Titre: Title:	PrismParser: a framework for implementing efficient P4-programmable packet parsers on FPGA
Auteurs: Authors:	Parisa Mashreghi-Moghadam, Tarek Ould-Bachir, & Yvon Savaria
Date:	2024
Type:	Article de revue / Article
Référence: Citation:	Mashreghi-Moghadam, P., Ould-Bachir, T., & Savaria, Y. (2024). PrismParser: a framework for implementing efficient P4-programmable packet parsers on FPGA. Future Internet, 16(9), 307 (19 pages). https://doi.org/10.3390/fi16090307

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/59166/
Version:	Version officielle de l'éditeur / Published version Révisé par les pairs / Refereed
Conditions d'utilisation: Terms of Use:	CC BY

Document publié chez l'éditeur officiel
Document issued by the official publisher

Titre de la revue: Journal Title:	Future Internet (vol. 16, no. 9)
Maison d'édition: Publisher:	Multidisciplinary Digital Publishing Institute
URL officiel: Official URL:	https://doi.org/10.3390/fi16090307
Mention légale: Legal notice:	© 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/)

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

MDPI

Article

PrismParser: A Framework for Implementing Efficient P4-Programmable Packet Parsers on FPGA

Parisa Mashreghi-Moghadam [1,*], Tarek Ould-Bachir [2] and Yvon Savaria [1,*]

[1] Department of Electrical Engineering, Polytechnique Montréal, Montréal, QC H3H 1T1, Canada
[2] Department of Computer and Software Engineering, Polytechnique Montréal, Montréal, QC H3H 1T1, Canada; t.ould-bachir@polymtl.ca
* Correspondence: parisa.mashreghi-moghadam@polymtl.ca (P.M.-M.); yvon.savaria@polymtl.ca (Y.S.)

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.

Keywords: SDN; packet parser; P4 language; FPGA; overlay

Citation: Mashreghi-Moghadam, P.; Ould-Bachir, T.; Savaria, Y. PrismParser: A Framework for Implementing Efficient P4-Programmable Packet Parsers on FPGA. *Future Internet* **2024**, *16*, 307. https://doi.org/10.3390/fi16090307

Academic Editor: Gianluigi Ferrari

Received: 16 July 2024
Revised: 12 August 2024
Accepted: 21 August 2024
Published: 27 August 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Software-defined networks (SDNs) have introduced the potential for dynamically programmable network infrastructure by decoupling control and data planes [1]. 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 [2]. 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 [3].

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 [4]. 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 [5,6].

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 [7]. 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 [3].

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 [8]. Whether software- or ASIC-based, traditional packet parsers face significant limitations in processing capabilities, latency, and flexibility [9]. 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 [10]. 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 [9,10], faced significant latency issues. Subsequent ternary content addressable memory (TCAM)-based methods by [3,11] reduced latency but compromised flexibility and scalability. Innovations like those from [12–15] introduced custom processors for reconfigurable packet parsing, but still grappled with latency. Recent advancements, such as those by [16–18], 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 [19]. Our design leverages parallel extraction and matching techniques to minimize latency and maximize throughput, supporting real-time reconfiguration through SDN control planes [20]. 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 2 outlines packet parsing strategies and reviews previous work. Section 3 highlights the methodology employed in our study. Section 4 presents the experimental results, and Section 5 concludes this work.

2. Protocol-Agnostic Packet Parsing: Background

2.1. 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 1 showcases a classic parse graph for enterprise networks, similar to what is explained in [21]. In this graphical representation, nodes show individual protocols or headers, and edges outline the potential transitions between them.

