The Binary Verdict Engine

An FPGA-based Verdict System for Assessing Structured Binary Data

MSc Thesis Computer and Embedded Systems Engineering

Stephen van der Kruk

Delft University of Technology

The Binary Verdict Engine

An FPGA-based Verdict System for Assessing Structured Binary Data

by

Stephen van der Kruk

to obtain the degree of Master of Science
in Computer and Embedded Systems Engineering
at the Delft University of Technology,
to be defended publicly on Monday June 23rd at 10:00.

Project duration:	May, 2024 – June, 2025	
Thesis committee:	Dr. ir. J.S.S.M. Wong,	TU Delft, Thesis Advisor
	Ir. F. Floros,	Technolution B.V., Daily Supervisor
	Dr. ir. G.C.M. Moura,	TU Delft, Committee Member

Cover: Binary X.509 certificate of www.tudelft.nl.

Abstract

Binary formats are used in low-level communication between systems. This binary data must be validated for correct structure and allowed content to ensure meaningful data exchange. This is especially important in high-security contexts such as air-gapped networks or classified communication channels. Such contexts can also be dynamic, requiring a flexible system to support quick changes to the allowed format.

This thesis presents the Binary Verdict Engine, an FPGA-based system that can assess a binary data stream by providing a verdict on the data that it has checked. It supports a wide range of binary formats without having to reconfigure the hardware to change formats. It achieves this through the programmability of its virtual machine architecture. The system executes instructions of a program binary, called a schema program. A language for writing schemas was created to define how the data should adhere to a specific binary format. Furthermore, a custom instruction set architecture was designed, consisting of instructions to traverse and assess data or to update control flow. Assessment consists of two types of assertions on the data. Field assertions exactly match or numerically compare a field of the data to a constant, and length assertions check whether the length of a section is equal to an earlier field specifying that length. The module design of the engine consists of: an input and output system that traverses the binary data per data field, a controller that executes instructions and manages verdict state, an instruction fetcher that provides instructions to the system and manages the instruction pipeline, and a stack for length assertions and control flow utility.

The design is implemented on an FPGA and evaluated for flexibility and performance. Benchmarks range from assessing externally defined flat formats, such as Internet packet headers, to self-describing hierarchical formats, such as ASN.1 DER. The varying use cases across benchmarks show the system's flexibility, which it trades off for a lower performance compared to fully custom FPGA designs. Synthetic benchmarks show a reciprocal decrease in throughput and a linear increase in latency once schemas become more complex, and show flexibility when switching schemas, as downtimes are minimal when switching between them. The system establishes itself as a flexible validation system for diverse and dynamic use cases.

Acknowledgements

This thesis marks the end of my internship at Technolution and the end of my time as a student at the TU Delft. It closes a chapter of my life where I have learned and experienced a lot, and have developed myself as a person and engineer. I take this opportunity to thank the people around me, as this would not have been possible without the incredible support from them.

I want to thank my daily supervisor from Technolution, Fanis Floros, for his invaluable guidance during the project. Your help and systematic approach to problems enabled me to continuously improve my work and made me a better engineer. I would also like to thank Sijmen Woutersen for his constructive insights during key design moments of the system.

I want to thank my thesis advisor from the TU Delft, Professor Stephan Wong, for his time and effort put into our meetings and in providing insightful feedback, keeping the project and thesis writing on track. Additionally, I would like to thank Professor Giovane Moura for his time and effort in being part of the thesis committee.

Finally, my heartfelt gratitude goes out to my friends and family. To my parents, I am forever thankful for your support throughout my life and education. To Wenjia, thank you for your encouragement and your continuous joy that made difficult times endurable. To all my friends from high school, university, and badminton, thank you for all the fun and experiences we've had so far. I look back on this chapter of my life with great fondness, and I look forward to what the future holds.

Stephen van der Kruk
Delft, June 2025

Contents

List of Abbreviations

Abbreviation	Definition
ASIC	Application-specific Integrated Circuit
ASN.1 DER	Abstract Syntax Notation One Distinguished Encoding Rules
AXI4l	Advanced eXtensible Interface 4 Lite
AXI4s	Advanced eXtensible Interface 4 Stream
BFM	Bus Functional Model
CPU	Central Processing Unit
DSL	Domain-specific language
FIFO	First-In-First-Out
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
ICMP	Internet Control Message Protocol
IoT	Internet of Things
IPv4	Internet Protocol version 4
ISA	Instruction Set Architecture
LUT	Lookup Table
MPLS	Multiprotocol Label Switching
P4	Programming Protocol-Independent Packet Processors
PC	Program Counter
RAM	Random Access Memory
RDL	Register Description Language
TCP	Transmission Control Protocol
TLV	Tag Length Value
UART	Universal Asynchronous Receiver-Transmitter
UDP	User Datagram Protocol
VHDL	VHSIC Hardware Description Language
VLAN	Virtual Local Area Network
VM	Virtual Machine

1 Introduction

Data exchange is at the core of communication between systems. Low-level machine-to-machine communication involves the data being transferred in binary formats. These are, for example, used in the lower layers in networking or in distributed embedded systems, where the focus is on fast, machine-to-machine-based communication. To ensure meaningful communication between such systems, this data must be correct in both structure and content.

In contexts where security is of major importance, such as air-gapped networks or classified communication channels, extra efforts must be made for security to ensure that the data is valid for safe processing. Such contexts can also be dynamic, where the format of allowed data can frequently change, requiring a flexible system to support future changes to the validation policy. A software-based system can support dynamic use cases, but such levels of flexibility allow for more ways to exploit such a system from a security perspective. On the other hand, a customised hardware implementation is usually targeting a specific use case, allowing for less flexibility in dynamic contexts.

1.1. Problem Statement

This introduces the need for a system that can, using a description of a format, interpret a binary format, assess its structured binary data, and provide a verdict on this data based on the assessment. Assessments include range checks, comparisons and exact matches. Different descriptions that can be interchanged allow for a wide range of binary formats to be interpreted and assessed. The system is preferably implemented on an FPGA as a custom hardware-based engine and takes in a data stream at the input, with the sole function of interpreting and assessing this stream. Having the hardware explicitly defined minimises the attack surface, as anything not described in the design cannot be exploited. Therefore, this benefits the overall security of the system.

The main design challenge lies in supporting a wide range of binary formats on an FPGA, without having to reconfigure the underlying hardware. Creating this ”reconfigure-once” design allows users of the system to create a description of the binary data and deploy this to the system without the need to generate new FPGA bitstreams. This makes the system more accessible to its users, as generating bitstreams requires advanced knowledge of the underlying FPGA platform and synthesis toolchain. Reconfiguring the hardware also introduces more security risk, as it can be reconfigured into anything that can exploit the system, as long as there are matching interfaces. The verdict system configuration process aims to be quicker than the process of generating bitstreams.

Therefore, the **main research question** this work aims to answer is:

> *How can we build a reconfigure-once, FPGA-based verdict system to assess structured binary data across different formats?*

The overarching topic of the research question is how we build a flexible system for our purpose. This flexibility is expressed in two dimensions. First, the system must support assessment within a class of binary formats, meaning it can handle different structures and data types to interpret and assess these formats. Second, the system must be able to quickly and drastically switch its interpretation and assessment from one format to another. The flexibility of the system is important for its effectiveness for its users. Hence, it is an important factor to evaluate in the system.

Two additional research questions this work aims to answer are:

- *What hardware architecture should be chosen for this verdict system?*
- *How can we describe binary data such that the verdict system can check adherence to this description?*

The following **goals** are formed to answer the research questions stated above:

1. Define the design requirements and hardware architecture for designing the binary verdict system.
2. Design a description method for binary data interpretation and assessment.
3. Design and implement the binary verdict system.
4. Evaluate the implemented binary verdict system in hardware for flexibility and performance.

1.2. Methodology

To achieve the aforementioned goals, the methodology to reach each goal is listed below.

Common steps for Goals 1 and 2

1. Investigate the properties of binary data formats.
2. Investigate how binary data can be interpreted and assessed.

Goal 1

1. Investigate what architectures other hardware-related systems that handle binary data have.
2. Enumerate the design requirements for the binary verdict system based on the performed research.
3. Select a suitable architecture based on the performed research and requirements.

Goal 2

1. Investigate related description methods for describing, interpreting and assessing binary data.
2. Design a binary description method that adheres to the design requirements.
3. (If time permits) Implement tooling to convert from a format description to a verdict system configuration.

Goal 3

1. Design the binary verdict system based on the design requirements, chosen architecture and designed description method.
2. Implement the design on the FPGA.

Goal 4

1. Define benchmarks for assessing the flexibility and performance of the binary verdict system.
2. Implement a hardware testbed for executing the benchmarks on the binary verdict system.
3. Execute the benchmarks on the binary verdict system and compile the results.
4. Evaluate the results of the implemented hardware and benchmarks and compare with related systems.

1.3. Thesis Outline

The thesis is divided into 6 chapters. After this introductory chapter, Chapter 2 describes the background required for the thesis. This discusses binary data formats, binary data assessment, and explores related works for various hardware architectures and binary description methods. Chapter 3 handles the design of the binary verdict engine. The created design requirements of the system are discussed, after which the binary description method and hardware architecture (ISA and modules) are explained. Chapter 4 explains the implementation of the system. As well as two implementations which help in the development of the system. Chapter 5 explains the implemented testbeds for evaluating the system and analyses the results obtained from the performed benchmarks. Chapter 6 summarises and provides the contributions of this thesis, and discusses various proposals for future work.

