agnostic packet processing on FPGAs. By combining templated hardware generation, memory-based overlays, and runtime configuration mechanisms, the proposed designs enable scalable, reprogrammable, and hardware-efficient implementations that evolve with the dynamic requirements of SDN environments.

1.10 Thesis Organization

This thesis is organized into nine chapters that address the motivations, methodologies, and contributions related to runtime-reconfigurable P4 data-plane architectures on FPGAs. The thesis progresses from foundational motivation and literature review to architectural design, implementation, evaluation, and discussion, providing a cohesive narrative from problem formulation to validated solution, as detailed hereafter.

- Chapter 2 presents a comprehensive review of the literature, covering programmable data planes, FPGA-based packet processing, P4 compilation frameworks, and overlay-based reconfiguration strategies. It identifies key architectural limitations in current P4-to-FPGA toolflows and positions the contributions of this thesis within the broader research landscape.

- Chapter 3 consolidates the evaluation methodology and presents a comparative analysis of all architectures introduced in this thesis. This includes performance benchmarks, resource utilization, latency measurements, and reconfiguration cost under varied FPGA implementations.

- Chapter 4 introduces the first contribution: a templated FPGA parser architecture based on static specialization. It describes the Python-assisted VHDL generation flow and demonstrates how compile-time parametrization enables terabit-class throughput while preserving P4 compatibility.

- Chapter 5 describes the second contribution: a runtime-reconfigurable overlay parser architecture. Unlike the templated approach, this design externalizes protocol behavior into embedded memories, allowing P4 parser graphs to be updated without resynthesizing the hardware.
- Chapter 6 presents *PrismParser*, a unified framework that integrates the templated, overlay, and pipeline variants into a coherent architectural and software design flow. The chapter evaluates scalability across variable bus widths, protocol graphs, and runtime scenarios.
- Chapter 7 introduces the fourth major contribution: a crossbar-free, runtime-programmable deparser architecture based on slice-and-shift composition. The design eliminates the need for wide switching fabrics while supporting memory-driven emission control in both static and overlay forms.
- Chapter 8 provides a general discussion that synthesizes the main contributions, evaluates them in relation to the original research objectives, and highlights theoretical and practical implications. It also reflects on trade-offs between throughput, flexibility, and hardware cost.
- Chapter 9 concludes the thesis by summarizing key findings and their impact. It outlines current limitations and discusses several promising directions for future research, including full overlay-based PISA pipelines and match–action reconfiguration.

CHAPTER 2 LITERATURE REVIEW

2.1 From Fixed Forwarding to SDN and Programmable Data Planes

In the early decades of computer networking, routers and Ethernet switches were designed as vertically integrated systems in which the control plane, responsible for routing, configuration and management, was tightly coupled with the data plane that performed packet forwarding. This monolithic integration restricted external programmability, as any modification to packet handling behavior required new firmware or hardware redesign by equipment vendors. As emphasized in early historical syntheses [7, 23], traditional network devices were closed appliances exposing limited configuration interfaces preventing researchers and operators from implementing new protocols or services within existing infrastructures. This inflexibility motivated subsequent research into Active Networks [24, 25], which sought to open the data path to computation and dynamic reconfiguration.

2.1.1 Active Networks: Computation in the Network

In the mid-1990s, the Active Networks vision proposed a radical rethinking of programmability. The seminal work by Tennenhouse and Wetherall [24] introduced two fundamental models:

- The capsule model, where packets carry executable code that routers interpret at runtime, and
- The node-program model, where routers host downloadable programs controlling their behavior.

The ambition was to enable rapid in-network innovation by executing application-specific logic directly inside the data plane. However, this paradigm encountered critical barriers:

1. Safety and isolation: untrusted code in the fast path required strict sandboxing and resource bounds.
2. Performance: general-purpose execution compromised deterministic throughput expected from ASIC pipelines.
3. Deployability: introducing new runtimes and trust models at every router made adoption infeasible.

Active networks thus failed to achieve wide deployment, yet their intellectual legacy and the desire for open programmability persisted.

2.1.2 Shifting the Boundary: Control-Plane Programmability

In the early 2000s, the focus of network programmability research shifted from in-band computation within the data path toward out-of-band control mechanisms. Rather than embedding executable logic directly inside routers, as proposed by the Active Networks model [24], researchers began exploring architectures in which programmability resided in an external control layer that dictated the forwarding behavior of simpler data-plane devices. This approach maintained deterministic performance while allowing global network policies to be defined and enforced through software. Notable examples include 4D [26] that formalized the separation of decision, dissemination, and data planes and demonstrated that centralized control could simplify configuration and improve network manageability. These systems marked a fundamental conceptual transition: the network was no longer programmed at the level of packet execution, but at the level of control and policy specification, laying the groundwork for the emergence of SDN.

Next came Ethane [27], which introduced centralized policy-driven control and marked a pivotal step towards modern networks. It implemented enterprise identity-based access control using a central controller that computed per-flow forwarding rules for simple switches. Ethane demonstrated that programmability could be achieved without embedding arbitrary computation in the fast path. Its immediate operational benefits, including simplified auditing, management, and security, provided a clear use case for network operators, distinguishing it from the abstract aspirations of Active Networks. Ethane directly inspired the OpenFlow protocol by formalizing the notion of a controller installing match–action rules in data-plane devices.

The introduction of OpenFlow [11] in 2008 marked the concrete realization of control/data-plane separation. OpenFlow defined a simple, low-level Application Programming Interface (API) that exposed existing match–action tables (flow entries, counters, actions) to external controllers. Unlike Active Networks, OpenFlow's design aligns perfectly with existing merchant silicon, utilizing match tables based on Ternary Content Addressable Memory (TCAM) [28] already present in switches, thereby avoiding hardware redesign. This hardware alignment enabled incremental deployment in research environments and catalyzed a new paradigm for programmable network control. Next came NOX [29], the first OpenFlow controller, which offered network-wide event handling, topology discovery, and flow rule installation. NOX showed that a logically centralized controller could operate in real time with global visibility, effectively serving as an operating system for the network. As deployments grew, Onix [30] extended this model by introducing a distributed control platform capable of sharing network state across multiple controllers. It supported strong consistency for operations that require an immediately synchronized global view, and eventual consistency for tasks that can tolerate temporary divergence to improve scalability and performance. This evolution from basic flow-table programming to scalable control platforms marked the maturation of the OpenFlow vision and paved the way for a more general architectural model.

The success of OpenFlow catalyzed the articulation of a broader architectural paradigm, later termed SDN [9]. While OpenFlow defined a specific protocol for controller–switch interaction, SDN generalized the concept by introducing a logically centralized control plane that manages a distributed collection of programmable forwarding devices. This abstraction enabled direct programmability of network behavior through software-defined policies rather than hardware-specific configurations. By decoupling the network's control logic from its forwarding implementation, SDN introduced unprecedented agility in management and innovation, allowing network operators to deploy new services, reroute traffic, or enforce security rules dynamically.

The formalization of SDN as a three-plane architecture, management, control, and data, occurred in the early 2010s, with the establishment of the Open Networking Foundation (ONF) in 2011 to promote standardization and interoperability among vendors [31]. These efforts defined the first generation of programmable networks that were software-defined but still relied on fixed-function forwarding hardware. Consequently, although OpenFlow and SDN enabled centralized reconfiguration, the underlying packet-processing pipelines remained statically designed around predefined protocol fields. To achieve true flexibility, the programmability offered by SDN needed to extend beyond control to encompass the data plane itself.

Despite its success in decoupling control logic from forwarding hardware, OpenFlow exposed only a limited notion of programmability. It allowed a controller to update forwarding rules at runtime but operated on a fixed parsing pipeline with protocol-dependent header formats. The set of fields available for matching and modification was predefined in the switch hardware, which constrained innovation and prevented deployment of new encapsulation mechanisms, evolving protocols, or application-specific packet formats. As a result, OpenFlow-based switches could only express packet-processing behavior within a narrow template, inhibiting the development of more advanced in-network functions such as telemetry, custom tunneling, and application-aware processing.

To overcome these restrictions, the switch microarchitecture had to evolve from fixed protocol semantics to protocol-independent forwarding. This shift resulted in the RMT architecture, later generalized as the PISA [12]. RMT/PISA extended the match–action abstraction introduced by OpenFlow and formalized a programmable packet-processing pipeline composed of three stages: a programmable parser for header extraction, a match–action pipeline for table-driven decision logic, and a deparser for packet reconstruction. Unlike OpenFlow's rigid pipeline, RMT/PISA decoupled forwarding behavior from hardware by representing protocol processing through configurable tables that define parsing transitions, lookup operations, and header modifications. This design retained the deterministic throughput of deeply pipelined switch hardware while enabling protocol-independent processing. As a result, RMT/PISA became the reference architectural model for high-speed programmable data planes, influencing both academic research and commercial switch ASICs such as Barefoot Tofino [14].

2.1.3 Toward Programmable Packet Pipelines

While PISA established a generic and protocol-independent forwarding architecture, it represented a hardware-centric abstraction. Configuring a PISA pipeline still required direct manipulation of match tables, parser specifications, and control logic through low-level APIs or vendor-specific configuration interfaces. This limited portability and increased the level of complexity for developers, who needed to understand the underlying hardware pipeline in order to program it effectively. Consequently, the research community began exploring ways to raise the abstraction level of the data plane, shifting from configuring forwarding entries to describing packet-processing intent. This transition laid the foundation for high-level data-plane programming models, ultimately leading to the introduction of programmable packet-processing languages. The progressive abstraction of the data plane did not emerge abruptly; instead, it resulted from successive attempts to make packet-processing behavior both programmable and portable across various hardware targets. As network processors and reconfigurable hardware matured, researchers sought high-level domain-specific languages (DSLs) that could describe packet transformations without requiring low-level manipulation of hardware registers or firmware. These efforts gradually shaped the concept of a programmable packet-processing pipeline.

Early initiatives, such as Click [32], developed in C++, proposed a modular framework where packet-processing elements, including classifiers, counters, or checksum verifiers, could be assembled graphically to form customized routers. Its extension NP-Click [33] adapted this model to network processors, exposing concurrency and pipeline mechanisms suitable for packet-level parallelism. Although these systems introduced composability at the software level, they remained limited to CPU-based execution and required manual adaptation for new protocol headers or specialized hardware.

Subsequent efforts focused on functional and low-level packet-processing languages. PFQ-Lang [34] proposed a functional abstraction to describe packet flows and filters while maintaining runtime efficiency. Despite this progress, it remained protocol-dependent, as every new protocol demanded language modification. Similarly, packetC [35] extended the C99 standard to support packet-oriented constructs and multi-threaded execution on massively multicore network processors. While it provided fine-grained control over packet-processing, it required intimate knowledge of the target microarchitecture and offered limited portability.

Around the same time, two languages attempted to unify packet-processing at the instruction level. Protocol-Oblivious Forwarding [36] redefined OpenFlow, introducing protocol independence by operating directly on byte offsets and field descriptors rather than fixed protocol definitions. This approach detached forwarding logic from protocol semantics but still relied on specialized hardware instructions. PX [37], a proprietary language from Xilinx, targeted FPGA-based switches and introduced an explicitly defined parsing and processing structure, anticipating the pipeline model later generalized by PISA.

The convergence of these early programming efforts marked a turning point as packet-processing could now be described using structured abstractions rather than hardware-specific interfaces. However, these languages lacked a unified model for expressing packet parsing, header manipulation, and match–action control in a way that was both protocol-independent and portable across hardware targets. As architectures such as PISA matured, it became clear that a dedicated, domain-specific language was required to express protocol-independent packet-processing behavior in a target-agnostic manner. This need motivated the development of a new data-plane programming language.

2.1.4 Introduction of the P4 Language

The next milestone in the evolution of programmable networking was the introduction of the P4 language by Bosshart et al. in 2014, followed by a major revision in 2016 [13, 38]. P4 was designed to enable developers to describe the entire packet-processing pipeline, including parsing, matching, and header modification, at a high level of abstraction.

Three design principles governed P4's conception:

1. Protocol independence: forwarding logic should not depend on fixed protocol formats and should remain correct as new headers are introduced;

2. Reconfigurability: data-plane behavior should be modifiable without hardware redesign, by recompiling or reloading the P4 program; and

3. Target independence: the same P4 program should be compilable for a broad range of hardware and software targets, including ASICs, NPUs, and FPGAs.

This language foundation paved the way for the development of data-plane architectures capable of executing protocol-independent pipelines in hardware. Building on the abstractions introduced by P4, architectures such as PISA and FPGA-based programmable pipelines emerged to realize flexible parsing, match–action processing, and header modification directly in high-performance switching devices. These architectures translate high-level packet-processing intent into efficient hardware execution, enabling true in-network programmability at line rate.

Overall, the evolution from fixed-function forwarding to fully programmable packet processing has been shaped by a series of architectural and conceptual milestones. Active Networks first articulated the vision of in-network programmability, but performance and safety constraints limited adoption. The shift toward control-plane programmability with Ethane and OpenFlow demonstrated the viability of centralized control and laid the foundation for Software-Defined Networking, which decoupled control logic from forwarding hardware. Nevertheless, SDN devices remained constrained by fixed protocol pipelines, motivating the development of protocol-independent switch architectures such as RMT and PISA. These architectures enabled programmable parsing, match–action processing, and deparsing, and ultimately led to the emergence of the P4 language, providing a portable, high-level abstraction for specifying packet-processing behavior. Together, these developments established the theoretical and architectural basis for modern programmable data planes.

Having laid the conceptual foundations for protocol-independent packet processing, the next section examines the hardware platforms that operationalize these ideas in practice. We survey contemporary programmable network devices.

2.2 From Programmable Abstractions to Programmable Hardware Platforms

The emergence of high-level data-plane languages such as P4 marked a shift from fixed-function forwarding pipelines to programmable packet processing. Realizing these abstractions at line rate, however, required corresponding advances in underlying hardware. Modern platforms expose different degrees of programmability while operating under strict throughput, latency, and memory-bandwidth constraints. General-purpose CPUs offer maximal flexibility due to their rich software ecosystems, but are fundamentally limited in sustaining high packet-rate workloads because of sequential execution and shared-memory contention. Network Processing Units (NPUs) extend programmability with specialized packet-processing cores and hardware accelerators, providing higher concurrency than CPUs but still relying on instruction-driven execution that constrains worst-case performance. At the opposite end of the spectrum, PISA-based switch ASICs employ deeply pipelined match–action architectures and dedicated memory hierarchies to deliver deterministic, multi-terabit packet forwarding, albeit with rigid resource and pipeline structures defined at fabrication time. FPGAs occupy an intermediate point in this space, enabling construction of fully customized packet-processing pipelines that remain reconfigurable after deployment, thereby offering greater architectural flexibility than ASICs and lower latency than general-purpose processors, at the cost of design complexity and lower clock frequencies. Together, these platforms illustrate the architectural trade-offs between flexibility, determinism, and line-rate performance that underlie contemporary programmable data planes.

2.2.1 P4 Compilation and Execution on General-Purpose Processors

General-purpose processors represent the most flexible platform for running P4 programs due to their rich execution models, mature software ecosystems, and support for rapid development and debugging. However, their sequential microarchitecture and shared-memory hierarchy limit their ability to sustain line-rate packet processing at high speeds. Constraints such as cache contention, branch-misprediction penalties, and finite memory bandwidth make CPUs suitable primarily for functional validation, experimentation, and software-based packet handling rather than deterministic high-throughput data-plane deployment. Two main strategies exist for executing P4 programs on general-purpose processors are compilation to an intermediate representation interpreted or executed by a runtime engine and direct generation of native executable code.

The first approach relies on software models that interpret a machine-readable representation of the P4 pipeline. The Behavioral Model version 2 (BMv2) software switch [39], maintained by the P4 community, is the canonical reference for functional validation. It consumes a JSON description generated by the P4 compiler (p4c) [40] and emulates the P4 processing pipeline in software. Additional toolchains such as p4c-ebpf [41], p4c-ubpf [42], and p4c-xdp [43] compile P4 programs to eBPF bytecode [44], enabling direct execution inside the Linux kernel and integration with modern programmable networking frameworks.

The second approach generates native machine code. PISCES [45] modifies Open vSwitch to synthesize P4 actions from a C-compiled program dynamically. Zanna et al. extend this model by compiling P4 programs for execution on the ZodiacFX evaluation and test board [46]. Finally, T4P4S [47] translates P4 programs into portable C code, enabling efficient execution on general-purpose hardware without requiring modifications to the software infrastructure.

These software-based approaches enable rapid prototyping of P4 pipelines and facilitate experimentation with new protocol processing behaviors. However, even with aggressive optimizations such as DPDK [48], XDP [49], and multi-threading, CPU-based solutions remain fundamentally constrained in achieving deterministic multi-100 Gb/s throughput at minimum packet size due to branch misprediction penalties, cache contention, and finite memory-bandwidth limits. These limitations motivate the adoption of more parallel and pipeline-oriented hardware architectures, including NPUs, programmable switch ASICs, and, increasingly, FPGAs, which offer a stronger balance between flexibility and line-rate performance.

2.2.2 P4 on Network Processing Units

Network Processing Units (NPUs) were introduced as a specialized class of packet-processing processors designed to combine software programmability with higher throughput than general-purpose CPUs. Unlike CPUs, NPUs consist of multiple lightweight processing cores optimized for fine-grained packet-level concurrency, often supported by hardware accelerators for tasks such as checksum computation, memory scheduling, and table lookups. On P4-enabled NPU platforms, programs are typically compiled into a microinstruction format or a vendor-specific intermediate representation, which is executed by the processing cores. This model offers greater flexibility than fixed-function switching silicon, while leveraging networking-aware hardware capabilities to achieve higher performance than software switches. NPUs have therefore been adopted in performance-critical network infrastructure and edge devices that require programmability, together with consistent packet throughput [50, 51].

However, NPUs remain instruction-driven architectures, meaning that performance is ultimately constrained by core count, shared memory bandwidth, and scheduling overhead. As a result, sustaining deterministic operation at very high speeds is challenging, particularly under minimum-size packet workloads or complex protocol pipelines. These architectural characteristics position NPUs between CPUs and fully pipelined hardware targets: more programmable than fixed-function devices, yet less capable of delivering predictable line-rate performance at multi-hundred-gigabit scales. This motivates the exploration of pipeline-centric architectures such as programmable switch ASICs and reconfigurable hardware platforms, including FPGAs.

2.2.3 P4 on Programmable Switch ASICs

Programmable switch ASICs represent the highest-performance execution platform for P4. Designed for data-center fabrics and backbone networks, they operate at multi-terabit-per-second rates with deterministic per-packet latency on the order of a few hundred nanoseconds. Architectures derived from the RMT model, such as Barefoot Tofino, implement deeply pipelined data paths with programmable match tables, fixed-latency action units, and high-bandwidth on-chip memories engineered for sustained line-rate forwarding. P4 programs targeting these ASICs do not execute instruction sequences as in CPUs or NPUs. Instead, the compiler configures parser transitions, match–action tables, and action parameters within a fixed execution template. This model is formalized by the PISA, which exposes a programmable parser, a structured match-action pipeline, and a deparser. PISA is often described as RISC-like because the pipeline is simple, regular, and deeply staged, and programmability is achieved by updating tables and metadata rather than modifying the datapath [52]. The pipeline structure remains static, and only configuration state changes, which ensures predictable cycle time and one-packet-per-cycle throughput under worst-case traffic.

These performance guarantees impose architectural rigidity. Pipeline depth, stage width, memory hierarchy, and available action primitives are fixed at fabrication time. Adding new protocol headers, expanding metadata formats, or modifying parsing and deparsing behavior frequently depends on vendor-specific extensions and may not be possible within the exposed abstraction. Prior work has highlighted this trade-off, showing that ASICs provide orders-of-magnitude higher throughput and energy efficiency than server-based processing, but at the cost of limited architectural flexibility [53]. Complementary experimental studies report similar findings: P4 programs that run unmodified on Tofino must conform to the fixed pipeline semantics, whereas FPGA targets can incorporate custom scheduling logic, modified metadata layouts, and specialized processing blocks to support evolving protocol behaviors [54]. These results reinforce that while ASICs excel in production environments with strict performance and power constraints, they are less suited for rapid protocol evolution or architectural experimentation.

As a result, programmable ASICs remain the optimal choice for large-scale, latency-sensitive deployments, but reconfigurable hardware platforms, particularly FPGAs, play a crucial role when protocol formats, pipeline stages, or metadata structures must evolve after deployment or when customized data-plane behavior is required.

2.2.4 P4 on FPGAs

FPGAs provide a reconfigurable hardware substrate capable of implementing custom packet-processing pipelines with high throughput and cycle-accurate datapath behavior. Unlike programmable switch ASICs, whose microarchitectures are fixed at fabrication time, FPGAs expose logic, memory, and interconnect resources that can be configured at compile time, enabling designers to construct P4-compliant parsers, match–action pipelines, and deparsers tailored to specific protocol sets or performance requirements. While the timing of individual pipeline stages can be precisely controlled, end-to-end packet latency is generally influenced by buffering, arbitration, backpressure, and I/O interactions, and is therefore not strictly constant under all traffic conditions. This flexibility has motivated major industrial and academic efforts to support P4 on FPGA targets.

Early commercial solutions relied on proprietary compilation flows. Xilinx SDNet [55] converts P4 programs into a vendor-specific intermediate representation and automatically generates RTL pipelines optimized for Xilinx devices, offering industry-grade determinism and timing closure for carrier-class networking systems. Netcope Technologies [56] introduced a similar P4-to-VHDL toolchain targeting their FPGA-accelerated NICs, providing a match–action pipeline with fixed, vendor-validated microarchitectural templates. These systems demonstrated the practicality of FPGA-resident P4 dataplanes in production environments but exposed limited internal architectural flexibility, since both relied on closed toolflows and predefined pipeline structures.

Beyond commercial flows, a growing body of academic work explores programmable data-plane and forwarding on FPGA platforms, compiler techniques, template-based hardware generation, and overlay architectures for runtime reconfigurations [17, 18, 57–65]. These efforts highlight the diversity of FPGA-resident P4 execution models, ranging from template-specialized pipelines to runtime-programmable overlays, reinforcing FPGAs as a compelling platform for flexible and high-performance data-plane architectures.

Subsequent research prototypes broadened programmability by exposing the pipeline microarchitecture directly to designers. These efforts demonstrated that high-throughput FPGA pipelines can sustain multi-gigabit to multi-hundred-gigabit data rates by exploiting wide streaming buses, explicit register placement, and statically scheduled pipelines with a one-cycle initiation interval. In contrast to CPUs and NPUs, which rely on instruction-driven execution and shared memory hierarchies, FPGA datapaths avoid general-purpose execution overheads and offer tightly controlled pipeline timing. However, end-to-end latency may still be affected by buffering and flow-control effects under realistic traffic conditions.

Compared to switch ASICs, FPGA pipelines provide post-deployment adaptability at the architectural level, enabling changes to header formats, table structures, PHV layouts, and action logic without replacing the physical device. In practice, such changes may require hardware regeneration, timing closure, validation, and controlled redeployment, and therefore do not always provide the same level of agility as software-based systems. Nevertheless, this ability to evolve dataplane functionality without new silicon distinguishes FPGAs as a complementary class of programmable data-plane hardware, particularly for emerging protocols, evolving encapsulation stacks, and deployment scenarios where architectural flexibility outweighs reconfiguration overhead.

This flexibility does not merely enable programmable packet processing; it aligns closely with the design philosophy of PISA. By structuring forwarding into a programmable parser, a table-driven match–action pipeline, and a deparser, PISA abstracts the packet path into modular and well-defined stages. Each of these stages maps naturally to FPGA fabrics: parsers benefit from bit-precise control, configurable state encoding, and timing-driven pipeline placement; match–action pipelines exploit heterogeneous on-chip memories and synthesizable action logic; and deparsers leverage structured header layouts, byte-aligned slicing, and staged emission pipelines. This natural correspondence allows FPGA-based data planes to deliver deterministic, deeply pipelined performance while retaining the freedom to evolve protocol semantics and resource allocation after deployment. The following sections examine each PISA component in detail and highlight the architectural properties that make FPGAs particularly well suited for protocol-programmable data planes.

2.3 PISA Pipeline in FPGA Hardware

2.3.1 Parser: DAG-structured extraction and pipeline specialization

The parser stage identifies protocol headers and extracts their fields according to the control-flow graph defined in the P4 program. This graph, typically a directed acyclic graph (DAG), encodes protocol dependencies and conditional transitions, and is mapped onto a sequence of parsing states in hardware. Each state inspects incoming packet bytes, extracts relevant fields, updates metadata, and determines the next state based on protocol-level conditions. The depth of the resulting parsing pipeline is governed by the longest valid protocol path in the DAG, while branching transitions enable selective progression through the graph based on packet contents [66].

FPGAs are particularly well suited to this execution model. Bit-accurate field extraction maps naturally to LUT-based logic, pipeline registers may be inserted at well-defined boundaries to satisfy timing constraints, and state encodings can be adapted as protocols evolve. Unlike state-of-the-art PISA-based switch ASICs, which expose programmability within a fixed microarchitectural context—such as a bounded number of parser states, table shapes, action slots, and pipeline stages—FPGA-based parsers permit modifications to protocol structure, metadata layout, and transition logic beyond these fixed structural limits. Such changes may be realized either through compile-time regeneration of a specialized hardware configuration or, in overlay-based architectures, through runtime updates of control structures while preserving a fixed datapath topology.

In hardware terms, the parser behaves as a statically scheduled finite-state machine whose state transitions are driven by protocol conditions extracted from the packet stream. While individual pipeline stages can be cycle-accurate, overall packet latency may be influenced by buffering and flow-control effects. Nevertheless, the combination of architectural flexibility and analyzable pipeline timing makes FPGAs a natural substrate for programmable packet parsing, particularly in contexts where protocol stacks evolve rapidly or require deployment-specific extensions.

2.3.2 Match–Action Pipeline: memory-centric lookup and synthesizable action execution

The match–action pipeline implements table-driven forwarding logic, where extracted header fields and metadata are used to perform lookups and apply associated actions. In the PISA model, each pipeline stage comprises one or more match tables that perform exact, prefix, or ternary lookups, followed by a set of programmable action units that update packet headers, modify metadata, and steer subsequent processing. This design mirrors a deeply pipelined and feed-forward execution structure, where each stage consumes the results of its predecessor and contributes a fixed amount of processing per cycle, maintaining a one-packet-per-cycle initiation rate.

On FPGAs, this abstraction maps naturally to the device's heterogeneous memory and logic fabric. Lookup structures can be implemented using distributed LUTRAM for small latency-critical tables, BRAM banks for large exact-match tables, and hybrid structures combining algorithmic indexing with narrow synthesized TCAM blocks for prefix and ternary matching. Cuckoo hashing provides deterministic lookup behavior with high memory utilization, while multi-banked memory organizations allow concurrent table accesses and predictable throughput [67]. Such flexibility contrasts with fixed-resource ASIC pipelines, where memory dimensions and access patterns are determined at design time.

Action execution in PISA resembles very-long-instruction-word (VLIW) semantics, where each stage exposes a set of primitive functional units capable of arithmetic, logical, and metadata updates [12]. On switch ASICs, the repertoire of supported actions is fixed in silicon; on FPGAs, action units are synthesizable logic, allowing designers to extend operation sets or introduce custom processing blocks while preserving cycle-accurate behavior. This permits integration of functions such as header compression, programmable telemetry metadata insertion, or lightweight stateful processing alongside standard forwarding logic.

Crucially, FPGA match-action pipelines preserve deterministic latency: each stage performs a bounded set of operations and propagates updated state every clock cycle. At the same time, they offer architectural malleability unavailable in fixed-function silicon, enabling reconfiguration of table structures, expansion of metadata formats, or replacement of action logic without redesigning hardware. As a result, FPGAs provide a compelling substrate for programmable forwarding engines that require both high-performance execution and long-term adaptability to evolving protocol and in-network processing demands.

2.3.3 Deparser: header serialization and packet reconstruction

The deparser is responsible for reconstructing the outgoing packet stream by serializing selected headers and payload bytes according to the logical emission order specified by the P4 program. In the PISA model, the deparser operates on a packet-header vector (PHV) populated by the parser and subsequently modified by the match–action pipeline. Each header instance is re-emitted only if it was marked valid, and its fields may have been updated by earlier stages, requiring the deparser to enforce both header ordering and field consistency guarantees.

In hardware terms, deparsing consists of controlled byte extraction, alignment, and concatenation. Barrel shifters, multiplexer networks, and explicit pipeline registers ensure deterministic emission even at high clock frequencies. Unlike fixed-function ASIC deparsers, which expose a finite grammar of allowable header transitions and field placements, FPGA deparsers may incorporate custom alignment units, extended metadata fields, or variable-width output interfaces without modifying underlying silicon.

Compared with parsing and table lookup, deparsing has received less dedicated research attention and is often treated as a structural counterpart to the parser, despite non-trivial alignment and emission-ordering challenges at line rate [68].

Deterministic timing is a core property of the deparser datapath: each cycle emits a fixed-width word, and the internal pipeline latency is fixed for a given configuration, independent of packet content. At the same time, FPGA fabrics permit architectural scalability. Datapaths may be widened to increase peak throughput, emission lanes can be replicated to support limited packet-level parallelism, and additional scheduling logic may be introduced to accommodate evolving encapsulation formats. While FPGA implementations do not aim to match switch ASICs in terms of clock frequency, integration density, power efficiency, or specialized memory resources, they can achieve cycle-accurate, high-throughput streaming behavior within a single pipeline, comparable in execution semantics to ASIC deparser stages. This balance between analyzable pipeline timing and architectural flexibility makes FPGA-based deparsers well suited for deployments requiring protocol extensibility and design-space exploration.

The architectural correspondence between PISA abstractions and FPGA resources motivates the exploration of specialized FPGA parsing and deparsing architectures. Early systems relied on compile-time specialization, while more recent efforts seek runtime-programmable datapaths that retain deterministic timing.

2.4 Synthesis and Motivation of FPGA-Based Parsing and Deparsing Approaches

This chapter traced the evolution from fixed-function forwarding to fully programmable data planes, highlighting how SDN decoupled control from forwarding while packet-processing pipelines largely remained rigid. The transition from OpenFlow's field-bounded flow tables to the RMT and PISA architectures marked a fundamental shift, since parsing, match–action processing, and deparsing became programmable modules configured by tables rather than fixed silicon semantics. In parallel, the P4 language introduced a portable and high-level abstraction capable of expressing protocol-independent processing intent across heterogeneous hardware targets.

The review then examined the execution platforms for P4 programs. General-purpose CPUs and NPUs provide broad programmability, but have difficulty guaranteeing deterministic, worst-case throughput at scale because of shared resources and instruction-driven execution. Programmable switch ASICs achieve multi-terabit performance with tightly bounded latency, however they expose only the operations defined at fabrication time and therefore cannot freely adapt to new protocol formats or pipeline behaviors. FPGAs occupy an intermediate position, providing cycle-accurate pipelining, predictable latency, and the ability to restructure parsers, match–action logic, and deparsers after deployment. This alignment between PISA abstractions and FPGA capabilities, including bit-precise parsing, memory-centric table lookup, and structured header emission, positions FPGAs as a compelling platform for protocol-independent data planes.

These observations motivate the central research question addressed in this thesis: how to achieve runtime reconfigurability on FPGA-based packet-processing pipelines while preserving deterministic timing and high throughput. The next chapter builds on the insights established here, presenting a structured research methodology that formalizes memory-driven reconfiguration for the parser and deparser, defines the associated compilation and configuration artifacts, and evaluates scalability across protocol graphs and data-path widths within a FPGA framework.

CHAPTER 3 RESEARCH APPROACH

3.1 Methodological Framework

3.1.1 Research Scope and Objectives

This chapter presents the methodological foundations of the research and explains how the individual publications form a coherent and cumulative contribution toward enabling runtime-programmable packet parsing and deparsing on FPGA platforms. The goal of this thesis is to introduce programmability into the data-plane first and last stages while preserving the deterministic, cycle-accurate behavior and high throughput traditionally associated with FPGA architectures. In this thesis, the focus was placed on architectural designs for the initial (parser) and final (deparser) stages of the PISA pipeline, where protocol extraction and packet reconstruction occur. To achieve this objective, the research employed a structured methodology that integrates conceptual analysis, hardware–software co-design, architectural prototyping, and experimental evaluation.

The investigation began by examining the limitations of existing P4-to-FPGA design flows, which predominantly relied on compile-time specialization. In such flows, the underlying hardware pipeline is regenerated and resynthesized whenever the P4 program changes. While effective in static environments, this method restricts rapid protocol evolution and conflicts with the runtime programmability expected in modern programmable networks. Recognizing this limitation motivated a two-phase research trajectory.

3.1.2 Overview of Research Phases

The first phase investigated a templated architecture design, in which protocol structure and transitions are exposed to the hardware through synthesis-time generics and compiler-generated configuration vectors. This phase required the systematic identification and decoupling of all protocol-specific attributes, including header layouts, field sizes, transition conditions, and alignment boundaries, from the underlying RTL implementation. By externalizing these attributes, the architecture ensures that the same hardware description can be reused for different P4 parser or deparser specifications simply by substituting new generic values during synthesis. As a result, the design shifts protocol semantics, ensuring that the pipeline remains structurally invariant while adapting to varied use cases. In other words, the hardware was made protocol-agnostic at the structural level, and protocol behavior was injected through synthesis-time metadata rather than hard-coded logic.

This approach preserves the benefits of static FPGA pipelines, including fixed latency, predictable timing closure, and high operating frequency, while substantially reducing the manual RTL effort typically associated with accommodating new protocols. Importantly, instead of relying on complex toolchains or heavyweight intermediate representations, a lightweight software was developed to extract the required node attributes from the P4 description and supply them as generics to the hardware. Although this step did not yet eliminate the need for resynthesis, it represented a major conceptual milestone, since it demonstrated that parsing logic could be abstracted into data-driven control information, thereby laying the foundation for the transition to the next phase of the research.

While templated designs demonstrated practicality and significantly reduced hardware specialization effort, their reliance on synthesis prompted the next methodological step. To fully support runtime flexibility, the architecture transitioned to an overlay-based operation, where protocol semantics and transitions are interpreted from configuration memory rather than being embedded in the RTL. All decisions regarding header extraction, alignment, and emission are driven by memory tables generated automatically from the P4 description. As a result, the datapath remains constant, and reconfiguration time is reduced to the memory update latency rather than the full FPGA compilation time. This evolution completed the shift from static, parameterized structures to dynamically configurable hardware capable of supporting evolving packet formats in the field.

With the overlay architecture defined, the next stage of the work focused on validating whether the proposed design principles could sustain high throughput, predictable timing, and correct protocol behavior across different P4 DAGs. This phase began with conceptual and analytical evaluation, confirming that the control structures, data alignment rules, and table-driven state transitions generated from P4 descriptions were consistent with the intended DAG semantics. The architecture was then tested in simulation to verify functional correctness, including header extraction, field alignment, and protocol sequencing in various protocol combinations. Test traffic included synthetically generated packets constructed using Scapy to ensure the correctness across a wide range of protocol paths.