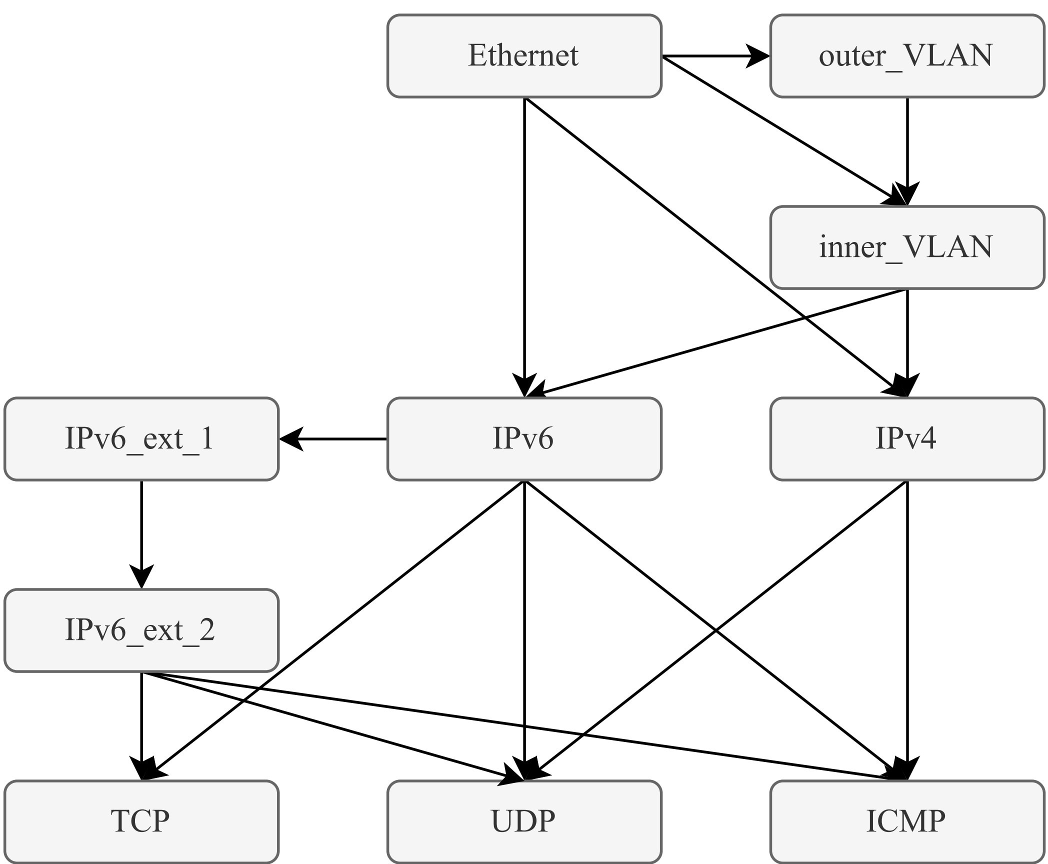

Figure 1. Example of an enterprise parsing graph.

2.2. 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 [9,10] were associated with considerable latencies. Subsequent TCAM-based methodologies [3,11] 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 [12–15,22]. These designs, while reconfigurable, grappled with latency issues, especially when integrating new protocols.

The automation of P4-programmable packet parsers, explored in [19,23], marked a significant shift. A notable contribution from Da Silva et al. [24] 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 [16,17], 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 [18]. 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.

3. 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.

3.1. Software Overview

3.1.1. 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 2.

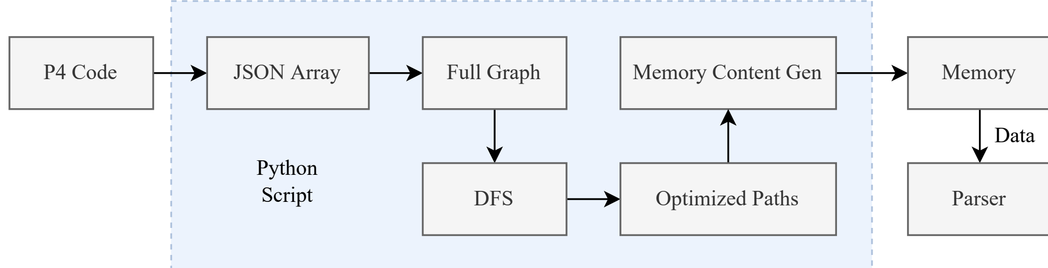

Figure 2. Proposed compilation workflow for generating control and configuration for the parser.

