From Patterns to Parsers: Automatic Generation of Efficient Hardware Parsers for FPGAs

Tushar Garg* and Andrew Boutros†
*Meta Platforms, Inc.
†Department of Electrical and Computer Engineering, University of Waterloo
Email: gtushar@meta.com, andrew.boutros@uwaterloo.ca

***Abstract*—This work presents an open-source tool for automatically generating efficient hardware parsers from high-level specifications. It uses a parsing intermediate representation (PIR) that decouples application-specific frontends from a common register-transfer level (RTL) generation backend. The backend produces optimized, human-readable SystemVerilog, handling FSM generation, byte-alignment, and multi-cycle field straddling for arbitrary datapath widths. The tool also extends pattern matching beyond simple equality checks by introducing custom symbolic tokens to support operations that existing parser generators cannot express, such as range validation, negation, and comparisons against external ports. We demonstrate two end-to-end flows using a P4 frontend for Ethernet protocol parsing and a Snort frontend for network intrusion detection, both using the same unmodified backend. The generated Ethernet parsers achieve up to 226% higher operating frequency and up to 97% fewer FPGA logic resources than prior work. A controlled synthetic study further shows that the tool's hierarchical pattern decomposition yields up to 8× resource utilization reduction over monolithic designs. Our open-source framework enables designers to rapidly implement high-performance, resource-efficient, vendor-agnostic hardware parsers for diverse applications.**

I. Introduction

Line-rate data parsing is a fundamental operation in modern high-performance computing. As data rates outpace general-purpose processor capabilities, dedicated packet processors on FPGAs [1], [2] or ASICs [3], [4] have become essential to meet throughput and latency targets. High-performance hardware parsing circuitry is central to all these accelerators. For instance, financial trading systems use FPGA-based parsers to decode market data with sub-microsecond latency, where parsing latency directly affects trading outcomes [5], [6]. Similarly, modern smart network interface cards (SmartNICs) implement security features that require parsing and inspecting packets in real-time without throttling bandwidth [7].

Implementing efficient hardware parsers requires manually crafting complex finite state machines (FSMs) for state tracking, byte alignment, and multi-cycle field straddling across arbitrary datapath widths, a process that is slow, error-prone, and brittle under evolving protocol specifications or datapath scaling. Tools generating hardware from domain-specific languages, such as P4 [8], have been proposed to automate this process. However, most commercial offerings produce encrypted, vendor-locked RTL [9], while open-source alternatives either rely on high-level synthesis (HLS) resulting in suboptimal implementation quality or have stagnated with deprecated dependencies [10], [11].

This work introduces a versatile open-source tool that bridges design productivity and implementation quality by automatically generating optimized RTL hardware parsers. Built on our parsing intermediate representation (PIR), the tool decouples domain-specific frontend(s) from the RTL generation backend, abstracting parsing specifications into a generalized pattern matching problem that automatically handles complexities such as dynamic byte alignment and multi-cycle fields. The tool also enables users to include symbolic tokens in their parsing specifications, which extends pattern matching beyond equality to support operations that no existing hardware parser generator can express natively, such as range validation, negation, and comparisons against external ports. The backend is target-agnostic and template-driven using Jinja2, enabling output in plain SystemVerilog. The generated designs are pipelined to achieve high frequencies yet remain structured and human-readable for further manual optimization, if needed. To showcase the tool, we implement two example frontends: a P4-based frontend for Ethernet/IP protocol parsing and a Snort frontend for network security rule parsing. In summary, our contributions in this work are:

- A versatile open-source[1] tool for automatic hardware parser generation, using a parsing intermediate representation that decouples application frontends from the RTL backend.
- Extended pattern matching with custom symbolic tokens for range validation, negation, and external port comparisons.
- Validation across two distinct domains (Ethernet protocol parsing and network intrusion detection) using the same unmodified backend, achieving up to 226% higher operating frequency and up to 97% and 71% fewer lookup tables (LUTs) and flip-flops (FF) compared to prior work.

II. Background & Related Work

Packet parsing involves two key stages: *pattern matching* to identify fields of interest, and *data extraction* to retrieve and align them for downstream processing. Strict sequential

This research, including the development and evaluation of the tool, was conducted in a personal capacity and by the University of Waterloo. The views, findings, and conclusions expressed in this paper are solely those of the authors and do not represent the official policy or position of Meta.

[1]Source code link: https://github.com/boutros-lab/patterns-to-parsers

dependencies between nested headers, where parsing one header depends on fields from the previous, require complex control FSMs. Header extraction is also resource-intensive, requiring large barrel shifters with additional complexity when fields straddle multiple data flits.

One hardware parser implementation approach [12], [11], [13] chains identical processing elements, each handling a single header and passing metadata along with the data flit to the next. While this handles arbitrary protocol stacks, every stage must incorporate full barrel shifters for byte alignment regardless of the limited shifting actually required, resulting in significant resource overhead. Parsing latency also scales linearly with header depth. In contrast, our tool generates flat custom parsers implementing only the minimal required data extraction multiplexing, yielding significant latency reduction and resource savings.

Some implementations use ternary content addressable memories (TCAMs), where parsing state machines are stored in runtime-configurable tables [14], [10], [3], [15]. While TCAMs enable parallel lookup and can reduce identification latency, they introduce significant area overhead and power consumption. These approaches also typically rely on speculative field extraction (i.e., extracting multiple potential header fields before identification completes) which can increase design complexity.