3.1.3 Design and Implementation Approach

Following functional validation, the designs were evaluated under realistic FPGA deployment conditions to confirm that frequency targets, timing behavior, and resource utilization remained within practical bounds. Measurements included the maximum achievable clock frequency, latency in clock cycles, LUT/FF/BRAM usage, as well as scalability, which was evaluated as protocol graph complexity and bus width increased. Hardware prototyping and synthesis were performed using Xilinx Vivado, targeting modern FPGA platforms, ensuring relevance to production-grade deployment conditions.

The shift to a memory-driven control path also required careful examination of configuration table organization, metadata mapping, and the interaction between control and data streams within the pipeline. These analyses ensured that introducing runtime programmability did not degrade correct operation or the expected performance characteristics of the hardware. Together, these conceptual, analytical, and experimental validation steps confirmed that the proposed templated and overlay architectures deliver runtime programmability while preserving the deterministic behavior and throughput requirements central to FPGA-based packet processing systems.

3.2 Content Organization

3.2.1 Paper 1: Templated Parser Architecture

This thesis follows the chronological and conceptual evolution of the research contributions. The papers begin by addressing the initial milestone: introducing a templated parser architecture capable of extracting key protocol information from P4 specifications and supplying it to hardware through synthesis-time generics. This first contribution demonstrates that protocol behavior can be externalized from the RTL, establishing a structured mechanism to derive parsing rules automatically from high-level descriptions. By defining this abstraction layer and proving the feasibility of metadata-driven parsing, the work provides a foundational step that enables the subsequent contributions.

3.2.2 Paper 2: Overlay Parser for Runtime Reconfiguration

Building on this foundation, the second contribution extends the templating concept toward full runtime programmability. Instead of embedding protocol metadata at synthesis time, this work transfers the generics into memory-driven control tables, forming an overlay parser architecture. This contribution shows how parse-graph semantics, header extraction rules, and transitions can be dynamically loaded into a fixed datapath without resynthesis, thus transforming the design into a runtime-programmable parser.

3.2.3 Paper 3: Unified Framework: Base, Overlay, and Pipeline Parsers

With the parsing stage established and runtime programmability demonstrated, the third contribution advances the research toward a performance-scalable and deployment-ready solution. This work introduces PrismParser, which evaluates a unified P4-to-FPGA methodology across three architectural forms: base, overlay, and pipeline. In this design, the base architecture demonstrates functional correctness and modularity with minimal logic resources, the overlay architecture introduces protocol flexibility by expressing parser behavior in configuration tables, and the pipeline architecture extends the base by parallelizing state evaluation across multiple stages to sustain higher throughput while preserving deterministic timing.

This contribution therefore bridges the conceptual runtime programmability achieved in earlier stages with practical, high-performance pipeline integration. By demonstrating that table-driven control scales from a static template to a reconfigurable model and that the same design principles can also be applied to build deeply pipelined, high-throughput parsers, this work reinforces the thesis objective and lays the foundation for the final step, where the complete parser–deparser runtime reconfigurable datapath is achieved.

3.2.4 Paper 4: Templated and Overlay Deparser Architectures

Building upon the principles established for the templated parser, the next contribution applies the same decoupling philosophy to the packet reconstruction stage. In this work, a templated deparser architecture is introduced, where protocol-specific header ordering, boundaries, and alignment rules are expressed as configuration parameters rather than fixed RTL logic. However, enabling such programmable permutation and emission behavior is traditionally achieved through wide crossbar networks, which incur significant routing complexity, LUT cost, and scalability limitations as the datapath widens or additional protocols are supported. To overcome this challenge, the proposed design replaces the conventional crossbar with a slice-and-shift mechanism that extracts, aligns, and forwards protocol fields based on parameterized offsets and sizes. This approach preserves timing and high throughput while reducing resource consumption, and it enables the deparser to support different protocol formats by simply adjusting configuration parameters.

3.2.5 Progressive Integration and Thesis Structure

These contributions establish a structured and cumulative research progression in which each step directly enables the next, both conceptually and architecturally. The thesis therefore evolves from identifying the need for protocol-agnostic control in FPGA packet processing, to demonstrating a templated parsing model capable of extracting and conveying protocol metadata, to enabling runtime reconfigurability through a memory-driven overlay parser, and finally to validating that the same principles can be extended to packet reconstruction with a resource-efficient, slice-and-shift deparser.

Across this sequence, the architectural abstractions become increasingly general and flexible, while still upholding strict FPGA constraints such as deterministic timing, high throughput, and minimal resource overhead. This organization ensures that the reader follows the natural trajectory of the research, seeing how initial templating concepts give rise to overlay mechanisms, how overlays mature into a pipelined implementation suitable for deployment, and how the final deparser work completes the end-to-end programmability vision. The chapters that follow present each contribution in detail.

CHAPTER 4 ARTICLE 1: A TEMPLATED VHDL ARCHITECTURE FOR TERABIT/S P4-PROGRAMMABLE FPGA-BASED PACKET PARSING

Parisa Mashreghi-Moghadam, Tarek Ould-Bachir, Yvon Savaria

Published in: International Symposium on Circuits and Systems (ISCAS)

Publication date: January 14, 2022

Preface: This chapter presents a templated VHDL architecture for P4-programmable packet parsing on FPGAs, achieving up to 549 Gb/s on a Xilinx Virtex-7 and approximately 1 Tb/s on an UltraScale+ device with small area footprint. The design employs a multi-stage header analysis pipeline, in which per-stage parameters, including protocol identifiers, key masks, shift offsets, and header size rules, are automatically extracted from P4 parsing graphs. As a templated solution, generic parameters are regenerated for each P4 program and the hardware is resynthesized, however, the process of generating new generics is automated and requires no manual RTL modifications. This work was peer-reviewed and published in the proceedings of the *2022 IEEE International Symposium on Circuits and Systems (ISCAS).*

Contributions: This research originated as part of my doctoral work at Polytechnique Montréal and was developed in collaboration with my co-authors. I contributed to the conception of the problem, the architectural design, testing and deployment of the system, literature review, analysis of the results, and preparation of the manuscript. My co-authors provided guidance, feedback, and supervision throughout the research and revision process.

Full Citation: Parisa Mashreghi-Moghadam, Tarek Ould-Bachir, Yvon Savaria *"A Templated VHDL Architecture for Terabit/s P4-programmable FPGA-based Packet Parsing,"* in *Proceedings of 2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, Austin, TX, USA.

DOI: 10.1109/ISCAS48785.2022.9937607

Copyright: © 2022 IEEE. Reprinted, with permission from the authors and publisher.

4.1 Abstract

This paper proposes a templated VHDL architecture for P4-programmable packet parsing on FPGAs offering high throughput while occupying a small area footprint. The architecture comprises a multi-stage header parser unit arranged in a pipelined structure. Each header analysis unit is characterized by a set of generic parameters reflecting unique features and relations of supported protocols retrieved from the P4 code that describes each stage along the pipeline. Synthesis results of the packet parser show up to 549 Gb/s throughput on a Xilinx Virtex-7 FPGA and 1 Tb/s on a Xilinx UltraScale+ for a twelve-stage pipeline. Compared with state-of-the-art solutions, our proposed architecture performs at higher throughput with acceptable resource utilization.

4.2 Introduction

Software-Defined Networking (SDN) revolutionized the traditional networking paradigm by breaking their classic vertical integration and decoupling the control plane from the data plane. With this separation, network switches become simple forwarding devices, and the control logic is implemented in a logically centralized controller [69]. OpenFlow is the standard protocol for implementing SDN [11]. Programming Protocol-Independent Packet Processors (P4) [13] is a Domain Specific Language (DSL) that mitigates OpenFlow's limitations, such as protocol dependency and rigidity, in order to facilitate supporting new protocols. The Protocol Independent Switch Architecture (PISA) [12] is a RISC-inspired pipelined architecture that uses P4 for programming the data plane. PISA consists of a programmable parser, a series of match action table engines, and a packet deparser.

A packet parser (PP), fixed or programmable, identifies and extracts appropriate packet header fields. As PISA's primary component, the PP must be fast and programmable to fulfill subsequent processing requirements. Among various devices proposed for networking purposes, FPGAs gained interest as good targets to maintain flexibility and high speed of SDN switches. Due to the general hierarchical nature of the protocols, a protocol field in a packet header is unknown until its preceding protocol element is parsed, so PPs are often proposed in a pipelined fashion. Design principles of a PP are presented in [66], and several approaches to generate FPGA-based PPs have been proposed in the literature.

VHDL implementations of P4-compatible parsers were investigated in recent works. Benácek et al. [16] introduced a high-speed automated P4-to-VHDL PP generator for FPGAs. Da Silva et al. [58] proposed a framework to convert P4 to HDL using HLS and templated C++ classes derived from P4. Cao et al. [70] presented a framework for a pipeline-based templated HDL architecture that converts P4 programs to VHDL. Cabal et al. [71] introduced a novel approach that can scale up to a Tb/s throughput on a Xilinx UltraScale+ FPGA; their methodology includes multiple packet placement on the data bus to feed parallel parsers.

This paper proposes a design methodology for P4-programmable protocol-agnostic PPs where header analysis (HA) units are arranged in a pipelined fashion. The HA units are defined in a templated VHDL code configured through a Protocol Identification Vector (PIV) and configuration parameters. The PIVs and the parameters are generated by a Python script that extracts critical information from the parsing graph obtained from a P4 code. The main contributions of this paper are as follows:

- A new header analysis multi-staged architecture that can be targeted to implement P4-programmable protocol-independent parsers;

- A demonstration that such an approach can achieve 1Tb/s throughput on a single FPGA using a templated VHDL implementation of the parser.

The remainder of this paper is organized as follows: Section 4.3 reviews prior works on network PPs. Section 4.4 presents the problem formulation of packet parsing, and gives an overview of the proposed architecture. Section 4.5 presents and discusses our experimental results. Finally, Section 4.6 concludes this work by summarizing our main findings.

4.3 Prior Works on Parser

Prior works and methodologies to develop packet parsers on FPGA are investigated in this section. Kangaroo [72] introduces a programmable parser with extremely high latency that is incompatible with today's switches. Attig and Brebner [19] constructed a high throughput reconfigurable parser; nonetheless, due to a long pipeline, this architecture encounters high latency and resource usage. Other approaches, including packet processing with an NoC-enhanced FPGA [73], and custom processor for protocol-independent packet parsing [74] were also examined. A 100 Gb/s P4 to VHDL PP generator was proposed in [16]. Moreover, Benacek et al. [57] presented a tool to convert the P4 description of high-speed input (Parser) and output (Deparser) network blocks. Da Silva et al. [58] proposed a framework to generate low latency and high-speed PP directly from P4 and used Vivado HLS to translate the pipeline description by C++ classes to RTL and achieved a 100 Gb/s throughput and lower resource utilization compared to the state-of-the-art. Yazdinejad et al. [75] presented an automatic parser generation from P4 to SDNet. Cao et al. [70] proposed a framework containing pipelined templated VHDL designs to generate P4 parsers on FPGAs. Cao et al. [20] presented a template-based framework to convert P4 network processor to VHDL and automatically implement on FPGAs. Cabal et al. [71] proposed a novel approach with multiple parallel parsers with high effective throughput. Unlike previous works, we managed to obtain 1 Tb/s throughput while consuming less resources on FPGA, compared to the state-of-the-art, without changing the data bus structure for a single parser.

4.4 Proposed Generic Template-based Solution

4.4.1 Problem Formulation

The parser section of a P4 specification can be translated into a directed acyclic graph (DAG), as in Fig. 4.1, where each node corresponds to a specific protocol, and each edge between nodes represents possible transitions. Additionally, the P4 code declares the headers of

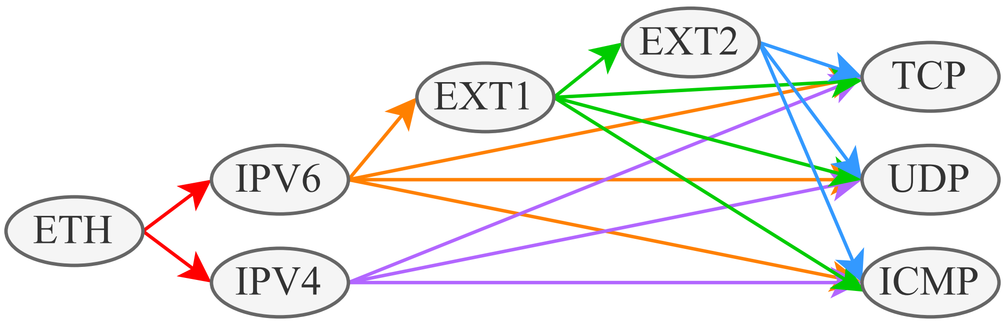


Figure 4.1 Example of a simple directed acyclic parsing graph.

desired protocols and the parsing state-machine. A JavaScript Object Notation (JSON) array, a lightweight data-interchange format, is extracted from P4 code using the P4C compiler; subsequently, a JSON parser produces a Full Graph as mentioned in [58]. Each node of the Full Graph represents a protocol that is labeled in topological order. This number, protocol id, is assigned to each node to identify the node's protocol. The node connections and inherited particular characteristics such as header length, field's location containing transition information and its size, etc., are stored in a specialization vector called PIV, as explained in detail in section 4.4.3. The process of generating the PIV from the Full Graph is scripted in Python and described in Fig. 7.2. This methodology allows using the same templated architecture for parsing elements when the input P4 parser changes by simply regenerating the PIVs and adjusting the generic parameters. Notably, the number of pipeline stages is provided as a generic parameter equal to the number of nodes in the graph.

All stages have the same microarchitecture and a PIV to determine which protocol a given stage can parse. Moreover, each pipeline stage parses one protocol per cycle. A header slice including all packet header data is supplied to the parser, similar to [70], to ensure that the parser parses one protocol each cycle. This slice must be wide enough to support the most extended combination of protocol headers. Every stage receives the header slice and a protocol bitmap as an input, where the protocol bitmap contains one bit per protocol, and its LSB represents a status bit to show that the parsing is done. The following sections take a top-down approach to explain the intricacies of our parser.

4.4.2 High-Level Architecture

Fig. 4.3 illustrates the high-level view of the proposed PP. A Header Analysis (HA) module parses one header per clock cycle. These HA modules are identical in all pipeline stages, and the Protocol ID identifies the protocol that each step will parse. One header slice enters the

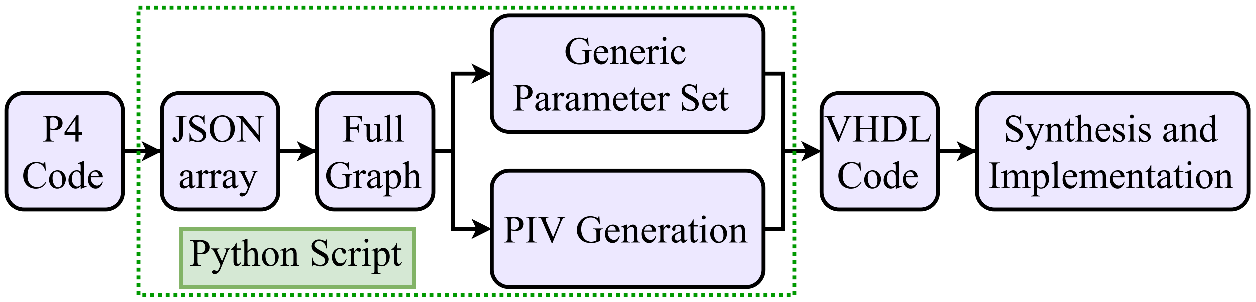


Figure 4.2 Proposed compilation workflow for the parser.

pipeline stream every clock cycle. If the input protocol bit is set in the input bitmap and the HA module detects a protocol transition, the protocol bitmap gets updated before being sent out, the header segment is stored as a Packet Header Vector (PHV), and the input data is aligned for the next stage being forwarded. Otherwise, the protocol bitmap is not updated, the PHV is filled with zeros, and the header slice is forwarded downstream without changes. The proposed PP generates an output after N clock cycles as N PHV vectors and a protocol bitmap that indicates the present protocols in a parsed header slice.

4.4.3 Hardware Specifications and Protocol Identification Vector

The PIV contains the necessary information for the HA block to perform the parsing. PIV has characterized fields as follows:

Header Size

The header size is a critical parameter to align the data for the next stage. To simplify the data offset calculations and yield a less complex barrel shifter for the PHV and extracted key, each stage of our pipeline aligns the incoming data stream before forwarding it to the next stage, similar to what is proposed in [58].

Key Location and Key Size

Each protocol contains a unique identifier (the key) to indicate the transition to the next protocol. This key is located at a certain position, called the key_location, and has a specific length, called key_size.

To further optimize the space in HAs, instead of storing the key_location and key_size, a

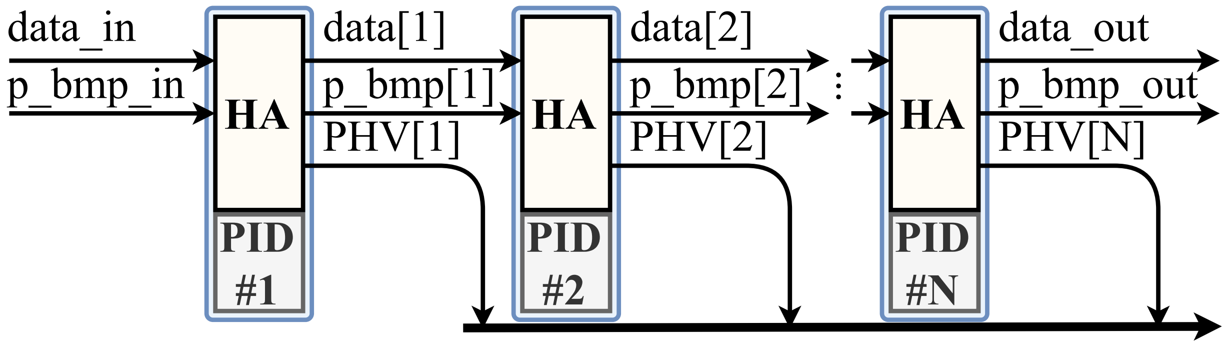


Figure 4.3 Multi-staged architecture: Pipeline structure with N Header Analysis (HA) modules, each of which is identified by a Protocol_ID (PID).

shift_value is stored in the PIV and is calculated as follows:

$$\begin{aligned} \text{shift\_value} &= \text{PHV\_SIZE\_BITS} \\ &- \text{key\_location} \\ &- \text{key\_size} \end{aligned} \tag{4.1}$$

The shift_value is the amount of logical right shift applied on PHV_size_data_in, and PHV_SIZE_BITS corresponds to the longest protocol in the parsing graph.

Protocol Mask

Applying shift right logical on PHV_size_data_in with shift_value, brings the key to LSB position. Since the key_size differs in every protocol, PIV has a reserved space to indicate the key_mask to extract the key from the header chunk.

Next Protocol IDs and Key Values

protocol_id grants a unique ID to every node.

The extracted key from the protocol header in the current stage must be compared with key_values regarding the connections derived from the P4 code. In case of a match, a transition is made to the correspondent node.

Furthermore, based on the maximum number of connections between two nodes in the parsing graph, next_protocol_id and key_value pairs may repeat in the PIV.

Variable Size Headers

As mentioned earlier, the header_size is critical to prepare the data for the next pipeline stage. Some protocols support variable-size headers, which must be calculated in case of the presence.

$$\begin{aligned}&\text{var\_header\_length} = \\ &(\text{var\_shift\_data\_masked} + \text{var\_add}) \ll \text{var\_shift}\end{aligned} \tag{4.2}$$

4.4.4 HA Microarchitecture

The proposed header analysis microarchitecture is depicted in Fig. 4.4. Output signals mirror the input signals, implying that the input signals are modified before being transmitted to the output. Furthermore, if the protocol is present in the header slice, the input segment corresponding to the current protocol is cropped and sent as the PHV. The length of this PHV is determined by the longest supported protocol in the graph. The parsed header will be inserted at the PHV_out LSB fields, and zeros will fill the remaining bits if the parsed header is shorter than the PHV vector size. data_in and protocol_bitmap_in enter the block and undergo a series of transformations before being outputted. protocol_bitmap indicates which protocols are present in a parsed header and includes one bit per protocol supported by the input graph and one bit to indicate that the parsing is done. Every HA is fully combinational, allowing each pipeline stage to detect a protocol in one clock cycle.

There are three main sub-blocks in our generic HA:

PHV Generator

Based on the longest protocol supported by the parser, a slice of the incoming data, called PHV_size_data, is fed into the PHV generator as an input. phv_shift_value is calculated by extracting the header_size value from the PHV_SIZE_BITS. The PHV_size_data is shifted to the right by phv_shift_value; therefore, only the part of the PHV_size_data containing the protocol header information remains.

State Transition

The sub-block receives PHV_size_data. Simpler logic is obtained by cropping the PHV_size_data because a smaller portion of the input data is fed to sub-blocks instead of the entire data_in. Notably, this PHV bus size supports the longest possible protocol, so smaller protocols can easily fit into this portion. The PHV_size_data is shifted to the right by shift_value, mentioned in equation 1, and masked with key_mask, from the PIV to generate the key vector.

Header Size Calculator

Header size is needed both in PHV calculations and in data_out calculations. If the protocol has a fixed size, the information regarding the protocol size is stored in a specific place in the PIV that can be easily used for further calculations. However, if the header has a variable size, the size is calculated using equation 2. The output data is prepared by using the header size to shift the data and align it prior to forwarding it to the next stage.

4.5 Results

The same two classes of experiments reported in [58] were conducted to evaluate our design. Two classes of parsers were implemented: 1) Simple parser: including Ethernet, IPv4/IPv6 (with two extensions), UDP, TCP, and ICMP/ICMPv6; 2) Full parser: Same as the simple parser but also includes MPLS (with two nested headers) and VLAN (inner and outer).

In Table 4.1, synthesis results drawn from the proposed implementation are compared with works presented in [16, 58, 70, 71] that support both fixed-sized and variable-sized headers. Our results are compared in terms of bus size, clock frequency, throughput, and resource utilization. To allow comparing our proposed design with [16, 58, 70], we target the Xilinx XC7VX6 Virtex-7 FPGA; and Xilinx XCVU7P UltraScale+ FPGA to compare with [71].

We proposed a generic framework and chose a bus size of 1280 bits to ensure supporting the most extended header slice combinations for different parsing graphs in a single clock cycle; moreover, we assigned a 153-bit vector for PIV. This vector contains five pairs of 20-bit sections of key_value and protocol_id, an 18-bit section for the var header length parameters, a 33-bit section for Key extraction information and the last two bits to indicate var_header and end_header for a particular protocol.

We compared our architecture with [58] implemented with Vivado HLS. As our architecture is modular, we could identify redundant logic that could be pruned to enhance speed. Considering their reported clock does not drop if the bus size quadrupled, our proposed design managed to get 1.37× their throughput. Compared to a similar templated architecture in [70], we achieved 1.73× higher throughput on the same FPGA since we developed a script to prepossess some fixed values to simplify the hardware used at run-time. In [71], the authors also reported 1 Tb/s throughput on an UltraScale+ FPGA; yet, their design contains multiple parsers running in parallel and a complex data bus structure that causes hardware complexity.

Furthermore, we present a general design that can be easily adapted to any P4 code by altering the generic parameters and recreating the PIVs using a Python script without going through all of the processes from P4 to HDL. With a bus size of 1280, we only use 37% more resources than [58] with a bus size of 320. We were able to achieve a small footprint by developing a custom parser, whereas [58] used Vivado HLS.

4.6 Conclusions

In this paper, we presented a templated VHDL protocol-agnostic P4-programmable PP implemented on FPGA. A Protocol ID identifies each pipelined stage, and a PIV vector assigns specific information for parsing each protocol. The pipeline component can be repeated in each pipeline stage, and a Python script can regenerate PIVs. Our pipeline has the same number of stages as the number of graph nodes. Every stage has the same microarchitecture, and its specific PIV varies according to the characteristics of each pipeline stage. The main results include a PP that offers up to 549 Gb/s throughput on a Xilinx Virtex-7 FPGA and 1 Tb/s on a Xilinx UltraScale+ for a twelve-stage pipeline. This performance is obtained with resource utilization comparable to previously reported designs.

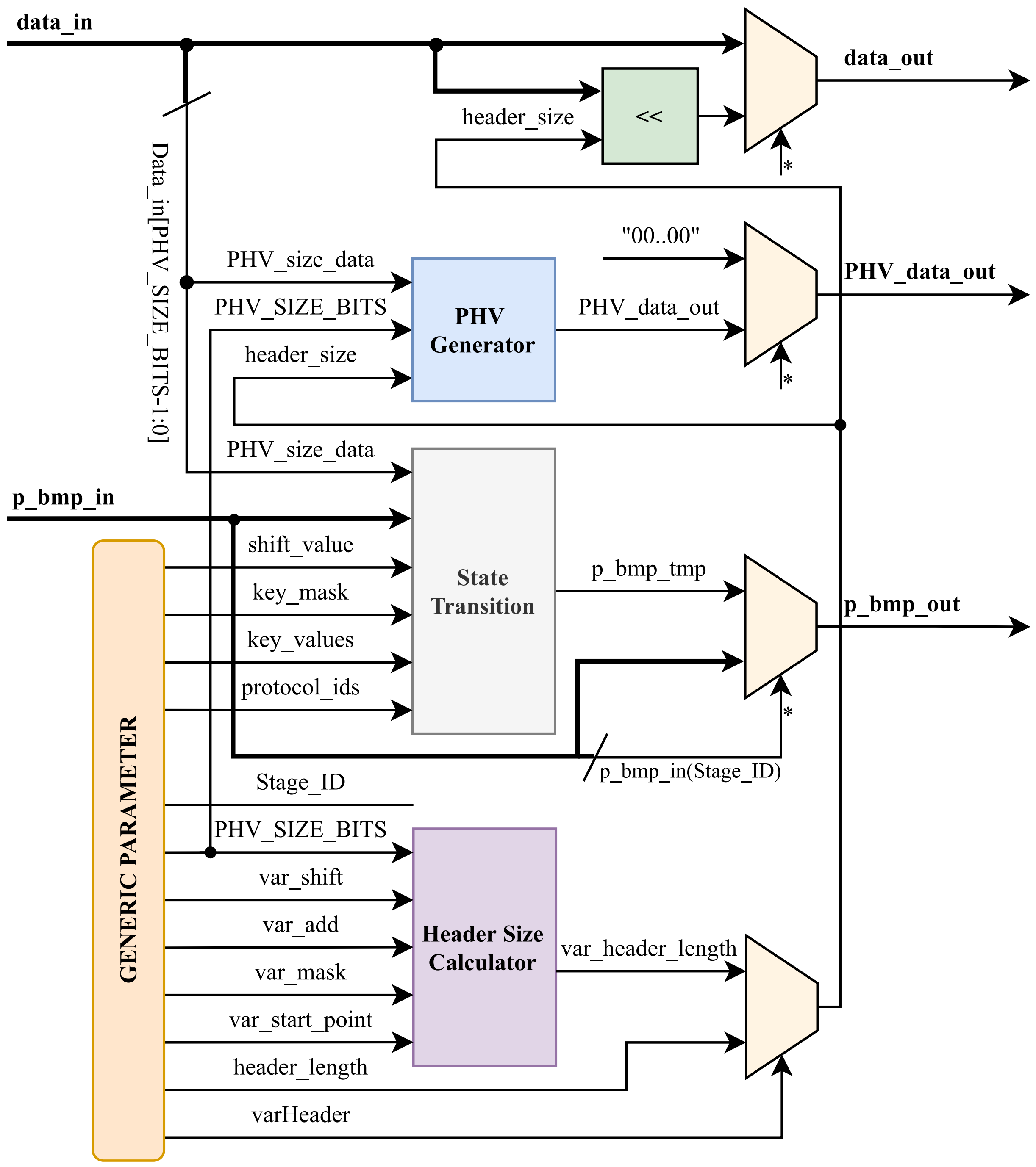


Figure 4.4 Architecture of the header analysis unit.

Table 4.1 Table of result comparison.

Work	FPGA	Performance				Resources			Extracted Fields
		Data Bus [bits]	Frequency [MHz]	Throughput [Gb/s]	Latency [ns]	LUTs	FFs	Slice Logic (LUTs + FFs)	
Simple Parser									
Golden [16]	Virtex-7	512	195.3	100	15	N/A	N/A	5,000	TCP/IP 5-tuple
[16]	Virtex-7	512	195.3	100	29	N/A	N/A	12,000	TCP/IP 5-tuple
[58]	Virtex-7	320	312.5	100	19.2	4,270	6,163	10,433	TCP/IP 5-tuple
[58]	Virtex-7	320	312.5	100	19.2	5,888	10,448	16,336	All fields
[70]	Virtex-7	1072	346	370	14.45	4,884	3,135	8,019	TCP/IP 5-tuple
[70]	Virtex-7	1072	334.4	358	14.55	10,495	6,886	17,381	All fields
[71]	UltraScale+	4096	244.1	1000	28	N/A	N/A	66,000	N/A
Proposed	Virtex-7	320	499.25	159.7	16.024	1,491	1,540	3,031	TCP/IP 5-tuple
Proposed	Virtex-7	320	453.3	145.05	17.648	3,670	4,010	7,680	All fields
Proposed	Virtex-7	1280	528.85	676.53	15.13	7,224	5,469	12,693	TCP/IP 5-tuple
Proposed	Virtex-7	1280	523.83	670.5	15.27	10,455	8,394	18,849	All fields
Proposed	UltraScale+	1280	877.9	1123.7	9.112	7,227	5,469	12,696	TCP/IP 5-tuple
Proposed	UltraScale+	1280	877.9	1123.7	9.16	10,948	8,888	19,836	All fields
Full Parser									
Golden [16]	Virtex-7	512	195.3	100	27	N/A	N/A	8,000	TCP/IP 5-tuple
[16]	Virtex-7	512	195.3	100	46.1	10,103	5,537	15,640	TCP/IP 5-tuple
[58]	Virtex-7	320	312.5	100	25.6	6,046	8,900	14,946	TCP/IP 5-tuple
[58]	Virtex-7	320	312.5	100	25.6	7,831	13,671	21,502	All fields
[70]	Virtex-7	1136	320.5	364	21.84	9,515	6,930	16,445	TCP/IP 5-tuple
[70]	Virtex-7	1136	279.3	317	25.06	16,888	12,033	28,921	All fields
[71]	UltraScale+	4096	244.1	1000	60	N/A	N/A	145,000	N/A
Proposed	Virtex-7	1280	429.1	549.3	27.96	12,265	10,628	22,893	TCP/IP 5-tuple
Proposed	Virtex-7	1280	428.6	549	27.996	15,787	13,689	29,476	All fields
Proposed	UltraScale+	1280	807.7	1033.9	14.856	12,207	10,628	22,835	TCP/IP 5-tuple
Proposed	UltraScale+	1280	800	1024	15	15,634	11,476	27,110	All fields

CHAPTER 5 ARTICLE 2: AN AREA-EFFICIENT MEMORY-BASED ARCHITECTURE FOR P4-PROGRAMMABLE STREAMING PARSERS IN FPGAS

Parisa Mashreghi-Moghadam, Tarek Ould-Bachir, Yvon Savaria

Published in: International Symposium on Circuits and Systems (ISCAS)

Publication date: January 20, 2023

Preface: This chapter presents a memory-centric, P4-programmable streaming packet parser for FPGAs that decouples protocol control flow from the datapath by encoding keys, masks, header lengths, and state transitions in configuration tables. This architecture separates protocol-specific behavior from the underlying hardware, enabling fast retargeting using P4-generated configuration data. This allows new protocol graphs to be deployed without modifying or resynthesizing RTL. The parser operates as a single generic analysis block and supports reprogramming through embedded memory updates. This work achieved 11 Gb/s on a Xilinx Virtex-7 device, utilizing only 312 LUTs and 1,135 FFs, and was published as an overlay-based approach to low-area, P4-configurable parsing. This work was peer-reviewed and published in the proceedings of the *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*.
Contributions: This research originated as part of my doctoral work at Polytechnique Montréal and was developed in collaboration with my co-authors. I contributed to the conception of the problem, the architectural design, testing and deployment of the system, literature review, analysis of the results, and preparation of the manuscript. My co-authors provided guidance, feedback, and supervision throughout the research and revision process.
Full Citation: Parisa Mashreghi-Moghadam, Tarek Ould-Bachir, Yvon Savaria *"An Area-efficient Memory-based Architecture for P4-programmable Streaming Parsers in FPGAs,"* in *Proceedings of 2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, Monterey, CA, USA.
DOI: 10.1109/ISCAS46773.2023.10182176
Copyright: © 2022 IEEE. Reprinted, with permission from the authors and publisher.

5.1 Abstract

Moving toward software-defined networking and function virtualization, flexibility and reconfigurability of the network have become more and more critical. Packet parsing, the first processing stage of programmable switches, requires high performance and reconfigurability to allow implementing low-latency and highly flexible data networks. This paper proposes an overlay architecture for an FPGA-based P4-programmable streaming packet parser. The purpose of this architecture is to allow supporting different functionality with a fixed hardware design by changing a program stored in an embedded memory. This program is derived from the parser section of a P4 code, describing a parsing graph. This approach eliminates a pipeline of parsing blocks in favor of a single parsing block, thereby reducing the design's

complexity. Our architecture offers an 11 Gb/s data rate on a Xilinx Virtex-7 XC7VX690 FPGA, while its implementation requires 312 LUTs and 1135 FFs.