2 Background

Based on the problem statement in Chapter 1, we perform a background study in this chapter to investigate relevant topics for this project. We investigate binary data regarding its structure, how to properly traverse and assess it, and what benefits FPGAs can provide for traversing and assessing binary data. To investigate the additional research questions, we perform a literature study to consider related systems in terms of their architecture and how they describe binary data. The insights received from this background study provide the necessary knowledge to create concrete requirements and design a system in the next chapter.

This chapter is organised as follows. Section 2.1 explains the focus of the system on binary data, and we create a classification of how binary formats organise their data. Section 2.2 considers the concepts behind binary parsing of binary formats. Section 2.3 provides a short explanation on why FPGAs were chosen as the implementation platform for this system. Section 2.4 covers the related work, providing an exploratory overview of related parsing and filtering systems and various domain-specific languages (DSLs) for describing binary data. Finally, Section 2.5 concludes this chapter.

2.1. Binary Data

To create a system that can process binary data, we need to consider how bits are used to represent data. Vectors of bits are used to represent more complex information; the structure and layout of these vectors are determined through a format. Text-based formats, such as XML or JSON, represent information in human-readable characters and require a character encoding, such as ASCII or UTF-8, to convert these characters to bytes, i.e., every bit vector consists of 8 bits, representing a character.

This work focuses on binary formats, which contrast with text-based formats as there is no human-readable representation; they are meant to be processed and exchanged by machines. The information is directly represented as data fields within the bit vector. The binary data we consider are bit vectors of finite length, such as packets, messages, certificates, etc.. The working definition we will use for a bit vector encoded in an arbitrary binary format is a **binary blob**. Protocols often make use of binary formats to have a compact representation of information in their message structure. As an example, many of these can be found in the TCP/IP protocol suite.

This section will further explain how binary formats relate to encodings and protocols, as they are frequently used alongside binary formats. Afterwards, we discuss how binary formats can be classified through different structural concepts. This relates to how the verdict system should eventually support these concepts to interpret various binary formats structurally. Finally, the binary data types that the system should be able to interpret are discussed.

2.1.1. Formats, Encodings, and Protocols

As these definitions are occasionally mixed up, it is good to set a working definition for binary formats, encodings, and protocols. This ensures there is a clear relation between the terms and that the discussed topics can build upon these definitions:

- A binary **format** specifies the structure and layout of the binary data, such that every field has properties such as a position, bit/byte width, and data type. When a machine knows the binary format, it can correctly store and interpret the binary data while adhering to the format.
- A binary **encoding** defines the rules on how to transform a higher-level data format, such as text, numbers, or data structures, into a specific binary format.
- A binary **protocol** is a set of rules which specify how two or more systems can communicate with each other. It specifies a binary format to give structure to its messages, and it specifies how messages are sequenced to ensure meaningful communication between systems.

Both a binary encoding and a binary protocol link in their definition to a binary format, but each has a distinct meaning.

2.1.2. Classifying Binary Formats

Binary formats can be classified according to how they structure their data. As different formats are designed for different trade-offs, domains, and use cases, there is no universal classification that can cover every binary format. Such a classification becomes either oversimplified or too granular to be practical. To this end, the current classification focuses on the base IPv4 header [1] (without extensions) and ASN.1 DER [2], which are two contrasting binary formats in the domain of networking and security. Creating a classification out of their underlying concepts provides a generalisation to see what other formats can be classified by this classification. The two dimensions of this classification are described below:

Externally described vs. Self-describing

The base IPv4 Header is an externally described format. Such formats have the structure of their fields declared in an external definition, such as a predefined specification. The formats provide a fixed position and width for each field, which a system can use as a blueprint to extract and interpret the fields. These formats transmit fields more compactly but are inflexible to changes.

ASN.1 DER is a self-describing format. These formats bundle the fields with metadata, such as tags or other structural markers to delimit and describe fields, allowing machines to extract and interpret fields without an external definition describing the structure and data type of each data field. In the case of ASN.1 DER, the tag describes the field type, which is followed by a length field, which refers to the length of the value field. This is called a Tag-Length-Value (TLV) format. Self-describing formats provide greater flexibility, as the structure of the data can be derived through processing the metadata. However, the metadata introduces an overhead for processing and transmission.

It should be noted that formats can make use of both methods to structure their data. This is useful in contexts where there needs to be a balance between efficiency and flexibility. The full IPv4 header includes this, as it has a fixed 20-byte header for fast processing, but it can include one or more optional ”Options” fields formatted as TLVs, allowing for more expressivity within the format.

Flat vs. Hierarchical

A flat format, like the base IPv4 header, organises the fields in a single sequential layer. This is a simpler way of extracting and interpreting fields, but it is not able to represent more complex relationships within the data. A hierarchical format allows fields or groups of fields to be nested within other data elements to represent a relation between data elements. ASN.1 DER supports this data hierarchy through their `SEQUENCE` and `SET` field types. While flat formats are faster to process due to their simpler structure, hierarchical formats allow for more complex relationships to be expressed.

Other Concepts

The base IPv4 header and ASN.1 DER do not cover all concepts for structuring data within a binary format. Other binary formats can contain other structural concepts which are not part of the current classification. This can be introduced as other classification dimensions, such as little-endian vs. big-endian data, and row-based vs. columnar formats. These are dimensions, for example, Viotti and Kinderkhedia [3] and Maltsev et al. [4] used in their classification. However, because of their focus on different domains or text-based formats, their dimensions are outside the scope of the project, emphasising again that a universal classification is not practical to create.

2.1.3. Binary Data Types

Binary formats consist of data fields of various data types. Data types have a binary representation which a system must recognise and support when assessing the data. Simple data types, such as null or boolean types, only have a single or limited number of binary representations, and can be assessed by checking if the value matches the binary representation. String data types are more complex to check as their underlying binary representation requires knowledge of the character encoding used to correctly interpret and check the string. ASN.1 DER, for example, has multiple string types, each for a different kind of character encoding.

More complex data types are numeric data types, namely, unsigned and signed integers, as well as fixed-point numbers. Their binary representation can be checked for an exact value, but can also be used for numeric comparisons. This implies that a system should include comparison hardware for such comparisons. Many binary formats also include floating-point data types, which can also be matched or compared, but require different hardware to support the comparison of floating-point types.

To highlight the difference with text-based formats, numbers in text-based formats cannot be directly used in numeric comparison, as numbers are represented as a string of decimal characters. Using this in a comparison leads to a lexicographic comparison, which has fewer practical use cases compared to using the number for an actual numeric comparison. For compatibility with numeric comparisons, a conversion from a numeric string to a binary number is required, which requires more hardware and introduces more processing overhead. Furthermore, there are losses in precision during this conversion, as decimal fractions often cannot be represented in binary with exact precision. Because of these disadvantages and the project's focus on binary formats, we will not include this conversion.

2.2. Binary Parsing

Binary parsing is the process of extracting the data fields from the binary data, as well as checking whether the data adheres to the format structure. The latter part of this process relates to this research, which is why we discuss binary parsing and subsequently, why we cover binary parsing systems in the related work (see Section 2.4). The following section discusses the implications of performing the binary parsing process in a single pass, as this is beneficial for handling stream-based data. Afterwards, we discuss how assertions are used in binary parsing to check that the binary data adheres to the binary format in both structure and content. Finally, it is discussed that a binary parser needs a description method, such that it can understand how to parse a binary format.

2.2.1. Single-pass parsing

In single-pass parsing, the data fields are extracted, and adherence to the structure is checked in a single traversal. The first advantage is that a single traversal over the data has the benefit of low latency and memory usage compared to other types of parsers. Other parsers might perform multiple passes or perform backtracking, which requires memory to buffer the input and introduces more latency as more processing is done on the same data. The second advantage is that single-pass parsing outputs its results in real time. As soon as a field enters from the input, a single-pass parser can extract, validate for content and structure, and send it to the output. This also enables early termination of the parsing process in case a field is found to be invalid. If only valid blobs are important, a parser can drop further processing on the invalidated blob and move on to the next. These advantages of single-pass parsing make it suitable for a streaming system, where it is desired to have minimal latency between the input and output stream, and where data invalidation needs to be recognised as soon as possible.

However, single-pass parsing does have limitations. The single traversal cannot handle complex relations within the data. For example, when a later data field influences the interpretation of an earlier field. This would require other parsing methods, such as using a lookahead buffer or performing multiple passes to process such relations. Furthermore, error recovery is minimal, as an invalidated field cannot be traversed again with a different interpretation, which might still validate the field. Resolving these ambiguities within the data is something a multi-pass parser would be able to do. These limitations are not major disadvantages in our context, as in general the binary formats that we consider (see Section 2.1.2) do not contain complex relations and contain unambiguous structure, therefore traversing the data in a single pass suffices for assessing these formats.

2.2.2. Assertions

Performing assertions on binary data is central to a binary parsing system, as it is the means to check the **content** and **structure** of a format. Assertions are used to find inconsistencies in the expected format, such as unexpected types, forbidden values, or missing data.

Assertions on content focus on whether data fields (not metadata) in the format adhere to a certain value. It requires interpretation of the data type, after which it can be compared with one or more expected values or ranges. For an externally described format, this is the only way to check whether the value of each field is correct, as there is no metadata to check for.