Other works rely on domain-specific compilers to automate parser generation from high-level descriptions. The authors of [16] compile P4 into C++ and use AMD Vivado HLS to generate hardware. While this simplifies the design process, the semantic gap between P4's packet processing model and C++'s sequential execution model forces insertion of synchronization logic, producing parsers with higher latency than hand-optimized counterparts due to conservative HLS scheduling. Similarly, P4FPGA [17] is an open-source P4-to-FPGA compiler generating Bluespec SystemVerilog implementations, but it is limited to P4 inputs and has not seen active development for a decade. The Xilinx SDNet commercial tool [18] translates P4 directly into RTL, but produces encrypted vendor-specific IP that hinders platform migration and prevents use-case-specific optimizations.

Unlike these approaches, our tool is not bound to a specific input language and does not sacrifice implementation quality by using generic HLS tools. It implements a fully decoupled RTL generation backend consuming a generic intermediate representation for pattern matching and data extraction. The generated RTL is structured, human-readable, and vendor-agnostic, enabling designers to inspect and optimize the output for their target platform as needed.

III. Tool Overview

Fig. 1 illustrates a high-level overview of our parser generation toolflow. The backend translates our generic PIR to an optimized RTL implementation based on provided configuration settings, while different application-specific frontends generate parsing specifications in PIR format. In addition, the backend also generates a SystemVerilog testbench code to validate the pattern matching and data extraction functionality using automatically generated synthetic input patterns. This section introduces the PIR format, configuration settings, and supported parsing features. We also implement two example frontends detailed in Sections IV and V.

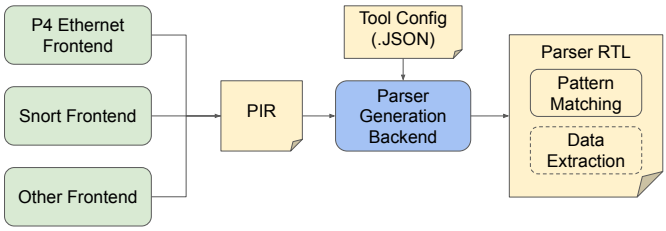


Fig. 1. High-level overview of our hardware parser generation toolflow.

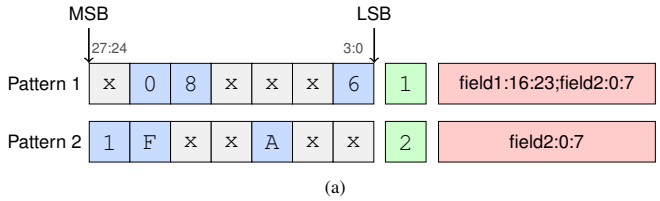
A. Parsing Intermediate Representation

Our PIR defines the parsing specification as a set of pattern strings along with their corresponding unique identifiers and (optional) data extraction requirements. Pattern strings are represented at 4-bit (nibble) granularity, chosen to align with the hexadecimal representation commonly used in networking protocol specifications and tools (e.g., `tcpdump` and Wireshark). The PIR supports three character classes:

- *Hexadecimal Digits* (`0-9, A-F`) that define required constant matches at the corresponding nibble positions.
- *Wildcards* (`x`) that indicate "do not care" positions masked during matching.
- *Custom Symbolic Tokens (any other character):* that enable complex comparisons beyond simple equality checks, such as comparisons against external input ports, inequalities, range checks, and set membership. Their meaning is defined in the tool configuration file discussed in the next subsection.

Each pattern string is coupled with a unique output identifier, specifying the value produced when the pattern is detected in the input data stream. Optionally, the PIR specifies data fields to extract when a pattern is matched, represented as tuples of field ID, start position, and end position. The backend uses these to implement minimal multiplexing for extracting specified fields across all pattern strings. If no extraction fields are specified, the generated parser implements only pattern matching. PIR patterns are prioritized by specification order; when input data matches multiple patterns, only the first matching pattern identifier is produced. This priority-based matching guarantees deterministic output for overlapping patterns and enables hierarchical matching where specific patterns take precedence over general fallbacks.

An example PIR with two pattern strings is shown in Fig. 2. Each pattern has 7 nibbles (28 bits) containing constant hexadecimal values (■) and wildcards (■), with output identifiers (■) set to `1` and `2`. The tuple (■) shows field extraction information per pattern. The tool extracts the bit positions and length of each contiguous set of nibble



(a)

```
//Pattern Matching RTL
module pattern_matching (
    input logic [27:0] data,
    output logic [1:0] out_id
);
always_comb begin
  if ( (data[23:16]== 8'h08) &&
     (data[3:0]==4'h6) )
    out_id = 1;
  else if ( (data[27:20]== 8'h1F) &&
            (data[11:8]==4'hA) )
    out_id = 2;
  else out_id = 0;
end
endmodule
```

```
//Data Extraction RTL
module data_extraction (
    input logic [27:0] data,
    input logic [1:0]  out_id,
    output logic [7:0] field1,
    output logic [7:0] field2
);
always_comb begin
  if (out_id == 1)
    field1 = data[23:16];
  else field1 = '0;
end
always_comb begin
  if ((out_id == 1) || (out_id == 2))
    field2 = data[7:0];
  else field2 = '0;
end
endmodule
```

(b)

Fig. 2. (a) Bit map of two PIR entries with their output identifiers and data extraction requirements shown in (■) and (■), respectively. (b) Generated RTL implementation for pattern matching and data extraction. The tool generates equality comparators for (■) regions and ignores (■) regions in the pattern matching module. The data extraction module steers the specified fields to the corresponding ports based on the pattern matching output.

```
1  "num_groups": 4,
2  "group_datawidth": 3,
3  "datapath_width": 6,
4  "invalid_match_output": 0,
5  "custom_tokens": [{
6     "id": "VLLL",
7     "type": "port",
8     "prefix": "myport",
9     "opcode": "RANGE" }]
```

Listing 1. Tool configuration file example.

characters and ignores the wildcards. Then, the template based RTL generator produces the pattern matching and data extraction implementations shown in Fig. 2b. When the first pattern is matched, the two fields `data[23:16]` and `data[7:0]` are steered to the `field1` and `field2` output ports, respectively. On the other hand, when the second pattern is matched, only `data[7:0]` is extracted and steered to the `field2` port.