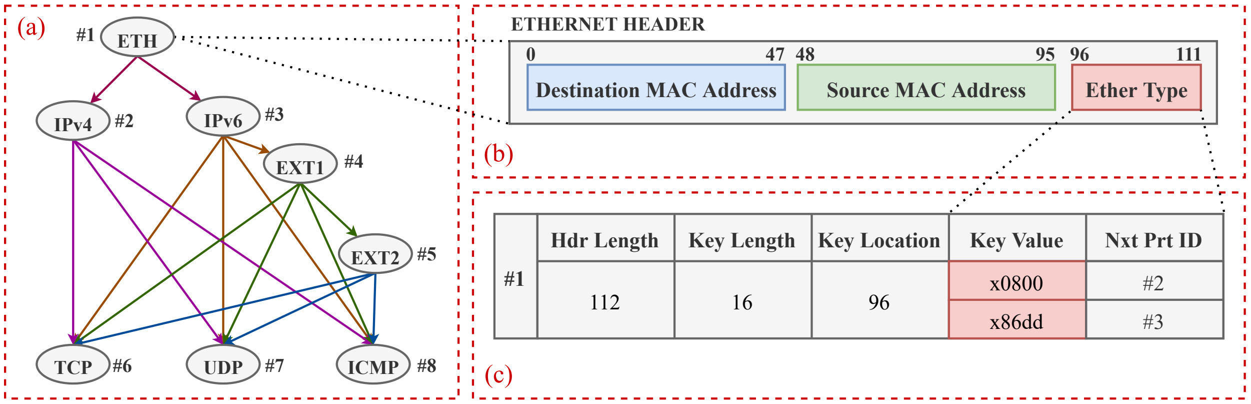
5.2 Introduction

In recent years, traditional IP data networks have grown in size, complexity, and speed. Despite the ubiquitous adoption of IP networks, they remain rigid, complicated, and challenging to manage [6] To help restore network flexibility, the Software Defined Networking (SDN) [9] paradigm has been created. The data plane, which is responsible for packet processing, and the control plane, which is logically centralized to command the data plane, are separated in the SDN paradigm so that they can evolve independently. McKeown et al. proposed OpenFlow [11] in 2008 to provide a standardized interface between the control and data planes. Bosshart et al. introduced the Protocol Independent Switch Architecture (PISA) [12] in 2013 as an abstraction for packet processing to solve the shortcomings of OpenFlow. PISA consists of a programmable parser, a series of match action table engines, and a packet deparser. In order to make the PISA model more programmable, Bosshart et al. proposed a Domain Specific Language (DSL) called Programming Protocol-Independent Packet Processors (P4) [13] with three main goals: reconfigurability in the field, protocol independence, and target independence. With SDN's separation of concerns, new protocols can be deployed on a centralized controller, and new forwarding rules can be compiled for the data plane without changing the underlying hardware. Thus, among the various networking devices presented, FPGAs have garnered interest as a possible target for maintaining the flexibility and high speed of SDN switches.

Packet parsing, as the initial phase of packet processing in PISA switches, looks at incoming packets to identify and extract appropriate packet header fields to be checked in the following processing stages. A highly efficient packet parser must be designed to achieve low-latency packet processing on the network. Aiming toward P4 programmable packet parsers, FPGA-based architectures have been developed and reported in recent years [16, 58, 66, 70, 71] as suitable hardware prototyping platforms. However, the downside of these works is that FPGA-based parsers are obtained by re-synthesizing an RTL (Verilog, VHDL, or other) intermediate code whenever there is a change in the input P4 source code. Thus a solid understanding of hardware design is required to make any changes to these designs.

The primary goal of this paper is to propose an area-efficient reconfigurable architecture for a streaming packet parser. The summary of the contributions of this paper are as follows:

- A novel generic parsing approach is proposed. Unlike previous works, it allows the same



	Hdr Length	Key Length	Key Location	Key Value	Nxt Prt ID
#1	112	16	96	x0800	#2
				x86dd	#3

Figure 5.1 Example of extracting specific information for one node of a parsing graph.

hardware architecture to be reused to parse various packet headers, as its functionality is defined by the content of a small embedded memory storing a program. The proposed approach thus reduces the pipeline of parsing blocks to a single generic parse block that can analyze any protocols.

- An end-to-end P4 parser to hardware framework is developed to generate the memory content, an executable machine language program. So, the parser can be reprogrammed whenever the P4 code changes.

- A dedicated parsing block can be assigned to each SFP+ interface [76]. The SFP+ provides a 10 Gbit/s throughput interface for high-speed network devices and is referred to as a parsing channel in this paper. It is demonstrated that a parsing block can be assigned to each parsing channel capable of processing incoming packets at a rate of 10 Gbps. As a result, multiple parsers can run in parallel to increase throughput if needed to serve multiple channels.

- Although programmability is obtained by adding a small embedded memory component to the parsing block, resource utilization is reduced as each parsing channel is composed of only one simple parsing block.

The remainder of this paper is organized as follows: Section 5.3 reviews prior works on P4-programmable parser Section 5.4 outlines the concept of the packet parsing problem and summarizes the main features of the proposed design. Section 5.5 presents and discusses our experimental results. Finally, Section 5.6 concludes this work by summarizing our main findings.

5.3 Prior Works on P4-Programmable Parser

This section explores previous efforts to obtain FPGA-based packet parsers. Different strategies have been reported. Early works, such as [19, 72] demonstrated programmable parsers with high latencies. TCAM-based designs [12, 66] provide low latencies but are not flexible, not scalable, and they require complex logic.

Custom processors and processors for packet parsing are provided in [62, 74, 77–79]. While some architectures are unchanged if additional protocols or features are to be supported; new instructions must be added. Therefore, these parsers are reconfigurable, and also they suffer from high latency.

Automatic generation of P4-programmable packet parsers is investigated in [16, 57, 75]. Furthermore, Da Silva et al. [58] provided a framework for directly generating low latency and high-speed packet parser from P4 using Vivado HLS to translate the pipeline description via C++ classes to RTL, achieving a 100 Gb/s throughput and lower resource consumption when compared to previously reported solutions. These designs are fast and reconfigurable, but all the P4 to RTL steps must be repeated whenever the input P4 code changes.

A templated packet parser architecture is presented in [59, 70]. This architecture is reconfigurable through generic parameters by keeping the same parser hardware architecture. Again, this design keeps the same RTL, but the code must be re-synthesized and implemented every time the P4 parser code and, by extension, the generic parameters change. In order to avoid the re-synthesis step, the design must be expressed by information stored in a memory. This is a common feature of overlay architectures.

5.4 Programmable Parser Design Methodology

This section provides a top-down description of the proposed memory-based programmable parser architecture. First, section 5.4.1, discusses the problem formulation. Second, section 5.4.2, illustrates the memory content generation from the input P4 parsing graph. Next, section 5.4.3 presents the high-level architecture view of the parser, and then, section 5.4.4 shows the architecture of the Header Analysis block.

5.4.1 Problem Formulation

Parsing is the process of detecting and extracting the proper fields from a packet header. A parser identifies different protocol headers in a hierarchical order, so a protocol header in a packet header is not known until its preceding protocol layer has been parsed. A directed

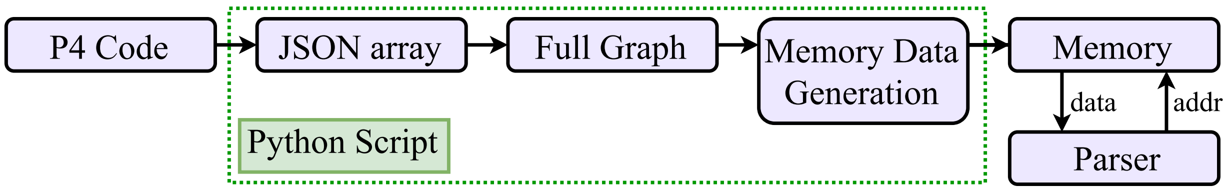


Figure 5.2 Proposed compilation workflow for generating the memory content

acyclic graph (DAG), Fig. 5.1(a), is used to represent the parser section of a P4 code. Each node in this graph portrays a different protocol, and each edge between nodes indicates a potential transition. Furthermore, as seen in Fig. 5.1(a), an example of a simple parsing graph, each node is labeled/numbered in topological order. This number, the protocol ID, is assigned to each node to identify the node's protocol.

Moreover, the parser section of the P4 code declares the headers of the desired protocols and the parsing state machine. In every protocol, there is a field that indicates the following encapsulated protocol in the frame's payload, which is called a Key in the present paper. Fig. 5.1(b) shows the different fields of the Ethernet header frame and the size of each field. For example, the Key for Ethernet is a two-Byte field called Ether Type, and the parser state machine retains the values to match against this field.

By separating the parsing logic from the critical parsing information for specific headers, a generic hardware architecture can parse different packet headers by storing the vital data of all graph nodes in a memory component. This memory stores specific characteristics about each node of the graph, including its header length, Key length, Key location, Key values that initiate a transition in the event of a match, and the subsequent protocol IDs to which they belong. The data stored in this memory for each protocol that is derived from the P4 code is called the Memory Data (MD), and it is stored at the same memory address as the protocol ID. This data is analogous to a program. Figure 5.1(c) depicts the information included in MD for the Ethernet protocol.

5.4.2 Memory Content

As previously mentioned, the parser's memory stores information about each node's characteristics and its connections' properties. The procedure that generates the Memory Data (MD) from input P4 code is outlined in Fig. 7.2.

We propose an end-to-end P4 parser to hardware framework which compiles the P4 code to a JSON array using the P4C compiler. As mentioned in [58], the Full Graph is generated from that JSON array. Subsequently, a Python script is developed to fill the memory content from

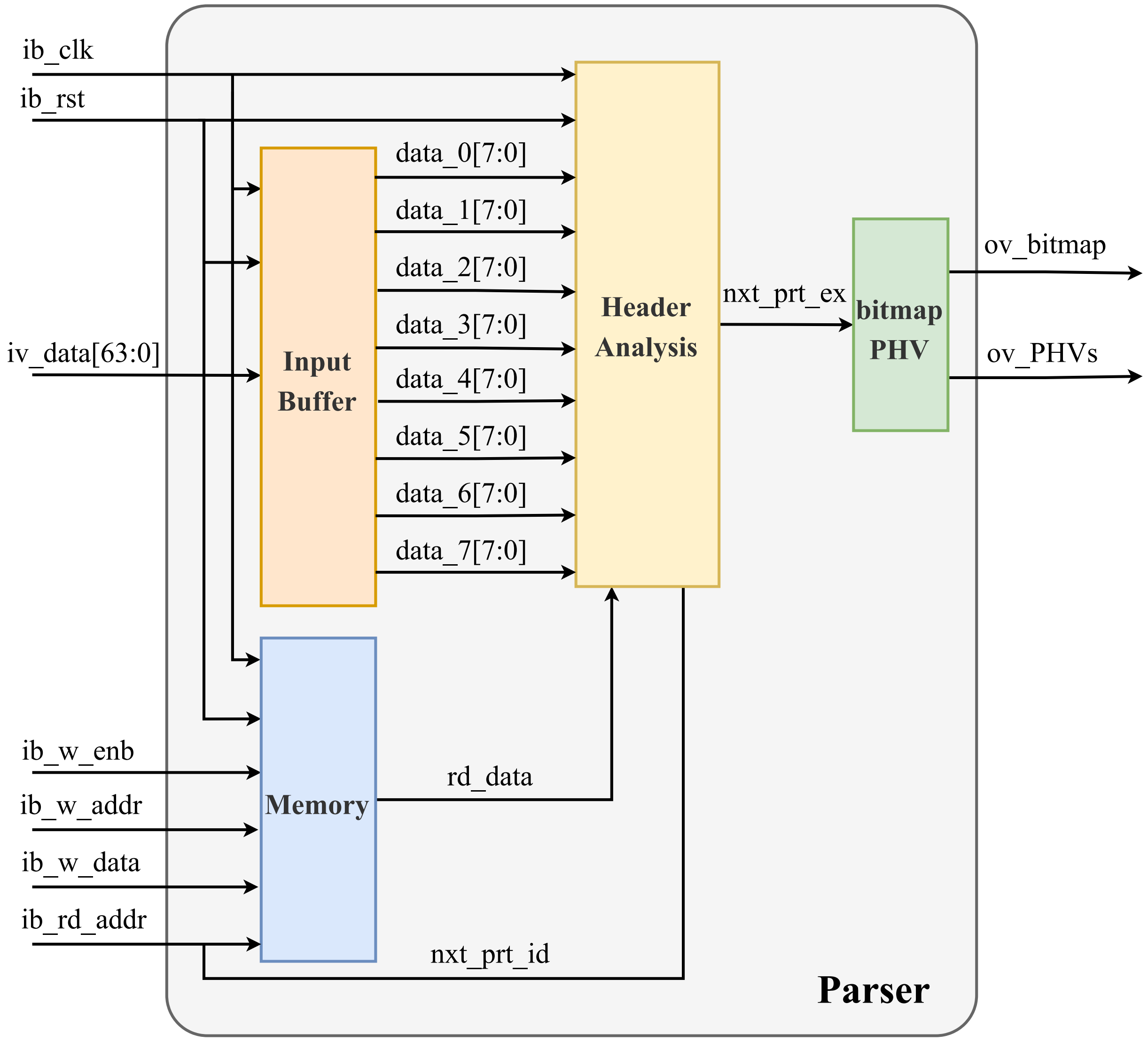


Figure 5.3 High-level architecture of programmable parser block.

the Full Graph, which the synthesis tool can use to instantiate the small embedded memory on the implemented hardware. After being instantiated, the memory can be reprogrammed with a new content file whenever there is a change in the input P4 parsing code.

5.4.3 High-Level Architecture

Figure 5.3 depicts the high-level view of the proposed programmable packet parser. In the proposed architecture, a single parsing block replaces the pipeline of parsing blocks found in other designs in which there is a dedicated block to parse each specific protocol. This simplification is achieved by decoupling the logic that parses the incoming packet, known as the Header Analysis (HA) block in this work, from the critical data associated with each protocol, which is stored in memory.

Input data is divided into eight parallel bytes, since the Keys are 1-byte aligned. These incoming data bytes are connected to the multiplexers inside the HA block. The Memory Data is fetched from the memory's protocol ID address and enters the HA block. The initial header type is commonly fixed for a specific network (e.g., Ethernet); consequently, this initial known protocol's information is stored in the first memory address, which is read from memory in the initial state.

Then the parser identifies the header field indicating the transition to the next protocol layer. The new parsing data is then fetched from this new address in the following clock cycle to parse the subsequent protocol if a transition is found.

The protocol bitmap indicates which protocols are included in a parsed header, with one bit representing each protocol supported by the input graph. If the HA block detects a protocol transition, the parser updates the protocol bitmap and sets the bit in the protocol ID position in the bitmap to one. Furthermore, the header segment is stored as a Packet Header Vector (PHV) for subsequent processing stages.

5.4.4 Header Analysis Block Architecture

Figure 5.4 depicts the proposed Header Analysis block architecture. Incoming data enters the HA block in eight parallel bytes, along with the protocol's MD fetched from memory. The eight parallel bytes are passed forward through two multiplexers inside the HA block. The first multiplexer detects the Key's first byte, while the second detects the Key's succeeding byte to form a 16-bit Key. This 16-bit Key is shifted and masked to form the desired Key of the incoming data.

The Control unit drives the select signals for the multiplexers based on Key location that is retrieved from the protocol's MD since the Key of different protocols has different sizes. Moreover, the Control unit regulates the other control signals, including the valid section detection. This block is the core of the HA block that makes decisions based on the information it extracts from the MD it receives. The MD contains the information, including the protocol header size, the Key location, and the Key size. The Control unit keeps track of the section of the input packet it parses, with a counter attached to this block. If the counter reaches the number of parsed bits as the Key location, it sets the valid section as true to update the next reading address from memory.

The Key extracted from the incoming data is compared with the Keys of the parsing protocol

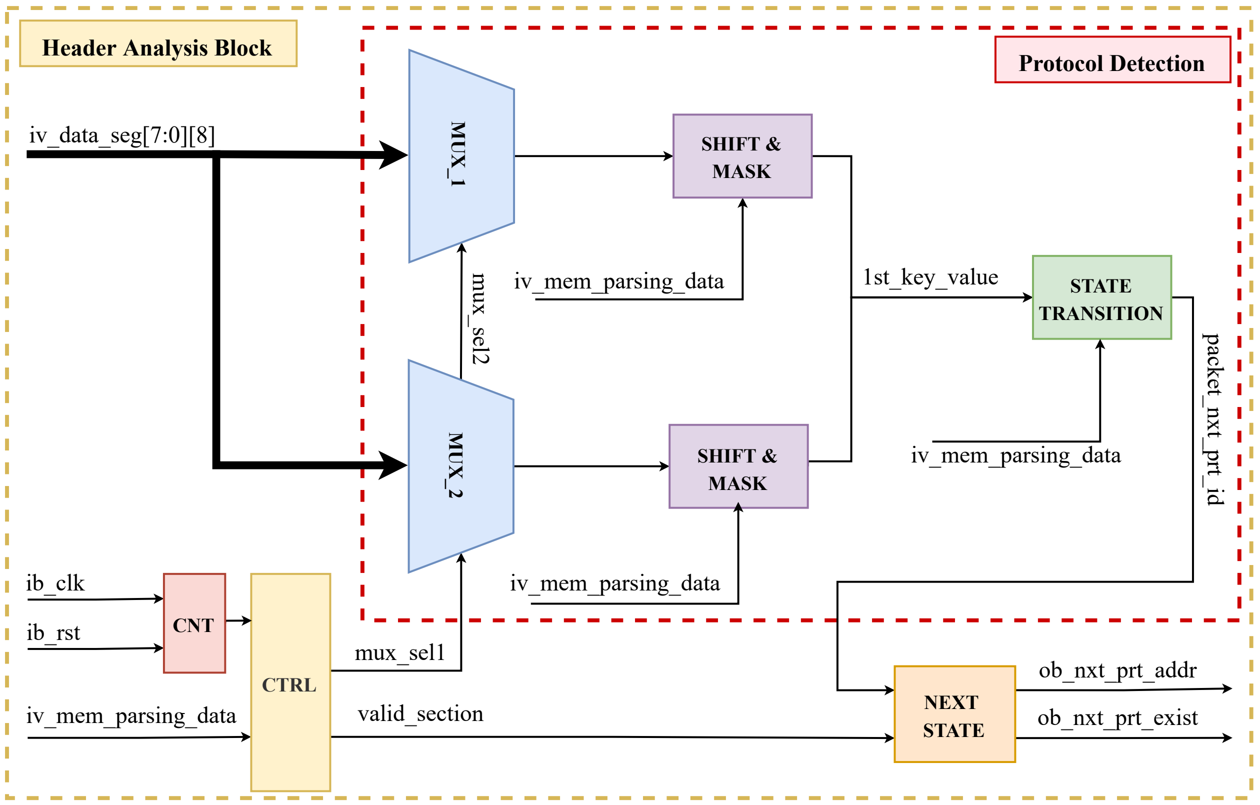


Figure 5.4 Architecture of Header Analysis (HA) block.

in the State Transition (ST) block. These Keys are included in the protocol's MD, and a protocol transition is initiated if the extracted Key matches any of the Key values. In the event of a transition, the next protocol ID is sent to the Next State (NS) block to update the memory read address if it is found in a valid section.

The section denoted by a red dashed line extracts a 16-bit Key from the incoming data, shifts, and masks it, and then employs the ST block to detect the new protocol. This block can be duplicated to detect more protocols within an HA block with some modifications to the Control unit. It is possible that more than one Key detection is required in an incoming packet segment, for example, if the protocol headers are small and their Keys are positioned in the same receiving packet segment. In that case, two consecutive Keys located in the same packet segment are received in one clock cycle, and multiple protocol detections are required in one clock.

5.5 Results

The proposed design was tested with the generated MDs for the Simple Parser to verify the design and to check its versatility. We used Vivado 2021.1 to synthesize the RTL code for a Xilinx Virtex-7 FPGA (XC7VX690T FFG1761-3). In this work, we proposed a generic parser with a 64-bit input bus size. Our implementation offers a 11Gb/s throughput, and uses 1,447 LUTs+FFs (see Table I). This implementation is programmable through the memory content.

Table 5.1 summarizes the performance of our design and its resource consumption and compares our results with the literature. These works use various approaches and architectures with varying levels of complexity, reconfigurability, and programmability. The latency reported in [58, 59, 66, 70] depends on the number of pipeline stages in the parser . [62] and the present work display latencies that depend on the parse graph's paths. The reported latency for [62] and this work is based on two parse graphs, Ethernet-IPv6-TCP and Ethernet-IPv4-TCP respectively. Moreover, in Table 5.1, programmability is defined as the ability to change the input parse graphs for the same implemented hardware architecture. It is important to note that programmability is not the same as reconfigurability, which refers to the ability to regenerate hardware for a new input parser code using hardware synthesis.

We considered using a small input bus to improve our design's effective throughput while maintaining area efficiency. According to [71], when bursts of the smallest possible packets are processed, the effective throughput of architectures that can only process one packet per clock cycle reduces to a small fraction of the reported raw throughput.

The old-fashioned TCAM-based design [66] has a remarkably complex logic and high power consumption, while in our architecture, the ratio of the Logic complexity (LUTs+FFs) over Throughput ratio is significantly improved to 130.36 (compared to 380.19 for the TCAM-based design).

In comparison to the HLS approach of converting pipeline descriptions via C++ classes to RTL directly from P4, we obtain a lower Logic over Throughput ratio.

The work in [70] and [59] reports a better Logic complexity over Throughput ratio. However, these designs suffers from an excessive bus width reducing the effective throughput below their reported raw throughput (130 vs 160).

The maximum throughput is reported in [71], achieved while maintaining a very low Logic complexity over Throughput ratio. This high effective throughput was achieved thanks to a bus segmentation method and running multiple parsers in parallel. However, adding more protocols to the parsing graph would increase complexity.

Finally, unlike previous works that were not programmable, [62] offers a custom programmable processor. Our design yields 4.9× lower Logic complexity over Throughput ratio while offering a much lower latency.

5.6 Conclusions

This paper proposes a novel area-efficient parser design that can parse any P4 parsing input graph with one transition per clock cycle through the content of its embedded memory. The hardware part of this design needs to be synthesized only once, and the proposed framework can change the functionality as specified in a P4 code by changing the content of a memory. Each parsing channel can handle 11 Gbps of incoming packets and can be assigned to a parsing block. Consequently, multiple parsers can operate in parallel to increase throughput if multiple channels must be served.

5.7 Acknowledgements

The authors thank Kaloom, Intel, Noviflow, Prompt Quebec, and the Natural Sciences and Engineering Research Council of Canada for their financial support and Jeferson Santiago da Silva, Thibaut Stimpfling, and Thomas Luinaud for their insightful comments.

Table 5.1 Results comparison.

Work	FPGA	Architecture	Performance				Resources			Logic/ Throughput	Programm -ability
			Data Bus [bits]	Frequency [MHz]	Throughput [Gb/s]	Latency [Cycle]	LUTs	FFs	Logic (LUTs+FFs)		
Simple Parser											
[66]	Virtex-7	TCAM	256	184.1	47	N/A	14,906	2,963	17,869	380.19	No
[58]	Virtex-7	Modular	320	312.5	100	6	5,888	10,488	16,336	163.36	No
[70]	Virtex-7	Templated	1072	334	358	5	10,495	6,886	17,381	48.55	No
[71]	UltraScale+	Modular	4096	244.1	1000	7	N/A	N/A	66,000	66	No
[62]	Alveo-U200	Instruction	64	125	8	22–23	3,538	1,578	5,116	639.5	Yes
Proposed	Virtex-7	Memory	64	173.4	11.1	10–7	312	1,135	1,447	130.36	Yes

CHAPTER 6 ARTICLE 3: PRISMPARSER: A FRAMEWORK FOR IMPLEMENTING EFFICIENT P4-PROGRAMMABLE PACKET PARSERS ON FPGA

Parisa Mashreghi Moghadam, Tarek Ould-Bachir, Yvon Savaria

Published in: MDPI, Future Internet

Publication date: August 27, 2024

Preface: This chapter presents PrismParser, a hardware–software framework that maps P4 parser descriptions to scalable FPGA datapaths across three architectural variants. The base architecture offers a compact and resource-efficient parser that evaluates one protocol state per cycle, delivering high performance with minimal hardware requirements. Building on this foundation, the overlay architecture relocates protocol behavior to memory-driven control tables generated from P4-compiled JSON, enabling reconfiguration without RTL modification or resynthesis. In parallel, the pipeline architecture extends the base design by evaluating parser states across multiple stages, allowing for several protocol checks to be in flight and achieving significantly higher throughput for wider datapaths. Together, these variants provide a flexible progression from lightweight static parsing to high-performance, runtime-programmable P4 processing. Experiments on Xilinx UltraScale+ hardware demonstrate performance ranging from 15.2 Gb/s for the base architecture to 289.1 Gb/s for the pipelined variant, confirming the scalability and efficiency of the framework. This work was peer-reviewed and published in the proceedings of the *MDPI - Future Internet*.
Contributions: This research originated as part of my doctoral work at Polytechnique Montréal and was developed in collaboration with my co-authors. I contributed to the conception of the problem, the architectural design, testing and deployment of the system, literature review, analysis of the results, and preparation of the manuscript. My co-authors provided guidance, feedback, and supervision throughout the research and revision process.
Full Citation: Parisa Mashreghi-Moghadam, Tarek Ould-Bachir, Yvon Savaria. 2024. *"PrismParser: A Framework for Implementing Efficient P4-Programmable Packet Parsers on FPGA"* Future Internet 16, no. 9: 307.
DOI: 10.3390/fi16090307
Copyright: © 2024 by the authors.

6.1 Abstract

The increasing complexity of modern networks and their evolving needs demand flexible, high-performance packet processing solutions. The P4 language excels in specifying packet processing in software-defined networks (SDNs). Field-programmable gate arrays (FPGAs) are ideal for P4-based packet parsers due to their reconfigurability and ability to handle data transmitted at high speed. This paper introduces three FPGA-based P4-programmable packet parsing architectural designs that translate P4 specifications into adaptable hardware

implementations called base, overlay, and pipeline, each optimized for different packet parsing performance.

As modern network infrastructures evolve, the need for multi-tenant environments becomes increasingly critical. Multi-tenancy allows multiple independent users or organizations to share the same physical network resources while maintaining isolation and customized configurations. The rise of 5G and cloud computing has accelerated the demand for network slicing and virtualization technologies, enabling efficient resource allocation and management for multiple tenants. By leveraging P4-programmable packet parsers on FPGAs, our framework addresses these challenges by providing flexible and scalable solutions for multi-tenant network environments.

The base parser offers a simple design for essential packet parsing, using minimal resources for high-speed processing. The overlay parser extends the base design for parallel processing, supporting various bus sizes and throughputs. The pipeline parser boosts throughput by segmenting parsing into multiple stages. The efficiency of the proposed approaches is evaluated through detailed resource consumption metrics measured on an Alveo U280 board, demonstrating throughputs of 15.2 Gb/s for the base design, 15.2 Gb/s to 64.42 Gb/s for the overlay design, and up to 282 Gb/s for the pipelined design. These results demonstrate a range of high performances across varying throughput requirements. The proposed approach utilizes a system that ensures low latency and high throughput that yields streaming packet parsers directly from P4 programs, supporting parsing graphs with up to seven transitioning nodes and four connections between nodes. The functionality of the parsers was tested on enterprise networks, a firewall, and a 5G Access Gateway Function graph.

6.2 Introduction

Software-defined networks (SDNs) have introduced the potential for dynamically programmable network infrastructure by decoupling control and data planes [69]. This architectural innovation allows for the agile deployment of new protocols through a centralized controller, streamlining the introduction of new forwarding rules to the data plane without altering the foundational hardware. Building on these foundational principles, the P4 language was introduced, enabling detailed specification of packet processing behaviors in the data plane [13]. The underlying concepts were embodied in the protocol-independent switch architecture (PISA), which supports dynamic packet processing and allows network devices to evolve with network requirements [80].

The rapid evolution of modern network infrastructures highlights the increasing importance of multi-tenant environments. Multi-tenancy allows multiple independent users or organizations to share the same physical network resources while maintaining isolation and customized configurations [22]. The rise of 5G and cloud computing has accelerated the demand for network slicing and virtualization technologies, enabling efficient resource allocation and management for multiple tenants [21, 81].

Field-programmable gate arrays (FPGAs) are crucial to networking evolution, especially in dynamic SDNs. They offer essential programmability, balancing fine-grained control with robust, power-efficient performance, making them ideal for SDN deployments [36]. As the industry recognized the potential of P4's flexibility, a need for hardware architectures to support this programmability emerged. PISA, inherently adaptable, pairs naturally with FPGAs, facilitating rapid prototyping and deployment of custom networking protocols [80].

PISA comprises three main components: a packet parser, a deparser, and match-action tables. The parser extracts relevant fields from incoming packets, the deparser reconstructs processed packets for transmission, and the extracted fields are matched against table entries. Given the dynamic nature of network traffic, FPGAs offer outstanding flexibility, enabling real-time adjustments [82]. Whether software- or ASIC-based, traditional packet parsers face significant limitations in processing capabilities, latency, and flexibility [72]. While software-based parsers offer programmability, they suffer from high latency and limited processing capacity. ASIC-based solutions, though faster, lack flexibility and have long development cycles [19]. In contrast, FPGA-based solutions present a favorable trade-off, providing high data rate processing and flexibility.

Early FPGA-based parsers, such as those by [19, 72], faced significant latency issues. Subsequent ternary content addressable memory (TCAM)-based methods by [66, 80] reduced latency but compromised flexibility and scalability. Innovations like those from [74, 77–79] introduced custom processors for reconfigurable packet parsing, but still grappled with latency. Recent advancements, such as those by [59, 61, 70], emphasized reconfigurability and software programmability, addressing some limitations but highlighting challenges in simultaneous protocol detection and scalability. Separating parsing logic from protocol-specific details and leveraging FPGA reconfigurability allows for adaptive, high-performance packet parsing, ensuring responsiveness to evolving network protocols.

This paper proposes a novel dynamic and configurable low-latency packet parser architecture based on FPGA technology, addressing the challenges of flexibility, scalability, and high performance in modern networks [16]. Our design leverages parallel extraction and matching techniques to minimize latency and maximize throughput, supporting real-time reconfiguration through SDN control planes [83]. The contributions of this paper are as follows:

- A scalable parser framework that efficiently extracts parsing information from the P4 definition and adjusts the hardware implementation on FPGA;
- A mechanism within the parser that allows for the integration and support of emerging protocols, offering a forward-compatible system;
- An optimized architecture that reduces the time complexity of protocol identification, enhancing the speed and efficiency of the parser;
- An intuitive software interface for users to easily define and modify protocol-specific information, enhancing the usability of the parser;
- Design considerations enabling third-party users to extend the parser's capabilities by setting design parameters and bus size to achieve their goals.

The remainder of this paper is organized as follows: Section 6.3 outlines packet parsing strategies and reviews previous work. Section 6.4 highlights the methodology employed in our study. Section 6.5 presents the experimental results, and Section 6.6 concludes this work.

6.3 Protocol-Agnostic Packet Parsing: Background

6.3.1 Introduction to Packet Parsing in P4

Data are sent in packets encapsulated using various protocol layers in digital communication. When the data is processed, the layers must be appropriately decoded, as parsing involves methodically examining incoming packets to identify and separate their headers. Given the layered nature of network protocols, headers are nested within each other, each signifying a specific protocol layer. For example, an Ethernet header can contain an IP header, which might enclose a TCP header.

In P4, a packet parser is implemented using a state machine to navigate through the packet's hierarchical structure. This state machine comprises states specifically configured to recognize and extract different headers based on predefined keys and their exact positions within the packet. The configuration of these keys, their respective locations, and the dimensions of the headers are all defined within the P4 code. Upon encountering a key at a specific location, the parser determines which header comes next, extracts it, and transitions to the appropriate state. This systematic process ensures the orderly extraction of headers, aligning with the structured layers of network protocols. This parsing procedure is often visualized using a parser graph, typically represented as a directed acyclic graph (DAG). Figure 6.1 showcases a classic parse graph for enterprise networks, similar to what is explained in [84]. In this graphical representation, nodes show individual protocols or headers, and edges outline the potential transitions between them.

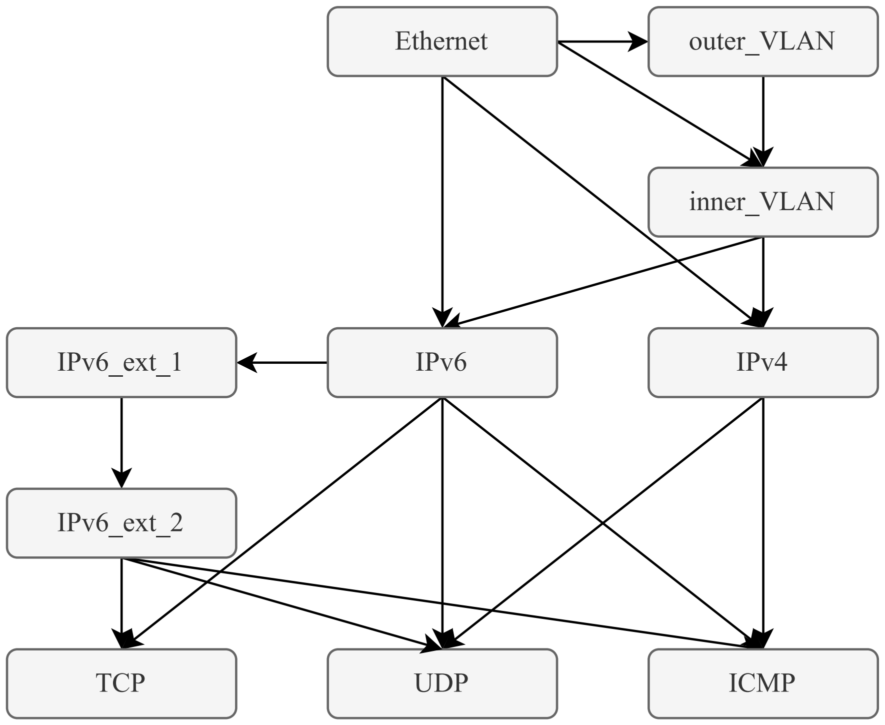


Figure 6.1 Example of an enterprise parsing graph.

6.3.2 Programmable Packet Parsing Overview

A programmable parser flexibly analyzes and dissects various packet formats and structures. Unlike fixed traditional parsers, it can be reconfigured to detect new or custom protocols, offering versatility in evolving networking environments. Combined with P4, it specifies which headers to recognize and in what sequence, maximizing its potential. FPGAs are ideal for implementing these parsers in hardware, effectively embodying the parsing logic described by P4. This combination of FPGAs and P4 offers the speed and low latency of dedicated hardware with software adaptability. Diverse methodologies and innovations have contributed to the evolution of FPGA-based packet parsers. Early programmable parsers such as [19, 72] were associated with considerable latencies. Subsequent TCAM-based methodologies [66, 80] achieved reduced latency but at the expense of flexibility, scalability, and the introduction of complex logic. As research advanced, custom processors for packet parsing emerged, as reported in [62, 74, 77–79]. These designs, while reconfigurable, grappled with latency issues, especially when integrating new protocols.

The automation of P4-programmable packet parsers, explored in [16,75], marked a significant shift. A notable contribution from Da Silva et al. [58] utilized Vivado HLS for P4-to-RTL conversion, achieving impressive throughputs but necessitated an entire translation cycle for each P4 code modification. The later innovations, such as the templated packet parser architecture in [59, 70], emphasized reconfigurability. However, they also highlighted the challenges of comprehensive re-synthesis with P4 code alterations, suggesting the potential advantages of software programmable solutions.

The merits of a software programmable design are explored in the work presented by [61]. This approach, where alterations in the P4 parser code necessitate only changes to the embedded memory content without mandating modifications to the hardware design, is particularly noteworthy. However, the design proposed has its limitations. Specifically, it supports detecting only one protocol field per clock cycle, and its logic would need substantial modification to detect multiple protocols simultaneously. Such changes introduce complexity and pose potential timing challenges.