3.1.2. 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 1 shows a simplified P4 compiled JSON file, which shows the header types, headers, and parsers for the Ethernet header:

Listing 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"}
    ]
  }]
}
```

3.1.3. 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 3.2 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 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 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 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 4 or Listing 5.

Listing 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 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
```

3.1.4. 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.

Input: `J`: The compiled P4 JSON file, `B`: The desired bus size, `S`: The desired multiplexers size
Output: `MemCode`: Memory configuration code

```
1  Initialization:
2  G ← Create an empty DAG
3  ProtocolBitmap ← Create an empty array
4  ProtocolMask ← Create an empty array
5  Matchvalues ← Create an empty array
6  NextState ← Create an empty array
7  MuxSelect ← Create an empty array
8  EnableSignals ← Create an empty array
9  NextStateBitmap ← Create an empty array
10 MemCode ← Create an empty array
11 HeaderInfo ← parseJSON(J)
12 foreach parseObj in J.get('parsers') do
13     header ← parseObj.name
14     G.addNode(header)
15     G.addData(HeaderInfo.get(header))
16     foreach transition in parseObj.get('transitions') do
17         G.addEdge(G.currentNode, transition.nextState, transition.value)
18 paths ← DFS(G)
19 reducedPaths ← removeUnusedPaths(paths)
20 foreach path in reducedPaths do
21     foreach header in path do
22         ProtocolBitmap.append(getProtocolBitmap(header,
            HeaderInfo.get(header)))
23         ProtocolMask.append(getProtocolMask(header, HeaderInfo.get(header)))
24         Matchvalues.append(getMatchvalues(header, HeaderInfo.get(header)))
25         NextState.append(getNextState(header, HeaderInfo.get(header)))
26         MuxSelect.append(getMultiplexerSelect(header, HeaderInfo.get(header), B,
            S))
27         EnableSignals.append(getEnableSignals(header, HeaderInfo.get(header), B,