B. Tool Configuration

The backend takes a `json` configuration file as input, as shown in Listing 1. This configuration enables the backend to: (1) decompose pattern matching into subproblems for optimized timing and resource utilization, and (2) generate logic for arbitrary datapath widths.

1) Hierarchical Pattern Matching Parameters: The user can implement two-level hierarchical pattern matching by breaking pattern strings into `num_groups` *pattern substrings*, each `group_datawidth` nibbles wide. Each group is treated as an independent subproblem. Repeated substrings within each group are deduplicated to reduce the size of the subproblem, and a *pseudo output identifier* is assigned to each unique substring. A second-stage

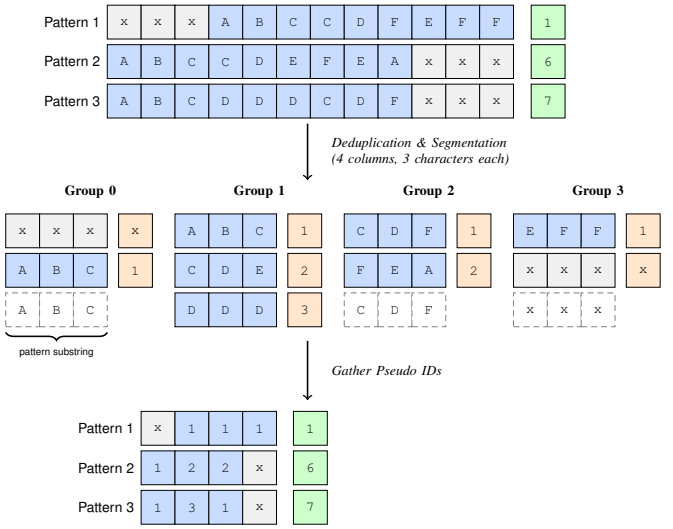


Fig. 3. Hierarchical pattern matching example with `num_groups`=4 and `group_datawidth`=3.

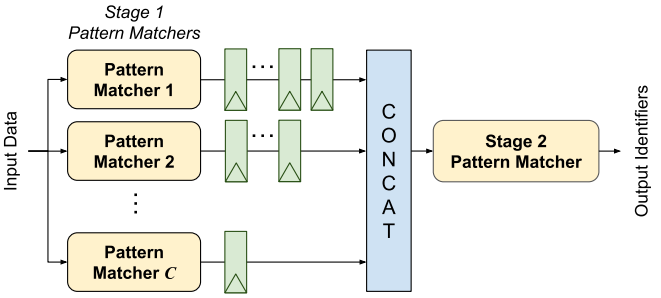


Fig. 4. Generated pattern matching hardware for pattern strings straddling multiple cycles.

module then maps the collected pseudo identifiers to final output identifiers. This organization reduces comparator bits when repeated substrings are prevalent, improving resource utilization and timing.

Fig. 3 shows an example with `num_groups` and `group_datawidth` set to 4 and 3, respectively. Each of the 12-nibble pattern strings is split into four 3-nibble pattern substrings. Following segmentation, the repeated pattern substrings `ABC`, `CDF`, and `xxx` are eliminated from groups 0, 2, and 3, respectively. Then, a unique group-level pseudo output identifier (■) is assigned to each pattern substring. These pseudo output identifiers are used in a second-stage pattern matcher to produce the final output identifiers (■). Substrings containing do-not-care characters within a group can overlap. For instance, `6x` and `x3` both match input `63`, causing a collision. The tool automatically resolves such cases by further splitting the substrings until no collisions remain.

2) Datapath Width Parameter: The `datapath_width` parameter specifies the pattern matching datapath width in nibbles and must satisfy:

$$\texttt{datapath\_width} \bmod \texttt{group\_datawidth} = 0$$

as each clock cycle must process integer number of complete groups. The number of cycles C the full pattern string

straddles is:

$$C \times \texttt{datapath\_width} = \texttt{num\_groups} \times \texttt{group\_datawidth}$$

The tool generates C distinct pattern matching modules processing flits in round-robin fashion. Pseudo output identifiers from the C modules are aligned using pipeline registers (see Fig. 4) and concatenated as input to a second-stage module producing the final output identifiers.

The `invalid_match_output` parameter specifies the output identifier produced when the input data does not match any PIR pattern string. These knobs enable architectural design space exploration; however, determining optimal configurations currently requires slow RTL synthesis in the loop. Analytical resource estimation, without invoking a synthesis tool, remains an avenue for future work.

C. Custom Symbolic Token Parameters

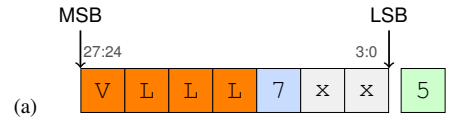
A key novel feature of our backend is extending the pattern matching alphabet beyond standard hexadecimal and wildcard characters via the `custom_tokens` configuration parameter. This parameter acts as a symbolic mapping, defining hardware generation rules for custom characters used in PIR pattern strings. Each custom token definition consists of two components:

- `opcode`: Defines the logical predicate for comparison. The tool extends standard equality `(==)` to support inequality `(NE)`, set membership `(EQOR)`, and range validation `(RANGE)`. The schema is extensible, allowing users to add arbitrary hardware predicates (e.g., bit-counting or parity checks) with minimal backend modifications.
- `type`: Determines the operand source:
  - `const`: Comparison operands are embedded directly in the configuration using the `values` parameter as immediate values in the generated RTL.
  - `port`: The tool creates top-level input port(s) in the generated RTL, allowing match conditions to be driven by external system state, other modules, or software-configurable registers. Port naming is controlled via `prefix` and port count is determined by `opcode` (e.g., `RANGE` requires two ports).