Furthermore, this limitation interferes with the design's scalability, mainly when larger bus sizes are essential for higher throughput demands. To truly harness the potential of such a design, it is imperative to evolve toward a more generic parser. The optimal parser would process all protocols from the parsing graph in each data bus cycle. This would allow simultaneous detection of all protocols delineated in a parsing graph, accommodating larger bus sizes and substantially improving throughput potential.

It is imperative to separate the parsing logic from the specific header details to achieve a versatile and efficient hardware architecture for packet parsing. The consistent set of operations, or the parsing logic, remains unchanged irrespective of the packet type. On the other hand, the critical parsing information, which includes specifics like header sizes and key locations, varies with each protocol. By storing this protocol-specific information, hardware can dynamically adapt to different packet headers, ensuring flexibility and optimized performance. Such an architecture, when implemented on an FPGA, can leverage the inherent reconfigurability of the platform. As network protocols evolve or new ones emerge, the FPGA-based parser can smoothly adapt, ensuring that the network remains agile and responsive to changing requirements. Furthermore, this adaptive approach provides a solid foundation for future enhancements.

6.4 Proposed P4-Programmable Parser

The parser framework presented in this paper offers a robust solution for translating P4 parser specifications into a generic, programmable hardware platform that combines software and hardware components. The software component generates header specifications from P4 configurations, which are then used by the hardware. The hardware architecture is designed to be generic and programmable, allowing easy reconfiguration when the input parsing graph changes.

6.4.1 Software Overview

Proposed Workflow

The initial step in extracting configurations and control data from P4 code is compiling the code using the p4c compiler. The p4c compiler translates P4 code into a JSON file, which is then processed to extract header and parser information. After that, the memory content is generated, which can be used to program the hardware via the SPI protocol. This process is described in Figure 7.2.

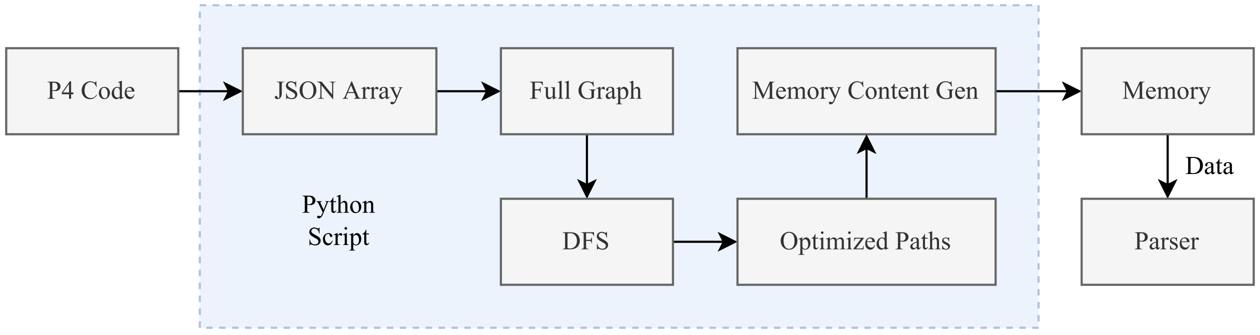


Figure 6.2 Proposed compilation workflow for generating control and configuration for the parser.

P4 Compiled JSON File

The most important fields within the JSON file are *header types*, *headers*, and *parsers*. The *header types* array defines the types of each header, where each object includes field names, lengths, and an ID number, referred to as the protocol ID in this paper. The headers array maps each header type to its respective header.

The most crucial part of the P4 compiled JSON file is the parser array. Typically, each object in this array represents a parsing state based on the current parsed header, its transition key (the fields of interest for parsing in the header), and transitions (which detail the next parsing state based on the value of each transition key). Therefore, parsing can be conceptualized as a directed acyclic graph (DAG), where each node symbolizes a header, and the edges denote transitions between headers based on the transition key and its value. Listing 6.1 shows a simplified P4 compiled JSON file, which shows the header types, headers, and parsers for the Ethernet header:

Listing 6.1 Header types, headers, and parsers for the ethernet header.

```
{
  "header_types": [{
    "name": "ethernet_t",
    "id": 1,
    "fields": [
      ["dstAddr", 48],
      ["srcAddr", 48],
      ["etherType", 16]
    ]
  }],
  "headers": [{
    "name": "ethernet",
    "header_type": "ethernet_t"
  }],
  "parsers": [{
    "name": "parse_ethernet",
    "transition_key": [{
      "type": "field",
      "value": ["etherType"]
    }],
    "transitions": [
      {"value": "0x0800", "next_state": "parse_ipv4"},
      {"value": "0x8100", "next_state": "parse_inner_vlan"},
      {"value": "0x9100", "next_state": "parse_outer_vlan"},
      {"value": "0x86DD", "next_state": "parse_ipv6"}
    ]
  }]
}
```

Hardware Configuration Generation

JSON processing and DAG generation are implemented using Python. The initial step involves extracting the required information from the JSON file, including header size, transition key lengths, and their locations relative to the start of the header, protocol ID, and transitions. This is achieved by matching the header types with the header arrays and looking for parser transition key field information within them. Based on this information, the DAG is generated. Subsequently, all simple paths from the root to the leaves of the DAG are found using a Depth-First Search (DFS) algorithm. Some paths to the leaf nodes are discarded since the last headers do not need to be considered parsing states. Now, for these paths and the extracted information from the JSON file, the following hardware configurations are generated per each path:

- **Protocol bitmap**: One-hot encoding representation of the *protocol_ID* and bit number of *protocol_ID − 1*. Bitmap length is equal to the number of parsable headers.
- **Protocol mask**: Determined by the transition key length and position within a 16-bit long chunk per each header in the path.
- **Match values and next state**: Determined by the extracted transitions per each header in the path.
- **Multiplexer select lines and enable signals**: Determined by the bus size, number of inputs per each mux, and transition key location.
- **Next state bitmap**: The OR value of the header protocol and next header bitmaps.

For example, for the Ethernet header, these contents are generated:

- **Protocol bitmap**: *"00000001"*; consider that there are seven headers and *protocol_ID* is 1.
- **Protocol mask**: *0x"FFFF"*, since the transition key *etherType* is 16 bits long.
- **Match values and next state**: Match value vector is *0x"08008100910086DD"* which is the concatenated transitions values. The next state is *0x"4325"*, which is *protocol_ID* for each match, meaning that when *etherType = 0x"9100"*, the next parser state will be *protocol_ID* 2, which is *parse_outer_vlan* in our example.

- **Multiplexer select lines and enable signals**:

  Section 6.4.2 explains that the design's input bus is 64 bits. The input data is split into four chunks of 16 bits, each connected to the input of a multiplexer. Since the key to be extracted is two bytes or less, one of these input chunks in a specific clock number contains the value that needs to be selected. The extracted key will be masked and matched if it is less than two bytes. Moreover, based on previously parsed values, the same protocol key may occur in different clock cycles. All possible scenarios for the key's occurrence are calculated to account for this. A bit vector is then generated to select the correct protocol at the appropriate clock cycle.

  Listing 6.2, an example of such a vector, and is provided below. To parse the correct key, the design must identify which protocols can occur in every clock cycle. The Enable bits indicate the protocols that can happen in a certain clock cycle. Each bit in the Enable vector corresponds to a specific transitioning node, representing a particular protocol. A one in the location number of the *protocol_ID − 1* signifies that this specific protocol can occur in that clock cycle. The select vector determines the selection of one of the four inputs of a multiplexer for an activated protocol. With two bits to select the binary range of *"00"* to *"11"*, one of the four inputs of the multiplexer is chosen. Consequently, each bit in the enable vector has two corresponding bits in the select vector.

  Listing 6.2 Example of select and enable arrays for protocol selection at a specific clock cycle.

```
select_enable_array_clk_3=
 ------SELECT-----------ENABLE---
 "00000000000000" & "0000100"&
 "00000000100000" & "0000110"&
 "00000011000000" & "0001000"&
 "00001000000000" & "0010000"
```

- **Next state bitmap**: The initial node is Ethernet, corresponding to the protocol bitmap *"0000000001"*, as it is the first protocol to be parsed. From this state, based on the discussion in the multiplexer select lines and enable signals section, protocols 2, 2 and 3, 4, or 5 can be detected when receiving 64-bit input data. Therefore, in the next clock cycle, the generated protocol bitmap needs to be checked against *"000001101"*, *"0000010101"*, *"0000001111"*, and *"0000010111"*, which indicate the possible paths in the graph up to number-of-clk-cycles * 64. The following are the possible bitmaps for the fourth clock cycle, which should be matched against the input bitmap from the previous cycle:

Listing 6.3 Bitmap array comparison for protocol transition at a specific clock cycle.

```
bitmap_array_clk_4=
"000001101"&
"0000010101"&
"0000001111"&
"0000010111"
```

After the process, the configurations and control data vectors will match those illustrated in Listing 6.4 or Listing 6.5.

Listing 6.4 Generated configuration.

```
configuration =
        mask_prt_1 & mask_prt_2 & ... & mask_prt_n &
        all_key_values_prt_1 & all_key_values_prt_2 & ... &
        all_key_values_prt_n & all_next_prt_ids_prt_1 &
        all_next_prt_ids_prt_2 & ... & all_next_prt_ids_prt_n
```

Listing 6.5 Generated control.

```
control =
        select_enable_array_clk_1 & select_enable_array_clk_2 & ... &
        select_enable_array_clk_m & bitmap_array_clk_1 &
        bitmap_array_clk_2 & ... & bitmap_array_clk_m
```

Memory Content Generation and Programming

After extracting all hardware configurations, alignment is performed based on the clock number. This is viable because the bus size is already known; therefore, the current position of the packet is determined. The path and header information are also available, allowing for the implementation of a counter in the hardware to address memory configurations based on the counter value. The SPI protocol is used to program these memory contents to the hardware. The Python program encodes these memory contents as binary vectors that can be transferred using the SPI protocol. Algorithm 1 shows the whole process:

Algorithm 1 Configuration and control extraction algorithm.

```
J: The compiled P4 JSON file, B: The desired bus size, S: The desired multiplexers size
MemCode: Memory configuration code **Initialization:**
$G \leftarrow$ Create an empty DAG
$ProtocolBitmap \leftarrow$ Create an empty array
$ProtocolMask \leftarrow$ Create an empty array
$Matchvalues \leftarrow$ Create an empty array
$NextState \leftarrow$ Create an empty array
$MuxSelect \leftarrow$ Create an empty array
$EnableSignals \leftarrow$ Create an empty array
$NextStateBitmap \leftarrow$ Create an empty array
$MemCode \leftarrow$ Create an empty array
$HeaderInfo \leftarrow$ parseJSON(J)
parseObj in J.get('parsers') $header \leftarrow$ parseObj.name
$G$.addNode(header)
$G$.addData(HeaderInfo.get(header))
transition in parseObj.get('transitions') $G$.addEdge(G.currentNode, transition.nextState, transition.value) $paths \leftarrow$ DFS($G$)
$reducedPaths \leftarrow$ removeUnusedPaths($paths$)
path in reducedPaths header in path $ProtocolBitmap$.append(getProtocolBitmap(header, HeaderInfo.get(header)))
$ProtocolMask$.append(getProtocolMask(header, HeaderInfo.get(header)))
$Matchvalues$.append(getMatchvalues(header, HeaderInfo.get(header)))
$NextState$.append(getNextState(header, HeaderInfo.get(header)))
$MuxSelect$.append(getMultiplexerSelect(header, HeaderInfo.get(header), B, S))
$EnableSignals$.append(getEnableSignals(header, HeaderInfo.get(header), B, S))
$NextStateBitmap$.append(getNextStateBitmap(header, HeaderInfo.get(header)))
$MemCode \leftarrow$ createMemContent(ProtocolBitmap, ProtocolMask, Matchvalues, NextState, MuxSelect, EnableSignals, NextStateBitmap, B)
$MemCode$
```

A key feature of the software component is its ability to support dynamic adaptability. It can handle real-time updates to P4 configurations, enabling the hardware to adapt to new parsing requirements on the fly. This adaptability is achieved through efficient data transfer mechanisms and a flexible hardware design.

6.4.2 Hardware Design Overview

The core of the packet parser presented in this study is conceptualized as a DAG, where each node symbolizes a protocol, and the edges denote transitions between protocols. It functions as an abstract state machine (ASM), systematically evaluating state transitions at each node within the parser. The states that link the initial state to the terminal state in the ASM delineate the set of protocols supported by the P4 parsing graph.

This design introduces a *base block* capable of reuse to accommodate larger bus sizes and higher data throughputs. The base design architecture is a streaming packet parser that operates without needing packet storage, explicitly tailored for 64-bit bus sizes. This bus size is strategically chosen to comply with byte-aligned network protocols and ensure reliable header key alignment, thereby preventing any header key from being split across two incoming data chunks. This careful balance in bus size selection avoids the complications of an overly intricate design while preventing excessive demands on software computations and hardware resources. This approach results in a highly flexible and scalable hardware design that can efficiently handle current demands and adapt to future network protocol developments.

Base Block Architecture and Microarchitecture

The base block, shown in Figure 6.3, is designed to support a 64-bit bus size with a maximum possible clock frequency to handle a data throughput of more than 10 Gbps efficiently. This throughput is commonly used in high-speed networks and ensures the design can meet industry-standard performance requirements while consuming a minimal area footprint. Table 6.1 compares the proposed design with previous programmable packet parsers using the same bus size.

The hardware design follows a straightforward principle: extract key information from a specific location within a header, match it against predefined key values, transition to the next state upon a match, and forward the information as a Packet Header Vector (PHV) for subsequent processing stages. To simplify the hardware design, the critical pieces of information are generated via software data structures essential for initializing the processing objects. The Software Overview section provides a detailed discussion of how this information is generated from P4 code. These configurations include protocol keys, masks, next protocol IDs, multiplexer selections, and hierarchical sets of bitmaps, all crucial for determining the processing path.

The base parser block also includes two critical sub-blocks designed to efficiently determine the next protocol header in the graph hierarchy with minimal dependencies and a low area footprint. When the protocol investigator unit identifies the next protocol to be parsed, it

Table 6.1 Results comparison for base block.

Work	**FPGA**	**Performance**				**Resources**			**Programmability**	
		Bus Size [bits]	**Freq. [MHz]**	**TP [Gb/s]**	**Max Latency [cycles]**	**LUTs**	**FFs**	**Total Logic**	**Method**	**Time**
[62]	Alveo U200	64	125	8	23	3538	1578	5116	Instr.	Hours
[61]	Virtex-7	64	173.4	11.1	10	312	1135	1447	Memory	Min.
Proposed	Alveo U280	64	237.69	15.2	9	1028	1873	2901	Register	Sec.

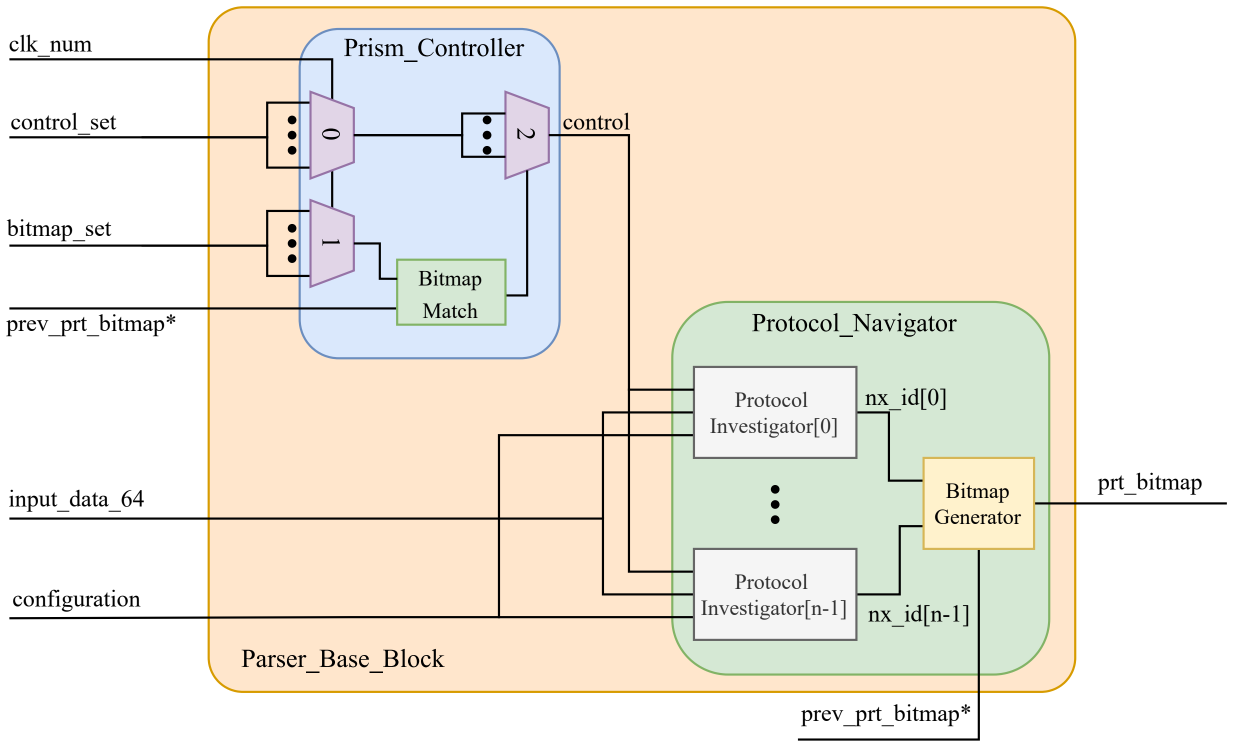


Figure 6.3 Base Block.

requires only bitmap updates to propagate to the block in the following cycle, enhancing the parser's efficiency and reducing the time and resources needed for protocol transitions.

The two main components of the base parser block are the *Prism Controller* and the *Protocol Navigator*. The Prism Controller manages control signals by calculating potential transition sets and using multiplexers to select protocol bitmaps and control signals based on clock cycles and comparison results, ensuring accurate control signal generation. As the guiding force, the Protocol Navigator directs the parser through both parsed and impending protocols. This block contains several Protocol Investigators sub-blocks, each tailored to specific protocols and responsible for retrieving header details from software-generated header configurations. The number of Protocol Investigator units corresponds to the transitioning nodes in the parsing graph, allowing detection of all possible protocols within the same clock cycle. As depicted in Figure 6.4, each Protocol Investigator is designed to be generic and extendable, adapting to diverse network protocols. It retrieves essential information, such as masks, keys, and next protocol IDs, under the control of the Prism Controller. It determines whether to activate its functionalities during each cycle and specifies which 2-byte input data segment to process. Using a protocol mask in a bitwise AND operation allows the extraction of relevant key bits from the input data. At the same time, a parallel search identifies a matching key through the Match Detector unit, which executes an XOR operation followed by a bitwise NOR operation to generate the match found signal. Additionally, the Bitmap Generator decodes the following protocol ID into a Protocol Bitmap, a vector with bits corresponding to the graph nodes. This newly encoded bitmap is combined with the previous cycle's protocol bitmap and outputs from other decoders using a bitwise OR operation. This process ensures a comprehensive representation of parsed protocols, facilitating efficient select operations in processing control stages. The protocol bitmap is an n-bit vector, where the position i indicates whether the $\mathit{protocol\ ID} - \mathit{1} \ = i$ has been detected.

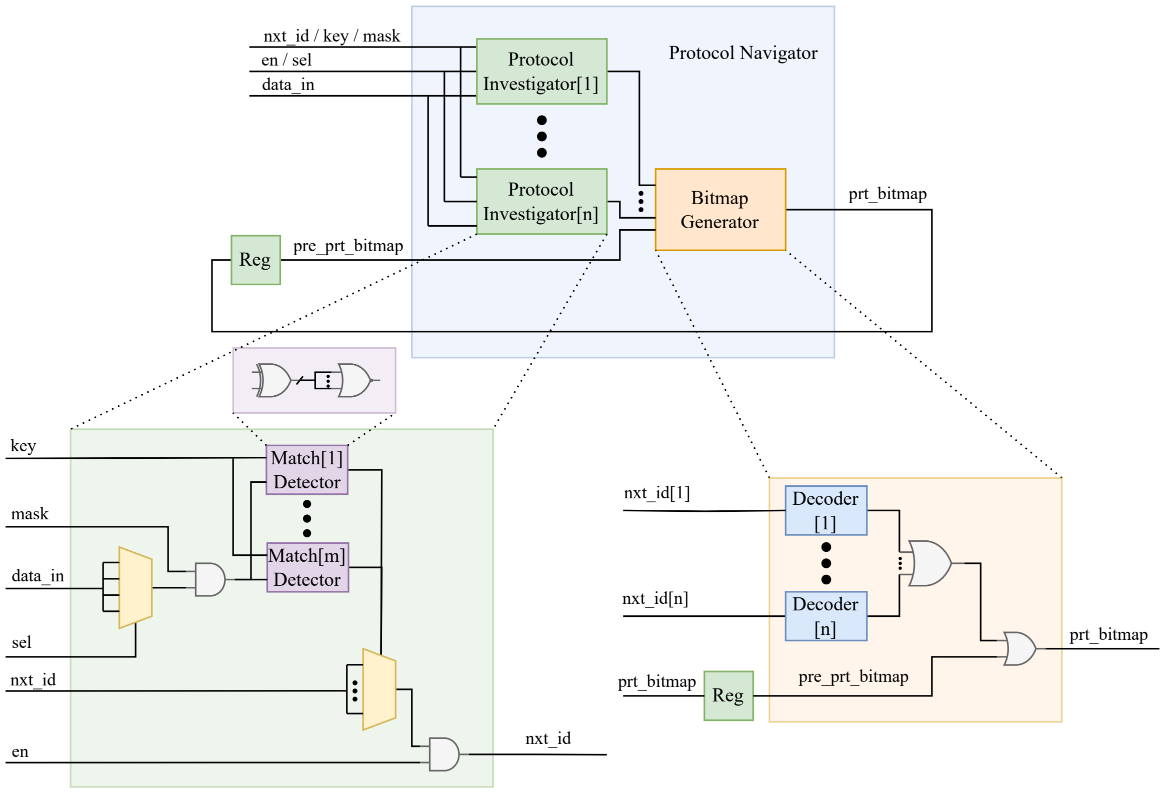


Figure 6.4 Protocol Navigator: Protocol Investigator with its Match Detector sub-block and Bitmap Generator.

Despite the design's support for a 15 Gbps data rate due to its 64-bit bus size and clock frequency limitations, it faces constraints such as limited bus width, required advanced fabrication for higher clock frequencies, latency from multiplexers and control logic, and sequential processing bottlenecks. To improve and support higher throughputs like 40 or 100 Gbps, the design could benefit from increasing bus width, implementing deeper pipeline stages, optimizing multiplexers, and utilizing parallel processing units.

6.4.3 Limitations and Evolution of the Base Block

Two primary solutions can be considered to enhance our parser's throughput: increasing its operating frequency or expanding the bus size. Alternatively, combining both approaches may also be employed to achieve optimal performance. However, the base design of our packet parser faces inherent challenges that constrain its operational efficiency. The primary issue is achieving higher frequencies, which is impossible with the base design because of its long critical path and single-cycle architecture.

The secondary issue lies in its limited throughput capabilities, primarily due to its support for only small bus sizes. Enhancing the parser through larger bus sizes involves maintaining the base design while scaling up the number of multiplexer inputs. While preserving the core architecture, this approach increases resource consumption and a longer critical path. Additionally, it necessitates more complex memory content generation and requires unrolling the Prism Controller to manage the expanded data flow effectively. This limitation impacts the parser's ability to handle high-volume network traffic effectively, necessitating enhancements to accommodate larger data loads. To mitigate these issues, a range of improvements are proposed to augment throughput, focusing on either refining the operational frequency or expanding the bus size capabilities:

- **Enhancement via Higher Frequency**:

  Increasing the operating frequency to enhance the packet parser's performance is inherently limited due to the current design's optimization boundaries. Achieving higher frequency involves extensive optimization of the control logic and minimization of the latency of multiplexers and other critical components. Given that the current frequency represents the highest possible frequency without compromising system functionality, further enhancements in performance must be sought through other means. Specifically, increasing the bus size offers a viable alternative to achieve higher throughput. This approach involves scaling up the number of multiplexer inputs and expanding the overall architecture to handle larger data loads effectively.

- **Enhancement via Larger Bus Sizes**:

  Two different approaches can enhance the parser through larger bus sizes. The first involves increasing the bus size, which results in larger multiplexers in the Protocol Investigator block. This makes the logic more complex, worsens the critical path, and requires the software-generated configuration and control data to consider the entire path, necessitating fundamental changes in the software script.

The second approach replicates the base block with modifications to enable parallel logic, extracting all headers in the input header chunk.

6.4.4 Overlay Optimization

To address the limitations of the base parser block and enhance throughput and scalability, the initial step was to double the bus size rather than the frequency, presenting fewer constraints and greater feasibility. Building on this foundation, the design was further expanded into an extendable block that can be replicated to support larger bus sizes and frequencies. This section explains the breakdown of the problem and the development of an adjustable design, controllable through generics, to optimize performance.

Doubling the throughput can be achieved by increasing either the frequency or the bus size. Since increasing the frequency is not feasible due to inherent limitations, the focus shifts to doubling the bus size while reusing the base block. The initial step involves employing two base blocks. By increasing the bus size from 64 to 128 bits, the first base block processes the first 64 bits and the second 64 bits by the second base block. However, the second base block cannot complete its calculations without the results from the first base block. Waiting for one clock cycle to obtain the results for the second base block would require processing another packet. If the packet is not stored, it will be dropped; if stored, it necessitates additional logic, memory, and more clock cycles to parse the header.

The solution, illustrated in Figure 6.5, is to modify the second base block to process all possible scenarios from the first block in parallel. This modified block, along with the original base block, runs concurrently. Before the result from the first block is available, the modified second block begins processing the data. The result from the first block drives a multiplexer that selects the appropriate result from the second block. Similarly, the modified base block can be replicated $x - 1$ times to support processing a $(x) \times$ (64-bit) bus size. The parameter x can be defined before FPGA implementation, thus creating an overlay design, as shown in Figure 6.6.

The operation of the modified base block is designed to process data in parallel across multiple blocks without waiting for the results from previous blocks, thereby improving throughput and reducing latency. Each new modified block increments the clock by one, simulating that it is working in the next cycle. However, in reality, all blocks work in parallel, and only the output is selected based on the result from the previous block. Multiplexer *#0*, which handles the control sets and is controlled by the clock number, remains the same across all blocks to ensure that each block receives the correct control signals based on the current clock cycle. Multiplexer *#1*, also controlled by the clock, selects the proper set of controls

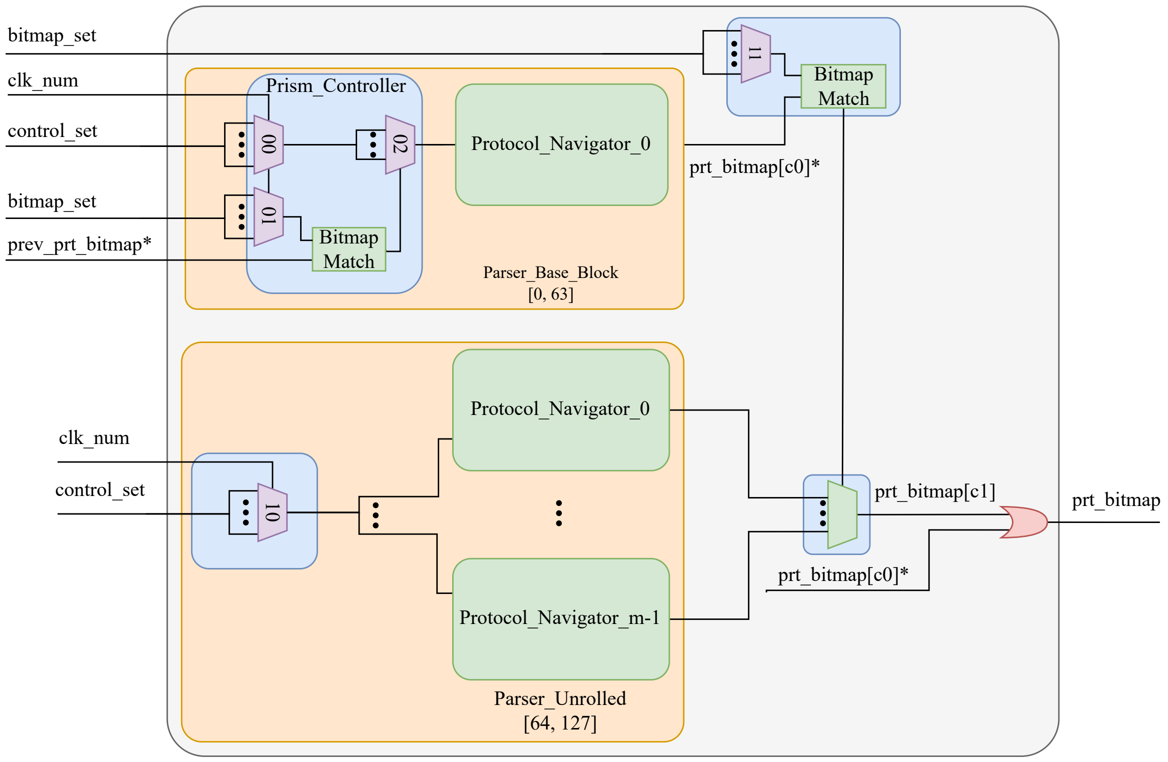


Figure 6.5 Overlay Block: In the multiplexer ID, the first digit indicates the parser block it belongs to, and the second digit indicates the multiplexer number, referred to in the text with a #.

by comparing the result bitmap from the previous block (i − 1) with the possible bitmap combinations for the current block (i). This comparison ensures that the correct control set is selected for the current block based on the outcomes of the previous block.

In the base design, Multiplexer *#2* was used to select one control set out of multiple m sets. In the modified design, this multiplexer is discarded. Instead, multiple Protocol Navigator blocks run in parallel, each handling a different control set. This parallel operation eliminates the need to wait for the results from the first block, allowing all possible scenarios to be processed simultaneously. The Prism Controller manages the control signals and ensures that each block operates based on the current clock cycle, sending the appropriate control signals to Multiplexer *#0* and Multiplexer *#1*. Each Protocol Navigator block processes data based on the control signals it receives, with multiple Protocol Navigators running in parallel to handle different control signals. The Bitmap Match unit compares the current bitmap with possible outcomes to select the appropriate data for processing, ensuring that the correct path is followed based on the previous block's results. This design removes the

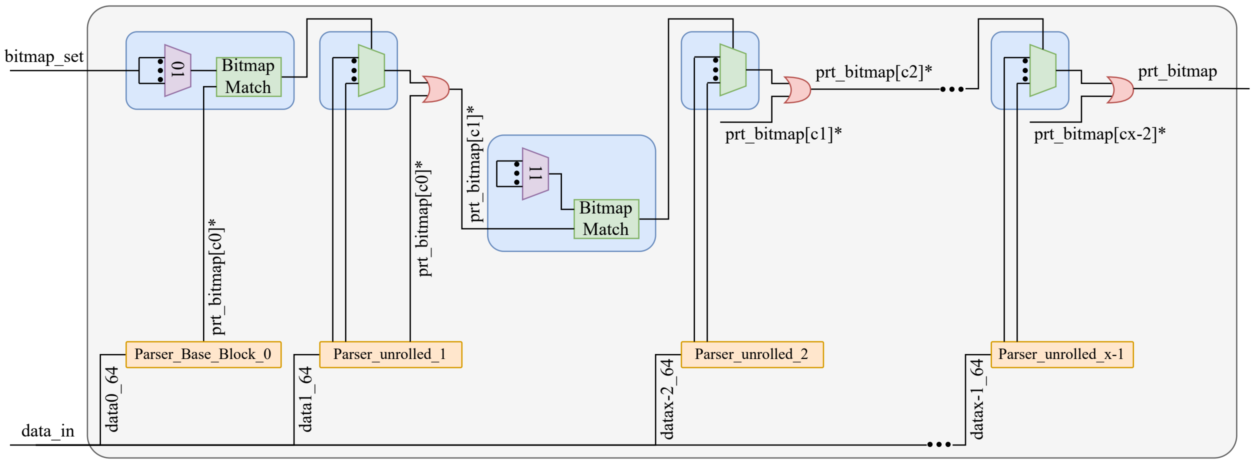


Figure 6.6 Overlay Block.

need to wait for a clock cycle to obtain the result from the first block, significantly reducing latency and increasing throughput. Figure 6.5 also illustrates how blocks from 0 to $m-1$ in Protocol Navigator blocks operate in parallel within the modified base block, resulting in a more efficient and faster processing pipeline.

The performance of different overlay sizes was evaluated by varying the parameter x. The results are summarized in Table 6.2, showing the throughput improvements achieved by increasing the number of modified base blocks.

Table 6.2 Results overlay block with different bus sizes.

Performance					Resources		
Data Bus [bits]	**Parameter x**	**Frequency [MHz]**	**TP [Gb/s]**	**Max Latency [cycles]**	**LUTs**	**FFs**	**Total Logic**
64	1	237.69	15.20	9	1028	1873	2901
128	2	193.38	24.80	5	3613	1938	5551
192	3	165.00	31.71	4	6841	2012	8853
256	4	145.51	37.20	3	8782	2066	10,848
320	5	97.50	31.21	2	17,427	2147	19,574
512	8	69.00	35.32	2	21,143	2311	23,454
1024	16	62.90	64.42	1	21,492	2880	24,372

6.4.5 Pipeline Optimization

As seen in Section 6.4.3, the overlay design has limitations due to the complex critical path caused by the selection process for the output of each parallel search. This complexity limits the maximum frequency for larger bus sizes. We propose a pipeline approach suitable for larger bus sizes to achieve better throughput for large bus sizes. The longest path in our graph can be parsed in a maximum of two clock cycles with a bus size of 512 bits and one clock cycle with a bus size of 1024 bits. Consequently, there will be no need for memory to store packets. We propose a pipeline parser and a bus selector unit. The bus selector either parallelizes the 1024-bit input into sixteen chunks of 64-bit data or, in a more complex manner, forwards the eight chunks of 512-bit input into the correct half of the pipeline.

The bus selector is a critical component in this design. It splits the 1024-bit input into sixteen parallel chunks of 64 bits each or divides the 512-bit input into eight parallel chunks, forwarding them into the correct half of the pipeline. This selective forwarding ensures that data is distributed evenly and processed efficiently across the pipeline stages.

The pipeline's parser unit is a simplified base block version. Since the block is replicated along the pipeline, the first two multiplexers of the Prism Controller can be omitted. The clock controls these multiplexers, but in our pipeline approach, each stage is associated with a specific clock cycle. This simplification makes the base block even more streamlined, reducing the clock frequency. Since block i depends on $i-1$, the bitmap of the latter is forwarded to the former, creating a streaming pipeline that accepts a packet per clock cycle.

As explained in the previous section, the Prism Controller consists of three multiplexers, two of which are controlled by the clock number. If the same base block is put in a pipeline fashion, these two multiplexers can be eliminated since each stage in the pipeline is dedicated to a particular clock number. The generated bitmap is forwarded for comparison to the next pipeline stage, simplifying the base block even further than the overlay design that includes only one multiplexer for control. This simplification also improves the performance frequency, enabling higher throughput for larger bus sizes.