            S))
28         NextStateBitmap.append(getNextStateBitmap(header,
            HeaderInfo.get(header)))
29 MemCode ← createMemContent(ProtocolBitmap, ProtocolMask, Matchvalues,
    NextState, MuxSelect, EnableSignals, NextStateBitmap, B)
30 return 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.

3.2. 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 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 1 compares the proposed design with previous programmable packet parsers using the same bus size.

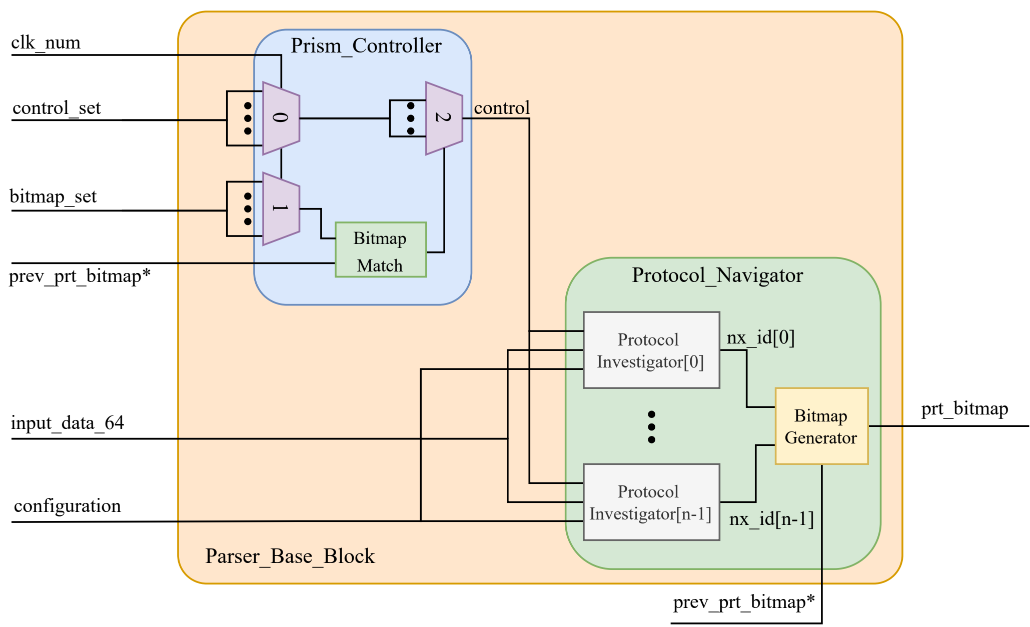

Figure 3. Base Block.

Table 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
[22]	Alveo U200	64	125	8	23	3538	1578	5116	Instr.	Hours
[18]	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.

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 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 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 $protocol\ ID - 1 = i$ has been detected.

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.

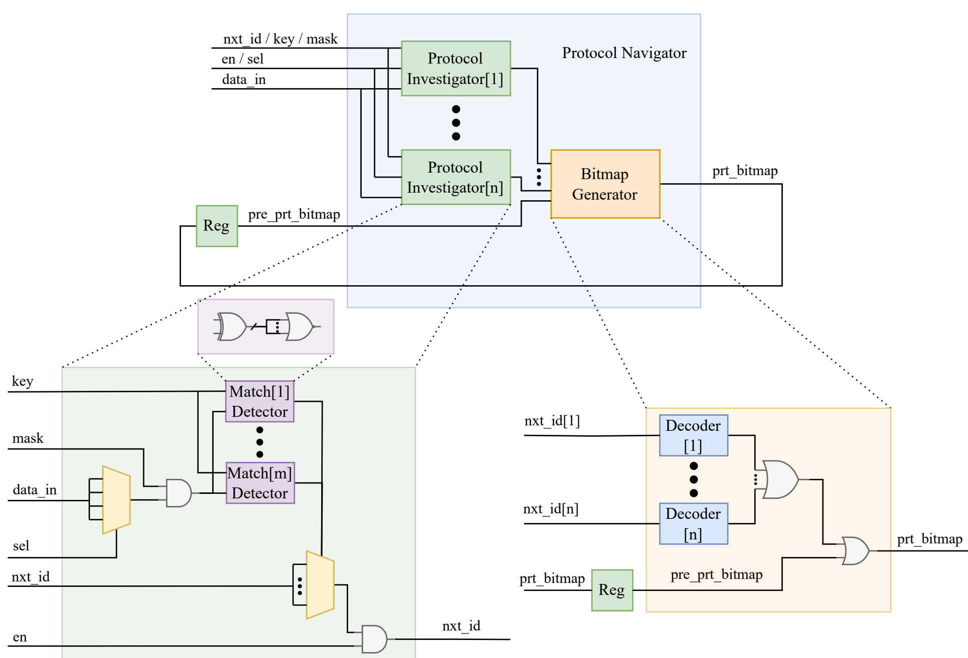

Figure 4. Protocol Navigator: Protocol Investigator with its Match Detector sub-block and Bitmap Generator.

3.3. 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.

3.4. 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 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.

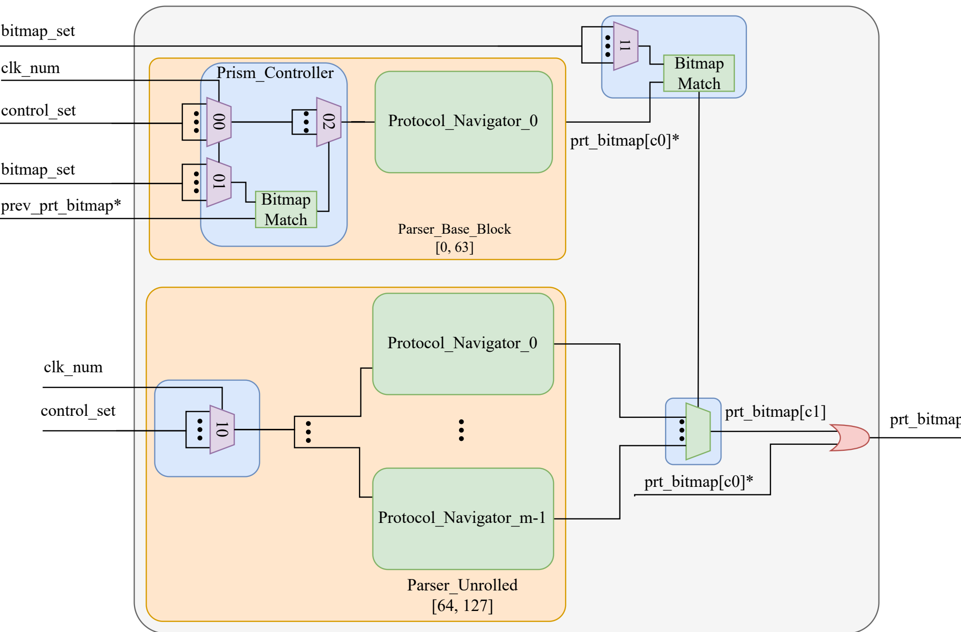

Figure 5. Overlay Block: In the multiplexer ID, the first digit indicates the parser block it belongs to, and the second digit indicates the multiplexer number, referred to in the text with a #.

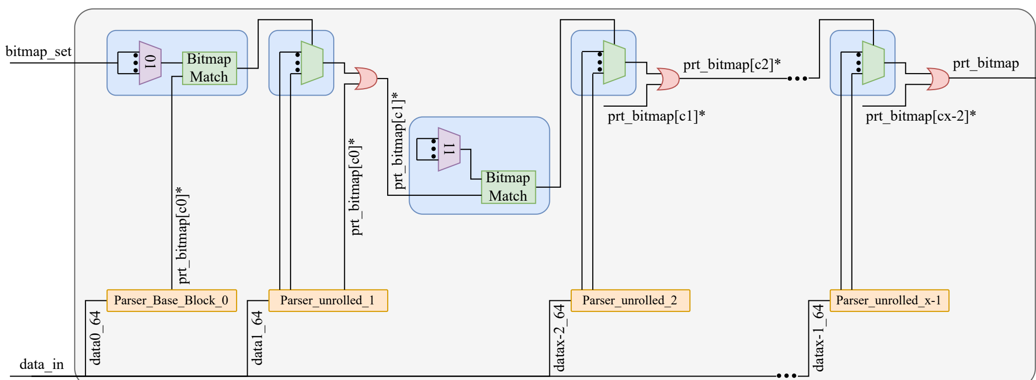

Figure 6. Overlay Block.

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 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 need to wait for a clock cycle to obtain the result from the first block, significantly reducing latency and increasing throughput. Figure 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 2, showing the throughput improvements achieved by increasing the number of modified base blocks.

Table 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

3.5. Pipeline Optimization