Fig. 5 illustrates an example PIR entry containing the 4-nibble custom token defined in Listing 1. The `custom` parameter maps the token `"VLLL"` to the `RANGE` opcode with source type `port`, instructing the backend to create two input ports and implement a magnitude comparator for the four nibbles specified by the custom token (i.e., `data[27:12]`).

D. Custom Sparse Data Extraction Multiplexing

Traditional protocol parsers rely on generic barrel shifters for field extraction at runtime. A generic barrel shifter selecting an N-bit field at any nibble-aligned offset within an M-bit datapath requires $N \times \lceil M/4 \rceil$:1 multiplexers. As datapath widths increase, these multiplexers cause significant area overhead, routing congestion, and timing degradation. Our backend instead performs compile-time static analysis to



(a)

(b)

```
//Pattern Matching RTL
module pattern_matching (
    input logic [27:0] data,
    input logic [15:0] myport_0,
    input logic [15:0] myport_1,
    output logic [1:0] out_id
);
always_comb begin
  if ( (data[27:12] >= myport_0) &&
       (data[27:12] <  myport_1) &&
       (data[11:8]  == 4'h7) )
    out_id = 5;
  else out_id = 0;
end
endmodule
```

Fig. 5. (a) Bit map of a PIR entry with its output identifier shown in (■) and a custom token shown in (■). (b) Generated RTL with range comparators for (■) region, and equality comparator for (■) region.

```
1 header udp_t {
2     bit<16> srcPort__sp__;  ⟶ extracted as sp
3     bit<16> dstPort__dp__;  ⟶ extracted as dp
4     bit<16> length_;
5     bit<16> checksum;
6 }
```

Listing 2. P4 header declaration with extraction annotations.

generate sparse, customized multiplexing networks. Because the frontend linearizes the parsing specification and assigns a unique output identifier to every possible pattern, the exact bit positions of every target field are known at generation time. The extraction logic therefore generates a sparse multiplexer accounting only for the limited set of offsets at which each field can appear. For example, extracting a 32-bit field from a 512-bit datapath would conventionally require 32×512:1 multiplexers, but if the PIR analysis determines the field can only appear at 5 distinct offsets, the tool generates 32×5:1 multiplexers, a significant reduction in area.

The following two sections demonstrate the tool's end-to-end capabilities through two case studies spanning distinct application domains: Ethernet protocol parsing (Section IV) and network intrusion detection (Section V). We then validate the impact of hierarchical decomposition on a controlled synthetic benchmark in Section VI.

IV. Case Study I: Ethernet Protocol Parsing

We implement a P4-based frontend that compiles a standard P4 program into the PIR and tool configuration file consumed by our backend, demonstrating the end-to-end flow from a high-level protocol specification to synthesizable RTL.

A. Frontend P4 Compiler and Backend Integration

The user defines parsing logic using a standard P4 program with header declarations, parser states, and `select()` transitions. To specify which fields should be extracted in generated RTL, we introduce a lightweight annotation

convention. Any header field whose name contains a double-underscore suffix __<id>__ is marked for extraction, where <id> becomes the field identifier in the PIR. Fields without the suffix are used only for pattern matching and are not extracted. As shown in Listing 2, declaring srcPort__sp__ instructs the frontend to populate the optional extraction tuple in the PIR entry for that path, recording the field identifier, start bit position, and end bit position. The backend then uses this information to generate the data extraction RTL alongside the pattern matching logic, as described in Section III. This convention keeps the P4 program fully compatible with standard P4 toolchains while providing our frontend with the extraction metadata it needs.

The frontend invokes the standard p4c [19] compiler with the BMV2 [20] backend to produce a json representation of the P4 program, encoding header types, field widths, and the parser state machine. It then traverses the parser's state graph via depth-first search to enumerate all valid protocol paths (e.g., Ethernet → IPv4 → TCP). Each root-to-leaf path is flattened into a nibble-granularity pattern string of hexadecimal constants and wildcards, assigned a unique output identifier, and annotated with id:startpos:endpos extraction tuples derived from __<id>__ suffixed fields, forming the PIR. The frontend also generates a backend configuration file using two user-provided frontend parameters: input_width and pattern_width, setting backend datapath_width and group_datawidth equal to input_width and computing backend num_groups as pattern_width / input_width. The backend then consumes the PIR and configuration file to produce the pattern matching and data extraction RTL as detailed in Section III.

B. Evaluation

To evaluate the end-to-end flow, we generate two parser types consistent with prior work [10], [11], [16]. We compare against these academic baselines as they represent the latest publicly available parser RTL generators. Commercial P4 compilers (e.g., AMD Vitis Networking P4 [9]) generate the parser, deparser, and match-action blocks as a single integrated pipeline, making it infeasible to isolate parser-only area and timing numbers for a fair comparison. Both parser types extract the standard five-tuple (protocol, source/destination IP, source/destination port) from the incoming packet stream:

- *Simple parser (SP):* Supports Ethernet, IPv4/IPv6 (with up to two extension headers), UDP, TCP, and ICMP/ICMPv6. The full pattern string length is 90 bytes.
- *Full parser (FP):* Extends SP with MPLS (up to two labels) and VLAN (up to two tags). The full pattern string length is 106 bytes.