By unrolling the two multiplexers controlled by the clock, the pipeline design eliminates the need for complex control logic at each stage. Each pipeline stage processes its specific data portion in parallel, and the results are forwarded to the next stage in real-time. This design allows for continuous data flow and high throughput, removing the need for intermediate storage. The improved frequency and reduced complexity make this pipeline design suitable for handling larger bus sizes and achieving higher overall performance. The results are summarized in Table 6.3, showing the throughput improvements achieved by this approach.

Table 6.3 Results comparison for base block.

Work	**FPGA**	**Performance**				**Resources**		
		Bus Size [bits]	**Frequency [MHz]**	**TP [Gb/s]**	**Latency [ns]**	**LUTs**	**FFs**	**Total Logic**
[16]—Golden	Virtex-7	512	195.30	100.00	27.00	N/A	N/A	8000
[16]	Virtex-7	512	195.30	100.00	46.10	10,103	5537	15,640
[58]	Virtex-7	320	312.50	100.00	25.60	7831	1367	21,502
[83]	Ultrascale+	512	250.00	128.00	36.00	2587	2395	4990
Proposed—Pipeline	Alveo U280	512	282.32	144.55	31.87	8435	3880	12,315
[70]	Virtex-7	1136	279.30	317.00	25.06	16,888	12,033	28,921
[59]	Ultrascale+	1280	800.00	1024.00	15.00	15,634	11,476	27,110
Proposed—Pipeline	Alveo U280	1024	282.32	289.10	31.87	8435	3880	12,315

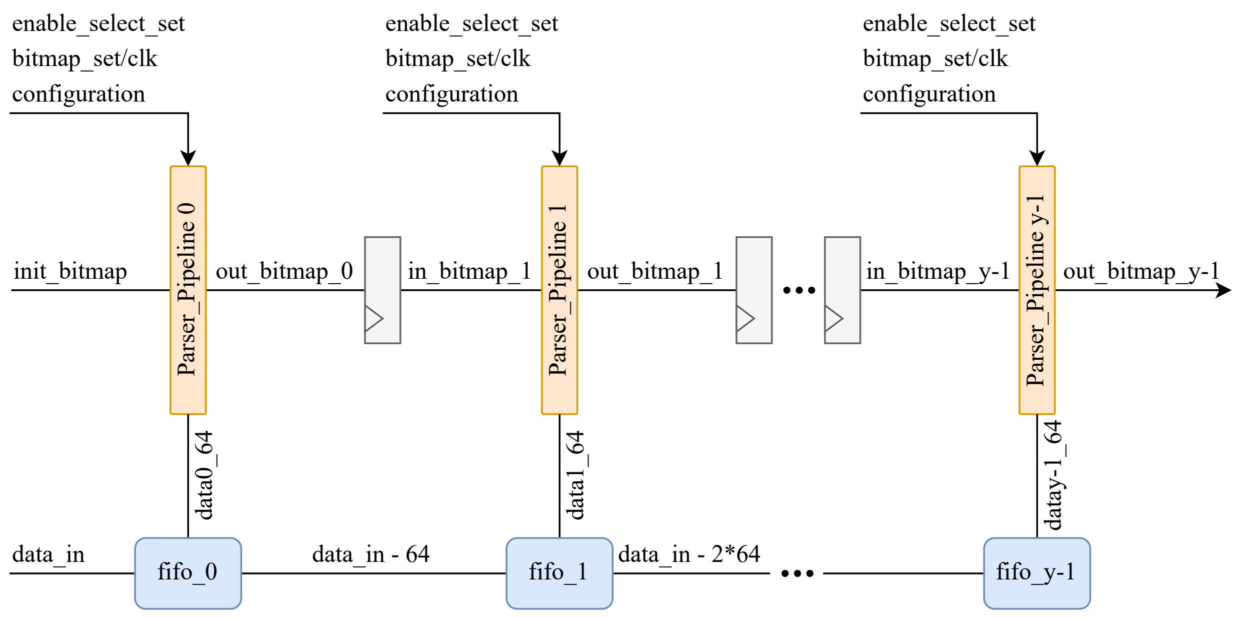


Figure 6.7 Parser Pipeline Block.

Adopting a pipeline design offers several advantages. It allows continuous data processing without waiting for the previous block's results, significantly increasing throughput. By dividing the processing task into stages, each packet part can be processed concurrently at different stages, enhancing overall efficiency. This design is especially beneficial in high-speed network environments where minimizing latency and maximizing data processing rates are crucial.

Unrolling both the bitmap and control multiplexers simplifies the parser block by dedicating one information set to each pipeline stage. This method is particularly effective for larger bus sizes, such as 512 and 1024 bits, as it improves block frequency and throughput. It allows the longest graph path headers to be parsed in one or two clock cycles, eliminating the need for packet storage and avoiding system back pressure. One significant advantage of this approach is that it does not require changes to the software to generate parsing information. This method enhances performance while maintaining simplicity and reliability. Additionally, the configuration and control data generated via software remain unchanged, ensuring compatibility and ease of implementation. Overall, this approach provides a throughput comparable to other high-performance designs, making it a robust solution for handling larger bus sizes efficiently.

6.5 Experimental Results

This section discusses the experimental results obtained by evaluating the proposed P4-programmable packet parser framework. The experiments were conducted on an Alveo U280 FPGA board. The hardware architecture was reprogrammed via software-generated configurations to process different input graphs. Configurations were generated for three different graphs: an enterprise network graph, a firewall graph, and an Access Gateway Function (AGF) graph. The enterprise network graph was the most complex, comprising ten nodes and seven transitioning nodes that require parsing states. The maximum number of connections between a node and its subordinate nodes in the hierarchy was four. Therefore, there are seven protocol investigators and four match detectors, and the protocol bitmap vector comprises 10 bits, one per protocol. The other two graphs fit within the boundaries considered for the enterprise network graph, and the same methodology can be applied to even more complex graphs if needed.

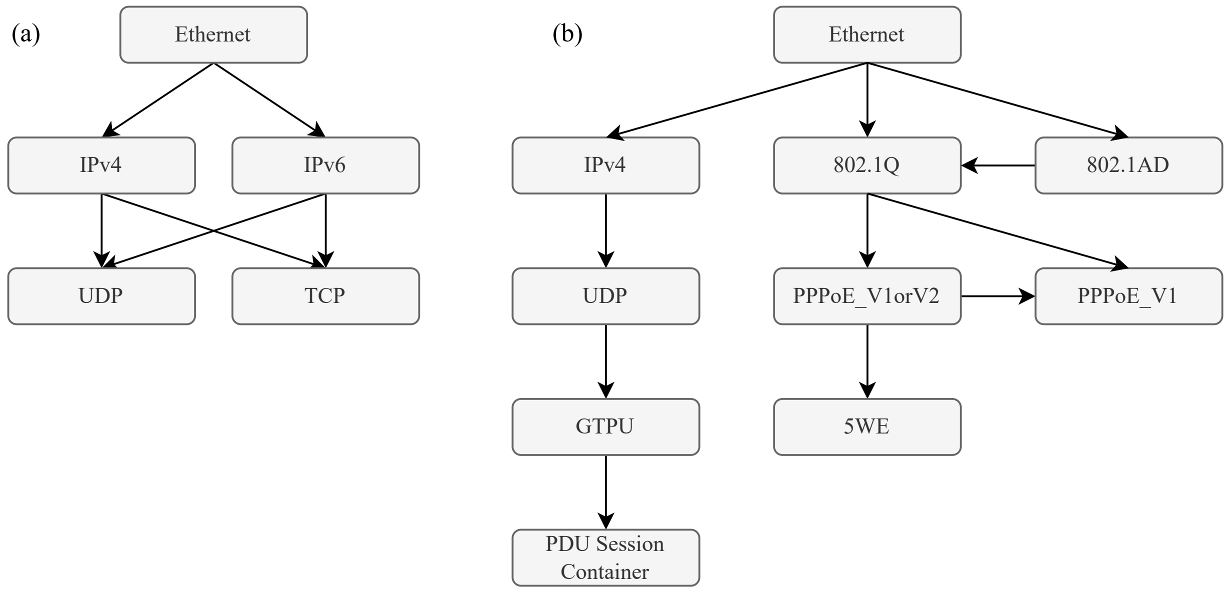


Figure 6.8 (**a**) Simple firewall graph. (**b**) Access Gateway Function flow graph.

Each of the seven protocols is allocated a 16-bit mask, resulting in a total of $7 \times 16 = 112$ bits. Given that this graph has a maximum of four connections, there are four keys for each protocol, each being 16 bits in size, totaling $7 \times (4 \times 16) = 448$ bits. Additionally, each key has an associated next protocol ID, representing each ID as a 4-bit array, leading to a total of $7 \times (4 \times 4) = 112$ bits. This means that the total number of bits required for Protocol Investigators is 672.

Each of the seven Protocol Investigators in the control unit requires a 1-bit *Enable* signal and a 2-bit *Select* signal to choose any 2-byte segment from the 64-bit input. This means that, for each cycle, $7 \times (1+2) = 21$ bits are essential to control all Protocol Investigators. Considering that the maximum number of distinct directions for various paths is determined to be four, there are $4 \times 21 = 84$ bits routed to the third prism multiplexer before the bitmap match. Given that the longest path in our graph is parsed over nine clock cycles, the cumulative number of control set bits directed to the second multiplexer of the prism controller amounts to $9 \times 4 \times 21 = 756$ bits. Additionally, four sets of 10-bit protocol bitmap sets are linked to the first multiplexer of the prism controller. Considering the nine clock cycles required for parsing, this results in a total of $9 \times 4 \times 10 = 360$ bits. Combining these two values yields a total of 1116 bits.

Tables 6.1–6.3 showcase our design outcomes using Vivado 2022.2 to synthesize the RTL code for an Alveo U280 board. Table 6.1 presents performance metrics and resource utilization, contrasting the programmability methods and the time taken to transition from a P4 code modification to hardware compared to other studies. Our design consumes 1028 LUTs and 1873 FFs and operates at 237.69 MHz. This frequency leads to a throughput of 15.2 Gb/s for a bus size of 64 bits per clock cycle. Table 6.2 presents the results of area and throughput for different bus sizes for the overlay design. Finally, Table 6.3 demonstrates and compares the results of the pipeline design with similar approaches in state-of-the-art works.

6.5.1 Base Block Design Performance

In terms of performance, our proposed design achieves the highest frequency at 237.69 MHz and the highest throughput at 15.2 Gb/s. This is a significant improvement over other designs, such as [62], which operates at 125 MHz with a throughput of 8 Gb/s, and [61], which achieves a frequency of 173.4 MHz and a throughput of 11.1 Gb/s. Additionally, our design maintains a maximum latency of nine cycles, matching the lowest latency observed among the compared works.

Resource utilization is another area where our proposed design excels. It uses only 1028 LUTs and 1873 FFs, considerably lower than the 3538 LUTs and 1447 FFs used by [62]. This efficient resource usage is further highlighted by the total logic usage, which is 2901 in our design, compared to 4029 in FPGA_paper. This demonstrates that our design not only delivers high performance but also does so with optimal use of hardware resources.

Regarding programmability, our design utilizes a register file for storing the configuration, similarly to other high-performance designs, ensuring low latency and fast reconfiguration. The configuration time for our design is in the order of seconds, which is significantly faster compared to [62], which requires hours for configuration. The quick reconfiguration capability, combined with our design's efficient resource usage and high performance, makes it highly adaptable and efficient for dynamic network environments.

Our proposed design offers superior performance, resource efficiency, and programmability compared to existing works. It achieves the highest throughput and frequency with minimal latency and resource utilization. It is an excellent choice for modern network environments that demand high-performance and scalable packet parsing solutions.

6.5.2 Pipeline Design Performance

The proposed pipeline design demonstrates excellent performance and resource efficiency compared to other works. Our design achieves a frequency of 282.32 MHz and a throughput of 144.55 Gb/s with a data bus size of 512 bits. Additionally, with a data bus size of 1024 bits, our design reaches a throughput of 289.10 Gb/s while maintaining the same frequency. The maximum latency observed is 31.87 cycles.

Compared with the work of [16], our proposed design shows superior performance. While their design achieves a frequency of 195.3 MHz and a throughput of 100 Gb/s, it uses significantly more resources, with 10,103 LUTs and 5537 FFs. Our design, on the other hand, utilizes only 8435 LUTs and 3880 FFs.

Similarly, our design outperforms [58] regarding resource efficiency and frequency. While they achieve a frequency of 312.5 MHz, their throughput remains the same at 100 Gb/s with a higher latency of 25.6 cycles. Our design achieves a comparable performance with significantly fewer resources, making it more efficient.

Compared to [83], our design offers a higher throughput of 144.55 Gb/s and 289.10 Gb/s for 512-bit and 1024-bit data buses, respectively, while using more resources. However, the increased performance justifies the difference in resource usage, making our design a robust choice for high-throughput applications.

The work of [70] achieves a higher throughput of 317 Gb/s with a data bus size of 1136 bits, but at the cost of significantly higher resource usage (16,888 LUTs and 12,033 FFs). Our design provides a balanced approach with competitive throughput and efficient resource utilization, making it suitable for various network environments.

Lastly, Ref. [59] achieves an impressive throughput of 1024 Gb/s with a data bus size of 1280 bits and the lowest latency of 15 cycles. However, their design requires significantly more resources (15,634 LUTs and 11,476 FFs), which might not be feasible for all applications. While achieving lower throughput, our design offers a more resource-efficient solution with excellent performance.

Overall, our proposed pipeline design offers a high-performance and resource-efficient solution for packet parsing, demonstrating superior scalability and flexibility for modern network environments.

6.6 Conclusions

This paper presents a versatile and efficient P4-programmable packet parsing framework that harnesses FPGAs' reconfigurability and high-speed data processing capabilities. Implemented as a hardware/software co-design, our software extracts definitions from a P4 code to generate memory content that programs the hardware. The hardware can be one of the three proposed architectural designs—base, overlay, and pipeline—each optimized for distinct packet parsing performance requirements. Experimental results validate the framework's efficiency and scalability, with detailed resource consumption metrics showing significant improvements in throughput, frequency, and resource utilization compared to existing solutions.

The base design achieves a frequency of 237.69 MHz and a throughput of 15.2 Gb/s using 1028 LUTs and 1873 FFs, showcasing a balance of high performance and resource efficiency. By doubling the bus size, the overlay design further enhances throughput while maintaining efficient resource usage. The pipeline design achieves remarkable throughput rates of 144.55 Gb/s and 289.10 Gb/s for 512-bit and 1024-bit data buses while maintaining low latency and high resource efficiency.

Regarding programmability, the proposed framework supports rapid reconfiguration in the order of seconds, significantly outperforming other designs requiring hours for similar transitions. This capability, combined with the proposed designs' high performance and resource efficiency, makes the framework highly adaptable to dynamic network environments.

The originality of our work lies in its ability to integrate high-throughput, resource-efficient packet parsing with rapid reconfiguration capabilities, addressing the limitations of existing solutions. Our pipeline design, achieving up to 289.10 Gb/s, surpasses the performance of many state-of-the-art designs, which often sacrifice speed or resource efficiency. Additionally, our framework's rapid reconfiguration in seconds substantially improves adaptability and responsiveness to network changes. This combination of high performance, resource efficiency, and quick reconfiguration establishes our framework as a superior solution for modern, scalable network environments.

In conclusion, the PrismParser framework offers a powerful solution for implementing efficient

P4-programmable packet parsers on FPGAs, particularly suited for multi-tenant networking environments. By supporting dynamic reconfiguration and high-speed processing, the framework addresses the unique challenges of network slicing and virtualization, ensuring robust performance and isolation for multiple tenants.

CHAPTER 7 ARTICLE 4: TEMPLATED AND OVERLAY HW/SW CO-OPTIMIZATION FOR CROSSBAR-FREE P4 DEPARSER FPGA ARCHITECTURES

Parisa Mashreghi Moghadam, Tarek Ould-Bachir, Yvon Savaria

Published in: MDPI - Electronics

Submission date: October 27, 2025

Preface: This chapter presents a crossbar-free P4 deparser architecture that extends the PrismParser framework to the packet reconstruction stage of PISA-based pipelines. The work introduces a unified hardware–software co-design that enables reprogrammable deparsing without sacrificing deterministic timing or throughput. Unlike specialized deparsers that require hardware resynthesis when protocols evolve, the proposed design externalizes emission order, start offsets, alignment rules, and chunk scheduling into compact overlay configuration tables derived from P4 metadata. This approach achieves scalable resource efficiency, predictable cycle-accurate behavior, and in-field protocol evolution. Experimental results on a Xilinx Virtex UltraScale+ device confirm sustained high throughput, with flexible reconfiguration supported through software-generated control tables. These findings demonstrate that deparsing, like parsing, can be made both high-performance and runtime-programmable through a clean separation of datapath and configuration. This work has been submitted for peer review and published in the proceedings of the *MDPI - Electronics*.
Contributions: This research originated as part of my doctoral work at Polytechnique Montréal and was developed in collaboration with my co-authors. I contributed to the conception of the problem, the architectural design, testing and deployment of the system, literature review, analysis of the results, and preparation of the manuscript. My co-authors provided guidance, feedback, and supervision throughout the research and revision process.
Full Citation: Parisa Mashreghi-Moghadam, Tarek Ould-Bachir, Yvon Savaria. 2024. *"Templated and Overlay HW/SW Co-Optimization for Crossbar-Free P4 Deparser FPGA Architectures"* Electronics 2025, 14(24), 4850.
DOI: 10.3390/electronics14244850
Copyright: © 2025 by the authors.

7.1 Abstract

The deparser stage in the Protocol-Independent Switch Architecture (PISA) is often overshadowed by parser and match-action optimizations. Yet, it remains a critical performance bottleneck in P4-programmable FPGA data planes. Challenges associated with the deparser stem from dynamic header layouts, variable emission orders, and alignment constraints, which often necessitate resource-intensive designs, such as wide, dynamic crossbar routing. While compile-time specialization techniques can reduce logic usage, they sacrifice runtime adaptability: any change to the protocol graph, including adding, removing, or reordering headers,

requires full hardware resynthesis and re-implementation, limiting their practicality for evolving or multi-tenant workloads. This work presents a unified FPGA-targeted deparser architecture that merges templated and overlay concepts within a hardware–software co-design framework. At design time, template parameters define upper bounds on protocol complexity, enabling resource-efficient synthesis tailored to specific workloads. Within these bounds, runtime reconfiguration is supported through overlay control tables derived from static deparser DAG analysis, which capture the per-path emission order, header alignments, and offsets. These tables drive protocol-agnostic, chunk-based emission blocks that eliminate the overhead of crossbar interconnects, thereby significantly reducing complexity and resource usage. The proposed design sustains high throughput while preserving the flexibility needed for in-field updates and long-term protocol evolution.

7.2 Introduction

The evolution of modern networking began with early, research-driven initiatives such as ARPANET, funded by the US Advanced Research Projects Agency (ARPA) in the 1960s. These initial experiments with packet-switched communication laid the foundation for the global Internet of today, which now connects billions of users and devices worldwide [1, 85]. As the Internet expanded, emerging applications such as video streaming, mobile services, and cloud computing required more than basic connectivity. Networks needed to provide low latency, high throughput, and in-network security [6,24]. Traditional fixed-function hardware was optimized for specific protocols, but it lacked flexibility, and even minor updates often required extensive hardware redesigns, a limitation widely recognized as network ossification [7, 8].

To address this, the networking community introduced Software-Defined Networking (SDN), which separates the control plane from the data plane, enabling centralized management and programmable behavior [10, 69]. Early SDN frameworks such as OpenFlow [11] represented an important step toward programmability, but they still supported only predefined protocol fields, and new features continued to require firmware and standard updates [12]. A major breakthrough followed the introduction of the Protocol-Independent Switch Architecture (PISA) [12], which defines a configurable pipeline comprising a parser, match-action units, and a deparser. Alongside PISA, the P4 language was developed to describe packet-processing logic in a protocol-agnostic and target-independent manner [13], allowing developers to define new protocols entirely in software for diverse targets, including ASICs, Network Interface Cards (NICs), and FPGAs. FPGAs have also emerged as attractive platforms for implementing P4-programmable data planes, offering high throughput and reconfigurability.

Several works have used FPGA flexibility to create programmable pipelines and prototype P4 applications, including high-speed packet parsers [59, 61, 66], P4-to-hardware compilation frameworks [20, 74, 79], and automatic P4-to-VHDL generators [16–18, 57]. These efforts demonstrate the practicality of FPGA-based data planes and motivate continued research on efficient and reconfigurable architectures for all PISA stages.

Although parsers and match–action stages have benefited from significant architectural innovation, the deparser, responsible for reconstructing outgoing packets, remains a performance bottleneck on FPGA platforms. This limitation arises from dynamic header layouts, variable emission orders, and alignment constraints, which often require wide crossbars or shift networks [73, 86]. Such interconnect structures consume large amounts of LUT resources, create routing congestion, and restrict achievable clock frequencies, which motivates research into more resource-efficient alternatives. Compilation-driven generation of hardware blocks such as P4-to-VHDL [57] further illustrates how P4 programs can be translated into static, program-specific VHDL components. However, these designs remain tied to a single P4 configuration and do not support runtime-reconfigurable deparsing and other work illustrates how general-purpose deparser designs quickly encounter scalability limits when attempting to match ASIC-level flexibility [55]. More recent efforts, including the work by Luinaud et al. [68], introduced vendor-independent bit-selector pipelines and strategic concatenation to reduce resource usage while maintaining high throughput. Their follow-up work [87] employed symbolic scheduling and partial evaluation to specialize deparsers for each P4 program, removing redundant logic. Cabal et al. [88] proposed the MFB Deparser, a high-throughput FPGA design that scales beyond 100 Gbps by processing multiple packets per cycle. Although these solutions are effective for their intended applications, they require custom deparsers to be regenerated for each P4 program and therefore do not provide runtime reconfigurability. Consequently, both the specialization-based approach of Luinaud et al. and the throughput-scalable MFB Deparser remain limited in environments that require protocol evolution, multi-tenant operation, or in-field updates.

Although existing FPGA deparser designs offer programmability through compile-time specialization, they remain structurally static once synthesized and cannot adapt to changes in protocol configurations. Any update to the protocol graph, including modifications to header order, insertion of new encapsulation layers, or changes in field size, requires hardware regeneration and resynthesis, which prevents in-field reconfiguration. On the other hand, although permutation engines and generic crossbar networks can, in principle, support runtime flexibility, their quadratic switching complexity, long critical paths, and routing congestion make them impractical for scalable FPGA deployment. To overcome these challenges, this paper addresses them by presenting a unified FPGA-targeted deparser architecture that integrates templated efficiency with overlay flexibility within a hardware–software co-design framework. The architecture eliminates wide crossbars through slice-and-shift operations and leverages overlay control tables for runtime adaptability. This balance of compile-time optimization and runtime reconfiguration enables high throughput while supporting evolving protocol graphs.

The contributions of this paper are as follows:

- A recursive-select deparser architecture with runtime overlay control to eliminate dynamic crossbars for programmable deparser (sec:hw-arch).
- A set of hardware building blocks (PHV operator, streamer, payload delay, and aligner) enabling scalable and synthesizable FPGA implementation (sec:sub-blocks).
- A formal complexity analysis showing linear growth compared to quadratic crossbar designs (sec:phv-operator).
- The proposed design sustains throughputs of more than 200 Gbps, achieving near-specialized levels of resource efficiency with larger bus widths, while uniquely preserving runtime adaptability through overlay reconfiguration (sec:results).

Although this work focuses on the deparser stage, the underlying concepts extend more broadly across the PISA architecture. Our earlier work introduced a templated hardware approach [59], in which a single reusable header-analysis architecture was instantiated across the parser pipeline using compile-time generic parameters derived from the P4 parser DAG description. This methodology decoupled parser functionality from manual RTL redesign but still required FPGA resynthesis whenever the protocol graph changed. The overlay concept advances this decoupling further by separating static datapath structures from memory-defined runtime control and can be applied directly to parser graph traversal [61, 89]. In this stage, decision structures such as protocol transitions and extraction rules are not implemented as fixed RTL state logic. Still, they are instead encoded in configuration memory as masks, match values, next-state identifiers, and control vectors that drive a protocol-agnostic execution engine. For instance, a parser transition from Ethernet to either IPv4 or IPv6 based on the 16-bit `EtherType` field is realized through a memory-defined 16-bit key mask, associated match values (e.g., $\{0x0800, 0x86DD\}$), and programmed next-state selections mapped to the corresponding IPv4 and IPv6 parser states. Additional control parameters stored in the configuration memory define the enable and select vectors that determine which processing units and multiplexing paths are active in each cycle and update the validity bitmap at runtime, rather than relying on hard-wired control logic. Consequently, modifying the parser

DAG, such as inserting additional protocol branches, requires only updating configuration entries without altering the datapath or performing FPGA resynthesis.

The remainder of this paper is organized as follows. Section 7.3 reviews the background of PISA pipelines and outlines key principles underlying templated and overlay approaches. Section 7.4 presents the proposed recursive-select deparser architecture, detailing its sub-blocks, execution model, and complexity analysis. Section 7.5 evaluates the design through synthesis results, scalability analysis, and comparisons with state-of-the-art FPGA deparsers. Finally, Section 7.6 concludes the paper and discusses the implications of our work for future P4-programmable FPGA platforms.

7.3 Background

The Protocol-Independent Switch Architecture (PISA) [12] defines a generic packet processing pipeline composed of three main stages: a parser, a match–action pipeline, and a deparser. The parser inspects incoming packets, extracting headers according to a Directed Acyclic Graph (DAG) specified in the P4 program [13]. Each node in this parser DAG represents a protocol header, and edges represent conditional transitions based on field values. Extracted header fields are stored in the Packet Header Vector (PHV) along with per–header validity bits. The match–action pipeline consumes the PHV and validity bits to perform lookups, modify fields or metadata, and can set or clear header validity before forwarding the packet to the deparser. The deparser reconstructs the outgoing packet by invoking emit on headers in a fixed, program-specified order, serializing each header only when its validity bit is set. Consequently, although the emission order is static, the subset of headers actually emitted may vary from packet to packet depending on parser outcomes and match–action modifications. On FPGA targets, the deparser must additionally account for bus-width granularity, byte-level alignment, and variable header lengths. Conventional implementations of such designs typically rely on wide crossbars or shifting networks to perform these alignments [73, 86], which substantially increase LUT utilization, add routing congestion, and limit the achievable clock frequency. These challenges have motivated ongoing research into alternative, more resource-efficient deparser architectures.

Luinaud et al. [68, 87] demonstrated that compiler-driven specialization can eliminate dynamic control in the deparser by hard-wiring the emission schedule and concatenation logic directly from the P4 program. This yields area efficiency and high clock frequency; however, the hardware becomes tied to a single protocol graph. Any modification to header order, encapsulation depth, or protocol composition requires full hardware regeneration, resynthesis, and place-and-route, a process that can take tens of minutes to hours on modern FPGAs.

This compile-time rigidity makes specialization impractical for deployments where protocol configurations evolve. Templated architectures [59] improve generality by constraining structural limits such as header width, parser pipeline depth, or field extraction granularity at synthesis time. Although they maintain high hardware efficiency, they remain compile-time configurable and do not support runtime reconfiguration once mapped to FPGA fabric. Overlay architectures [61,89] overcome this limitation by decoupling datapath hardware from control and loading configuration from on-chip memory. This enables runtime reconfiguration in a few nanoseconds without resynthesis, significantly reducing deployment latency, avoiding FPGA downtime, and enabling hardware reuse across evolving P4 programs. However, flexibility alone is not sufficient, as generic crossbar-based emission fabrics introduce quadratic hardware cost and long interconnect delays. To address this, the proposed deparser delivers crossbar-free runtime programmability using a recursive-select architecture that scales linearly in hardware and achieves throughput comparable to specialized designs while remaining reconfigurable at runtime.

7.3.1 Design Principles for Overlay Architecture

The proposed overlay architecture targets a reusable deparser that can support diverse P4 workloads within predefined overlay boundaries. Unlike the compiler specialization of Luinaud et al. [68,87], which generates a fixed emission schedule tailored to one P4 program, our approach fixes only the structural limits of the hardware at synthesis time. These limits, which cover the maximum number of headers, transitions, maximum header size, and maximum bus width, define the template capacity once at compile time. At runtime, protocol variations are accommodated by reprogramming an overlay configuration memory, which encodes emission sequences, header alignments, and offsets derived from static analysis of the P4 compiler (P4C) [40] output. This separation ensures the datapath remains constant after synthesis, while the deparser's behavior can be reconfigured for heterogeneous applications and runtime requirements.

Table 7.1 summarizes the maximum boundary parameters that define the design space. Additional notation specific to timing analysis and PHV operator internals will be introduced later.

Table 7.1 Notation summary for maximum boundary parameters.

Symbol	**Meaning**
W_{PHV}	Width of the Packet Header Vector (Bytes)
$H_{\max}$	Maximum number of headers supported by template
$L_{\max}$	Maximum header length (Bytes)

Software Optimizations for Deparser Graphs

The deparser's behavior can be formalized through a graph abstraction. Luinaud et al. [68,87] introduced the notion of a deparser DAG, derived directly from the program's emit order. In this model, nodes represent header bytes and edges capture their possible successors in the serialized packet stream. This global DAG is then partitioned into per-lane sub-DAGs, each driving the multiplexer network that feeds a single output byte lane. Their compiler performs a one-time traversal of these sub-DAGs at compile time, yielding fully specialized concatenation schedules that minimize resource usage but remain tied to a single P4 program, without runtime adaptability.

Based on this foundation, our approach adopts a similar high-level decomposition into global DAGs and sub-DAGs. Still, it uses the P4 compiler's JSON to generate them automatically and compile them into metadata rather than hardwired logic. Specifically, we record all legal emission sequences, their lengths, and alignment requirements. From these, we derive maximum boundary parameters that bound the design space at synthesis time. At the same time, the emission details per path are encoded in compact overlay control tables that guide the datapath at runtime. This hybrid scheme preserves the efficiency of DAG–based scheduling while enabling reprogrammability: a single synthesized instance can support multiple P4 applications as long as they remain within these bounds, with behavior adapted dynamically through overlay memory updates.

At synthesis time, the following characteristics are extracted and enforced as maximum values for boundaries:

- The maximum number of distinct protocols present in the deparser DAG,
- Maximum total size (in Bytes) of all headers across the DAG,
- Maximum number of headers along any single emission path
- Maximum size (in Bytes) of an individual header,

- The target data bus width (e.g., 64, 128, or 256 bits).

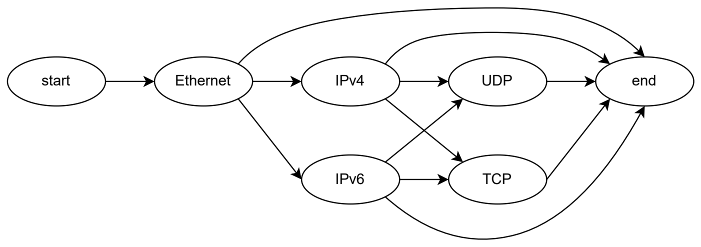
Figure 7.1 demonstrates (a) T1, (b) T2, (c) T3 deparse DAGs and (d) configuration parameters and the boundaries for them. These parameters act as a blueprint to bound the maximum values, while the per-path overlay tables generated by our software provide runtime flexibility. This blueprint ensures that the hardware achieves high throughput within its predefined bounds, yet remains adaptable to protocol evolution without requiring resynthesis. By statically analyzing the deparser DAG, the system determines the emission order, lengths, start offsets, and alignment rules for all valid sequences. This data is then used to generate static schedules, configure chunk-based emission units, and populate overlay control memory. Bounding the maximum headers, transitions, and alignment patterns within the boundaries avoids unnecessary logic replication and over-provisioning. A single synthesized design can support multiple P4 applications within the established limits. The approach balances flexibility and efficiency: boundary parameters such as header count, path depth, and maximum header size are configurable via VHDL generics, whereas the bus width must be fixed during synthesis, as it defines the datapath and is not runtime-reconfigurable.

Software Optimizations for Overlay Memory Generation from P4

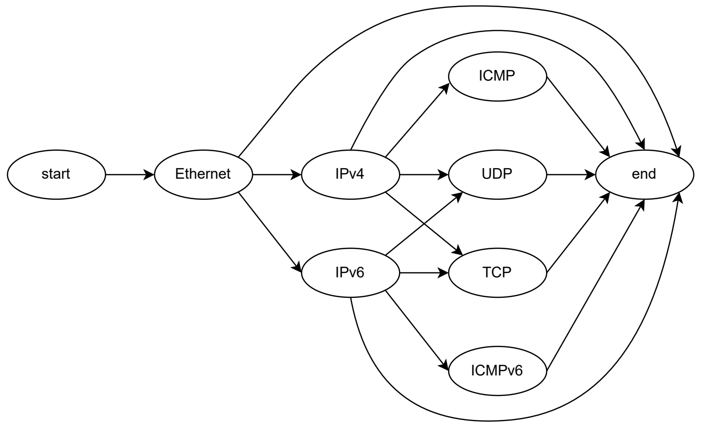
Once the boundaries have been defined and the hardware synthesized accordingly, the deparser must be configured to handle different combinations of protocol headers. This is achieved through a memory-resident overlay control table that stores precomputed metadata for each legal emission path. These schedules are generated by a companion software core that analyzes the JSON output of the P4 compiler and derives the necessary emission parameters from the deparser DAG.

Each configuration entry is indexed by a unique path identifier or header validity bitmap (`config_addr`) and contains:

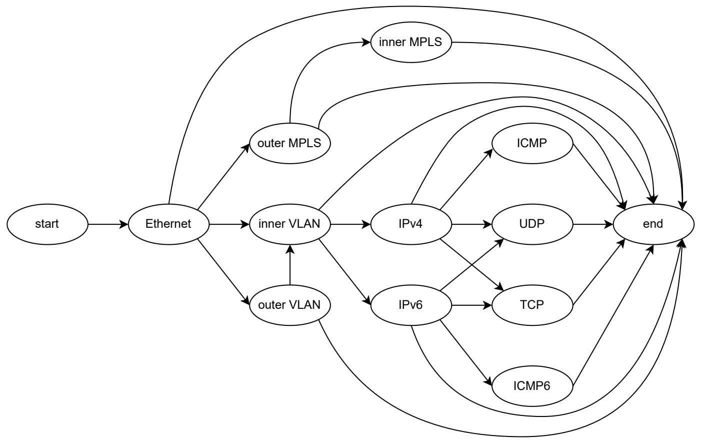
- The emission order of headers,
- Start offsets and lengths of each header within the PHV bus,
- Emission positions in the serialized output stream,
- Alignment padding, and inter-header spacing,
- The number of bus-sized chunks per header,
- Total cycle count and payload delay.