As seen in Section 3.3, 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 (Figure 7). 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.

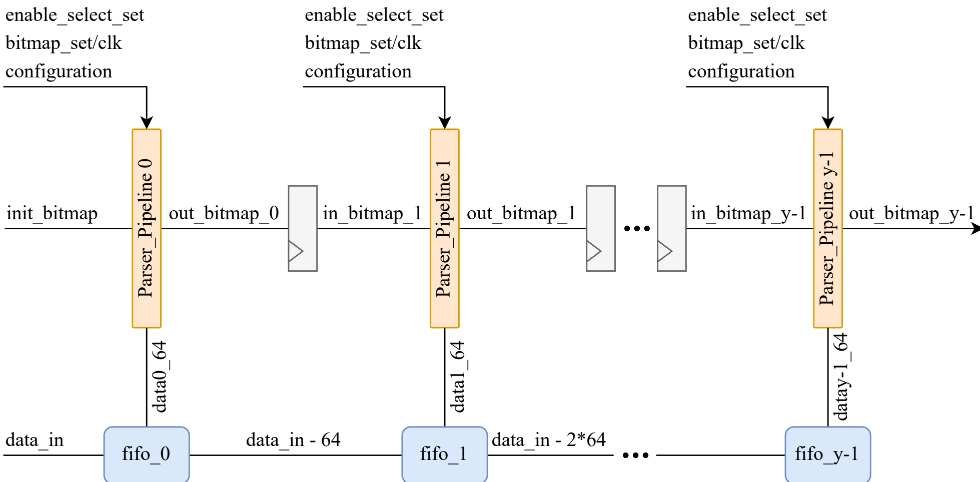

Figure 7. Parser Pipeline Block.

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 3, showing the throughput improvements achieved by this approach.

Table 3. Results comparison for pipeline block.

Work	FPGA	Performance				Resources		
		Bus Size [bits]	Frequency [MHz]	TP [Gb/s]	Latency [ns]	LUTs	FFs	Total Logic
[19]—Golden	Virtex-7	512	195.30	100.00	27.00	N/A	N/A	8000
[19]	Virtex-7	512	195.30	100.00	46.10	10,103	5537	15,640
[24]	Virtex-7	320	312.50	100.00	25.60	7831	13,67	21,502
[20]	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
[16]	Virtex-7	1136	279.30	317.00	25.06	16,888	12,033	28,921
[17]	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

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.

4. 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 (Figure 8). 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.

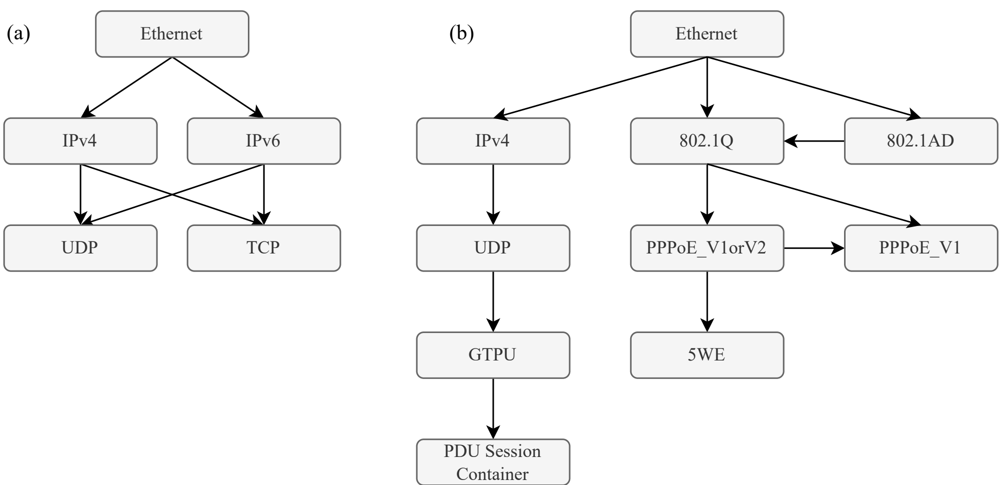

Figure 8. (**a**) Simple firewall graph. (**b**) Access Gateway Function flow graph.

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 1–3 showcase our design outcomes using Vivado 2022.2 to synthesize the RTL code for an Alveo U280 board. Table 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 2 presents the results of area and throughput for different bus sizes for the overlay design. Finally, Table 3 demonstrates and compares the results of the pipeline design with similar approaches in state-of-the-art works.

4.1. 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 [22], which operates at 125 MHz with a throughput of 8 Gb/s, and [18], which achieves a frequency of 173.4 MHz and a throughput of 11.1 Gb/s. Ad-

ditionally, 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 [22]. 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 [22], 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.

4.2. 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 [19], 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 [24] 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 [20], 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 [16] 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. [17] 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.