1) Comparison against prior work: Table I compares our generated designs against prior work on an AMD Virtex-7 (xc7vx690tffg1761-2), which is the target platform used in these prior works. To ensure a fair comparison, we match the exact datapath width used by each prior work.

TABLE I. Comparison of Parser Results across Benchmarks (N/R = Not Reported; N/C = Not Calculated)

	Work	Freq [MHz]	T-put [Gb/s]	Latency [ns]	LUTs	FFs
SP (256b)	[10]	178.6	46	N/R	6865	1851
	ours	583	149.2	15.4	179	2116
	Gain/Loss %	*+226%*	*+224%*	*N/C*	*-97%*	*14%*
SP (320b)	[16]	312.5	100	19.2	4270	6163
	Hybrid [16] and [11]	312.5	100	28.8	4699	7254
	ours	562	179.8	16	180	2564
	Gain/Loss %	*+80%*	*+80%*	*-17%*	*-96%*	*-58%*
FP (64b)	[10]	172.2	11	N/R	3789	1425
	ours	379	24.3	81.8	1331	2240
	Gain/Loss %	*+120%*	*+121%*	*N/C*	*-65%*	*57%*
FP (320b)	[16]	312.5	100	25.6	6046	8900
	Hybrid [16] and [11]	312.5	100	41.6	6450	10308
	ours	359	114.9	25.1	1357	2619
	Gain/Loss %	*+15%*	*+15%*	*-2%*	*-78%*	*-71%*
FP (512b)	[11]	195.3	100	46.1	10103	5537
	ours	379	194	18.5	1352	2938
	Gain/Loss %	*+94%*	*+94%*	*-60%*	*-87%*	*-47%*

All designs were synthesized using Vivado 2025.2 in out-of-context (OOC) mode with default settings. The synthesized RTL uses fully registered inputs and outputs with valid-ready flow control. We sourced the resource utilization numbers for prior works from [16], which consolidates results from earlier works. For Gain/Loss calculations, we consider the *best* result from prior work, ensuring our reported gains are conservative.

Our framework pipelines the generated designs heavily and applies (* DONT_TOUCH = TRUE *) on all FFs to avoid tool retiming. This aggressive pipelining leads to a higher FF count compared to prior work in some cases. Our generated designs use 65–97% less LUTs, while the FF utilization varies with datapath width; wider datapaths use 47–71% less FFs, while the narrowest configurations (*SP*(256b) and *FP*(64b)) use more FFs than prior work (+14% and +57%, respectively). For a fixed pattern length, the datapath width determines the number of groups (a narrower datapath splits the pattern across more groups), and each group adds a pipeline stage, so more groups use more pipeline registers. At low datapath widths this group count is highest, and combined with our aggressive pipelining, this inflates the FF count. As for Fmax and throughput, *FP*(320b) shows the smallest gain (+15%), as prior work already operated at 312.5 MHz. All other benchmarks achieve gains of 80% or more, reaching +226% for *SP*(256b) where prior work ran at only 178.6 MHz. Across all benchmarks, our latency is lower than or equal to prior work, with the largest improvement at *FP*(512b) where we reduce end-to-end latency by 60% (from 46.1 ns to 18.5 ns).

2) Scaling across datapath widths: To characterize how our designs scale beyond the datapath widths used in prior work, we sweep datapath_width from 64 to 512 bits on two modern FPGAs using the highest speed grade available from each vendor: Altera Agilex 5 (A5DD064AB32BI1V) and AMD Versal (xcvc1902-vsva2197-3HP-e-S). Results are shown in Fig. 6 and Fig. 7.

Fig. 6. ALM/FF utilization vs. datapath width for Altera Agilex 5 FPGAs.

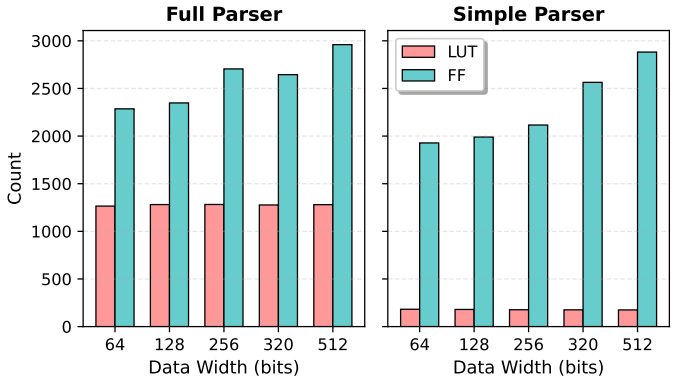


Fig. 7. LUT/FF utilization vs. datapath width for AMD FPGAs.

Both parsers exhibit sub-linear LUT/ALM growth as the datapath widens. For a fixed pattern size, the datapath width determines how the pattern is partitioned into groups, setting `num_groups` and `group_datawidth`. A wider datapath has more bits per group, and therefore wider pipeline registers are used resulting in higher FF utilization as datapath width increases. Also, wider groups require larger field comparators, resulting in an increase in logic utilization which is more noticeable on the Agilex 5 device.

V. Case Study II: Snort Rule Hardware Mapping

This case study demonstrates the versatility of our parser generation backend by applying it to network intrusion detection. Rather than parsing protocol headers, the backend is used to generate hardware rule matchers for Snort [21], the de facto standard for network intrusion detection and prevention. Crucially, this case study requires no modifications to the backend; only a different frontend is developed, validating that our PIR and custom symbolic tokens generalize beyond conventional protocol parsing.