(a) T1 deparser DAG with Ethernet, IPv4, IPv6, TCP and UDP headers



(b) T2 deparser DAG with Ethernet, IPv4, IPv6, TCP, UDP and ICMP headers



(c) T3 deparser DAG with Ethernet, Vlan, IPv4, IPv6, TCP, UDP and ICMP headers

Parameter	**T1**	**T2**	**T3**
Header Count	5	7	11
Maximum Total Size (Bytes)	102	110	126
Maximum Tuples	3	3	5
Maximum Header Size (Bytes)	40	40	40

(d) Summary of configuration parameters values for T1, T2, and T3

Figure 7.1 Example of deparser DAGs and their associated template boundaries.

This structured metadata allows the hardware to reconstruct packets correctly for any P4 program within the bounds, without resynthesis. Algorithm 2 presents the pseudo-code for extracting the overlay configuration entries. The use of each parameter to enable a generic header-agnostic datapath will be explained later sections.

The overlay configuration tables are generated by a lightweight, fully automated software toolchain that operates directly on the P4 compiler's JSON output. The workflow in Figure 7.2 consists of three stages. First, the tool parses the P4C-generated JSON file and extracts all relevant protocol metadata, including header definitions, field sizes, parser states, transition conditions, and extraction operations. Second, using this information, it reconstructs the DAG and enumerates all reachable protocol paths. Each path is assigned a validity bitmap and a deterministic header-emission order. Third, for every path, the tool computes the full overlay configuration entry required by the deparser. Since the analysis involves only graph traversal and simple arithmetic over header metadata, the end-to-end generation process completes in a few tens of milliseconds for representative P4 programs. Crucially, no resynthesis, recompilation, or hardware regeneration is required.

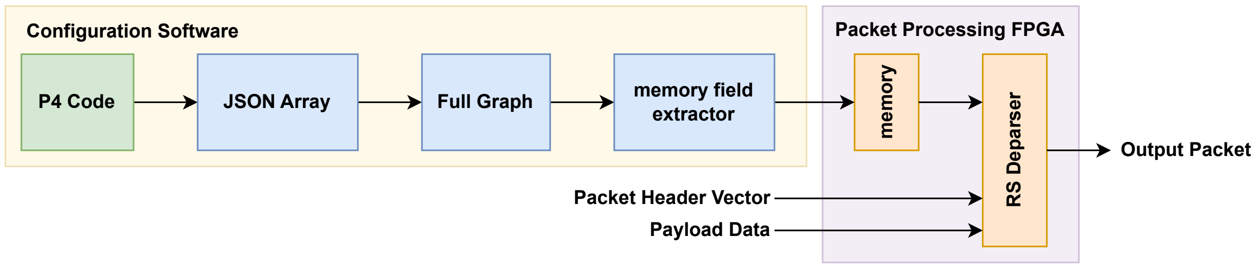


Figure 7.2 Proposed compilation workflow for generating configuration for the deparser.

Beyond specifying the format of each configuration entry, it is also necessary to account for the configuration memory's size and access latency. If each overlay entry has width E_{size} and the total number of valid paths from the deparser DAG is N_{paths}, then the required configuration-memory footprint is

$$M_{\text{config memory}} = E_{\text{size}} \times N_{\text{paths}}.$$

The configuration table maps to embedded block memories on FPGA, which offer discrete depth and width granularities and allow efficient packing of intermediate-sized tables. This means that the configuration memory typically occupies only a few percent of the total available BRAM resources on FPGA, depending on E_{size} and N_{paths}. Accessing an entry in the configuration memory introduces fixed, architecture-defined read and write latencies determined solely by the characteristics of the underlying FPGA memory primitives. Let L_{read} denote the memory-read latency and L_{write} the memory-write latency associated with updating the configuration table or routing its fields to the emission logic. The worst-case lookup latency to read a single config is therefore

$$L_{\text{config memory, max, read}} = L_{\text{read.}}$$

Algorithm 2 Overlay Configuration Memory Generation

```
J: Compiled P4 JSON file  B: Data bus width (bytes) ConfigMem: Overlay control table for
deparser $G_d \leftarrow$ Build deparser DAG from J (emit order graph)
$HeaderInfo \leftarrow$ Extract header *byte* lengths/types from J
$EmitOrder \leftarrow$ Linear emit order extracted from J
$ValidPaths \leftarrow$ Enumerate all legal emission paths in $G_d$
$ConfigMem \leftarrow$ Initialize empty table
path in $ValidPaths$ $Bitmap \leftarrow$ validity bitmap for headers in *path*
$n_h \leftarrow \text{popcount}(Bitmap)$
$EmissionOrder \leftarrow EmitOrder$ filtered by headers in *path* Initialize lists: $PHVOffsets$ ($s_h$),
$OutOffsets$ ($o_h$), $HeaderLengths$ ($\ell_h$), $ChunkList$
$LenAcc \leftarrow 0$ header $h$ in $EmissionOrder$ $\ell_h \leftarrow HeaderInfo[h].length$
$s_h \leftarrow ComputePHVOffset(h)$
$o_h \leftarrow LenAcc$
$chunks_h \leftarrow \left\lceil \frac{\ell_h}{B} \right\rceil$
$LenAcc \leftarrow LenAcc + \ell_h$
$Len \leftarrow LenAcc$
$T_{\text{hdr}} \leftarrow Len \bmod B$
$Alignment \leftarrow (B - T_{\text{hdr}}) \bmod B$
$W_{\text{hdr}} \leftarrow \left\lceil \frac{Len}{B} \right\rceil$
$L_{\text{mem}} \leftarrow 2$
$Delay \leftarrow (W_{\text{hdr}} - 1) + L_{\text{mem}}$
$Entry \leftarrow$ Encode($PHVOffsets$, $OutOffsets$, $HeaderLengths$, $ChunkList$, $Alignment$, $Delay$,
$n_h$)
$ConfigMem[Bitmap] \leftarrow Entry$
$ConfigMem$
```

In the current implementation, the access interface to the configuration memory uses a lightweight valid–address–data protocol, which introduces no additional cycle overhead beyond the native read/write latency of the memory primitive. If a custom wrapper is added then the wrapper's handshake and transport latency must also be included in the total access time. In such cases, the effective per-access latency becomes

$$L_{\text{config memory, program}} = L_{\text{custom\_wrapper}} + L_{\text{write}},$$

where $L_{\text{custom\_wrapper}}$ accounts for any arbitration, handshaking, framing, or transfer delays introduced by an external interface wrapper, and in the present design, this term is zero. When reprogramming the entire configuration table, the total update time scales with the number of entries N_{paths} due to it's pipelined nature. Assuming each entry is written once, the worst-case full-configuration programming time is

$$L_{\text{config memory, max, program}} = L_{\text{custom\_wrapper}} + N_{\text{paths}} + L_{\text{write}},$$

and, in real time,

$$T_{\text{config memory, max, program}} = \frac{L_{\text{config memory, max, program}}}{f_{\text{clk}}}.$$

This provides an upper bound for reloading all overlay entries; in most deployments, only a subset of entries requires updating when protocol configurations evolve.

7.4 Overlay Hardware Architecture

7.4.1 Architectural Overview

The deparser plays a critical role in the PISA pipeline Figure 7.3a by reconstructing outgoing packets from the Packet Header Vector (PHV) and payload. The proposed recursive-select deparser Figure 7.3b organizes configuration control separately from packet data processing. It consists of two main blocks: the configuration memory and the deparser core, which communicate through control and data interfaces to enable scalable packet reconstruction without resorting to dynamic routing logic. The configuration memory stores software-generated overlay tables indexed by a per-packet validity bitmap. The configuration entry specifies the emission order, PHV offsets, output offsets, alignment requirements, chunk scheduling, and payload timing for each legal protocol path. At runtime, the control logic retrieves the active entry and provides these parameters to the deparser core.

The deparser core in Figure 7.3b comprises five sub-blocks:

- PHV_operator,
- PHV_streamer,
- payload_delay,
- payload_aligner, and
- streamer_selector.

Guided by the control information, the PHV_operator extracts the required header segments according to the specified offsets and lengths. These segments are then streamed as contiguous bus-width words by the PHV_streamer, avoiding costly byte-level crossbars. The payload is held in the payload_delay and then aligned to the bus boundary by the payload_aligner. Finally, the streamer_selector appends the payload to the emitted headers, yielding the reconstructed packet.

The recursive-select execution model denotes a uniform, header-agnostic loop: select the next scheduled header, resolve its per-packet size and PHV offset, stream its bytes if valid, then advance the output pointer. This process is repeated up to the maximum bound value on headers per path. Because only overlay-provided indices, offsets, and lengths are consumed, the same hardware boundaries supports different P4 programs within the configured bounds by updating configuration entries rather than resynthesizing hardware. Overall, this modular architecture cleanly decouples static datapath logic from runtime control, preserving high throughput while enabling runtime reconfiguration. Operational details for a single-header combination are given in Section 7.4.3.

7.4.2 Sub-Blocks

The proposed overlay deparser comprises several interconnected sub-blocks, each responsible for a specific function in the reconstruction pipeline. As illustrated in Figure 7.3, these blocks collectively manage configuration, control, header processing, and payload handling to ensure correct and efficient packet emission. The main sub-blocks are summarized below:

- **Configuration memory**: overlay entries per legal path (emission order, PHV offsets/lengths, output positions, chunk counts, delay/alignment).
- **Control logic**: selects the active entry, computes the header count $n_h = \text{popcount}(\mathbf{V})$, and sequences exactly n_h iterations of the recursive-select loop.

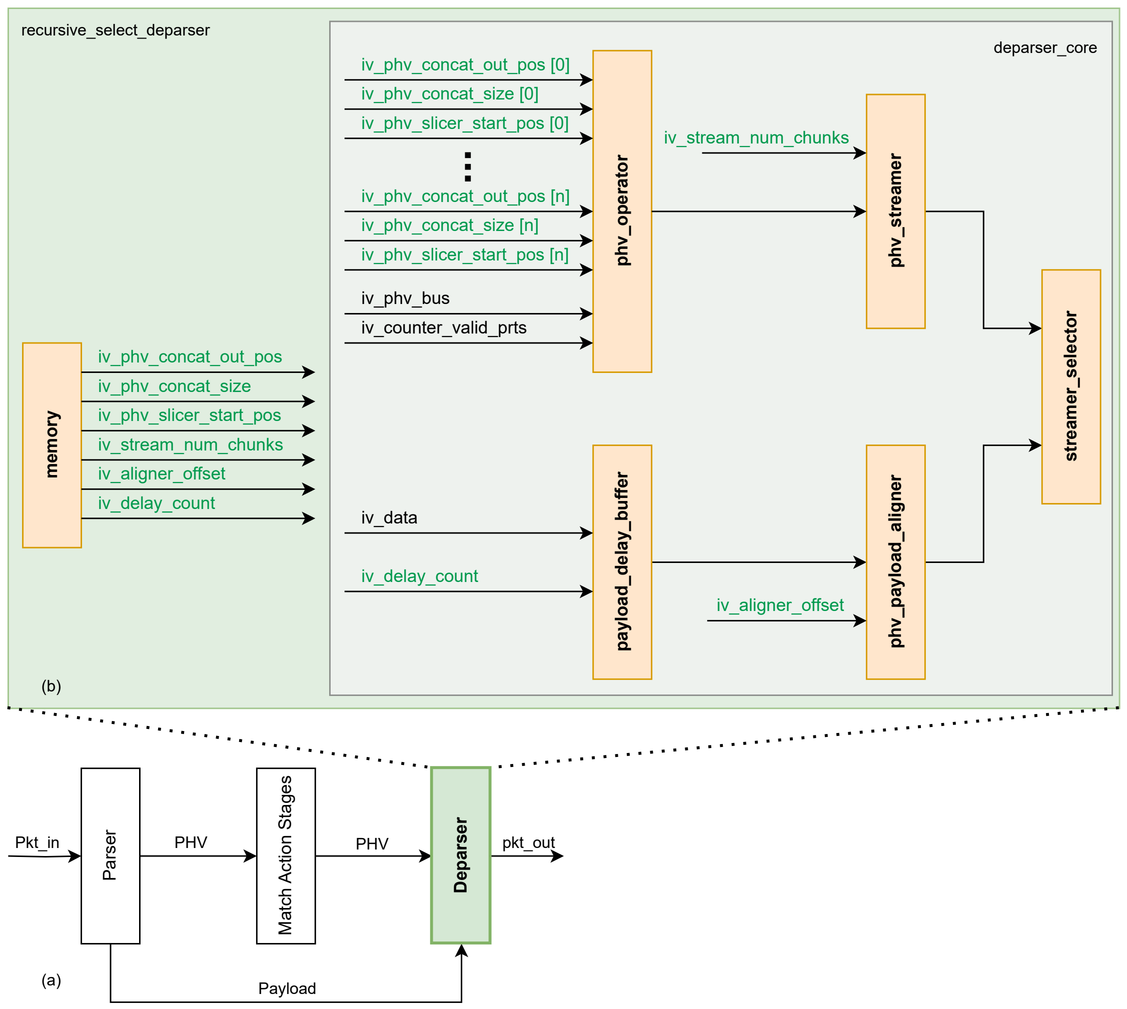


Figure 7.3 (**a**) PISA architecture comprising a Parser, Match–Action processing stages and a Deparser. (**b**) Recursive-Select Deparser architecture and sub-blocks.

- **PHV_operator**: extracts each valid header, positions it at the correct output offset, and OR-reduces slices into a contiguous header region (no dynamic crossbar).
- **PHV_streamer**: [-15]serializes bus-width words at one word per cycle in the scheduled order.
- **payload_delay**: buffers payload for the programmed delay to match header emission time.
- **payload_aligner**: applies a single alignment computed from the total header bytes (no per-header padding) for seamless, byte-accurate handoff.
- **streamer_selector**: multiplexes header and payload at the exact boundary cycle, without bubbles.

Together, these components form a modular datapath that avoids dynamic crossbars while guaranteeing line-rate emission. The following sections explore their interaction further, explaining the per-packet execution flow and timing behavior in more detail.

7.4.3 Per-Packet Execution (Single Header Combination)

At the input cycle, the deparser receives the PHV data vector, the per-header validity bitmap, and the first payload word with its valid. The control logic uses the bitmap to index the overlay configuration entry for this packet path. Each incoming packet triggers a lookup into the overlay configuration memory. While the entry is being fetched, the PHV, bitmap, and the payload are buffered. This ensures that data and control emerge in step with the memory read latency, keeping them time-aligned. Once the entry is available, the control logic sequences exactly $n_h = \mathrm{popcount}(\mathbf{V})$ iterations of the recursive-select loop, where n_h is the number of valid headers. In each iteration, the PHV operator slices and repositions one header, which is then streamed toward the output. After all headers are emitted, the payload delay block releases the buffered payload at the correct boundary, fusing the last header word with the first payload bytes. This process guarantees bubble-free handoff between headers and payload.

The configuration entry specifies which headers are present, their PHV start offsets and lengths, their output start positions, and the number of bus-sized chunks each header occupies. Once available, the header-assembly stage (PHV_operator) extracts each valid header from its predefined slice of the PHV, masks it to its exact length, shifts it to the correct output position, and OR-reduces the result into an intermediate header buffer. This operation converts header bytes that were scattered across the PHV bus into a single contiguous region of header data in the correct order, without requiring a dynamic crossbar.

The PHV_streamer then emits this contiguous header region one bus-width word per cycle. During this time, the payload remains buffered for exactly as many cycles as needed to match the total header emission time (*Delay*, defined in Section 7.4.4). If the final header word is only partially filled, the payload_aligner shifts the payload by the precomputed offset (*Alignment*, defined in Section 7.4.4) so that the remaining bytes of that word are completed by the first payload bytes in the same cycle. From the next cycle onward, only the payload is selected at the output. After the handoff, payload streaming continues at one word per cycle with the same alignment established at the boundary. The final payload word carries its own byte-valid count, and no bubbles are inserted between the tail of the header region and the start of the payload.

Table 7.2 summarizes the notation for timing parameters introduced in Section 7.4.4. These symbols will be used throughout the latency and alignment analysis of the PHV operator and streamer pipeline.

Table 7.2 Notation summary for timing and alignment analysis.

Symbol	**Meaning**
B	Payload/output bus width (Bytes)
ℓ_h	Length of header h (Bytes)
Len	Total header footprint
W_{hdr}	Header word count
L_{mem}	Configuration memory read latency
$Delay$	Payload delay
T_{hdr}	Header tail
$Alignment$	Payload aligner offset
n_h	Number of headers in the emission path
$\mathbf{V}$	Validity bitmap for headers in the path

7.4.4 Timing and Alignment Mathematics

Let B be the bus width in bytes. For each valid header h, let s_h be its PHV start offset (bytes), ℓ_h its length (bytes), and o_h its output start position (bytes). Let $\mathbf{V}$ denote the per-header validity bitmap and define the header count.

$$n_h \;=\; \text{popcount}(\mathbf{V}),$$

which is the number of headers in the path and therefore the number of loop iterations in the recursive-select deparser. The total header footprint is

$$Len \;=\; \sum_{n_h} \ell_h.$$

To calculate the delay, we need to know the header word count. Headers occupy $W_{\text{hdr}} = \left\lceil \frac{Len}{B} \right\rceil$ cycles (header word count). Let $L_{\text{mem}} = 2$ denote the configuration-memory read latency (cycles). The payload delay is therefore

$$Delay \;=\; (W_{\text{hdr}} - 1) + L_{\text{mem}} \;=\; (W_{\text{hdr}} - 1) + 2.$$

Let the header tail $T_{\text{hdr}} = Len \bmod B$ be the number of valid bytes in the final header word ($T_{\text{hdr}} = 0$ means the last header word is full). The aligner offset applied to the payload is

$$Alignment \; = \; (B - T_{\text{hdr}}) \bmod B.$$

To avoid a spurious one-word shift, $\bmod B$ is applied after computing $(B - T_{\text{hdr}})$; equivalently, the piecewise form is

$$Alignment \; = \; \begin{cases} 0, & T_{\text{hdr}} = 0, \\ B - T_{\text{hdr}}, & T_{\text{hdr}} \neq 0 \; . \end{cases}$$

[-10]If $T_{\text{hdr}} \neq 0$, the last header word contains T_{hdr} header bytes and $B - T_{\text{hdr}}$ bytes from the (shifted) payload in the same cycle; from the next cycle onward, only the payload is emitted.

PHV Operator

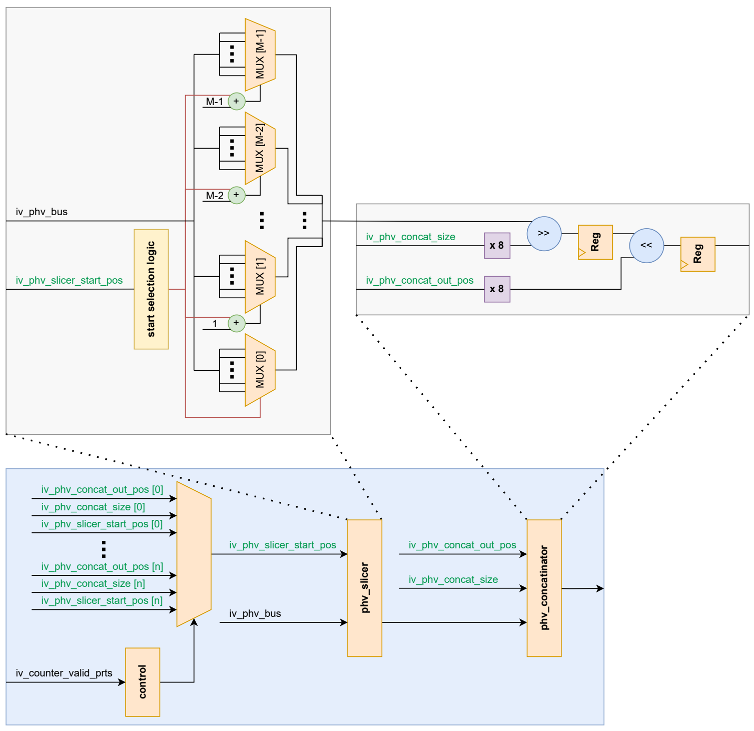
Table 7.3 summarizes the notation used for the PHV operator. These symbols describe how headers are sliced, shifted, and reconstructed, and will be referenced throughout Section 7.4.4.

Table 7.3 Notation summary for symbols used in the PHV operator.

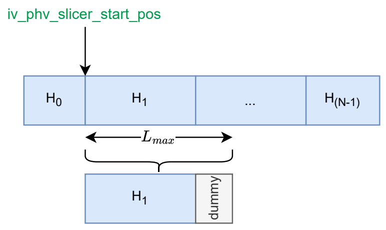
Symbol	**Meaning**
W_{PHV}	PHV width (Bytes)
$L_{\max}$	Maximum header length / slice size (Bytes)
s_h	PHV start offset of header h (Bytes)
o_h	Output stream offset of header h (Bytes)
δ_h	Shift distance: $\delta_h = o_h - \text{slice\_MSB\_position}$ (Bytes)
S_h	Shifted and masked slice for header h
H_{hdr}	Reconstructed header region, $H_{\text{hdr}} = \bigvee_{h=1}^{n_h} S_h$

The PHV operator is responsible for extracting protocol headers from the PHV and placing them into their correct positions in the output stream. The operator is decomposed into two stages to avoid the quadratic number of cross-point cost of a full byte-level crossbar: slice selection with MSB alignment and shift-based repositioning as demonstrated in Figure 7.4a.

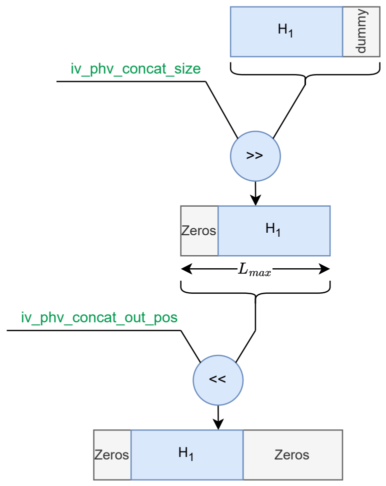
First stage is MSB-aligned slice selection. The slicer, as shown in Figure 7.4b, produces a fixed $L_{\max}$-byte window from the PHV starting at byte offset s_h (MSB-first indexing). If $s_h < L_{\max}$, the missing MSB portion is zero-padded. If the requested window runs past the PHV boundary ($s_h + L_{\max} > W_{\text{PHV}}$), the missing LSB portion is zero-padded. After applying



(a) PHV operator architecture



(b) Slicer header architecture



(c) Concatinator header architecture

Figure 7.4 PHV operator architecture.

the specific situation that applies, the output is always a $L_{\max}$-byte vector with valid PHV bytes aligned to the MSB side and any shortfall filled with zeros. Second stage is Shift-based re-positioning. In the second stage, as shown in Figure 7.4c, each MSB-aligned slice is then placed at its configured packet offset o_h. Instead of routing every byte through a crossbar, the design applies a logical shift by

$$\delta_h = o_h - \text{slice\_MSB\_position}, \tag{7.1}$$

which displaces the slice from its MSB alignment to the target output offset. After shifting by δ_h, the slice is masked to its actual header length ℓ_h to ensure padded bytes are suppressed. All slices are then combined with a bitwise OR:

$$H_{\text{hdr}} = \bigvee_{h=1}^{n_h} S_h, \tag{7.2}$$

where S_h is the shifted and masked slice of header h, and $\bigvee$ denotes a logical OR across all slices. This guarantees that headers are reconstructed at the correct offsets without overlap, while avoiding the quadratic crosspoint complexity of a crossbar.

Complexity analysis for the proposed design conveys that a conventional byte-level crossbar requires $O(W_{\text{PHV}}^2)$ byte multiplexers, since each of the W_{PHV} output positions must be able to select from all W_{PHV} PHV bytes. In contrast, the proposed PHV operator avoids this quadratic cost by decomposing reconstruction into three more straightforward steps: slice selection, per-header alignment, and output positioning.

- **Slice selection:** An $L_{\max}$-byte contiguous window is extracted from the W_{PHV}-byte PHV starting at offset s_h. This can be achieved by a barrel-shaped depth shifter $O(\log_2 W_{\text{PHV}})$ applied on $L_{\max}$ bytes, giving a cost of $O(L_{\max} \log_2 W_{\text{PHV}})$.

- **Re-positioning:** Each valid header requires two shift operations inside the concatenator. First, the slice is right-shifted so that its ℓ_h valid bytes occupy the least significant positions, with zero padding applied to the left. Second, the slice is left-shifted to its final packet offset o_h. Both operations touch at most $L_{\max}$ bytes per header, so the overall cost per header remains $O(L_{\max})$.

- **Masking and OR-combination:** After shifting, each slice is masked to its true length ℓ_h and OR-combined into the contiguous header buffer. Across all n_h headers, this results in $O(n_h L_{\max})$ work, with an OR-tree depth of $O(\log_2 n_h)$.

The overall complexity of the operator is therefore:

$$O(L_{\max} \log_2 W_{\text{PHV}} + n_h L_{\max})\,.$$

Although the OR-combination stage involves a reduction tree of depth $O(\log_2 n_h)$, this factor reflects the parallel logic depth rather than additional operations, and thus does not affect the overall asymptotic work complexity. This complexity grows linearly with the maximum header size $L_{\max}$ and the number of valid headers n_h, rather than quadratically with W_{PHV} as in a full crossbar. In practice, both $L_{\max}$ and n_h are much smaller than W_{PHV}, which leads to substantial reductions in LUT usage and easier timing closure on FPGAs. In the VHDL design, the phv_slicer module implements slice selection, while the phv_concat module performs re-positioning, masking, and OR-combination without requiring a wide broadcast network.

PHV Streamer

The PHV streamer serializes header chunks produced by the PHV Operator onto the output data bus. Driven by configuration fields such as the number of chunks and the start position, it emits one bus-width word per cycle. Internal counters track the progress of emissions to ensure that multi-word headers are transmitted contiguously before advancing to the next header in the schedule. Once a header is completely processed, the counters reset automatically, enabling seamless back-to-back emission of headers.

Payload Delay Buffer

After all headers in a packet are emitted, the payload must be forwarded without violating timing or alignment constraints. In our design, the payload data is buffered during the header emission cycles and then released after a programmed delay derived from the configuration memory. This controlled buffering ensures that payload emission begins precisely when the last header chunk has been transmitted, preventing data hazards and alignment errors.

Its size scales with the payload bus width and the maximum payload-delay depth derived from the longest path of the DAG:

$$M_{\text{payload}} = B \times \mathit{Delay}_{\max}.$$

Since both parameters scale linearly with protocol complexity, B is set by the target throughput requirement and $\mathit{Delay}_{\max}$ depends on the longest header chain, the resulting memory footprint also scales linearly. In terms of latency, the payload delay buffer introduces a fixed, architecture-defined memory-access delay that depends only on the characteristics of the underlying FPGA memory primitives and not on the buffer depth. Let L_{access} denote this fixed access latency (in cycles). The worst-case buffering latency, measured from the cycle at which a payload word is written into the buffer to the cycle at which it is first read out, is then:

$$L_{\text{payload delay buffer, max}} = L_{\text{write}} + Delay_{\text{max}} + L_{\text{read}}.$$

PHV–Payload Aligner

Due to variable header sizes and alignment requirements, the transition from header emission to payload emission often requires byte shifting or word realignment. The `phv_aligner` module performs this task, adjusting the starting point of the payload data so that it follows immediately after the last emitted header byte. This alignment step uses precomputed padding values from the configuration memory, ensuring zero gaps and maintaining protocol compliance without introducing dynamic shifting logic in the critical path.

Streamer Selector

The streamer_selector arbitrates between multiple possible output sources that are the PHV stream for header data and the payload buffer for payload data. Based on runtime control signals derived from the emission schedule, the selector switches the active data source at the exact cycle when the header-to-payload transition occurs. This ensures a seamless concatenation of headers and payload in the output stream, preserving throughput and avoiding redundant cycles.

Overall, this modular microarchitecture, backed by precomputed overlay control tables, eliminates wide, dynamic routing fabrics while enabling runtime flexibility. Each block has a clearly bounded function, mapped to synthesizable VHDL modules, allowing the architecture to scale to diverse P4 workloads within the defined boundary limits.

7.4.5 Deparser Packet Flow

Figure 7.6 illustrates the detailed flow of packet reconstruction in the proposed recursive-select deparser. The figure shows how headers are extracted from the PHV, realigned, padded if necessary, concatenated into a contiguous block, serialized, and finally merged seamlessly with the payload. Each subfigure highlights one step of the datapath, emphasizing how the design avoids costly dynamic crossbars while maintaining precise alignment.

Figure 7.6a: Header Extraction from PHV

The Packet Header Vector has width W_{PHV} (Bytes) and stores all parsed headers at fixed offsets s_h together with their validity bits V. The slicer module inspects the bitmap V and extracts a fixed $L_{\max}$-byte slice starting at offset s_h. If the requested window extends beyond the PHV boundary, the missing portion is zero-padded so that the output is always an $L_{\max}$-byte vector.

Figure 7.6b: Fixed-Size Chunks

Each extracted slice is represented as a fixed-size chunk of $L_{\max}$ bytes. Although the actual header length is ℓ_h, standardizing to $L_{\max}$ ensures uniform handling for all headers. These chunks may include unused or overlapping bytes, but they simplify downstream logic by enforcing a common width.

Figure 7.6c: Zero Padding and Right Alignment

To isolate only the valid portion, each slice of length $L_{\max}$ is shifted right so that the ℓ_h valid bytes of header h occupy the vector's least significant side (LSB). The remaining $L_{\max} - \ell_h$ positions are zero-padded. Formally,

$$L_h = \Big(\text{Slice}(s_h, L_{\max}) \gg (L_{\max} - \ell_h)\Big) \ \&\ \text{Mask}(\ell_h).$$

Figure 7.6d: Contiguous Header Reconstruction

Each header h is placed at its absolute output offset o_h, which is computed as the cumulative length of all previously emitted headers:

$$o_h = \sum_{k<h} \ell_k.$$

The placement is achieved by shifting the MSB-aligned slice L_h by this offset:

$$S_h = L_h \ll o_h.$$

This formulation is consistent with the shift distance definition introduced in Section 7.4.4, where

$$\delta_h = o_h - \text{slice\_MSB\_position}.$$

In the cumulative-offset view of Figure 7.6d, the slice is already MSB-aligned, so the displacement reduces to $\delta_h = o_h$.

Finally, all shifted slices are combined with a bitwise OR to form the contiguous header block:

$$H_{\text{hdr}} = \bigvee_{h=1}^{n_h} S_h,$$

where $n_h = \text{popcount}(\mathbf{V})$ is the number of valid headers in the path.

Figure 7.6e: Streaming of Header Block

The PHV_streamer serializes H_{hdr} into bus-width words of B bytes, one per cycle. The number of header cycles and the corresponding payload delay follow directly from the derivations in Section 7.4.4.

Figure 7.6f: Header–Payload Handoff

At the boundary, the payload_aligner applies a precomputed offset so that the first payload bytes fill any unused space in the final header word. The exact expressions for residual bytes and alignment (T_{hdr} and *Alignment*) are given in Section 7.4.4.

Overall, Figure 7.6 demonstrates the recursive-select execution model: the deparser iterates through valid headers, extracts them at offsets s_h, right-aligns them to length ℓ_h (producing L_h), shifts them to their output offsets o_h (producing S_h), OR-combines them into H_{hdr}, streams bus words through the `PHV_streamer`, and finally appends the payload with alignment as derived in Section 7.4.4. By replacing dynamic crossbars with slice-and-shift logic guided by overlay control metadata, the architecture scales linearly with header size $L_{\max}$ and header count n_h, while sustaining one bus-width word per cycle at runtime.

7.4.6 Templated Hardware Architecture

The templated architecture is introduced as a compile-time variant of the overlay. It preserves the same recursive-select datapath and header reconstruction pipeline but removes runtime configurability by eliminating the configuration memory and its associated control interface. Instead of storing per-path control information in memory, it replaces these contents with VHDL generics that are resolved during synthesis. This approach preserves protocol independence within predefined template bounds while binding control parameters statically. The templated architecture is motivated by ease of regeneration and design stability. Unlike

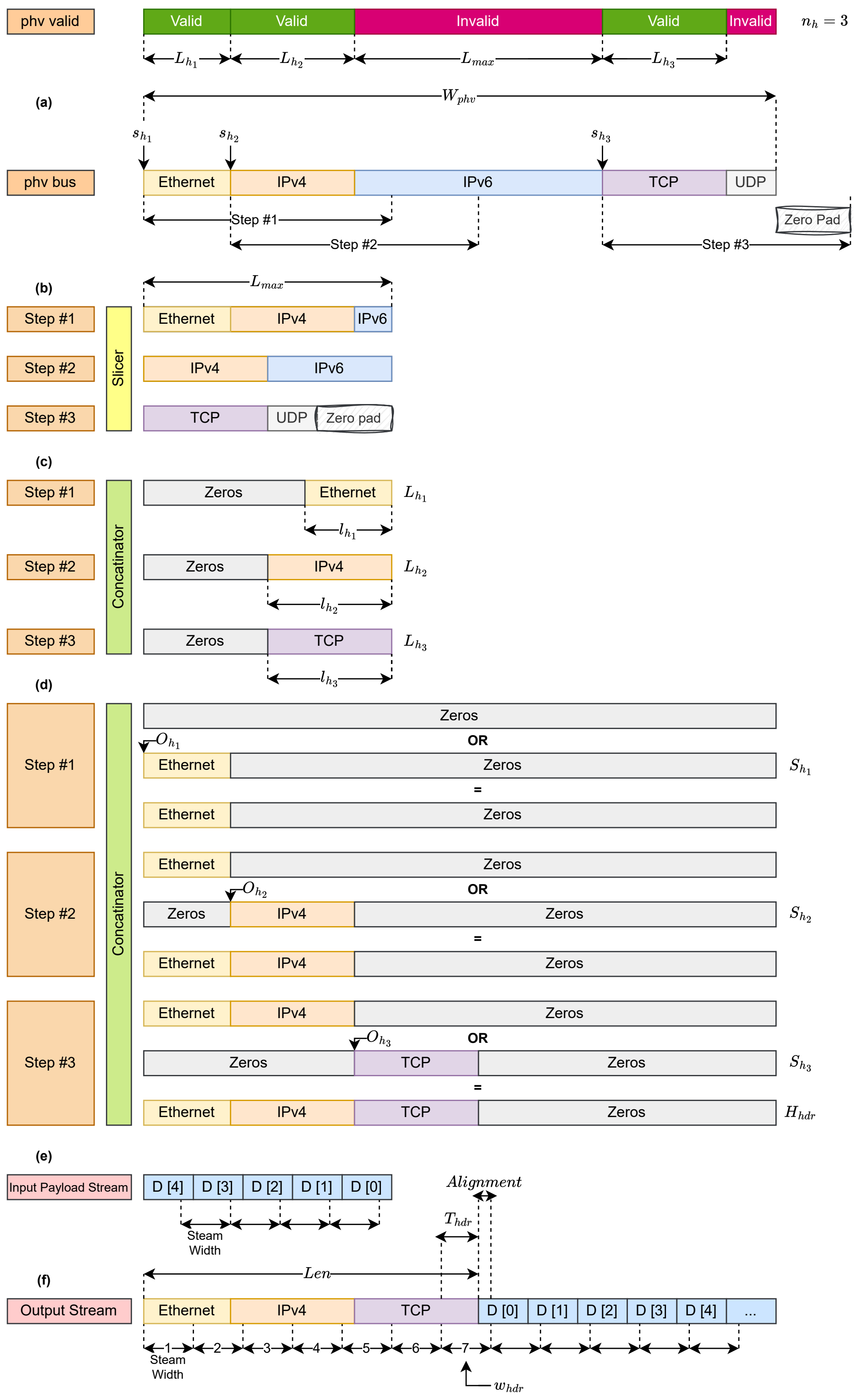


Figure 7.6 Deparser Packet Flow.

compiler-specialized approaches that regenerate RTL for every P4 program, the templated variant maintains a fixed hardware structure and avoids automatic code generation. This makes regeneration significantly simpler from a design and usability perspective, because only generic parameters are updated rather than regenerating tailored RTL. Although it still requires resynthesis, it avoids build-chain complexity and provides a reusable and maintainable hardware implementation.