Assertions on structure are used for checking metadata fields, which are relevant for self-describing formats. These ensure that tags, lengths, and other structural markers have the appropriate position and value. These assertions need to take the relation of the metadata with the content data into account. For example, when a tag field determines that a data field will follow with a certain width, data type, and value, the appropriate content assertions are required on the data field to check that the value is correct given the width and data type. When length fields are included, it is important to cross-check the length field with the actual length of the section to which it is referring, otherwise, this can lead to cascading errors in parsing if the length is assumed to be correct.

On top of that, assertions can be configured to be stricter than what the data format requires. For example, an assertion can be placed on an address field to permit only addresses from a certain range, rather than the entire address space. Similarly, assertions on structure can enforce stricter constraints, such as allowing only specific tags or types to appear in a certain nesting. This enables finer control, ensuring that only a subset of the binary format is accepted according to the user's needs.

2.2.3. Describing Formats for Parsing

For binary parsing, we require a language to describe a binary format, such that a parser knows how to parse a format. Such a language provides a higher-level abstraction to the system, where a user can describe the organisation of the data by specifying fields, lengths, relationships, etc. This description needs to be converted to an input for a parser to understand. To our end, this means a description needs to be converted to assertions that check whether the binary data adheres to the specified format. Section 2.4.5 discusses such DSLs from related research.

2.3. FPGAs

The platform for implementation is a field-programmable gate array (FPGA). To elaborate on why it was chosen for this task, the following points were considered:

1. An FPGA is better suited to streaming applications compared to a software-based solution, as the design can be oriented around processing a data stream, allowing for much quicker access to the data than what a software program can achieve. A program running on a CPU often requires many instructions to be executed before anything functional to the process is performed.
2. FPGAs are not limited to byte-aligned processing. Therefore, bit-level operations on an FPGA can be performed much more efficiently compared to software. This property is desirable for processing binary formats, which often have fields that represent data at the bit level. Software programs running on a CPU are bound to operations on byte-aligned units, requiring shifting and masking to operate on values at the bit level.
3. A CPU contains more functionality than required. It can run any kind of program as a CPU contains a Turing-complete instruction set. This additional functionality increases the risk of vulnerabilities due to extra hardware that does not contribute to the purpose of checking whether binary data adheres to a user-specified format, making its use undesirable in the context of security for this project. A design on an FPGA can be customised to contain the minimum hardware necessary for its goal, lowering the risk of exploits.

These points showcase why an FPGA was chosen over a software-based solution on a CPU.

In contrast, the hardware implementation in an ASIC was not considered for this project, as it was outside the scope of the implementation. Although it would likely result in a faster system, designing a system for an ASIC takes much longer and is much more expensive than implementing a design on an FPGA. It is only worthwhile for the large-scale production of such an ASIC. An FPGA design may be slower compared to an ASIC, but the process of conceptualising, prototyping and implementing on an FPGA will be much quicker, which is needed for this project.

2.4. Related Work

This section will discuss the related binary parsers and filters presented in related research. We performed a literature review to find a suitable architecture to form the basis of our system. Related FPGA-based parsers and filters have been researched that have a degree of configurability in what they can parse. An important note is that the FPGA-based systems that were found during this literature review solely focus on the domain of networking. The presented parsers are built around parsing the headers of various protocols in the TCP/IP protocol suite. We need to consider this when selecting a suitable architecture from related research, as our system needs to support a wider category of formats. This section is structured as follows. Sections 2.4.1 to 2.4.4 each introduce a category of FPGA-based packet parsing/processing systems and explain how they achieve their configurability. This is then followed up by a showcase of related research. Finally, a discussion is held on how the category relates to the desired functionality of the to-be-implemented verdict system. Finally, Section 2.4.5 discusses related packet parsing DSLs.

2.4.1. Generated HDL systems

The first category of FPGA-based systems are packet parsers/filters/processors that are created from a generated HDL framework. The rise of Software-Defined Networking (SDN) has driven the need for more flexible packet parsers. This category addresses this need through FPGA-based systems that generate a hardware description or implementation from a higher-level packet parsing, filtering, or processing description. This enables such systems to be flexible to a degree while still being able to create a customised hardware solution to process different kinds of packet formats.

One of the earlier frameworks to generate HDL is the system by Benáček et al. [5] introduces a pipeline which converts a P4 parse-graph (P4 is a DSL for programmable packet parsing and processing) description into synthesizable VHDL code for deployment. In hardware, the packet header goes through a pipeline in parallel, consisting of multiple extraction blocks where every block extracts the fields from a different protocol. Cabal et al. [6] build upon this generator and can generate parsers with a maximum throughput of 1 Tbps by handling multiple packets in parallel. Santiago da Silva et al.[7] take a similar approach but allow multiple protocol extraction blocks to execute in parallel, making them more flexible regarding which protocols are used within the packet. Mashreghi-Moghadam et al. [8] built a generic packet parser that makes use of pipelined, templated VHDL blocks, each parsing one header layer per clock cycle. These blocks are configured through bit vectors and parameters, allowing for much hardware reuse. With their design, they also managed to reach a throughput of 1 Tbps.

Besides purely hardware-based packet parsers being generated, ClickNP from Li et al. [9] generates hardware from a custom C-like description for an entire packet processor. This system can parse and filter, but can also perform more network functions on packets. Wang et al. [10], Ibanez et al. [11] and Yazdinejad et al. [12] do the same but generates hardware from a P4 program, which is a more standardised way of programming network devices. Making such frameworks compatible with other P4 programs.

Finally, Fiessler et al. [13] created a packet filter system that splits the processing of packets against user-defined filter rules to an FPGA-based filter as well as a software firewall. It differentiates complexity in the rules, where simple rules can be processed at high speed in the FPGA, while complex rules are handled in software. Showing that the more complex the analysis, the more flexible the system must be.

Generated HDL systems offer high performance by generating a hardware design for a specific binary format from a user-defined high-level description. Packets pass through in parallel, allowing quick extraction of data fields. In some cases, multiple packets can be processed in parallel, allowing for scalable solutions. However, these systems are not flexible for changing the format at runtime, as any change in the parser or filter description requires new hardware to be generated and synthesised, which is a time-consuming process and requires expertise in the tooling to integrate it into a desired

FPGA platform. Furthermore, the data must be complete before it is parsed by the system. This makes the processing less real-time in the case of a data stream, as the data arrives in segments. Even with these challenges, generated HDL systems are powerful for high-speed networking in contexts where performance is more important than having to reconfigure frequently.

2.4.2. Templated Generic Parsers

The second category, templated generic parsers, offers a more flexible approach to packet parsing on FPGAs. Unlike the first category, which requires hardware to be updated for changing or updating protocols, templated generic parsers allow changes during runtime to support different binary protocol formats without updating or generating hardware. This is usually done by the architecture, consisting of generic hardware structures for extracting fields. These are parametric so that they can be configured during runtime. An overview of notable research is showcased below.

Switchblade from Anwer et al. [14] is a packet-forwarding platform that makes a packet go through a pipeline for parsing and forwarding. Within the pipeline, hardcoded modules can be selected through register access for parsing and processing different protocols. This provides a coarse level of configurability during runtime. Similarly, Puš et al. [15] work with the same idea for parsers, where at runtime the parsing format can be switched to a different module to recognise different protocols, but without any further granularity for altering the layout of the format within the modules.

Attig and Brebner [16] designed a packet parsing system that makes use of programmable modules. Each module parses one protocol layer of a packet header and is parametrised using 'microcodes'. These are configuration vectors that determine how the protocol is laid out and contain more parsing functionality, such as configuring conditional checks to determine which header to parse next. Allowing this amount of granularity in these vectors to be configurable at runtime allows for a highly flexible parsing system for packets. Similarly, Lixin et al. [17] provides the same granularity through a software-based configuration, which generates a descriptor that contains information on how to parse the various headers per packet. Furthermore, matching rules are stored in RAM, which can be changed during runtime. The system uses this to extract fields and determine which next protocol format to parse in the header. Finally, Sun and Guo [18] created a packet parser which uses one parametric parsing module. This is used to parse a stack of protocol headers iteratively. Matching and configuration settings are stored in RAM and are used by the state machine to extract relevant fields and to find the next protocol header. When the next protocol is known, the next settings are retrieved from RAM to configure the module, after which the next parsing iteration starts. This architecture has the benefit of a low hardware footprint.

Templated generic parsers offer a more flexible approach to parsing configuration than generated parsers, while still containing a fast, parallelised extraction of data fields from the packet header. Having configuration (partially) possible during runtime allows quicker changes to the parsing pipeline behaviour compared to the first category. However, due to the general architecture consisting of multiple generic stages that can only extract fields, there are difficulties in extending such a template for implementing assertions and supporting data hierarchy. For number comparison assertions, it would require a stage to make comparisons of arbitrary field width for an arbitrary number of fields that arrive in parallel. For checking data hierarchy, it would only support one nesting per stage by design. This is fine for packets where one protocol is nested inside another, but any more complex hierarchy (multiple nestings in one layer or variable-length nestings) would be more difficult to implement. Another constraint would be the number of pipeline stages, as this will determine the number of operations that can be performed on the data. If there were more assertions than what the pipeline supports, resynthesis would be required to add more stages. This leaves this category with complex issues to solve for the desired purpose of the system.

2.4.3. Match-action systems

The third category of related systems is match-action systems. These systems operate on packets where they parse headers and match them to values in a table. A matching operation can be an exact match, ternary matching, or longest prefix matching. A successful match is then linked to an action. This can be, for instance, a modification of the header or forwarding the packet to a port. Multiple match-action operations are put in a pipeline, making it possible to perform multiple operations on a packet.