Snort rules specify packet matching conditions using header fields (IPs, ports, protocol) combined with payload inspection (content strings at specific offsets). Many rules employ features such as Classless Inter-Domain Routing (CIDR) blocks for network ranges, comma-separated IP lists, negation operators, and port ranges, operations that go beyond simple equality matching. Existing hardware parser generators, which are designed around P4's equality-based match semantics, cannot express these operations. Our backend's custom symbolic tokens (`RANGE`, `EQOR`, `NE`) map directly to these Snort constructs, enabling hardware rule matching that would otherwise require hand-crafted RTL.

TABLE II. Snort Rule Features and Hardware Mapping

Feature	Supported	Backend Operator
IP exact match	✓	equality
IP list (comma-separated)	✓	`EQOR`
CIDR blocks	✓	`RANGE`
IP ranges	✓	`RANGE`
Negation (`!`)	✓	`NE`
Port ranges	✓	`RANGE`
Content string	✓	equality
Content offset/depth	✓	rule expansion
PCRE (regex)	×	—
Flow/stream state	×	—

Systems such as Pigasus [7] target full intrusion detection and prevention system (IDS/IPS) pipelines at 100 Gbps, implementing TCP reassembly, multi-string pattern matching, and offloading complex features like Perl Compatible Regular Expression (PCRE) to the CPU. Our tool targets a complementary point in this design space: lightweight, full hardware implementation of rule matchers for header fields and simple content inspection suitable for first-stage packet filtering, protocol violation detection, and IP-based blocklisting. In such cases, line-rate performance is critical and full stateful inspection can be handled downstream or offloaded if needed. Table II summarizes the Snort rule features supported by our frontend and their mapping to backend operators. We do not support PCRE regular expressions or stateful flow tracking, which are outside the scope of static pattern matching.

A. Frontend Compiler

The Snort frontend compiler parses rule strings from a text file and maps them to PIR entries through the following steps:

- *Packet Pattern Construction:* The compiler constructs an Ethernet/IPv4/TCP (or UDP/ICMP) packet pattern from the rule header, placing specified field values (source/destination IPs, ports, protocol) at their correct byte offsets and marking all unspecified fields as wildcards (`x`).
- *Custom Token Assignment:* Fields requiring non-equality comparisons are assigned custom symbolic tokens in the pattern string, with corresponding entries generated in the backend `json` configuration file.
- *Content Positioning and Rule Expansion:* When a rule specifies `content` with `offset` o and `depth` d, a content string of length k can appear at any byte position in $[o, d-k]$. The frontend generates $(d-o-k+1)$ separate PIR entries, one for each valid starting position. This compile-time expansion converts sliding-window searches into parallel static pattern matches, trading hardware area for throughput.
- *Action Encoding:* The rule action (e.g., `alert`, `drop`), rule index, and message identifier are encoded together

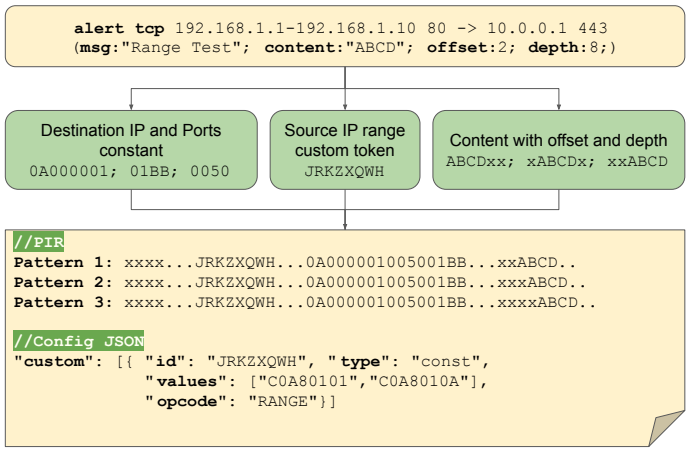


Fig. 8. Snort rule compilation flow: the frontend maps header fields to hex constants and custom tokens, then expands the content constraint into multiple PIR entries. The backend configuration maps the custom token to a `RANGE` comparator with the specified IP bounds.

as a compact binary literal and assigned as the PIR `output_id`. This enables the generated hardware to report which rule matched and what action to take without additional lookups.

B. Compilation Example

Fig. 8 illustrates the complete compilation flow for a rule that triggers an `alert` for TCP packets from source IPs in the range 192.168.1.1–192.168.1.10 on port 80, destined for 10.0.0.1 on port 443, containing `"ABCD"` within byte offsets 2–8 of the payload.

The source IP range cannot be encoded as a fixed hex value in the PIR, so the compiler assigns a placeholder custom random token (`JRKZXQWH`) and generates a `RANGE` opcode entry in the configuration with bounds `192.168.1.1(0xC0A80101)`–`192.168.1.10(0xC0A8010A)`.

The destination IP and L4 ports are encoded directly as hex constants. The content constraint `"ABCD"` with `offset:2` and `depth:8` allows the 4-byte string to start at payload offsets 2, 3, or 4, so the compiler generates three separate pattern entries, each identical except for the payload offset where `ABCD` appears.

C. Evaluation

To evaluate the Snort frontend, we compile a benchmark of 10 rules that collectively exercise all supported features: exact IP matching, CIDR blocks, IP lists, IP ranges, negation, port ranges, and content with offset/depth expansion. As noted above, our goal in this case study is not to implement a production-grade IDS/IPS. This 10-rule benchmark instead showcases that the same backend, reusing the same PIR and custom symbolic tokens without any modifications, can be coupled with a non-P4 frontend to generate functionally correct, synthesizable, and efficient hardware for a domain beyond protocol parsing, across all supported feature types.