5. 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.

Author Contributions: Conceptualization, P.M.-M., T.O.-B., and Y.S.; methodology, P.M.-M., T.O.-B. and Y.S.; software, P.M.-M.; validation, P.M.-M.; formal analysis, P.M.-M.; investigation, P.M.-M.; resources, Y.S., T.O.-B. and P.M.-M.; data curation, P.M.-M.; writing—original draft preparation, P.M.-M.; writing—review and editing, Y.S., T.O.-B. and P.M.-M.; visualization, P.M.-M.; supervision, Y.S., T.O.-B.; project administration, Y.S.; funding acquisition, Y.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the NSERC Kaloom-Intel-Noviflow Industrial Chair of Professor Savaria IRCPJ-548237-18 CRSNG, by Polytechnique Montreal, and by discovery grant RGPIN-2019-05951 CRSNG (AV: 05295-2014/6574-09) to one of the authors.

Data Availability Statement: The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Kreutz, D.; Ramos, F.M.; Verissimo, P.E.; Rothenberg, C.E.; Azodolmolky, S.; Uhlig, S. OpenFlow: Enabling Innovation in Campus NetworksSoftware-defined networking: A comprehensive survey. *Proc. IEEE* **2015**, *103*, 14–76. [CrossRef]
2. Bosshart, P.; Daly, D.; Gibb, G.; Izzard, M.; McKeown, N.; Rexford, J.; Schlesinger, C.; Talayco, D.; Vahdat, A.; Varghese, G.; et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 87–95. [CrossRef]
3. Bosshart, P.; Gibb, G.; Kim, H.S.; Varghese, G.; McKeown, N.; Izzard, M.; Mujica, F.; Horowitz, M. Forwarding metamorphosis. *ACM SIGCOMM Comput. Commun. Rev.* **2013**, *43*, 99–110. [CrossRef]
4. Krude, J.; Hofmann, J.; Eichholz, M.; Wehrle, K.; Koch, A.; Mezini, M. Online reprogrammable multi tenant switches. In *Proceedings of the ENCP 2019—Proceedings of the 1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms, Part of CoNEXT 2019, Orlando, FL, USA, 9 December 2019*; Association for Computing Machinery, Inc.: New York, NY, USA, 2019; pp. 1–8.