Match-action systems make networking flexible because these match tables can be reprogrammed to perform a different match operation or link to another action. Programming the match tables is done with P4 [19]. Systems that use P4 programs to process packets were described earlier in Subsection 2.4.1. However, such FPGA-based systems required resynthesis when the system had to run a new P4 program. In this subsection, FPGA-based match-action systems are discussed, which can update to a new P4 program during runtime.

Bosshart et al. [20] introduced the Reconfigurable Match Tables (RMT) model, a hardware architecture for match-action packet processing. It was later coined as PISA (Protocol-independent switch architecture). It works with a parser that extracts packet header fields. The extracted fields go through stages, where each stage contains a match table and action primitives. The parser, match tables and action primitives can be programmed at runtime, allowing different packet formats to be parsed and processed. This provides quick adaptability to changing network requirements. The research of Bosshart et al. achieved a 10 Gb/s throughput for a single pipeline. Luinaud et al. [21] refined the architecture by alleviating some bottlenecks seen on FPGA-based PISA implementations and performing a design space exploration. They achieved a maximum throughput of 786 Gb/s for a single pipeline.

Match-action systems have also been expanded upon in terms of functionality in [22], where programmable ASICs for match-action processing with fixed actions were expanded in functionality using FPGAs. Thus, the P4 program used to program the ASIC could be extended with functional modules for the different hardware targets. Furthermore, match-action systems were made stateful in [23]. This was done by the idea that stages in the pipeline can make use of an FSM. This allows for more complex network functions that use state, such as firewalls that track packet flows. This shows that match-action systems can be extended to allow even more functionality.

Match-action systems are powerful for packet processing because they allow for complex network functions to be executed while keeping up with networking speeds. For our purpose, however, the match-action pipeline contains more functionality than necessary for providing verdicts on binary data. The functionality regarding parsing and matching is also built around Internet packet headers. Exact matches are very fast and efficient, but number comparisons are not supported. Similarly to the second category, there is a maximum limit to the size of the pipeline, resulting in a finite number of matches and actions that can be done.

2.4.4. Hardware-based VM

Hardware-based Virtual Machines (VMs) are systems that operate similarly to a conventional processor, where processing happens through the execution of instructions. This architecture is inherently runtime-programmable because the behaviour of the system can be changed by updating the instruction memory. The presented VMs are built for packet parsing and processing and perform this task through the execution of instructions.

Zolfaghari et al. [24] created a custom processor for header parsing. Instructions are decoded by a control unit, which then controls the header. The control unit extracts fields from the header at a specific layer and determines which header to parse next, and can branch to a subroutine to parse the next header. It also keeps track of length sections in the payload using a stack, returning once the length has been reached. The has the same functionality as the parser from the RMT model [20] but without making use of Ternary Content Addressable Memory (TCAM) for matching fields, which would otherwise consume more power and area. A single pipeline achieves a faster throughput compared to the RMT parser, namely 27 Gb/s.

hXDP by Brunella et al. [25] introduced an FPGA-based model that runs the XDP[26] framework in hardware. XDP allows quick access to the incoming network traffic and runs an eBPF[27] program for processing the packet. Similar to hXDP, Pacifico et al. [28] created eBPFlow, an FPGA-based multi-core environment with 16 parallel cores which can execute eBPF programs. Both systems can offload network functions to the FPGA, such as firewalling or Deep Packet Inspection (DPI), while still allowing the same programmability from a software-based environment using eBPF.

Hardware-based VMs are flexible systems because they allow different packet formats to be parsed and processed based on the flow of instructions. The main difference compared to the previous categories is that the VMs operate on the data using a ”run-to-completion” model, meaning they need to complete processing one packet before another can enter the system. The other categories could support multiple packets simultaneously in the system through pipelining, allowing for higher throughput. Zolfaghari et al. [29] point out this performance difference, showing that pipelined header processing achieved a 40x higher throughput compared to run-to-completion. However, they note that run-to-completion is more flexible in its sequence of actions, whereas pipelined designs have a relatively short sequence of actions due to the length of the pipeline. Furthermore, extending the pipeline with another stage requires resynthesis. For a VM, however, this is much more straightforward because the only limit to the number of actions that can be executed for a packet is the size of the instruction memory, and swapping instructions can be done without having to change the hardware.

2.4.5. Packet Parsing DSLs

From the showcased packet parsing research, custom DSLs have been introduced to describe the binary formats of protocols to a packet parser. They provide an abstraction for the user to the underlying system, allowing the user to define how the system should parse incoming data. In existing research, a format description is used to generate a hardware description for the parser or is compiled to input for the underlying parser, such as configuration vectors or instructions.

These languages share several concepts for describing packet formats:

1. **Ordered field declaration**: Fields within the packet format are declared in the sequence which reflects their actual order in transmission, and therefore the order in which they should be parsed.
2. **Labels for declared fields**: Each field can be given a unique label, used to reference the field when required for its value. For example, when a field determines what the next header will be, we refer to it via its label.
3. **Bit/Byte width indicators per field**: Every field is given a bit or byte width, such that the parser knows the exact layout of the format for extraction.
4. **Conditional Header Parsing Notation**: Notation is included for conditionally determining the sequence of headers to be parsed. This is essentially a conditional statement, where the value of a specified ”next header” field determines which protocol header is parsed next. This allows for the implementation of parse graphs for headers, which is a directed graph containing the supported headers, with the order of protocol headers that can follow after another.

From the discussed research, examples of DSLs are the PP language from Attig and Brebner [16] and the DSL from Zolfaghari et al. [24]. Both make use of the aforementioned concepts to build header format descriptions. Their descriptions are compiled down to microcodes or instructions, respectively, that the parsing hardware can use to configure itself. Furthermore, P4 from Bosshart et al. [19] is the standard for programming network devices. Besides containing functionality to define a parsing description for the header data, it contains much more functionality for implementing network functions. From the discussed research, it is shown to be platform-independent as well, as many networking systems use P4 to generate parsing hardware or configure existing hardware. Making P4 the most versatile DSL in the networking domain.

These DSLs inspire our own ”schema” language (see Section 3.2). This language would expand beyond the discussed DSL concepts to support the notation for data types, self-describing formats, hierarchy, and assertions. This would enable the language to support a wider range of binary formats (see Section 2.1.2), which go beyond the domain of protocol headers, and it would cater to our need to assess whether the binary data adheres to the declared format.

2.5. Conclusion

In this chapter, we discussed the required background for the thesis project. We set the project scope on binary formats. A binary format classification was created to describe the binary formats that needed to be supported, where we identified that formats could be externally described or self-describing (or a hybrid), and flat or hierarchical. We described how binary formats hold data fields with various data types and how these types can be checked, such as exact matching on strings and comparison on integers. Afterwards, the binary parsing process was explained, specifically the advantages and disadvantages of single-pass binary parsing. We discussed how binary parsing relies on assertions to check for the format structure and content, and why a DSL was required to provide parsing configurations. Finally, for the general background, we discussed how FPGAs provided advantages over CPUs related to better data streaming, bit-aligned operations, and purpose-built hardware. We also briefly touched upon why FPGAs were preferred over ASICs for this project. Afterwards, the related work was discussed and provided an exploration of various categories of configurable hardware-based packet parsing systems. The first category, generated HDL parsers and filters, generated hardware based on a parsing/processing description. They provide high performance for field extraction but are inflexible to change. The second category, templated generic parsers, is more flexible as it supports changes in the format description during runtime through generic hardware stages, but has challenges related to supporting assertions and data hierarchy. The third category of systems, match-action systems, are hardware-based systems with wide functionality for executing network functions. It also remained a challenge for this category to support assertions and data hierarchy. The final category, hardware-based VMs, are programmable systems which are very flexible, at the cost of high-performance parsing. Lastly, we considered what concepts several packet parsing DSLs had in common in terms of notation. The showcase of related research provided the knowledge to make a design direction for the system.

3 Design

The researched topics from the background study in Chapter 2 led to the creation of concrete requirements, hardware architecture, and binary data description method, which this chapter discusses. These form the basis for a top-down design process of our verdict system, named the Binary Verdict Engine. We start with explaining a schema language, thereafter discuss the ISA, and finally, handle the module design. This provides a complete design which is ready to be implemented on a platform, which we discuss in the next chapter.

This chapter is organised as follows. Section 3.1 discusses the requirements of the system and the chosen architecture after having performed the background study. Section 3.2 will introduce the schema language, which provides an overview of the desired functionality of the system. Section 3.3 discusses the Instruction Set Architecture with provided pseudo notation for programming schemas. Section 3.4 connects the schema language to instructions, showing how a program should be compiled down to instructions for the system. Section 3.5 introduces all the modules within the system and provides a top-level module overview of the full system. Finally, Section 3.6 concludes this chapter.