For simpler protocol graphs, the templated variant uses fewer resources compared to the overlay since it removes the configuration memory and runtime selection logic. However, as protocol graphs grow in complexity, the number of valid header combinations increases and the control logic must explicitly encode these combinations. This creates a statically synthesized decision tree that expands proportionally to protocol variability. Since this decision structure grows in logic rather than memory, LUT usage and routing complexity increase more rapidly than in the overlay. The overlay, by contrast, reserves hardware based on template bounds and stores per-path behavior in memory. This introduces modest fixed overhead for small designs, but its advantage becomes clear as protocol complexity increases. Since additional protocol combinations only expand memory contents rather than logic, growth remains more predictable. The templated design therefore serves as an intermediate option: easier to regenerate than compiler-specialized RTL and lighter than an overlay for simple configurations, but increasingly inefficient as the per-path decision tree grows with protocol complexity.

7.5 Results

This section presents the results obtained with the proposed recursive-select and templated overlay deparser. We begin by describing the experimental setup and the protocol stacks under test. We then analyze FPGA resource scaling to validate the theoretical complexity model and to show how the slice-and-shift operator eliminates the rigidity of crossbars. Next, we compare our results with those reported for previously published specialized deparsers. Finally, we demonstrate runtime programmability by reconfiguring the synthesized hardware to operate with the Access Gateway Function (AGF) stack's DAG as described [90]. To evaluate scalability beyond isolated bus-width or protocol demonstrations, we deliberately stress the architecture across increasing graph complexity, wider buses, and growing PHV sizes. This section shows that resource usage is governed primarily by protocol depth rather than PHV width, confirming that the recursive-select overlay scales predictably even as protocol graphs grow in size and heterogeneity.

The design was synthesized on a Xilinx Virtex UltraScale+ (XCVU3P) device, selected for both technical and methodological reasons. The same FPGA family has been used in prior deparser studies, enabling fair and direct comparison with established baselines. The XCVU3P is also widely deployed in industrial FPGA accelerators making it a practical and representative target for P4-programmable data-plane research. In addition, its balanced mix of LUTs, flip-flops, BRAM, and high-speed transceivers provides ample resources to implement both the templated and overlay variants without routing congestion or resource saturation. To evaluate the design, throughput is obtained from post-synthesis and post-implementation results, using the standard Fmax $\times$ bus width calculation followed in prior FPGA deparser studies. Functional validation is ensured through an automated Python 3.12.-driven VHDL simulation script that exercises all valid header paths, PHV_valid patterns, and supported bus-width configurations, and validates each scenario to ensure complete coverage of header ordering.

We evaluated three protocol stacks of growing complexity:

- **T1:** Ethernet, IPv4/IPv6, TCP/UDP
- **T2:** Ethernet, IPv4/IPv6, TCP/UDP, ICMP/ICMPv6
- **T3:** Ethernet, double VLAN, double MPLS, IPv4/IPv6, TCP/UDP, ICMP/ICMPv6

For each stack, boundary overlay parameters were derived from the deparser DAG and enforced at synthesis time. These include the maximum number of headers H_{max}, the maximum per-header size L_{max}, the maximum depth, and the total PHV footprint and bus width. Concretely, T1 required $H_{\text{max}} = 5$, $L_{\text{max}} = 40$ Bytes with a PHV footprint ($W_{\text{PHV,max}}$) of 102 Bytes; T2 expanded to $H_{\text{max}} = 7$, $L_{\text{max}} = 40$ Bytes with a PHV footprint of 110 Bytes; and T3 exercised the upper bound with $H_{\text{max}} = 11$, $L_{\text{max}} = 40$ Bytes and a PHV footprint of 126 Bytes, as shown in Figure 7.1d. Within these template bounds, runtime adaptation of emission order, header offsets, and alignment is achieved entirely through overlay configuration memory.

In addition to resource scaling, timing behavior is a key factor in evaluating deparser performance. While graph complexity and bus width influence logic growth, they also determine the pipeline depth and overall emission latency. The proposed overlay architecture introduces a fixed baseline latency of nine clock cycles, plus the maximum number of iterations required for recursive selection, five in the most complex case (T3). This latency originates from additional stages introduced by the recursive-select architecture, including a two-cycle configuration-memory fetch, slice selection and shift alignment performed by the PHV_operator, sequencing of the recursive loop and streamer activation by the control logic, and the payload-delay and aligner stages that ensure correct emission order. After this offset, the latency depends on the total number of header bytes relative to the bus width. The worst-case packet emission delay is therefore expressed as:

$$\text{Latency} = \left\lceil \frac{\text{Total header length}}{\text{Bus width}} \right\rceil + N_{\text{sel}} + 9, \tag{7.3}$$

With wider buses, more header bytes are emitted per cycle, reducing the variable component of the latency compared to narrower datapaths.

To provide a complete view of system behaviour, the memory characteristics of the overlay were assessed alongside the pipeline-latency analysis. Table 7.4 reports, for T1–T3 configuration, the configuration-memory footprint $M_{\text{config memory}}$, the payload-delay buffer size $M_{\text{payload delay buffer}}$, the corresponding BRAM usage on the XCVU3P device, and the worst-case latencies derived from the models in Sections 7.3.1 and 7.4.4. For all evaluated stacks, both memories exhibit fixed, device-characterized read and write latencies. The configuration memory requires one write cycle to update an entry and two read cycle to fetch it during operation. At the same time, the payload-delay buffer introduces the same fixed access delay for both read and write operations, as determined by the underlying UltraScale+ memory primitives.

Since each configuration entry is written in exactly one cycle, the total time required to reprogram the configuration memory grows linearly with the number of valid paths extracted from the deparser DAG. The worst-case programming time is therefore proportional to the number of entries. On UltraScale+ devices, both the configuration memory and the payload-delay buffer are implemented using distributed BRAM primitives, which internally consist of paired BRAM18 blocks forming logical BRAM36 units. This structure allows the memories to be mapped efficiently onto fixed-width BRAM tiles, with each table or buffer aligned to the native port widths of the underlying blocks.

To provide a quantitative view of memory occupancy, the BRAM usage is expressed as a percentage of device resources. The XCVU3P device contains 720 BRAM36 blocks in total. For the smallest configuration, corresponding to the T1 stack at 64 bits, the overlay architecture consumes 2.5 BRAM36 blocks and the templated design consumes 1 BRAM36 block, which corresponds to 0.35 percent and 0.14 percent of the available BRAM resources. For the largest evaluated configuration, corresponding to T3 at 512 bits, the templated design requires 7.5 BRAM36 blocks and the overlay requires 9.5 BRAM36 blocks, corresponding to 1.05 percent and 1.33 percent of the device capacity. These values confirm that even at the highest complexity levels the memory footprint of both designs remains very small relative to

Table 7.4 Configuration and payload memory characteristics for different bus widths.

Test	Bus Width (Bits)	M_{config} (Bits)	M_{payload} (Bits)	BRAM (Config + Payload)	Read Latency (Cycles)	Write Latency (Cycles)	Programming Time (Cycles)	Programming Time (ns)
T1	64	80×7	64×32	$1.5 + 1$	2	1	8	13
	128	80×7	128×32	$1.5 + 2$	2	1	8	14
	256	80×7	256×32	$1.5 + 4$	2	1	8	14
	512	80×7	512×32	$1.5 + 7.5$	2	1	8	14
T2	64	80×9	64×32	$1.5 + 1$	2	1	10	13.5
	128	80×9	128×32	$1.5 + 2$	2	1	10	14
	256	80×9	256×32	$1.5 + 4$	2	1	10	13
	512	80×9	512×32	$1.5 + 7.5$	2	1	10	14
T3	64	126×33	64×32	$2 + 1$	2	1	34	54
	128	126×33	128×32	$2 + 2$	2	1	34	54
	256	126×33	256×32	$2 + 4$	2	1	34	54
	512	126×33	512×32	$2 + 7.5$	2	1	34	59

the available resources. For the proposed architecture, bandwidth limitations are not imposed by either memory components, since both operate well within the capacity of the underlying BRAM primitives. The configuration memory is written in a single cycle per entry during reprogramming and is read with a fixed two-cycle latency during operation. The payload delay buffer is accessed once per cycle at the system clock frequency, with one bus-width word issued sequentially on every cycle, and its access pattern remains strictly linear and free of contention. Since both memories are operating at the same frequency as the deparser pipeline and are only used for control lookups and streaming, their bandwidth demands remain modest, and no bottlenecks arise for any of the evaluated bus widths. These characteristics, along with the resulting memory footprints, latencies, and programming times for each test stack, are summarized in Table 7.4. The table further reports the measured overlay reconfiguration latency, which ranges from approximately 13 ns for the smallest evaluated configuration T1 graph to 59 ns for the largest configuration graph T3.

The synthesis data is demonstrated in Table 7.5 and implementation data is shown in Table 7.6. For the overlay, LUT and FF usage remain below 15 k even for T3 with bus size of 512 while sustaining ≈296 Gb/s, whereas the templated variant surpasses this value as the graph expands. This behavior confirms that the recursive-select overlay preserves throughput while providing more predictable, balanced resource scaling. Table 7.5 compares the recursive-select framework, including both the templated and overlay implementations, against representative FPGA deparsers. Compared with the bit-selector pipeline of Luinaud [68], which hardwires concatenation schedules at compile time, both variants sustain higher throughput within a comparable area budget. For T2 with bus size of 256, the templated deparser achieves up to 1.45× higher throughput (≈174 Gb/s vs. 120 Gb/s), while the overlay maintains 1.2–1.4× improvement at modest additional logic cost. Against the symbolic specialization approach of Luinaud [87], which minimizes area by entirely eliminating runtime control, the templated and overlay deparsers exhibit 1.6–10× higher LUT usage but retain competitive throughput (up to ≈296 Gb/s for T3 at bus size of 512 vs. 311 Gb/s in [87]) while uniquely supporting dynamic reconfiguration. The key distinction is that both proposed designs are reusable across multiple P4 workloads, whereas compiler-specialized pipelines must be regenerated for each configuration. The recursive-select architectures more than double this throughput without relying on replicated or time-multiplexed datapaths. The slice-and-shift datapath used in both the templated and overlay variants achieves comparable throughput and scalability with significantly lower structural complexity.

Implementation-level results in Table 7.6 reinforce these findings for a fixed 512-bit datapath. The overlay sustains approximately 232–254 Gb/s after place-and-route, while synthesis estimates in Table 7.5 reach about 296 Gb/s. This difference reflects timing closure at implementation and remains well above prior work. LUT usage stays below 15 k and BRAM utilization under 10.5 blocks, outperforming Xilinx SDNet [55]. The deparser generated with Xilinx SDNet consumes ~8× more LUTs and ~10× more FFs than the deparser generated in this work. The templated design achieves slightly better timing closure for the simpler graphs (T1–T2). Still, it exhibits steeper resource growth for T3 due to its fully unrolled structure, confirming that overlay-based configurability scales more efficiently with increasing graph complexity. Compared with Luinaud [68], which attains 140–220 Gb/s with larger LUT footprints and higher BRAM usage, the two proposed designs reach higher throughput with reduced reliance on wide multiplexers and deep buffers. Compared with Luinaud [87], the overlay sustains slightly lower peak throughput for T3 (approximately 296 Gb/s synthesis versus 310 Gb/s reported) but remains fully reconfigurable at runtime, a capability absent from compile-time-specialized designs.

The observed scalability behavior is further quantified using the normalized LUT/Gbps metric in Table 7.6, which shows that the overlay design maintains an almost constant cost across all protocol stacks (57.5–61.7 LUT/Gbps from T1 to T3) while providing full runtime programmability. For the most complex configuration (T3), this corresponds to approximately twice the normalized cost of the specialized and optimized symbolic design of Luinaud et al. [87] (≈30 LUT/Gbps). This remains lower than the compile-time specialized deparser of Luinaud et al. [68] (≈100 LUT/Gbps), and is more than an order of magnitude lower than the results obtained with the Xilinx SDNet baseline [55] (>600 LUT/Gbps). The templated variant spans 32.8–79.2 LUT/Gbps, achieving competitive efficiency for simpler graphs and increasing for deeper DAGs due to static unrolling. To further validate runtime programmability while stressing scalability beyond T3, we evaluated the overlay architecture using a more demanding configuration derived from T3 by augmenting it with IPv6 extension headers. This extended configuration preserved the same maximum per-header size (40 bytes) but increased structural complexity to 13 total nodes, 7 conditional transitions, and a PHV footprint of 142 bytes. Table 7.7 summarizes the synthesis results for this configuration. Since the overlay is synthesized once for the most demanding configuration within its template bounds, it naturally accommodates any subgraph without resynthesis. In practice, this means that a single bitstream generated for the extended T3 configuration can also execute T1, T2, and T3 by simply loading their respective configuration memory at runtime. This downward compatibility is inherent to the overlay approach meaning that when synthesizing for the worst-case graph, it can safely covers all simpler protocol stacks within the same design envelope.

Table 7.5 Comparison of FPGA resource usage and throughput across bus widths for the proposed recursive-select deparser and prior FPGA deparsers.

Test	**Bus Width (bits)**	**Work**	**Worst Latency (cycles)**	**LUTs**	**FFs**	**BRAMs**	**Freq. (MHz)**	**Throughput (Gbps)**
T1	64	Proposed Overlay	25	6,032	5,414	2.5	613	39.2
		Proposed Templated	23	4,458	6,011	1.0	579	37.0
		Luinaud et al. [68]	19	1,517	402	0	448	28.7
		Luinaud et al. [87]	NA	614	397	0	740	47.3
	128	Proposed Overlay	19	6,838	5,740	3.5	562	71.9
		Proposed Templated	17	4,666	6,334	2.0	578	73.9
		Luinaud et al. [68]	13	2,066	784	0	448	57.3
		Luinaud et al. [87]	NA	NA	NA	NA	NA	NA
	256	Proposed Overlay	16	8,503	6,424	5.5	568	145.4
		Proposed Templated	14	5,946	6,974	4.0	572	146.4
		Luinaud et al. [68]	10	2,862	1,522	0	448	114.7
		Luinaud et al. [87]	NA	1,557	1,373	0	684	175.0
	512	Proposed Overlay	14	12,994	7,763	9.0	577	295.4
		Proposed Templated	12	9,690	8,256	7.5	566	289.9
		Luinaud et al. [68]	8	9,127	3,002	0	469	240.1
		Luinaud et al. [87]	NA	5,489	2,690	0	628	321.5
T2	64	Proposed Overlay	26	7,749	7,461	2.5	613	39.2
		Proposed Templated	24	4,504	6,606	1.0	566	36.2
		Luinaud et al. [68]	20	1,798	404	0	448	28.7
		Luinaud et al. [87]	NA	718	402	0	740	47.3
	128	Proposed Overlay	19	8,573	7,780	3.5	576	73.7
		Proposed Templated	17	4,742	6,523	2.0	717	91.8
		Luinaud et al. [68]	13	2,664	808	0	448	57.3
		Luinaud et al. [87]	NA	NA	NA	NA	NA	NA
	256	Proposed Overlay	16	10,369	8,451	5.5	653	167.2
		Proposed Templated	14	6,102	7,187	4.0	679	173.9
		Luinaud et al. [68]	10	4,879	1,602	0	469	120.1
		Luinaud et al. [87]	NA	1,514	1,378	0	662	169.4
	512	Proposed Overlay	14	14,623	9,785	9.0	579	296.4
		Proposed Templated	12	10,052	9,035	7.5	542	277.8
		Luinaud et al. [68]	8	11,212	3,197	0	448	240.1
		Luinaud et al. [87]	NA	5,552	2,693	0	625	320.0
T3	64	Proposed Overlay	30	8,045	8,464	3.5	606	38.8
		Proposed Templated	28	7,781	7,563	1.0	394	25.2
		Luinaud et al. [68]	22	2,137	390	6	448	28.7
		Luinaud et al. [87]	NA	1,053	375	0	666	42.6
	128	Proposed Overlay	22	8,835	8,783	4.5	625	80.0
		Proposed Templated	20	8,483	7,879	2.0	424	53.4
		Luinaud et al. [68]	14	3,962	780	14	448	57.3
		Luinaud et al. [87]	NA	NA	NA	NA	NA	NA
	256	Proposed Overlay	18	10,435	9,447	6.5	619	158.5
		Proposed Templated	16	11,191	8,544	4.0	417	106.9
		Luinaud et al. [68]	10	6,598	1,525	32	448	114.7
		Luinaud et al. [87]	NA	3,452	1,507	0	662	169.4
	512	Proposed Overlay	16	14,966	10,775	10.0	579	296.4
		Proposed Templated	14	16,273	9,816	7.5	397	203.0
		Luinaud et al. [68]	8	14,287	3,116	41	469	240.1
		Luinaud et al. [87]	NA	9,400	3,130	0	609	311.8

Table 7.6 Comparison of Deparser Implementation with Previous Work with an Output Bus of 512 Bits.

Code	Work	Slices	LUTs	FFs	BRAMs	Throughput (Gbps)	LUTs /Gbps	Runtime Programmability
T1	Proposed Overlay	N/A	13.8k	9.8k	9	232.4	59.4	Yes
	Proposed Templated	N/A	9.4k	8.5k	7.5	244.6	38.4	No
	Xilinx SDnet [55]	N/A	78k	95k	116.5	256	304.7	No
	Luinaud et al. [68]	3.1k	9k	3k	0	200	45.0	No
	Luinaud et al. [87]	N/A	N/A	N/A	N/A	N/A	N/A	No
T2	Proposed Overlay	N/A	14.6k	10.3k	9	253.8	57.5	Yes
	Proposed Templated	N/A	9.4k	9.4k	7.5	286.3	32.8	No
	Xilinx SDnet [55]	N/A	98k	119k	149.5	240	408.3	No
	Luinaud et al. [68]	3.9k	11.2k	3.2k	0	220	50.9	No
	Luinaud et al. [87]	2.2k	5.6k	2.7k	0	320	17.5	No
T3	Proposed Overlay	N/A	14.9k	11.3k	9.5	241.3	61.7	Yes
	Proposed Templated	N/A	16.1k	10.3k	7.5	203.3	79.2	No
	Xilinx SDnet [55]	N/A	139k	165k	229.5	220	631.8	No
	Luinaud et al. [68]	4.7k	14k	3k	20.5	140	100.0	No
	Luinaud et al. [87]	2.5k	9.4k	3.2k	0	310	30.3	No

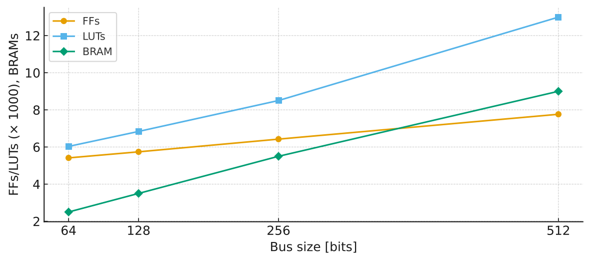
Table 7.7 Synthesis/implementation results for the Extended T3 configuration under the proposed overlay design.

Bus Width (Bits)	LUTs	FFs	BRAMs	Throughput (Gbps)
64	9664	9182	6.5	39.3
128	9779	9482	7.5	74.4
256	12,169	10,176	9.5	143.9
512	16,350	11,404	13	290

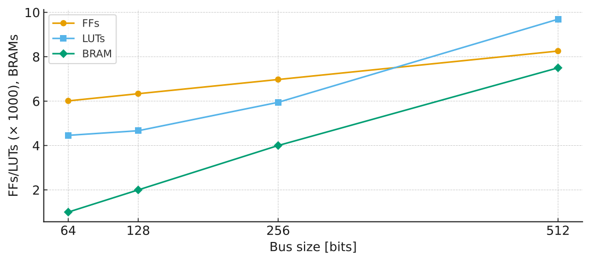
Furthermore, runtime programmability is not limited to protocol subgraphs from the same stack family. As demonstrated using the Access Gateway Function (AGF) stack [90], which consists of 10 headers (Ethernet, IPv4, 802.1Q, 802.1AD, UDP, GTP-U, 5WE, PPPoE_V1/V2, PPPoE_V1, and the PDU Session Container) with a maximum header size of 20 bytes and 7 transitioning nodes, the overlay can be repurposed for entirely different protocol graphs. Only the configuration memory contents are regenerated to encode AGF-specific emission order, header offsets, and alignment, while the datapath remains unchanged. This demonstrates horizontal adaptability across heterogeneous protocol applications. Together, these results confirm that the overlay enables hardware reuse across both sub-stacks and new protocol configurations without resynthesis, validating its ability to support diverse P4 workloads using a single bitstream. To reprogram the overlay for a new protocol stack, only the configuration memory is updated, without modifying or resynthesizing the hardware. The P4 JSON description of the new program is analyzed to extract all legal emission paths and compute the associated control parameters. For each validity bitmap, one configuration entry is produced that contains $H_{\max}$ per-header fields: the PHV start offsets $\mathbf{s} = \{s_0, s_1, \ldots, s_{H_{\max}-1}\}$, the header byte lengths $\boldsymbol{\ell} = \{\ell_0, \ell_1, \ldots, \ell_{H_{\max}-1}\}$, the output offsets $\mathbf{o} = \{o_0, o_1, \ldots, o_{H_{\max}-1}\}$, and the number of bus-width chunks $\mathbf{W} = \{W_0, W_1, \ldots, W_{H_{\max}-1}\}$. Each entry also includes path-level metadata: the total header length *Len*, the payload alignment *Alignment*, the payload delay *Delay*, and the number of valid headers $n_h = \text{popcount}(\textit{bitmap})$.

Figure 7.7 evaluates the impact of increasing the output bus width from 64 to 512 bits for both proposed overlay and templated architectures. while compile-time–specialized designs [87] have reported a significant increase in resource usage when widening the output bus including LUT usage for the T3 configuration increases from 1.05 k at 64 bits to 9.40 k at 512 bits and the FF usage increases from 375 to 3.13 k, corresponding to an 8.9× and 8.3× increase respectively for an eightfold bus-width expansion. In contrast, the proposed recursive-select architecture exhibits a much smaller growth in resource usage for the same increase in bus width, LUT utilization increases from 8.05 k to 14.96 k and FF utilization increases from 8.46 k to 10.77 k, corresponding to only 1.9× and 1.3× growth respectively.

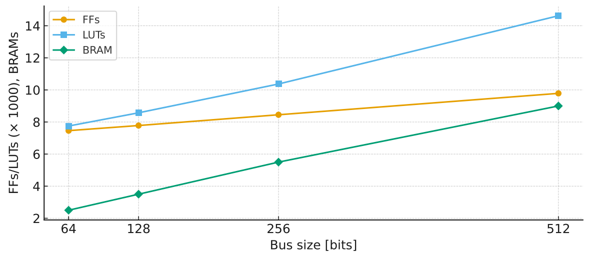
Figure 7.8 evaluates the scaling behavior of the proposed overlay architecture with respect to protocol complexity and PHV width. Figure 7.8a isolates the effect of increasing the number of supported protocol headers while fixing the output bus width at 512 bits. The results show that LUT and FF utilization grow approximately linearly with the number of headers n_h. For example, increasing protocol depth from T1 ($n_h = 5$) to T3 ($n_h = 11$) raises LUT usage from 12.99 k to 14.97 k and FF usage from 7.76 k to 10.78 k, corresponding to only 15% and 38% growth respectively. This confirms that resource usage is primarily dominated by header count, consistent with the analytical complexity term $O(n_h L_{\max})$ derived in Section 7.4.4. In contrast, compile-time–specialized designs must replicate logic for each protocol path and therefore scale poorly as protocol graphs grow. Figure 7.8b evaluates the sensitivity of the architecture to PHV width W_{PHV}, which reflects the total header footprint extracted from the parser. Resource usage increases only slightly as W_{PHV} grows from 102 bytes (T1) to 142 bytes (Extended T3), with LUT usage increasing from 12.99 k to 16.35 k (26%) and FF usage from 7.76 k to 11.40 k (47%). This mild growth validates the weak logarithmic dependency predicted by the $O(L_{\max} \log_2 W_{\text{PHV}})$ term. Unlike byte-level crossbar implementations, which exhibit quadratic growth with PHV width, the recursive-select architecture uses slice extraction and shift-based alignment, avoiding wide multiplexers and routing congestion. Overall, Figure 7.8 shows that protocol depth is the dominant scaling factor, while PHV width has a secondary and controlled impact on FPGA resource usage. These results highlight that scaling trends are dominated by control complexity rather than datapath width, which motivates a closer comparison between the overlay and templated implementations, since the templated version encodes control logic statically while the overlay stores it compactly in memory and therefore scales more efficiently for larger protocol graphs.



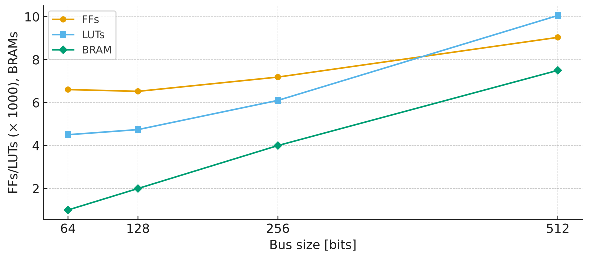
(a) T1 Overlay.



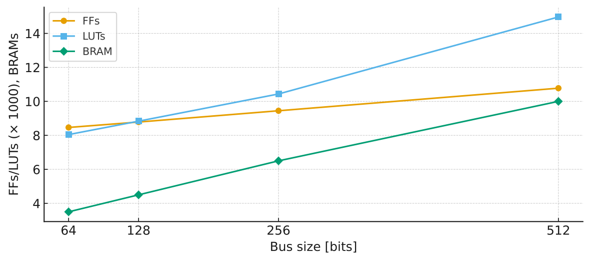
(b) T1 Templated.



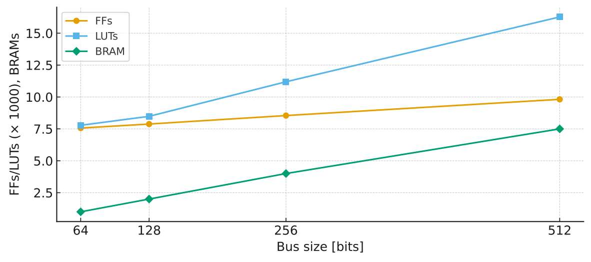
(c) T2 Overlay.



(d) T2 Templated.

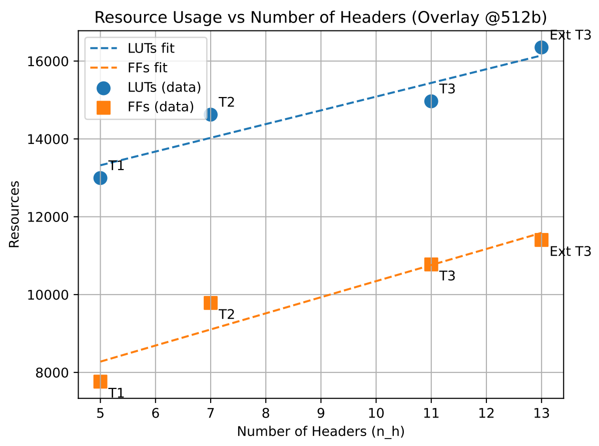


(e) T3 Overlay.

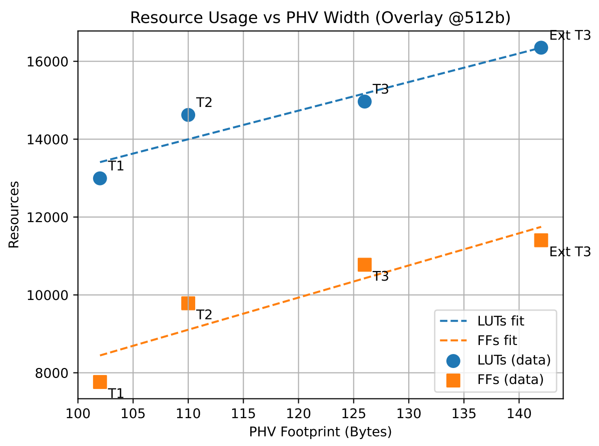


(f) T3 Templated.

Figure 7.7 FPGA resource utilization across bus widths of 64–512 bits for test graphs T1–T3 under the proposed overlay and templated architectures. Each subplot reports LUT, FF, and BRAM scaling with bus size.



(a) Resource usage vs protocol headers



(b) Resource usage vs PHV width

Figure 7.8 Resource scaling behavior of the proposed overlay architecture: (**a**) variation with increasing number of protocol headers; (**b**) variation with PHV width.

Compiler-specific deparsers require complete RTL regeneration followed by synthesis and place-and-route whenever the protocol graph changes. On contemporary FPGA toolflows, these steps often incur tens of minutes to several hours, which limits their practicality in environments with evolving protocols or multiple coexisting applications. In contrast, the proposed overlay deparser applies behavioral changes through lightweight updates to configuration memory, completing in nanoseconds without halting the datapath or regenerating hardware. Although this generic architecture introduces a higher initial cost than a specialized deparser for very simple graphs, its hardware complexity scales far more favorably as protocol structures grow. The slice-and-shift operator increases approximately linearly with the number of headers and only weakly with the PHV width, while crossbar permutation-engine designs incur $O(W_{\text{PHV}}^2)$ complexity and compiler-specialized deparsers scale with $O(n_h \cdot W_{\text{PHV}})$. As a result, the overlay grows with a smaller slope. At the same time, the initial gap is noticeable for T1-level graphs, but narrows for T2 and T3, confirming that the proposed design becomes increasingly efficient alternative as protocol graphs deepen and PHV widths expand.

Taken together, these results show that the recursive-select framework redefines the performance-versus-programmability boundary for FPGA deparsers. For small and moderately complex graphs (T1–T2), the templated implementation achieves slightly lower resource usage than overlay by eliminating configuration memory and dynamic control. However, as graph complexity increases, the static unrolling in templated and compiler-specialized designs results in rapid logic replication and timing closure stress. Unlike the templated variant, whose logic footprint grows rapidly with graph complexity, the overlay exhibits a much flatter resource growth trend because additional protocol paths expand memory contents rather than LUT logic. This makes it increasingly advantageous for deeper and more diverse protocol graphs. This trend is confirmed by Table 7.7, where the overlay handles an extended T3 configuration (13 headers, 7 transitions) with only a modest increase in LUTs and FFs, while remaining within timing and BRAM limits. For the compiler-driven specialization the logic must be regenerated for every protocol stack change and scales more steeply beyond fixed header pipelines, while the overlay sustains predictable linear growth and reusability from T1 through extended T3 and the latter can be used for all other sub-graphs without resynthesis. These results indicate that the overlay is advantageous in scenarios where protocol configurations evolve, offering reusable hardware with controlled resource growth. In contrast, static and specialized designs must be regenerated and scale less predictably.

7.6 Conclusions

This work presented a unified hardware–software framework for FPGA-based P4 deparsers that bridges compile-time specialization and runtime reconfigurability. The proposed recursive-select architecture replaces wide dynamic crossbars with a slice-and-shift datapath, achieving near-linear computational complexity and predictable scalability as protocol depth and datapath width increase. Two architecture variants were introduced: a templated deparser, optimized for static deployments with fixed protocol graphs, and an overlay deparser that stores per-path behavior in configuration memory to enable runtime adaptation. Reported results show that for small to moderate protocol graphs (T1–T2), the templated implementation achieves a lower area footprint and slightly higher clock frequency due to the absence of configuration memory. However, as graph complexity increases (T3 and beyond), compiler-specialized and static approaches exhibit steeper growth in logic utilization and face timing closure challenges due to logic replication. In contrast, the overlay maintains a flatter resource scaling trend, sustaining line-rate throughput up to 296 Gb/s (synthesis) and 241–254 Gb/s (implementation) on a 512-bit datapath while remaining within modest LUT and BRAM budgets. These findings validate the analytical complexity model $O(L_{\max} \log_2 W_{\text{PHV}} + n_h L_{\max})$ and confirm that runtime-configurable deparsing can be achieved without crossbar switching. Compiler-specialized deparsers remain programmable but require complete regeneration of RTL, synthesis, and place-and-route for every protocol update. This limits their practicality when protocol stacks evolve or when multiple P4 applications must share FPGA resources. In contrast, the recursive-select overlay accepts a modest area overhead and a slight reduction in maximum clock frequency compared to fully specialized designs in exchange for long-term flexibility: protocol behavior is defined in configuration memory, allowing in-field updates without modifying hardware or interrupting service. This capability enables hardware multitenancy, where different tenants or applications may load distinct protocol configurations on the same FPGA fabric.

This work opens several avenues for future research. A direct extension of the framework is the construction of fully overlay-defined PISA pipelines, where the parser, match–action, and deparser stages are all controlled by configuration-memory tables, enabling end-to-end pipeline reconfiguration without regenerating RTL. The overlay approach is well-suited to dynamic, multi-tenant data-plane designs, allowing each tenant to operate with distinct protocol stacks, header formats, or service policies. Such capabilities align closely with emerging network-slicing requirements, where multiple virtual network instances must coexist on shared FPGA resources. Another avenue for advancement involves integrating the overlay design with FPGA partial reconfiguration so that only the affected region of the pipeline is updated when protocol-graph limits or header layouts change, eliminating the need for full-design resynthesis.

Overall, this work demonstrates that high performance and runtime programmability can be co-designed in FPGA data planes without sacrificing scalability. Beyond the deparser itself, the overlay approach aligns with the long-term goal of realizing a fully programmable PISA pipeline on FPGA, where the parser, match-action, and deparser stages are all overlay-defined and reconfigurable at runtime. Such architectures enable evolvable packet processing, rapid protocol deployment, and efficient FPGA sharing across heterogeneous workloads, forming a practical foundation for future reconfigurable network systems.