Table III presents the synthesis results on a 1024-bit datapath for a 10-rule benchmark designed to exercise all supported feature types (Table II). Note that rules using content matching with offset/depth windows expand into multiple PIR entries (one per valid starting position), so area grows with both the number of rules and the content expansion factor.

TABLE III. Snort rule matcher synthesis results (10 rules).

Device	Fmax [MHz]	LUTs/ALMs	FFs
AMD Versal	714	234	97
Altera Agilex 5	712	706	99

VI. Impact of Hierarchical Decomposition

Hierarchical decomposition is a key feature of our tool. It enables pattern matching across arbitrary datapath widths and assists vendor tool compilers by breaking a monolithic matching problem into smaller, independently optimizable subproblems. The Ethernet parsers evaluated in Section IV use this decomposition, and the results in Table I reflect its benefit. To isolate and quantify the impact of decomposition independently of protocol-specific effects, we design a controlled synthetic benchmark suite with varying levels of substring repetition. All patterns consist solely of hexadecimal characters with no wildcards or special tokens. We generate 4,096 pattern strings each 80 bits wide and assign a unique output identifier to each, effectively constructing an 80-bit input, $\lceil \log_2(4096 + 1) \rceil$-bit output lookup table (+1 for an invalid match output identifier). The pattern count and width are chosen arbitrarily to demonstrate how decomposing the matching problem into subproblems results in producing faster and more resource-efficient designs.

For benchmark construction, we treat the dataset as a 4096×80 matrix. We partition this into four 4096×20 groups. This allows us to independently control the repetition characteristics within each group. The benchmark suite sweeps two independent dimensions of repetition across these groups:

- *Uniqueness per group (u):* The number of distinct 20-bit substrings within a group, drawn from {256, 512, 1024}, corresponding to each unique substring appearing 16×, 8×, and 4× across the 4,096 patterns, respectively.
- *Number of controlled groups (ug):* Specifies how many of the four groups are assigned a controlled *u*.

Crossing these two dimensions yields 12 configurations. For each configuration, we generate two functionally identical RTL variants: a *flat* design (`num_groups` = 1) that treats each 80-bit pattern as a single monolithic unit, ignoring the group structure entirely, and a *decomposed* design (`num_groups` = 4, `group_datawidth` = 5) that aligns its partitioning with the benchmark groups, solving each 20-bit substring independently. The expectation is that as repetition increases, either through lower *u* or higher *ug*, the *decomposed* design should show greater improvement in terms of area and speed over the *flat* design, since more repetition within groups translates directly into more sharing that the decomposed subproblems can exploit.

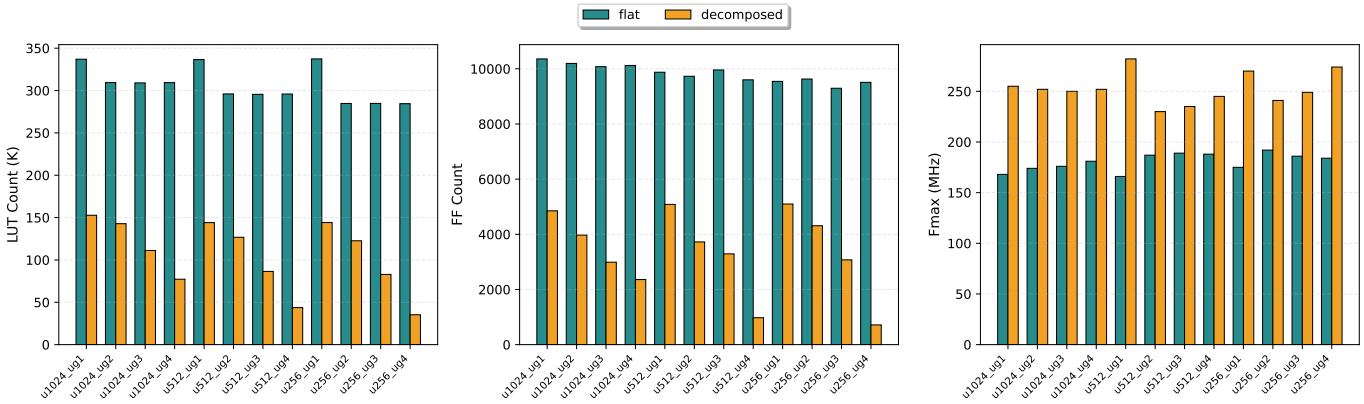


Fig. 9. Comparison of synthetic benchmarks highlighting the advantages of pattern decomposition in terms of LUT count, FF count, and frequency on an AMD FPGA.

A. Results

Fig. 9 reports Fmax, LUT, and FF for all 12 configurations, comparing the *flat* and *decomposed* variants across the full repetition sweep on an AMD Versal (`xcvc1902-vsva2197-3HP-e-S`) using Vivado 2025.2. As in our other experiments, we mark all FFs with `dont_touch` synthesis directives such that Vivado does not retime pipeline registers, which otherwise distorts FF counts without improving Fmax.

1) LUT utilization: The *flat* design exposes a fundamental limitation: it is largely insensitive to substring repetition. For a fixed u, its LUT count drops slightly from ug=*1* to ug=*2* and then plateaus (e.g., at u=*256* it holds at ∼285K LUTs for ug=2,3,4). Because the full 80-bit pattern is presented as a single monolithic unit, the vendor compiler cannot discover or share the repeated substrings.