3.1. Requirements and Design Direction

The problem statement and the performed background study led to the following requirements. These are categorised into data assessment and architectural requirements. The data assessment requirements describe all required functionality for assessing streams of binary blobs. The architectural requirements describe functionality that the architecture should support and how it should handle input and output. All requirements are explained in Sections 3.1.1, 3.1.2, and 3.1.3

Data Assessment Requirements

- **REQ1**: The system shall output a verdict per binary blob.
- **REQ2**: The system shall update the verdict as soon as the system knows the data is invalid.
- **REQ3**: The system shall check for adherence to the data against a schema.
- **REQ4**: The system shall sequentially traverse the binary data in segments of specified bit lengths.
- **REQ5**: The system shall recognise data fields as raw binary, unsigned, signed, unsigned fixed-point and signed fixed-point data types.
- **REQ6**: The system shall perform equality and numeric comparison assertions on the data fields.
- **REQ7**: The system shall be able to perform multiple assertions on a data field.
- **REQ8**: The system shall be able to use the current data field as the size of the next field.
- **REQ9**: The system shall be able to switch its control flow based on an assertion condition on a data field.
- **REQ10**: The system shall be able to check the length of the section.
- **REQ11**: The system shall be able to repetitively assess repeating sections in a binary blob.

- **REQ12**: The system shall be able to repetitively assess repeating sections until an exact length is reached.

Architectural Requirements

- **REQ13**: The system shall be implemented on an FPGA.
- **REQ14**: The system shall take a stream with arbitrary length as input.
- **REQ15**: The system shall be able to change schemas without having to reconfigure the hardware.
- **REQ16**: The system shall be designed and operate with security in mind. (see Section 3.1.3)
- **REQ17**: The system shall keep input and output interfaces to a minimum.
- **REQ18**: The system shall keep the content of the output stream identical to the input stream.

3.1.1. Derivation of Data Assessment Requirements

Chapter 2 discussed how binary formats organise their data by data fields. Systems discussed in the related work base their operation around data fields, either for extraction, filtering or processing. REQ4 generalises this and notes that our system will base its operations on segments of specified bit lengths, taken from the input stream, which can be either single data fields or multiple fields.

Furthermore, related systems make use of DSLs to define a binary format and what operations are performed on them. Our system will also include a language, called the ”schema language”, where a schema for a binary format can be defined and include what assessment operations are performed on each data field. This is what REQ3 denotes. Adherence of the data to this schema determines whether the data adheres to a binary format and other data constraints. Schemas are explained in Section 3.2.

Data assessment REQs 5, 6 and 7 include the system’s functionality related to assertions on data fields. This includes interpreting data fields as various data types, and being able to perform various assertion operations on these data fields, with multiple assertions possible per field. These requirements already allow for flat and externally described formats, such as the base IPv4 header, to be assessed for validity.

Data assessment REQs 8 to 12 extend the assessment functionality to support hierarchical and self-describing formats, such as Tag-Length-Value formats like ASN.1 DER. The background study identified that such formats, besides checking for value, also need to be checked for structure. This can require handling variable data field sizes (REQ8), conditional statements (REQ9), length checking of data sections (REQ10), and repetition of various data sections (REQ11 and REQ12).

3.1.2. Architecture Selection

Section 2.4 discussed four categories of related systems. Out of these four, the Hardware-based VM design described in Section 2.4.4, where the instructions execute the functionality of the schema, suits the system best, as this shows flexibility in multiple dimensions:

- **Format variability**: Formats differ in structure and content, therefore, assessments highly differ per format. A VM design can support many schemas that each describe different formats and assertions in different orders and lengths.
- **Runtime programmability**: A VM is inherently programmable. Schemas can be changed during the runtime of the system, without having to change the underlying hardware (fulfils REQ15).
- **Implementing and extending system functionality**: The instructions are used to implement the required data assessment functionality. They are implemented independently from each other and can therefore be changed in their functionality without affecting other instructions. If new functionality is required, new instructions can be implemented to support new functionality.

No hardware implementations with a similar function were found in the researched literature, but the design of the VM-based system is deemed viable for implementation, as it has similar properties to a conventional hardware processor. Therefore, we can design for a hardware-based implementation on an FPGA with this architecture (REQ13) while providing the flexibility found in software-based systems.

The binary blob input can be of any size, structure, and layout, and multiple blobs can follow in a sequence. This data will be in a stream. The VM-based architecture will therefore perform its operation on this data stream (REQ14) and require a verdict per binary blob to be synchronised with the stream (REQ1). At the start of each blob, this verdict is positive (true or 1) and stays positive until an assertion fails. This updates the verdict to a negative verdict (false or 0) as soon as the system knows the data is invalid (REQ2).

The other categories of architectures from the related work take a more parallelised approach to process the data. In general, this forms the basis for a higher throughput and lower latency. However, these architectures fall short of providing the desired flexibility. They have a maximum size per binary blob and are limited in the number of assertions that can be performed when the blob propagates through the system. Flexibility regarding how quickly a binary format description (i.e. schema) can be changed is also limited compared to a VM architecture. Further desired functionality, such as data interpretation of various data types and assertion operations on the data, is limited in these systems or is built around the domain of packet parsing, therefore being less relevant for processing other formats.

3.1.3. Consideration of Security

We elaborate on REQ16, as it does not state security as a main design requirement. The main goal of the project is to prototype a design and implementation of the system and have it be functional. However, various considerations will be taken into the design to facilitate future secure operation, as we identified that the most likely application domain will be for secure networking purposes. This is why REQ16 is formulated as having ”security in mind” rather than to be fully secure.

The first consideration is formulated as REQ2; once the data is determined to be invalid, the verdict on the binary blob is updated as soon as possible. This is to let the recipient of the blob know as soon as possible that the data is not safe for further processing. Secondly, REQ17 notes that input and output interfaces to the system are kept to a minimum, as to minimise the attack surface for potential malicious activity. An additional benefit is that minimal interfaces make it easier to use the system as part of a larger system. Finally, REQ18 notes that the system should keep the output stream identical to the input stream, and implies that the system is not allowed to mutate the data.

Relating the security requirements to the Instruction Set Architecture, we strive for a minimum set of instructions to fulfil the data assessment requirements, where each instruction has a clear and specific purpose. The ISA should not contain operations to be able to mutate the data going to the output, through arithmetic operations, for example. Nor should the ISA be able to reset the verdict after invalidation, prior to the arrival of the next blob.

3.2. Schema Language Design

This section introduces the design and notation of the schema language. This language is used to describe binary formats, such that the system knows how to check the blobs that it receives. A schema dictates in what order the data fields are arranged and traversed by the system. Furthermore, it declares the assertions that need to be performed on the data to result in a positive verdict.

Note that this language was not used to program the system, as building a corresponding compiler fell outside the project timeframe. Nevertheless, defining the notation and functionality played an important role in the design process, as it provided a top-down approach for designing the system. First, Section 3.2.1 will explain the schema notation. Thereafter, Section 3.2.2 describes a matrix which links every language feature to the data assessment requirements from the previous section.

3.2.1. Schema Notation

The schema language is a DSL with an imperative programming style that follows a top-to-bottom execution of statements. We explain the notation starting with the basic constructs, such as declaring fields and assertions, to gradually more complex functionality, such as callable schemas and loops.

Declaring fields

Section 2.4.5 discussed related research containing custom DSLs for describing binary formats. Four commonly used concepts were identified that were used to describe packet formats. The first three characteristics - ordered field declaration, labelled fields, and width indicators per field - form the basis of the schema language. They allow the declaration of data fields in a blob in order of appearance. In operation, the system will traverse the data in the order of these declared fields in the schema program.

A basic field declaration in the schema language consists of a **width**, **data type**, and **name tag**. In the case of many binary formats, including the IPv4 header and ASN.1 DER encoded data, their blobs are byte-aligned or word-aligned. The schema language covers this by declaring the width of a field with the `byte<n>` notation, where `n` is the number of bytes belonging to that field. There is no explicit notation to adhere to a word alignment in the schema. In the case that a format has a word alignment and includes padding bytes, an extra field can be declared to serve as padding.

Although the blob as a whole, encoded in a binary format, is byte-aligned. Fields can internally consist of bit-aligned fields. Such fields can be specified using the `bitfield` keyword. This is used as a safeguard to prevent misalignment of the fields, as the programmer (or a future compiler) can check whether the bitfields add up to the number of bytes to maintain byte alignment. Finally, bitfields can also be used for simulating don't care bits within a field. The don't care bits are declared in the bitfield, but no assertions are put on those bits.

Declared fields in a schema traverse the blob from top to bottom in a **single pass**. This is in line with the stream-based nature of the system, where the arriving data is assessed for its adherence to this field order. The code block below showcases how both bytefields and bitfields are declared.

```
schema MySchema {
    byte<3> unsigned MyField1; // A basic field declaration.
    byte<2> bitfield {
        /* The bitfields must add up to byte<2>. */
        bit<5>  unsigned MyField2;
        bit<11> unsigned MyField3;
    }
}
```

Data types specify information about how the field should be interpreted. The schema datatypes are `unsigned` for unsigned integers (also used for raw binary), `signed` for signed integers and `fixed<x,y>` and `ufixed<x,y>` for signed/unsigned fixed-point numbers, where `x` indicates the number of bits in the integer part, and `y` indicates the number of bits in the fractional part. Signed numbers are encoded in 2's complement representation. Future inclusion of data types such as floats would also be declared here as a separate `float` data type. The code block below shows fields with various data types.

```
schema MySchema {
    byte<4> unsigned      MyFieldUnsigned;
    byte<2> signed        MyFieldSigned;
    byte<4> fixed<12,20>  MyFieldFixed;
    byte<3> ufixed<2, 22> MyFieldUFixed;
}
```

Assertions on fields

When the system starts checking the blob, the verdict is positive. The fields are checked for structure and content through assertions. If a field is determined to be invalid, the verdict becomes negative and is only reset when the next blob is ready to be processed.