5. Chartsias, P.K.; Amiras, A.; Plevrakis, I.; Samaras, I.; Katsaros, K.; Kritharidis, D.; Trouva, E.; Angelopoulos, I.; Kourtis, A.; Siddiqui, M.S.; et al. SDN/NFV-based end to end network slicing for 5G multi-tenant networks. In *Proceedings of the EuCNC 2017—European Conference on Networks and Communications, Oulu, Finland, 12–15 June 2017*; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2017.
6. Kfoury, E.F.; Crichigno, J.; Bou-Harb, E. An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends. *IEEE Access* **2021**, *9*, 87094–87155. [CrossRef]
7. Song, H. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, Hong Kong, China, 16 August 2013; p. 127.
8. Sivaraman, A.; Subramanian, S.; Alizadeh, M.; Chole, S.; Chuang, S.T.; Agrawal, A.; Balakrishnan, H.; Edsall, T.; Katti, S.; McKeown, N. Programmable packet scheduling at line rate. In Proceedings of the SIGCOMM 2016—Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication, Salvador, Brazil, 22–26 August 2016.
9. Kozanitis, C.; Huber, J.; Singh, S.; Varghese, G. Leaping Multiple Headers in a Single Bound: Wire-Speed Parsing Using the Kangaroo System. In Proceedings of the 2010 Proceedings IEEE INFOCOM, San Diego, CA, USA, 14–19 March 2010; pp. 1–9.
10. Attig, M.; Brebner, G. 400 Gb/s Programmable Packet Parsing on a Single FPGA. In Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems, Brooklyn, NY, USA, 3–4 October 2011; pp. 12–23.
11. Gibb, G.; Varghese, G.; Horowitz, M.; McKeown, N. Design principles for packet parsers. In Proceedings of the Architectures for Networking and Communications Systems, San Jose, CA, USA, 21–22 October 2013; pp. 13–24.
12. Zolfaghari, H.; Rossi, D.; Nurmi, J. An Explicitly Parallel Architecture for Packet Parsing in Software Defined Networks. In Proceedings of the 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP), Milano, Italy, 10–12 July 2018; pp. 1–4.
13. Zolfaghari, H.; Rossi, D.; Nurmi, J. Low-latency Packet Parsing in Software Defined Networks. In Proceedings of the 2018 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC), Tallinn, Estonia, 30–31 October 2018; pp. 1–6.
14. Zolfaghari, H.; Rossi, D.; Cerroni, W.; Okuhara, H.; Raffaelli, C.; Nurmi, J. Flexible Software-Defined Packet Processing Using Low-Area Hardware. *IEEE Access* **2020**, *8*, 98929–98945. [CrossRef]