CHAPTER 8 GENERAL DISCUSSION

8.1 Chapter Overview

Programmable data planes constitute a central pillar of modern networking systems, yet delivering both flexibility and line-rate performance on reconfigurable hardware remains challenging. The motivation for this thesis stems from the observation that, although P4 and the PISA architecture introduced protocol-agnostic packet processing on programmable devices, the FPGA domain has primarily relied on compile-time specialization or translation pipelines that regenerate hardware for each P4 program, thereby preventing rapid protocol evolution and imposing significant synthesis overhead. This chapter discusses how the contributions of this work address these challenges, interprets the results obtained across the proposed templated and overlay architectures, situates them within the broader literature, evaluates how the original research objectives have been achieved, and reflects on the implications, limitations, and avenues for further research.

8.2 Synthesis of Contributions

8.2.1 Template-Based Architectures for High-Performance Parsing

Previous approaches to FPGA-based P4 parsing largely depended on tool-generated pipelines or vendor-specific translation frameworks, which constrained portability and forced designers to adopt compiler-driven control flows. Early P4-to-hardware systems such as the P4-to-VHDL compiler [16] and HLS-based translation strategies such as [58] demonstrated that high-throughput P4 header parsers could be realized on reconfigurable logic, yet both approaches required full regeneration of RTL or intermediate C++ code when protocol structures evolved. As a result, hardware recompilation remained tightly coupled to the development workflow, limiting design agility and tying architectural evolution to the availability of synthesis cycles and the toolchain.

High-performance parser architectures, such as the matrix-parallel design in [71], extended P4-to-hardware compilation by introducing structured datapath parallelism capable of sustaining terabit-class throughput. Their approach scales throughput by instantiating multiple protocol analyzers and widening the datapath, thereby processing several headers and multiple packets per cycle. While this achieves exceptional line-rate performance, it does so through proportional growth in logic resources and routing complexity as protocol sets and throughput targets increase, and protocol behavior remains fixed at synthesis time.

In contrast, the templated parser introduced in this research preserves deterministic, bit-accurate performance while eliminating the need for manual RTL redesign, relying instead on P4-derived parameters and configuration vectors to adapt to different protocol graphs. This maintains hardware efficiency without binding the architecture to a specific compiler backend or introducing instruction-driven control, providing a more flexible and portable foundation for P4-programmable FPGA parsing.

8.2.2 Overlay Architectures for Runtime Reconfigurability

The transition from compile-time templating to runtime overlays represents a critical step, as overlays enable protocol transition rules, header offsets, and extraction lengths to reside in memory, allowing for behavioral modification without impacting the RTL. Early table-driven parser overlays demonstrated the feasibility of decoupling parser behavior from hardware logic. Still, they relied on micro-instruction execution pipelines that introduced fetch-and-decode latency and limited deterministic performance [62]. The first overlay parser developed in this work validated the ability to externalize P4 parse-graph behavior into configuration memory and support runtime reconfiguration with a compact hardware footprint. However, its throughput remained modest compared to fully specialized pipelines, reflecting the inherent trade-off between flexibility and deep pipelining also observed in prior programmable or instruction-oriented data-plane designs.

Subsequent refinement through the PrismParser framework addressed these limitations by introducing balanced pipelining and shortening critical paths, enabling higher operating frequency while preserving deterministic flow control, and automating table generation. These advances preserved cycle-accurate behavior without introducing microcoded engines or instruction decoding, enabling multi-hundred-gigabit performance across diverse P4 protocol graphs while retaining runtime reconfigurability through table-driven control. The framework introduced three complementary design paths: a lightweight base architecture that validates the template and provides a compact static datapath, an overlay architecture that enables protocol updates through configuration tables rather than RTL changes while supporting different bus widths, and a pipelined static architecture that applies stage unrolling and parallel state evaluation to increase maximum operating frequency and throughput. While PrismParser significantly improves performance and area efficiency while supporting dynamic protocol evolution, its peak throughput remains below that of the fully templated parser in [59], which achieves terabit-class performance at the cost of compile-time specialization and requires synthesis for protocol changes.

8.2.3 Extending the Model to Deparser Design

Moving from the parser to the next architectural element under investigation, the deparser, the same principles of structural decoupling and memory-driven control were applied. The deparser introduces an additional challenge, since packet emission requires a permutation engine that is traditionally implemented using wide crossbars, resulting in significant logic cost and heavy routing demand as datapath widths increase. This work eliminates that overhead by introducing a recursive-select mechanism that composes slice-and-shift units under configuration memory control, replacing monolithic permutation networks with a structured sequence of localized selection and alignment operations. The resulting architecture achieves near-linear scaling with respect to packet-header vector size and bus width, while preserving predictable timing behavior and cycle-accurate operation. Although highly specialized deparsers [87] tailored to fixed protocol graphs can retain a slight area advantage in narrow cases, evaluation showed that the proposed design starts approaching their footprint as graphs get more complex, with the benefit of uniquely enabling runtime modification of emission schedules without hardware regeneration. In doing so, it extends flexibility beyond what fixed designs provide while sustaining high throughput, demonstrating that deparsing can be made both programmable and efficient through memory-driven control.

8.3 Assessment of Research Objectives

The four research objectives defined at the outset can now be assessed in light of the architectural developments and empirical results achieved throughout this work.

First, exploring the design space of FPGA-based P4 data planes revealed two fundamental mechanisms for reconciling performance with programmability: compile-time templating, which enables structural reuse through generic parametrization, and runtime overlay control, which externalizes behavioral configuration to memory. This dual strategy overcomes the rigidity of static translation flows such as P4-to-VHDL pipelines, which regenerate RTL for each program and thereby inhibit practical deployment agility.

Second, the objective of establishing a memory-driven reconfiguration methodology was fulfilled through the formulation of compact control tables that encode parsing and deparsing transitions, extraction offsets, header lengths, and deparser emission schedules. These tables serve as the sole programmable interface to the datapath, enabling behavioral evolution through lightweight configuration updates without requiring modifications or resynthesis of hardware. This validates the central hypothesis that packet processing flexibility can be decoupled from FPGA reconfiguration by relocating protocol logic into distributed on-chip memory.

Third, a unified reconfiguration methodology for parser and deparser programmability was developed by applying the same templated and overlay principles to both PISA stages within a consistent hardware–software co-design flow. This framework shares a common configuration model, addressable control space, and automated software backend, demonstrating that parser and deparser reconfiguration need not be treated as disjoint problems. Instead, they can be coordinated through a shared abstraction that maintains protocol consistency while supporting runtime updates.

Finally, scalability and performance validation was conducted across multiple protocol graphs and datapath widths using synthesis, timing, and resource utilization analyses. Results confirmed that templated architectures preserve the efficiency characteristics of compile-time specialization, sustaining high operating frequencies and predictable resource scaling. At the same time, overlay architectures deliver practical runtime reconfigurability with bounded hardware overhead, maintaining multi-gigabit throughput and deterministic latency while avoiding synthesis toolchain delays. Collectively, these findings demonstrate that the proposed framework achieves the research goal of combining FPGA-class performance with runtime programmability.

8.4 Theoretical and Practical Implications

The results of this work carry several theoretical and practical implications that extend beyond the implemented architectures.

8.4.1 Theoretical Implications

From a theoretical standpoint, they motivate a refinement of the design principles traditionally applied to FPGA-based data planes. In particular, the findings support a configuration–datapath separation principle, analogous in spirit to the separation of control and forwarding planes in SDN, where datapath microarchitectures remain structurally stable while protocol behavior is externalized into re-writable control memories. This shifts the role of hardware away from encoding protocol semantics toward providing a programmable substrate governed by compact configuration state. Such a shift enables new forms of formal reasoning; parser and deparser behavior can be treated as table-driven state transition systems rather than hardwired RTL, allowing the configuration space to be analyzed independently of the underlying pipelines. This abstraction opens the door to correctness guarantees, bounded configuration synthesis, and offline validation of protocol evolution policies, all without reengaging FPGA implementation tools.

8.4.2 Practical Implications

Practically, the overlay methodology demonstrated in this work enables a deployment model that is aligned with operational requirements in cloud, telecom, and edge environments. Because behavioral changes are applied through control table updates rather than bitstream reconfiguration, protocol evolution can be achieved in the field within microseconds. The approach also supports multi-tenant or multi-service environments by partitioning configuration memory into isolated regions, ensuring that separate logical pipelines may coexist without interference. This reduces the operational costs traditionally associated with FPGA-based systems, eliminating lengthy resynthesis cycles and simplifying version management and rollback. In addition, the crossbar-free emission fabric developed for the deparser component demonstrates that high-bandwidth header reconstruction can be achieved without the quadratic complexity associated with permutation networks. This resolves a longstanding scalability challenge in FPGA-based deparser design and suggests that high throughput, protocol-flexible architectures can be realized without prohibitive routing congestion or timing degradation. Collectively, these implications indicate that runtime reconfigurable FPGA-based P4 data planes are not only viable but well aligned with modern requirements for agility, isolation, and performance in programmable networking systems.

CHAPTER 9 CONCLUSION

9.1 Considerations on Programmable FPGA Data Planes

The emergence of programmable data planes has reshaped the evolution of network infrastructure, enabling fine-grained control over packet processing at high speeds. Although languages such as P4 and architectures like PISA provided the necessary abstractions for protocol-independent packet handling, practical deployment on reconfigurable hardware remains constrained by rigid compilation flows, static pipeline structures, and long synthesis times. This thesis addressed these challenges by proposing a set of FPGA-based designs and methodologies that bridge the gap between programmability and performance, enabling runtime-reconfigurable packet parsers and deparsers while retaining throughput scalability.

The primary objective of this work was to propose an architecture for protocol-agnostic packet processing on FPGAs that does not require hardware resynthesis. To achieve this, we introduced templated and overlay architectures that decouple protocol semantics from hardware implementation, enabling hardware reuse across multiple P4 specifications. Through this decoupling, parsing and deparsing behaviors are expressed as memory-stored configuration tables generated automatically from P4 input descriptions. This approach significantly reduces development overhead and enables on-the-fly reconfiguration, making the proposed architectures suitable for SDN.

9.2 Summary of Main Contributions

9.2.1 Separation of Protocol Logic from Hardware Structure

This thesis proposed reconfigurable approaches to FPGA-based packet processing for protocol-independent parsing and deparsing. The central idea developed throughout this work is the separation of protocol semantics from hardware structures, enabling a single, reusable FPGA design to support different P4 parsing and deparsing graphs without requiring redesign of the underlying datapath. Protocol logic is extracted from the P4 description and expressed as configuration values, externalizing the decision flow from hardware nodes. In the case of templated variants, this decoupling allows the RTL architecture to remain unchanged across P4 programs, requiring only updated generics before resynthesis. For overlay variants, configuration memories replace compile-time logic, allowing for runtime updates without requiring any hardware resynthesis. Through this direction, the thesis reconciles the trade-off between compile-time specialization and in-field hardware adaptability, enabling the practical deployment of P4-programmable FPGA dataplanes in dynamic network environments.

The first contribution of this work focuses on designing a high-speed packet parser architecture that efficiently maps P4 descriptor graphs to FPGA hardware. To demonstrate the performance limits of static specialization, a templated parser was implemented in VHDL using a Python-assisted compilation backend. The parser architecture leverages deep pipelining, parallel field extraction, and static transition resolution to sustain data rates of up to 1 Tb/s on a Xilinx UltraScale+ FPGA at a 256-bit datapath width. Despite targeting protocol independence, the design maintains compact area utilization by using bitmask-based extraction and transition key comparison, eliminating unnecessary hardware duplication. This implementation demonstrated that template-driven hardware generation can hit a few hundred Gb/s in a single streaming datapath on a large FPGA while remaining fully compatible with P4 descriptions.

9.2.2 Runtime Reconfigurability

However, compile-time specialization inherently lacks runtime adaptability, which motivated the next stage of this work. The second contribution introduced a memory-based overlay parser architecture. Instead of using compile-time generics to generate fixed RTL logic, the overlay design externalizes state transition control, header extraction offsets, and decision flow into on-chip configuration memories. These memory tables can be modified at runtime, allowing protocol graph updates without altering the RTL. As a result, the architecture avoids the long delay and resource cost associated with FPGA resynthesis while allowing reconfiguration by simply reloading control tables. The overlay parser achieved sustained throughput of 11.1 Gb/s on a Xilinx Virtex-7, demonstrating that runtime reconfiguration can be achieved while maintaining high-performance packet streaming. This contribution established a scalable and practical approach to runtime P4 parser reconfiguration on FPGA platforms.

To expand runtime configurability beyond a single architectural model and address scalability across protocol sizes and datapath widths, a third contribution extended this work toward a unified and scalable parsing framework by integrating the base, overlay, and pipeline designs within a structured hardware–software toolchain. This framework demonstrated that throughput can be systematically increased by widening the datapath and exploiting architectural parallelism while preserving P4 programmability. The pipeline variant eliminated intermediate control dependencies and enabled fully parallel parsing over wide bus widths, achieving up to 289 Gb/s on Xilinx Alveo accelerator cards.

Extending beyond parsing, the fourth contribution introduced runtime programmability to the deparser stage. In contrast to parsing, deparsing presents a greater architectural challenge because it must reconstruct packets by emitting variable-length headers in a configurable order. Conventional solutions rely on permutation engines such as crossbars or large multiplexing networks, both of which scale poorly with increasing datapath width and protocol complexity and introduce significant routing congestion and timing degradation on FPGAs. To overcome these limitations, a crossbar-free recursive-select deparser architecture was proposed. This design employs deterministic header emission through slice-and-shift operations, eliminating wide switching fabrics while maintaining correct byte alignment. The deparser is available in both templated and overlay forms, maintaining runtime reconfigurability through memory-controlled emission rules. It achieves over 200 Gb/s on modern Xilinx FPGAs with predictable resource utilization and high timing closure reliability.

9.2.3 Overall Findings

These results confirm the central hypothesis of this thesis, namely that protocol-agnostic packet processing can be achieved on FPGAs without sacrificing throughput or scalability. Across both parsing and deparsing stages, templated designs provide high-performance packet processing for fixed protocol graphs, while overlay variants enable true runtime programmability with predictable and controlled resource growth.

9.3 Limitations

Although this thesis introduces scalable and runtime-programmable FPGA architectures for both parsing and deparsing, certain limitations remain. The templated variants retain a dependency on FPGA synthesis and bitstream generation for structural changes to the P4 program, which limits their responsiveness to rapidly evolving protocol graphs. For overlay architectures, programmability is constrained by the architectural bounds defined at design time. Extending beyond these limits, such as supporting additional protocols, deeper encapsulation, or wider datapaths, requires redefining the overlay parameters and generating a new hardware instance. For the deparser, although the proposed crossbar-free architecture reduces the quadratic routing cost associated with traditional crossbar networks to a linear cost while sustaining high throughput, it still underperforms compared to highly specialized deparsers that target a single fixed emission graph.

In exchange for programmability, the overlay and templated variants incur a modest area overhead compared to fully specialized designs. Additional design iterations and architectural refinements could further optimize resource utilization. Moreover, achieving a fully programmable data plane would require extending runtime configurability to the match–action pipeline. In this thesis, the focus was placed on architectural mechanisms for the initial (parser) and final (deparser) stages of the PISA pipeline. Enabling equivalent reconfigurability in the match–action stage represents a promising direction for future work and would complete an end-to-end dynamically programmable data-plane architecture.

9.4 Future Research

As discussed in the limitations, this thesis focused on the parser and deparser stages of the PISA pipeline. Extending the same overlay methodology to the match–action stage represents a compelling direction for future research. Achieving a fully overlay-based PISA pipeline would enable complete runtime programmability across all data-plane stages, allowing the entire packet-processing pipeline to be synthesized once and subsequently reconfigured on the fly without hardware regeneration. Such a capability would enable multiple programmable datapaths to coexist within a single FPGA, allowing for true plane-level hardware virtualization. In this model, each tenant or network slice could dynamically instantiate its own P4 processing pipeline, isolated in terms of control and data plane resources, while sharing the same physical infrastructure. This vision aligns closely with emerging requirements in cloud-networking environments, network slicing, multi-tenant FPGA deployments, and the broader goal of providing FPGA-accelerated data-plane services as a dynamically reconfigurable, shared platform for heterogeneous applications.

Moreover, while the proposed deparser architecture eliminates the quadratic resource and routing overhead traditionally associated with crossbar-based emission networks, it nonetheless incurs additional resource utilization compared to fully specialized designs. This overhead is an inherent consequence of supporting arbitrary protocol graphs and runtime programmability. A promising avenue for future work is therefore the development of optimization strategies that reduce the area footprint of the overlay deparser while preserving its flexibility and high throughput. Achieving these optimizations would further narrow the gap between overlay-based and fully dedicated architectures, enabling more efficient deployment of runtime-programmable deparsers in resource-constrained FPGA environments.

REFERENCES

[1] L. Kleinrock, “An early history of the internet [History of Communications],” *IEEE Communications Magazine*, vol. 48, no. 8, pp. 26–36, 8 2010. [Online]. Available: https://ieeexplore.ieee.org/document/5534584/

[2] R. Zakon, “Hobbes’ Internet Timeline,” Tech. Rep., 1997. [Online]. Available: https://www.rfc-editor.org/info/rfc2235

[3] C. Annual and I. Report, “White paper Cisco public,” Tech. Rep., 2018.

[4] ——, “White paper Cisco public,” Tech. Rep., 2023.

[5] C. Partridge, *END-TO-END ARGUMENTS IN SYSTEM DESIGN.* Artech House, 1988.

[6] T. Benson, A. Akella, and D. Maltz, “Unraveling the Complexity of Network Management,” pp. 335–348, 4 2009. [Online]. Available: https://dl.acm.org/doi/10.5555/1558977.1559000

[7] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and J. Van Der Merwe, “The case for separating routing from routers,” in *Proceedings of the ACM SIGCOMM 2004 Workshops*, 2004.

[8] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K. J. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic, and S. Mangiante, “De-Ossifying the Internet Transport Layer: A Survey and Future Perspectives,” pp. 619–639, 1 2017.

[9] O. N. Fundation, “Software-defined networking: The new norm for networks.” ONF White Paper,, 2012, pp. vol. 2, pp. 2–6.

[10] M. Casado, N. McKeown, and S. Shenker, “From ethane to SDN and beyond,” *ACM SIGCOMM Computer Communication Review*, vol. 49, no. 5, pp. 92–95, 11 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3371934.3371963

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 3 2008. [Online]. Available: https://dl.acm.org/doi/10.1145/1355734.1355746

[12] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM.* New York, NY, USA: ACM, 8 2013, pp. 99–110. [Online]. Available: https://dl.acm.org/doi/10.1145/2486001.2486011

[13] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 7 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2656877.2656890

[14] Barefoot Networks, "The World's Fastest & Most Programmable Networks," https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/, 2013.

[15] R. Das and A. C. Snoeren, "Memory Management in ActiveRMT: Towards Runtime-programmable Switches," in *SIGCOMM 2023 - Proceedings of the ACM SIGCOMM 2023 Conference.* Association for Computing Machinery, Inc, 9 2023, pp. 1043–1059.

[16] P. Benacek, V. Pu, and H. Kubatova, "P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* IEEE, 5 2016, pp. 148–155. [Online]. Available: http://ieeexplore.ieee.org/document/7544769/

[17] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4FPGA : A rapid prototyping framework for P4," in *SOSR 2017 - Proceedings of the 2017 Symposium on SDN Research.* Association for Computing Machinery, Inc, 4 2017, pp. 122–135.

[18] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4-&gt;NetFPGA Workflow for Line-Rate Packet Processing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* New York, NY, USA: ACM, 2 2019, pp. 1–9. [Online]. Available: https://dl.acm.org/doi/10.1145/3289602.3293924

[19] M. Attig and G. Brebner, "400 Gb/s Programmable Packet Parsing on a Single FPGA," in *2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems.* IEEE, 10 2011, pp. 12–23. [Online]. Available: http://ieeexplore.ieee.org/document/6062708/

[20] Z. Cao, H. Su, Q. Yang, J. Shen, M. Wen, and C. Zhang, “P4 to FPGA-A Fast Approach for Generating Efficient Network Processors,” *IEEE Access*, vol. 8, pp. 23 440–23 456, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8976091/

[21] P. K. Chartsias, A. Amiras, I. Plevrakis, I. Samaras, K. Katsaros, D. Kritharidis, E. Trouva, I. Angelopoulos, A. Kourtis, M. S. Siddiqui, A. Vines, and E. Escalona, “SDN/NFV-based end to end network slicing for 5G multi-tenant networks,” in *EuCNC 2017 - European Conference on Networks and Communications.* Institute of Electrical and Electronics Engineers Inc., 7 2017.

[22] J. Krude, J. Hofmann, M. Eichholz, K. Wehrle, A. Koch, and M. Mezini, “Online reprogrammable multi tenant switches,” in *ENCP 2019 - Proceedings of the 1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms, Part of CoNEXT 2019.* Association for Computing Machinery, Inc, 12 2019, pp. 1–8.

[23] B. A. A. Nunes, M. Mendonca, X. N. Nguyen, K. Obraczka, and T. Turletti, “A survey of software-defined networking: Past, present, and future of programmable networks,” *IEEE Communications Surveys and Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.

[24] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden, “A survey of active network research,” *IEEE Communications Magazine*, vol. 35, no. 1, pp. 80–86, 1997.

[25] D. L. Tennenhouse and D. J. Wetherall, “Towards an Active Network Architecture,” Tech. Rep., 2002. [Online]. Available: http://www.tns.lcs.mit.edu/.

[26] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, “A clean slate 4D approach to network control and management,” *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 5, pp. 41–54, 10 2005.

[27] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, “Ethane,” in *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM ’07. New York, NY, USA: ACM, 8 2007, pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/1282380.1282382

[28] K. Qiu, J. Yuan, J. Zhao, X. Wang, S. Secci, and X. Fu, “Fast lookup is not enough: Towards efficient and scalable flow entry updates for TCAM-Based OpenFlow Switches,” in *Proceedings - International Conference on Distributed Computing Systems*, vol. 2018-July, 2018.

[29] N. Gude, T. K. Hiit, J. Pettit, B. Pfaff, M. Casado, N. Mckeown, and S. Shenker, "NOX: Towards an Operating System for Networks," Tech. Rep. [Online]. Available: http://www.noxrepo.org

[30] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A Distributed Control Platform for Large-scale Production Networks," Tech. Rep.

[31] Open Networking Foundation, "OpenFlow Switch Specification," 12 2014. [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.0.pdf

[32] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, 8 2000. [Online]. Available: https://dl.acm.org/doi/10.1145/354871.354874

[33] N. Shah, W. Plishker, K. Ravindran, and K. Keutzer, "NP-Click: A Productive Software Development Approach for Network Processors," *IEEE Micro*, vol. 24, no. 5, pp. 45–54, 9 2004. [Online]. Available: https://doi.org/10.1109/MM.2004.53

[34] N. Bonelli, A. D. Pietro, S. Giordano, and G. Procissi, "A Purely Functional Approach to Packet Processing," in *Proceedings of the ACM SIGCOMM 2014 Conference (Poster/Demo/CCR track)*, 8 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2658260.2658269

[35] R. Duncan and P. Jungck, "packetC Language for High Performance Packet Processing," in *Proceedings of the 11th IEEE International Conference on High Performance Computing and Communications (HPCC)*, 6 2009, pp. 450–457. [Online]. Available: https://doi.org/10.1109/HPCC.2009.89

[36] H. Song, "Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 8 2013, pp. 127–132. [Online]. Available: https://doi.org/10.1145/2491185.2491190

[37] G. Brebner and W. Jiang, "High-Speed Packet Processing Using Reconfigurable Computing," *IEEE Micro*, vol. 34, no. 1, pp. 8–18, 1 2014. [Online]. Available: https://doi.org/10.1109/MM.2014.19

[38] M. Budiu and C. Dodd, "The P416 Programming Language," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 1, pp. 5–14, 9 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/3139645.3139648

[39] P4 Language Consortium, "behavioral-model: The reference P4 software switch (BMv2)." [Online]. Available: https://github.com/p4lang/behavioral-model

[40] ——, "p4c: The P4 Compiler." [Online]. Available: https://github.com/p4lang/p4c

[41] ——, "p4c - eBPF Backend (p4c/backends/ebpf)." [Online]. Available: https://github.com/p4lang/p4c/tree/main/backends/ebpf

[42] ——, "p4c - uBPF Backend (p4c/backends/ubpf)." [Online]. Available: https://github.com/p4lang/p4c/tree/main/backends/ubpf

[43] VMware Inc., "P4C-XDP: A P4 Compiler Backend for XDP/eBPF." [Online]. Available: https://github.com/vmware/p4c-xdp

[44] eBPF Project, "What is eBPF? — An Introduction and Deep Dive into the eBPF Technology." [Online]. Available: https://ebpf.io/what-is-ebpf/

[45] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "PISCES: A programmable, protocol-independent software switch," in *SIGCOMM 2016 - Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication*, 2016.

[46] P. Zanna, P. Radcliffe, and K. G. Chavez, "A Method for Comparing OpenFlow and P4," in *2019 29th International Telecommunication Networks and Applications Conference, ITNAC 2019*, 2019.

[47] P. Voros, D. Horpacsi, R. Kitlei, D. Lesko, M. Tejfel, and S. Laki, "T4P4S: A target-independent compiler for protocol-independent packet processors," in *IEEE International Conference on High Performance Switching and Routing, HPSR*, vol. 2018-June, 2018.

[48] Data Plane Development Kit Project, "DPDK Documentation," 2025. [Online]. Available: https://core.dpdk.org/doc/

[49] eBPF Project, "Program Type 'BPF_PROG_TYPE_XDP' — eBPF Docs," 2025. [Online]. Available: https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_XDP/

[50] Intel Corporation, “Intel Corporation-Intel ® IXP2400 Network Processor-2nd Generation Intel ® NPU,” Tech. Rep. [Online]. Available: https://www.cs.ucr.edu/~bhuyan/cs162/IXP2400.pdf

[51] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, “A survey on data plane programming with P4: Fundamentals, advances, and applied research,” *Journal of Network and Computer Applications*, vol. 212, p. 103561, 3 2023. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1084804522002028

[52] S. Pontarelli, M. Bonola, and G. Bianchi, “Smashing SDN "built-in" actions: Programmable data plane packet manipulation in hardware,” in *2017 IEEE Conference on Network Softwarization: Softwarization Sustaining a Hyper-Connected World: en Route to 5G, NetSoft 2017*, 2017.

[53] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soul, and N. Zilberman, “The case for in-network computing on demand,” in *Proceedings of the 14th EuroSys Conference 2019*, 2019.

[54] R. Kundel, A. Rizk, J. Blendin, B. Koldehofe, R. Hark, and R. Steinmetz, “P4-CoDel: Experiences on Programmable Data Plane Hardware,” in *IEEE International Conference on Communications*, 2021.

[55] Xilinx Inc., “SDNet Packet Processor User Guide (UG1012 v2017.1),” 6 2017. [Online]. Available: https://www.xilinx.com/support/documents/sw_manuals/xilinx2017_1/UG1012-sdnet-packet-processor.pdf

[56] I. Red Hat, “NETCOPE P4,” 2025. [Online]. Available: https://catalog.redhat.com/en/software/applications/detail/207717

[57] P. Benáček, V. Puš, H. Kubátová, and T. Čejka, “P4-To-VHDL: Automatic generation of high-speed input and output network blocks,” *Microprocessors and Microsystems*, vol. 56, no. October 2017, pp. 22–33, 2018.

[58] J. Santiago da Silva, F.-R. Boyer, and J. P. Langlois, “P4-Compatible High-Level Synthesis of Low Latency 100 Gb/s Streaming Packet Parsers in FPGAs,” in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, vol. 2018-Febru. New York, NY, USA: ACM, 2 2018, pp. 147–152. [Online]. Available: https://dl.acm.org/doi/10.1145/3174243.3174270

[59] P. Mashreghi-Moghadam, T. Ould-Bachir, and Y. Savaria, "A Templated VHDL Architecture for Terabit/s P4-programmable FPGA-based Packet Parsing," in *2022 IEEE International Symposium on Circuits and Systems (ISCAS).* IEEE, 5 2022, pp. 672–676. [Online]. Available: https://ieeexplore.ieee.org/document/9937607/

[60] M. Abbasmollaei, T. Ould-Bachir, and Y. Savaria, "P4THLS: A Templated HLS Framework to Automate Efficient Mapping of P4 Data-Plane Applications to FPGAs," *IEEE Access*, 2025.

[61] P. Mashreghi-Moghadam, T. Ould-Bachir, and Y. Savaria, "An Area-efficient Memory-based Architecture for P4-programmable Streaming Parsers in FPGAs," in *Proceedings - IEEE International Symposium on Circuits and Systems*, vol. 2023-May, 2023.

[62] K.-S. Hsu and C.-A. Shen, "The Design of a Configurable and Low-Latency Packet Parsing System for Communication Networks," *SSRN Electronic Journal*, 2022. [Online]. Available: https://www.ssrn.com/abstract=4031275

[63] M. T. Arashloo, "Stateful Programming of High-Speed Network Hardware," Tech. Rep., 2019.

[64] J. Santiago, "Fully Programming the Data Plane: a Hardware/Software Approach," 2020.

[65] T. Luinaud, J. M. P. Langlois, and Y. Savaria, "Titre: Title: Optimisation de la compilation de déparseurs pour processeurs réseau implémentés sur FPGA Directeurs de recherche," Tech. Rep., 2022.

[66] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown, "Design principles for packet parsers," in *Architectures for Networking and Communications Systems.* IEEE, 10 2013, pp. 13–24. [Online]. Available: http://ieeexplore.ieee.org/document/6665172/

[67] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2161, 2001.

[68] T. Luinaud, J. Santiago da Silva, J. P. Langlois, and Y. Savaria, "Design Principles for Packet Deparsers on FPGAs," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: ACM, 2 2021, pp. 280–286. [Online]. Available: https://dl.acm.org/doi/10.1145/3431920.3439303

[69] D. Kreutz, F. M. V. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-Defined Networking: A Comprehensive Survey,” 6 2014. [Online]. Available: http://arxiv.org/abs/1406.0440

[70] Z. Cao, H. Zhang, J. Li, M. Wen, and C. Zhang, “A Fast Approach for Generating Efficient Parsers on FPGAs,” *Symmetry*, vol. 11, no. 10, p. 1265, 10 2019. [Online]. Available: https://www.mdpi.com/2073-8994/11/10/1265

[71] J. Cabal, P. Benáček, L. Kekely, M. Kekely, V. Puš, and J. Kořenek, “Configurable FPGA Packet Parser for Terabit Networks with Guaranteed Wire-Speed Throughput,” in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, vol. 2018-Febru. New York, NY, USA: ACM, 2 2018, pp. 249–258. [Online]. Available: https://dl.acm.org/doi/10.1145/3174243.3174250

[72] C. Kozanitis, J. Huber, S. Singh, and G. Varghese, “Leaping Multiple Headers in a Single Bound: Wire-Speed Parsing Using the Kangaroo System,” in *2010 Proceedings IEEE INFOCOM*. IEEE, 3 2010, pp. 1–9. [Online]. Available: http://ieeexplore.ieee.org/document/5462139/

[73] A. Bitar, M. S. Abdelfattah, and V. Betz, “Bringing programmability to the data plane: Packet processing with a NoC-enhanced FPGA,” *2015 International Conference on Field Programmable Technology, FPT 2015*, no. Figure 1, pp. 24–31, 2016.

[74] H. Zolfaghari, D. Rossi, and J. Nurmi, “A custom processor for protocol-independent packet parsing,” *Microprocessors and Microsystems*, vol. 72, p. 102910, 2020. [Online]. Available: https://doi.org/10.1016/j.micpro.2019.102910

[75] A. Yazdinejad, A. Bohlooli, and K. Jamshidi, “P4 to SDNet: Automatic Generation of an Efficient Protocol-Independent Packet Parser on Reconfigurable Hardware,” in *2018 8th International Conference on Computer and Knowledge Engineering (ICCKE)*. IEEE, 10 2018, pp. 159–164. [Online]. Available: https://ieeexplore.ieee.org/document/8566590/

[76] SFF Committee, “SFP+ (Enhanced 10 Gbps pluggable module),” Tech. Rep., 2009. [Online]. Available: www.t10.org

[77] H. Zolfaghari, D. Rossi, and J. Nurmi, “An Explicitly Parallel Architecture for Packet Parsing in Software Defined Networks,” in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2018-July. IEEE, 7 2018, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/document/8445123/

[78] ——, “Low-latency Packet Parsing in Software Defined Networks,” in *2018 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC).* IEEE, 10 2018, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/8573461/

[79] H. Zolfaghari, D. Rossi, W. Cerroni, H. Okuhara, C. Raffaelli, and J. Nurmi, “Flexible software-defined packet processing using low-area hardware,” *IEEE Access*, vol. 8, pp. 98 929–98 945, 2020.

[80] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, “Forwarding metamorphosis,” *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 9 2013. [Online]. Available: https://dl.acm.org/doi/10.1145/2534169.2486011

[81] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, “An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends,” *IEEE Access*, vol. 9, pp. 87 094–87 155, 2021.

[82] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S. T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, “Programmable packet scheduling at line rate,” in *SIGCOMM 2016 - Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication*, 2016.

[83] Y. Sun and Z. Guo, “The Design of a Dynamic Configurable Packet Parser Based on FPGA,” *Micromachines*, vol. 14, no. 8, 8 2023.

[84] H. Liu, Z. Qiu, W. Pan, J. Li, and J. Huang, “HyperParser: A High-Performance Parser Architecture for Next Generation Programmable Switch and SmartNIC,” in *ACM International Conference Proceeding Series*, 2021.

[85] Cisco, “Cisco Annual and Internet Report,” Tech. Rep. 3, 2018.

[86] M. S. Abdelfattah and V. Betz, “Design tradeoffs for hard and soft FPGA-based networks-on-chip,” in *FPT 2012 - 2012 International Conference on Field-Programmable Technology*, 2012.

[87] T. Luinaud, J. M. Langlois, and Y. Savaria, “Symbolic Analysis for Data Plane Programs Specialization,” *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 1, 11 2022.

[88] J. Cabal, P. Benacek, J. Foltova, and J. Holub, “Scalable P4 Deparser for Speeds Over 100 Gbps,” in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* IEEE, 4 2019, pp. 323–323. [Online]. Available: https://ieeexplore.ieee.org/document/8735524/

[89] P. Mashreghi-Moghadam, T. Ould-Bachir, and Y. Savaria, “PrismParser: A Framework for Implementing Efficient P4-Programmable Packet Parsers on FPGA,” *Future Internet*, vol. 16, no. 9, 9 2024.

[90] E.-M. Makhroute, M.-A. Elharti, V. Brouillard, Y. Savaria, and T. Ould-Bachir, “Implementing and Evaluating a P4-based Access Gateway Function on a Tofino Switch,” in *2023 6th International Conference on Advanced Communication Technologies and Networking (CommNet).* IEEE, 12 2023, pp. 1–7.