For performing assertions on the data, we use a list-like notation per declared field where assertions can be listed. An assertion consists of an assertion operator and a constant to which the data is being compared. The field is always placed on the left-hand side of the assertion operation, and the constant on the right-hand side. The possible assertions are:

- **Equality (`== x`)**: All bits for the width of the field are equal with constant `x`.
- **Inequality (`!= x`)**: At least one bit for the width of the field is unequal to constant `x`.
- **Less than (`< x`)**: The value of the field is less than the value of the constant `x`.
- **Less than or equal to (`<= x`)**: The value of the field is less than or equal to the value of the constant `x`.
- **Greater than (`> x`)**: The value of the field is greater than the value of the constant `x`.
- **Greater than or equal to (`>= x`)**: The value of the field is greater than or equal to the value of constant `x`.

- **Has range (`range x..y`)**: The field has a value between constant `x` and constant `y`. This is syntactic sugar for performing a >= `x` AND-ed by a <= `y` assertion.
- **Does not have range (`!range x..y`)**: The field does not have a value between constant `x` and constant `y`. This is syntactic sugar for performing a < `x` OR-ed by a > `y` assertion.

Additional notes about assertions are:

- Constants used in assertions can be declared in hexadecimal, decimal, or raw binary format. This requires a conversion step (in the future compiler) for hexadecimal and decimal values to be converted into a binary constant before they can be used in the system.
- Assertion comparisons support all four data types.
- If an assertion fails, the associated blob is invalidated (i.e., it receives a negative verdict), and processing proceeds to the next blob.

Multiple assertions are possible on a field through Boolean operators. The "`&`" operator performs an AND operation between two assertions. The "`|`" operator will perform an OR operation. These operations evaluate assertions accumulatively from left to right, making them best used when all assertions need to be AND-ed or OR-ed. A more complex order of operations can be handled using the conditional statements explained further below. The code block below shows how assertions are declared.

```
schema MySchema {
    byte<4> unsigned MyField1 [== 0xFB5C923C]; // Assertion with hexadecimal constant.
    byte<1> unsigned MyField2 [>= 0b01010]; // Constant will be padded with leading 0's.
    byte<2> signed   MyField3 [range -25..5000]; // Signed range comparison
    byte<4> unsigned MyField4 [== 0xFB5C923C | == 123456789]; // Multiple assertions
    byte<1> unsigned MyField5 [!= 0b00000000 & <= 0b010101110];
}
```

Variable width fields

In the case of ASN.1 DER and other TLV-based formats, its TLV structure has a *length* field which refers to the length of the *value* field, resulting in value fields which can be of variable width. To support the notation of a field with variable width, the `byte<n>` notation can have `n` refer to the value of the previous field (which in the case of TLVs is the length field). The code block below shows how an ASN.1 DER integer with variable width can be declared.

```
schema MySchema {
    byte<1>            unsigned MyIntTag [== 0x02];
    /* ASN.1 DER Length fields denote the size in bytes. */
    byte<1>            unsigned MyIntLength [<= 32]; // Assert integer width <= 32.
    byte<MyIntLength> signed   MyIntValue;
}
```

Conditional statements

Section 2.4.5 discussed that related DSLs share four characteristics. The fourth characteristic, conditional header parsing notation, is less relevant for describing binary formats, as determining the next header is related to packet parsing. The schema language instead includes conditional statements for conditional schema processing. This is for a more general purpose of describing binary formats beyond packet headers.

Conditional statements are used in the schema language when different structures or values in the data will follow, based on the value of the current field. For instance, the IPv4 header has the Protocol field that indicates which next header will follow. In the schema language, all assertions in a conditional statement are performed on the last declared field before the conditional statement. Per `if` or `else if` case, different assertions can be declared to determine which branch is taken. Multiple conditional assertions can be listed per case, and they are evaluated in the same method as field assertions (cumulatively, from left to right). This also allows for more complex field assertions, as an order of operations for assertions can be programmed. The code block below showcases how a conditional statement is used for an IPv4 schema. Based on the Protocol field, a TCP or UDP header is expected to be the next protocol.

```
schema IPv4_header {
    ...IPv4 fields...
    byte<1> unsigned Protocol; // This is the field used in the conditional statement.
    // Check which protocol follows.
    if (Protocol == 0x06) {
        ... // Parse IPv4 remainder + TCP
    } else if (Protocol == 0x11) {
        ... // Parse IPv4 remainder + UDP
    } else {
        fail; // Fail the schema
    }
}
```

A notable constraint about the schema language is shown in the code block above, which is that the conditional statement needs to be declared right after the declaration of the field used in the statement. This results in the issue that it is not possible to first finish processing the IPv4 header, and afterwards check the Protocol field for the next header.

This would be solved by saving the value of the Protocol field, but we found that saving such state increased the complexity of the system due to introducing state that requires random access writes and reads to memories. Writing conditional statements for the current field is a way to reduce this complexity. This is therefore something the programmer should keep in mind when writing a schema.

Callable Schemas

Schemas are callable to allow for better schema code organisation and code reuse. A schema can be called by writing the schema name at the desired location within the caller schema. Multiple schemas do mean that a starting schema needs to be determined. This starting schema is the topmost schema in the file. This follows the natural reading order, making it straightforward for a programmer to know where the schema execution starts. The code block below shows a revised version of the previous schema example with callable schemas.

```
schema IPv4_header {
    ...IPv4 fields...
    byte<1> unsigned Protocol;
    // Check which protocol follows. NOTE: We cannot save the value of the Protocol field for
         later. We perform the assertion now and process the remainder of the IPv4 fields.
    if (Protocol == 0x06) {
        remainder_IPv4; // Process remaining IPv4 fields
        TCP;            // Process TCP
    } else if (Protocol == 0x11) {
        remainder_IPv4; // Process remaining IPv4 fields
        UDP;            // Process UDP
    } else {...}
}

schema remainder_IPv4 {...remaining IPv4 fields...}

schema TCP {...TCP fields...}

schema UDP {...UDP fields...}
```

Length Assertions

Self-describing hierarchical formats, like TLV, can include length fields which refer to the length of a section that follows. This length field needs to be cross-checked with the actual length of the section. The schema language supports this with a notation for making length assertions. The `length` keyword is used to declare that the current field is a length field. This is followed up by a `bytes`, `bits` or `fields` keyword. This indicates that the length field either denotes the number of bytes, bits, or fields. The indented section between the { } is where the section which will be checked for its length can be declared. Finally, length assertions can be nested as well. The code block below illustrates how length assertions and nested length assertions are declared. (Note: the example is not ASN.1 DER but a TLV-like format)

```
schema LengthAssertionExample {
    byte<1> unsigned sequence_tag [== 0x01];
    byte<1> length bytes { // Sequence denotes its length as bytes.
```

```
        /* Indented section for length assertion. */
        byte<1> unsigned nested_array_tag [== 0x02];
        byte<1> length fields { // Array denotes its length as fields.
            ...
        }
        process_remainder; // Process remaining fields in the sequence
    }
}
```

Repeat loops

In the case that a schema has repetition in its structure, repeat loops can repeat the execution of the desired routine to improve schema code organisation. A repeat loop can be declared using the `repeat` keyword followed by the number of repeats. The code block below shows how repeat loops are declared.

```
schema MySchema {
    repeat 42 { // Repeat section 42 times
        byte<1> unsigned MyTag;
        byte<1> unsigned MyValue [range 1..3];
    }
}
```

A special type of repeat loop is a **repeat "while" loop**. This repeats a routine until the accumulated section length (either in bytes, bits or fields) has been exactly reached. To illustrate: a hierarchical structure can have a length field stating the length of that section. The section has several fields, but it is not known beforehand how many fields in this routine fit inside the length of the hierarchical structure, and of what length they are. This type of repeat loop can be interpreted as: "While the current counted length has not exactly reached the value of the declared length field, rerun the routine." If an iteration happens to overshoot the length, the length field is invalid and the blob is given a negative verdict. The code block below illustrates the function of the repeat while loop for processing ASN.1 DER.

```
schema MySchema {
    byte<1> unsigned sequence_tag [== 0x30];
    byte<1> unsigned sequence_len;
    /* while loop: repeats until a 'sequence_len' amount of bytes has passed. */
    repeat sequence_len bytes {
        process_integer; // A variable amount of variable-length integers.
    }
}

schema process_integer { // Process a variable-length integer
    byte<1> unsigned int_tag [== 0x02];
    byte<1> unsigned int_len [range 1..2];
    byte<int_len> signed int_val [range 1..350];
}
```

Ending a schema

The schema execution can end in 5 ways, as shown by the code block below:

```
schema MySchema {
    byte<1> unsigned MyCommand;
    if (MyCommand == 0x01) {
        byte<1> unsigned MyField [== 39 | == 100]; // 1. An assertion might fail.
    } else if (MyCommand == 0x02) {
        byte<1> length bytes {
            process_something;
        } // 2. A length assertion might fail
    } else if (MyCommand == 0x03) {
        byte<1> unsigned MyWhileLength;
        /* Repeat while loop. */
        repeat MyLength bytes { // 3. A repeat while loop might fail.
            process_something;
        }
    } else {
        fail; // 4. An explicit fail call is made.
    }
} // 5. The schema ends normally.
```

1. **Field assertion fail**: the schema execution can end with a negative verdict through a failed assertion of a field. For multiple assertions on a field, it depends on the cumulative assertion result to determine whether the field assertion failed or not.
2. **Length assertion fail**: the schema execution ends if a length field does not match the section to which the length is referring.
3. **Repeat while loop fail**: the schema execution can be stopped with a negative verdict through the accumulated length overshooting the supposed length of that section. This means the length field does not match the actual length of that section.
4. **Explicit fail call**: An explicit `fail` statement stops the execution of the schema and sets a negative verdict. For example, in an else case of a conditional statement, when one of the conditional blocks must be taken and the schema execution must be stopped otherwise.
5. **Normal end**: the schema execution ends normally if the last statement of a schema is executed and the schema was not a called schema (otherwise it returns execution to the caller schema).