The *decomposed* design solves each 20-bit group independently and therefore exploits this repetition directly, with gains that scale with how much repetition is exposed. Even at ug=*1*, where almost no repetition is controlled, splitting the problem into smaller subproblems already yields a ∼2.3× LUT reduction from structure alone. As repetition increases (higher ug or lower u) the reduction compounds, reaching 6.7× at u=*512,* ug=*4* (296K vs. 44K LUTs) and a peak of 8× at u=*256,* ug=*4* (284K vs. 35K LUTs).

2) Fmax: Decomposition also raises the clock frequency across every configuration, from the *flat* design's 166–192 MHz to the *decomposed* design's 230–282 MHz, roughly a 1.5× improvement. The gain comes from partitioning the 80-bit match into four independent 20-bit subproblems, which shortens the combinational critical path regardless of the repetition structure.

3) FF utilization: The same partitioning also reduces FF utilization. The *flat* design uses ∼9.3–10.4K FFs independent of repetition, whereas the *decomposed* design uses far fewer FFs that are reduced from ∼5.1K FFs at ug=*1* down to 715 FFs at u=*256,* ug=*4* as repetition increases.

Overall, these results confirm that vendor compilers cannot exploit substring repetition in *flat* designs. Aligning the decomposition with the repetition structure yields up to 8× fewer LUTs and ∼1.5× higher Fmax, with the benefit growing directly with the degree of repetition.

VII. Limitations and Future Work

While our tool demonstrates strong results across both case studies, several limitations present opportunities for future improvement. Custom symbolic tokens cannot be split across group boundaries during hierarchical decomposition; a token must reside entirely within a single group, constraining `num_groups` and `group_datawidth` for patterns with wide tokens. Additionally, the token system currently supports a fixed set of predicates (`RANGE`, `NE`, `EQOR`). Extending this to operations such as bit counting, parity checks, or masked comparisons, and relaxing the cross-group constraint, would broaden applicability to domains beyond networking.

VIII. Conclusion

This paper presented a versatile open-source tool for automatic hardware parser generation, built on a parsing intermediate representation (PIR) that decouples application-specific frontends from an optimized RTL generation backend. The custom symbolic token system extends pattern matching beyond equality to support range validation, negation, and external port comparisons, operations that no prior hardware parser generator can express natively. Two case studies validated the tool's generality: an Ethernet protocol parsing frontend that outperforms prior academic baselines across most reported metrics, and a Snort intrusion detection frontend that generates efficient hardware rule matchers using the same unmodified backend. A controlled synthetic study further confirmed that hierarchical decomposition, by breaking monolithic matching into independently optimizable subproblems, yields substantial area and frequency improvements that vendor tool compilers cannot achieve on flat designs. Unlike commercial P4 compilers that produce encrypted, vendor-locked IP, our template-driven approach generates structured, human-readable, and vendor-agnostic RTL. The tool is open-sourced to enable rapid hardware acceleration across diverse pattern-matching domains.

REFERENCES

[1] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung *et al.*, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 51–66.

[2] J. Boo, Y. Chung, E. Baek, S. Na, C. Kim, and J. Kim, "F4T: A Fast and Flexible FPGA-based Full-stack TCP Acceleration Framework," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2023, pp. 1–13.

[3] 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," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[4] I. Burstein, "Nvidia data center processing unit (dpu) architecture," in *IEEE Hot Chips 33 Symposium (HCS)*, 2021, pp. 1–20.

[5] J. W. Lockwood, A. Gupte, N. Mehta, M. Blott, T. English, and K. Vissers, "A Low-Latency Library in FPGA Hardware for High-Frequency Trading," in *IEEE Symposium on High-Performance Interconnects (HOTI)*, 2012, pp. 9–16.

[6] A. Boutros, B. Grady, M. Abbas, and P. Chow, "Build Fast, Trade Fast: FPGA-based High-Frequency Trading using High-Level Synthesis," in *IEEE International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2017, pp. 1–6.

[7] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry, "Achieving 100Gbps Intrusion Prevention on a Single Server," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 1083–1100.

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[9] Advanced Micro Devices, Inc., "Vitis Networking P4 User Guide," in *(UG1308)*, 2025.

[10] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown, "Design Principles for Packet Parsers," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2013, pp. 13–24.

[11] P. Benácek, V. Pu, and H. Kubátová, "P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 148–155.

[12] M. Attig and G. Brebner, "400 Gb/s Programmable Packet Parsing on a Single FPGA," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2011, pp. 12–23.

[13] P. Mashreghi-Moghadam, T. Ould-Bachir, and Y. Savaria, "A Templated VHDL Architecture for Terabit/s P4-Programmable FPGA-based Packet Parsing," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2022, pp. 672–676.

[14] C. Kozanitis, J. Huber, S. Singh, and G. Varghese, "Leaping Multiple Headers in a Single Bound: Wire-Speed Parsing using the Kangaroo System," in *IEEE International Conference on Computer Communications (INFOCOM)*, 2010, pp. 1–9.

[15] K. Kumarasamy, D. Vaithiyanathan *et al.*, "FPGA Architecture for High-speed TCAM-based Packet Parsing," in *IEEE International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)*, 2024, pp. 1–6.

[16] 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 *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2018, pp. 147–152.

[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 *ACM Symposium on SDN Research (SOSR)*, 2017, pp. 122–135.

[18] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4-> NetFPGA Workflow for Line-Rate Packet Processing," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019, pp. 1–9.

[19] P4.org, "p4c: P4_16 reference compiler," https://github.com/p4lang/p4c.

[20] ——, "Behavioral model (bmv2)," https://github.com/p4lang/behavioral-model.

[21] Snort Team, "Snort 3: The next generation network intrusion detection system," https://www.snort.org/snort3, 2021, accessed: 2025-05-22.