15. Zolfaghari, H.; Rossi, D.; Nurmi, J. A custom processor for protocol-independent packet parsing. *Microprocess. Microsyst.* **2020**, *72*, 102910. [CrossRef]
16. Cao, Z.; Zhang, H.; Li, J.; Wen, M.; Zhang, C. A Fast Approach for Generating Efficient Parsers on FPGAs. *Symmetry* **2019**, *11*, 1265. [CrossRef]
17. Mashreghi-Moghadam, P.; Ould-Bachir, T.; Savaria, Y. A Templated VHDL Architecture for Terabit/s P4-programmable FPGA-based Packet Parsing. In Proceedings of the 2022 IEEE International Symposium on Circuits and Systems (ISCAS), Austin, TX, USA, 27 May–1 June 2022; pp. 672–676.
18. Mashreghi-Moghadam, P.; Ould-Bachir, T.; Savaria, Y. An Area-efficient Memory-based Architecture for P4-programmable Streaming Parsers in FPGAs. In Proceedings of the Proceedings—IEEE International Symposium on Circuits and Systems, Monterey, CA, USA, 21–25 May 2023.
19. Benacek, P.; Pu, V.; Kubatova, H. P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers. In Proceedings of the 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Washington, DC, USA, 1–3 May 2016; pp. 148–155.
20. Sun, Y.; Guo, Z. The Design of a Dynamic Configurable Packet Parser Based on FPGA. *Micromachines* **2023**, *14*, 1560. [CrossRef] [PubMed]
21. Liu, H.; Qiu, Z.; Pan, W.; Li, J.; Huang, J. HyperParser: A High-Performance Parser Architecture for Next Generation Programmable Switch and SmartNIC. In Proceedings of the ACM International Conference Proceeding Series, Virtual Event, 20–25 June 2021.
22. Hsu, K.S.; Shen, C.A. The Design of a Configurable and Low-Latency Packet Parsing System for Communication Networks. *SSRN Electron. J.* **2022**. [CrossRef]
23. Yazdinejad, A.; Bohlooli, A.; Jamshidi, K. P4 to SDNet: Automatic Generation of an Efficient Protocol-Independent Packet Parser on Reconfigurable Hardware. In Proceedings of the 2018 8th International Conference on Computer and Knowledge Engineering (ICCKE), Mashhad, Iran, 25–26 October 2018; pp. 159–164.
24. Santiago da Silva, J.; Boyer, F.R.; Langlois, J.P. 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, New York, NY, USA, 25–27 February 2018; pp. 147–152.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.