When a schema has ended execution, any remaining data in the input stream is ”flushed” through the system. The execution restarts from the top of the schema for the next blob. Flushing is explained in Section 3.5.1.

3.2.2. Requirements and Schema Notation Matrix

Table 3.1 shows which requirement is related to what functionality in the schema language. Only the data assessment requirements are considered, as these are what need to be implemented by the schema language. We also include REQ14, as a stream-based system is related to how the data is declared and traversed in the schema language.

	Single-pass traversal	Declaring fields	Data types	Assertion operations	Multiple assertions	Variable width fields	Conditional statements	Callable schemas	Length Assertions	Repeat loops	While loops	Explict fails	Schema restart
REQ1: Verdict per blob				X	X				X			X	X
REQ2: Verdict update				X	X				X			X	X
REQ3: Schema adherence		X	X	X	X	X	X	X	X	X	X	X	
REQ4: Data traversal	X	X											
REQ5: Data types			X										
REQ6: Assertion operations				X									
REQ7: Multiple assertions					X								
REQ8: Field is next size						X							
REQ9: Control flow changes	X						X	X					
REQ10: Check section length								X	X				
REQ11: Repetition										X			
REQ12: Repetition until length											X		
REQ14: Stream-based	X												

Table 3.1: The requirements and schema notation matrix.

3.3. Instruction Set Architecture

This section explains the Instruction Set Architecture (ISA). The ISA is a 32-bit architecture consisting of 9 instructions that execute the functionality of the schema language. Table 3.2 briefly describes every instruction. The next subsection explains how the ISA was derived. This is followed by an explanation of the instruction pseudo notation. Sections 3.3.1 to 3.3.9 discuss each instruction in detail, explaining their functionality, describing each instruction field, and providing the corresponding pseudo notation. Finally, Section 3.3.10 explains the design decisions behind various instruction field widths.

Instruction	**Description**
shift	Sets a new data field by shifting bits from the input onto the data field bus, moves the previous field to the output, and updates bit/byte/field counters.
end	End the operation on the current binary blob. Either output the current accumulated comparison ”cmp” verdict or explicitly fail the blob.
load immediate	Loads a constant into the constant register for comparison. Consecutive load immediate instructions create bigger constants.
compare	Compare two operands together and add the comparison result to a cumulative verdict. This is accumulated for a data field assertion or branching.
branch	Branch to another instruction based on the accumulated branching ”brch” verdict.
jump	Jump unconditionally to another instruction.
call	Push a stack entry to the stack and jump to a subroutine.
return	Pop the topmost stack entry from the stack and return to the caller address.
increment	Increment the topmost stack entry on the stack.

Table 3.2: Instruction overview

Deriving the ISA

As mentioned in Section 3.1.3, we strive for a minimal ISA that can fulfil the data assessment requirements. The previous section translated these requirements into concrete functionalities of the schema language. Therefore, we strive for a minimal set of instructions to fulfil the schema functionality. Reading Sections 3.3.1 to 3.3.9 first provides an understanding of each instruction's functionality and pseudo notation. Subsequently, Section 3.4 demonstrates how the 9 instructions collectively use their function to implement the functionality of the schema language.

We chose a 32-bit instruction set as it contains the right balance of instruction compactness without compromising on the required fields per instruction. To put this decision into perspective, earlier iterations of the ISA were 64 bits wide. This provided a larger space for instruction fields and contained larger address fields and immediate values. However, this field space went unused for many instructions. Furthermore, every instruction had compound functionality: every instruction had the same fields for shifting input data. This approach was later deemed impractical, as certain instructions depended on the data shifting to happen first, before executing their own functionality. This made it harder to implement these instructions in a single clock cycle.

Furthermore, previous iterations of the ISA included instructions for saving and loading constants in comparison operations. This implied the usage of another memory in the system that can contain these constants. This made the system harder to program, as either the instruction memory had to contain the instructions to fill this memory with constants, or an external port had to fill this memory with constants. The latter also enlarged the attack surface of the system, as a new external interface would be required. Removing these instructions and reverting to a single instruction memory to program the system was therefore safer and less complex.

Our final 32-bit instruction set contains 9 instructions, where every instruction has a single function. The instructions from earlier iterations were split to exclude the shifting fields and instead include a standalone shift instruction. This made it more achievable to implement this functionality in a single clock cycle. The functionality of each instruction could be categorised by either contributing to **advancing data flow** (shift), **advancing control flow** (end, branch, jump, call, return, increment), or **performing field or length assertions** (load immediate, compare, call, return).

Instruction Pseudo Notation

We program the system by writing the instructions manually, due to the absence of a compiler. The instructions are written in a pseudo notation, which is a textual representation of instructions that makes writing programs and reasoning about programs more accessible and understandable. We introduce additional notation to explain the pseudo notation and various accessible fields and counters accessed in the instruction in the code block below.

```
// HELPING NOTATION
(...) -> optional, can be left out.
<...> -> varying value (e.g. an address).
/  -> selection, one of the options must be selected.

// ACCESSIBLE FIELDS
inp       // The current value of the data field bus.
stacktop  // The entry_value of the stack top-level entry.
bit_cnt   // The number of bits counted so far in the current blob.
byte_cnt  // The number of bytes counted so far in the current blob.
field_cnt // The number of fields counted so far in the current blob.
```

3.3.1. Shift (`sft`)

The `sft` instruction moves data through the system. It moves a segment of bits from the current section of the input stream onto the data field bus based on the shift amount (More on the system data flow in Section 3.5.1). The bit and byte counters of the system are incremented by the amount that is shifted. The field count is incremented by one. At the same time, the previous value on the data field bus and its specified width move to the output, and the verdict at the output is updated. If the outgoing field is deemed invalid, the data is flushed, and the program restarts with the next blob. Furthermore, there is no restriction that every field needs to be shifted individually onto the data field bus. Therefore, one or more fields can be ”shifted through” in a single shift amount, which is useful if those fields can be skipped. Finally, the shift instruction also functions as a no-op instruction when all fields are set to 0.

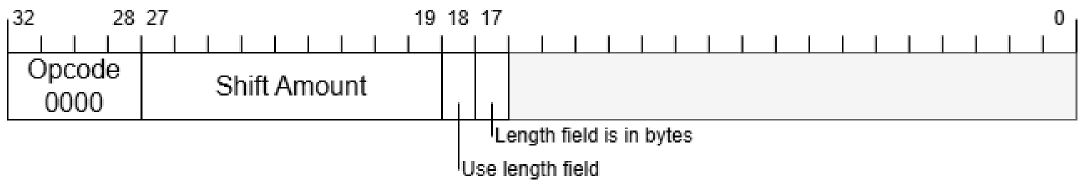

Figure 3.1: Shift instruction with bit positions.

Field overview in order from left to right:

- **Opcode** (4 bits): Opcode of the instruction.
- **Shift amount** (9 bits): Determines the number of bits to ”shift in”. The value can range from 0 to 256. This value is used to update the current bit and byte count
- **Use length field** (1 bit): Flag indicating that the current data field value will be used as the next shift amount (for variable width fields).
- **Length field in bytes** (1 bit): Flag indicating that the current data field value indicates a byte amount, rather than a bit amount.
- **Reserved** (17 bits)

The pseudo notation for the shift instruction is provided in the code block below.

```
sft
    0..256/bitinp/byteinp

// Examples
sft 42      // Shift in a new field of 42 bits
sft bitinp  // Shift in a new field with a width of the current data field in bits.
sft byteinp // Shift in a new field with a width of the current data field in bytes.
```

3.3.2. End (`end`)

The `end` instruction is used to end the operation on the current binary blob. This is used, for instance, in conditional statements where the else case implies an invalidation of the data. Besides being able to set an explicit 'fail' to set a negative verdict, the message can also end normally. This is set at the end of the normal schema flow and outputs the assertion verdict of the last field. In both cases, the remaining data of the packet is flushed to the output, and the system jumps to the first instruction and starts processing the next blob.

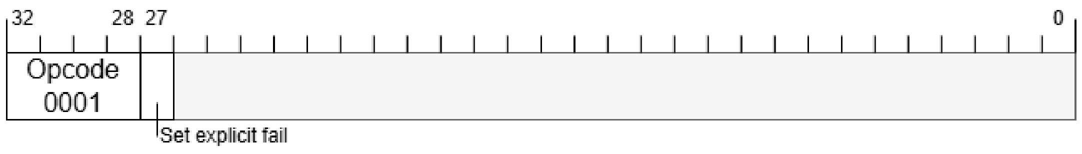

Figure 3.2: End instruction with bit positions.

Field overview in order from left to right:

- **Opcode** (4 bits): Opcode of the instruction.
- **End or Fail flag** (1 bit): (0) sets normal end, (1) sets explicit fail.
- **Reserved** (27 bits)

The pseudo notation for the end instruction is provided in the code block below:

```
end
    (fail)

// Examples
end       // Normal end
end fail // Explicit fail end
```

3.3.3. Load Immediate (`ldim`)

The `ldim` instruction loads a constant value into the constant register for comparison. Consecutive 'ldim' instructions will shift the existing contents in the register to the right, allowing the new immediate value to be inserted. From this, bigger constants can be made up to the maximum data field width.

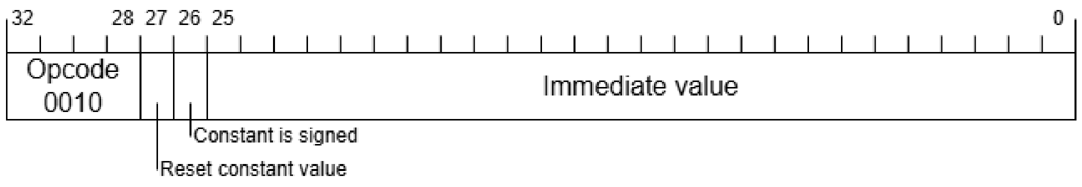

Figure 3.3: Load Immediate instruction with bit positions.

Field overview in order from left to right:

- **Opcode** (4 bits): Opcode of the instruction.
- **Reset constant value** (1 bit): This flag resets the constant register, therefore indicating that a new constant is being loaded in.
- **Constant is signed** (1 bit): This flag indicates that the immediate value is signed, therefore, the constant register should sign extend based on the MSB of the immediate value. In case of multiple consecutive `ldim` instructions, sign extension is only relevant for the first immediate value. Therefore, the "reset constant value" flag will also need to be 1 in case this flag is raised.
- **Immediate value** (26 bits): The immediate value to be inserted in the const register.

The pseudo notation is included in the compare instruction (next section).

3.3.4. Compare (`cmp`)

The `cmp` instruction compares two operands together and checks whether the comparison output bits (either smaller <, equals =, bigger >) matches with one of the bits in the expected result field. **We call the result of this match operation the assertion result**. The assertion result is cumulatively AND-ed or OR-ed in either a `cmp_verdict` bit for field assertions or a `brch_verdict` bit for branching. Both these bits are initialised to 1 at the start of the execution at the current binary blob. Meaning that successful assertions keep the verdict to 1, and failing assertions will set the verdict to 0. The verdict of the system will be set to the `cmp_verdict` bit at the next shift instruction.

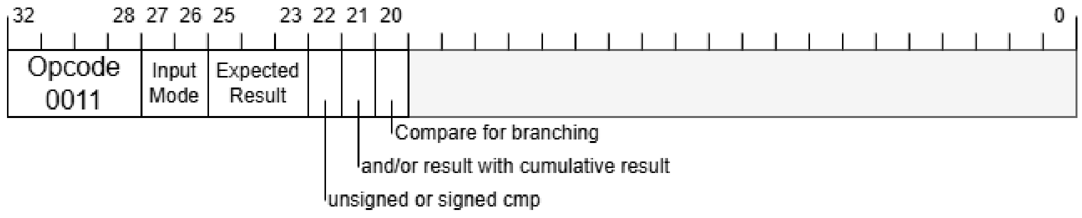

Figure 3.4: Compare instruction with bit positions.

Field overview in order from left to right:

- **Opcode** (4 bits): Opcode of the instruction.
- **Input mode** (2 bits): Determines which two operands are used for comparison. They are:
  - **00**: the **input** field is compared against a **constant**
  - **01**: the **stack top entry** is compared against a **constant**
  - **11**: the **stack top entry** is compared against a **counter**. The selected counter (bit/byte/field) is dependent on the `length_mode` in the top stack entry. See `call` instruction: length mode.
- **Expected result** (3 bits): Used to encode which output is expected from the comparator. The comparator has 3 outputs: <, =, >. Therefore, each of the 3 bits refers to the comparator output being expected. The assertion result (1 pass, 0 fail) is based on whether the comparator matched with one of the bits of the expected result. See table 3.3 for all possible combinations.
- **Unsigned or signed compare** (1 bit): Flag to set to use unsigned (0) or signed (1) comparison.
- **And/or the result** (1 bit): Flag to set whether the current assertion result will be ANDed (0) or ORed (1) with either the `cmp` or `brch` verdict.
- **Compare for assertion or branching** (1 bit): Determines whether the assertion result contributes to the `cmp` or `brch` verdict bit.
- **Reserved** (20 bits)

Bit config	Comparator Expected outputs	Notes
000		Not Used
001	>	
010	=	
011	= or >	Essentially >=
100	<	
101	< or >	Essentially !=
110	< or =	Essentially <=
111	< or = or >	Not Used

Table 3.3: The possible ”expected result” field combinations.

The pseudo notation for the compare instruction includes shifting and constant loading in the same statement. This allows for a direct conversion from a field with a single assertion in the schema language to this pseudo notation. Converting this pseudo notation to actual instructions would be split into `sft`, `ldim`, and `cmp` instructions. (More on schema-to-instruction conversions in Section 3.4.) The pseudo notation for the compare instruction is provided in the code block below.

```
sft
    0..256/bitinp/byteinp
cmp
    (brch) // omitting means field assertion instead of branching assertion
    (or)   // omitting means AND-ing the result
    unsigned/signed
    inp/stacktop // lhs operand of the compare
    >/==/>=/</!=/<= // comparison operation
    const=<const_value>/bit_cnt/byte_cnt/field_cnt // rhs operand of the compare

//Examples
sft 32 cmp unsigned inp > const=42          // Field assertion
sft 8 cmp brch or signed inp <= const=0x1A // Branching assertion
sft 0 cmp unsigned stacktop == byte_cnt     // Length assertion
```

3.3.5. Branch (`brch`)

This instruction branches to another address in the instruction memory when the ‘brch‘ verdict is true (1). Otherwise, the flow continues with the next instruction. The brch verdict bit is afterwards reset to 1 to be ready for the next branching assertion.

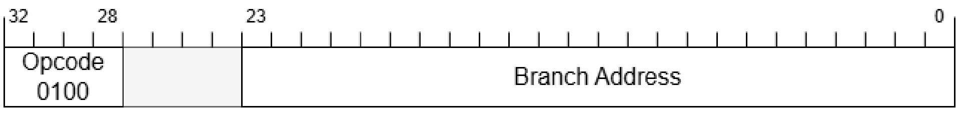

Figure 3.5: Branch instruction with bit positions.

Field overview in order from left to right:

- **Opcode** (4 bits): Opcode of the instruction.
- **Reserved** (4 bits)
- **Branch address** (24 bits): Address to branch to.

The pseudo notation for the branch instruction is provided in the code block below:

```
brch
    <subroutine_name>

// Example
brch MyBrchRoutine
```

3.3.6. Jump (`jmp`)

This instruction unconditionally jumps to a given address.

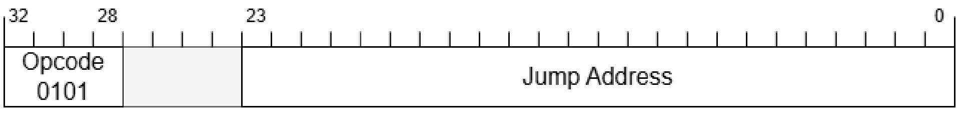

Figure 3.6: Jump to instruction with bit positions.

Field overview in order from left to right:

- **Opcode** (4 bits): Opcode of the instruction.
- **Reserved** (4 bits)
- **Jump address** (24 bits): Address to jump to.

The pseudo notation for the jump instruction is provided in the code block below:

```
jmp
    <subroutine_name>

// Example
jmp MyJmpRoutine
```

3.3.7. Call (`call`)

This instruction pushes a stack entry (see Table 3.4) onto the stack and jumps to a subroutine.

Field:	`length_mode` (2 bits)	`call_mode` (2 bits)	`entry_value` (256 bits)	`return_address` (24 bits)
Description:	The unit of the `entry_value`. Either bytes, fields, bits or custom.	Determines what the `entry_value` represents, as well as what checks are performed during the return instruction.	The value of the stack entry in amount of `length_mode`.	The return address of the entry.

Table 3.4: An overview of a stack entry. Stack entries are 284 bits wide, requiring a stack of the same width.

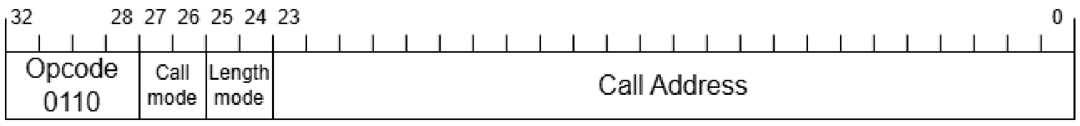

Figure 3.7: Call instruction with bit positions.

Field overview in order from left to right:

- **Opcode** (4 bits): Opcode of the instruction.
- **Call mode** (2 bits): Determines what data is put into the stack entry and how it is used. The call mode configuration is also written to the stack entry such that the return operation knows what to evaluate when popping that stack entry. The different call modes are:
  - **00: Function call mode** $\rightarrow$ Pushes only the return address to the stack. Other fields are unused.
  - **01: Counter mode** $\rightarrow$ Used for fixed repeat loops. Pushes a value of 0 to the entry and the return address.
  - **10: Exact length check mode** $\rightarrow$ Used for checking the exact length of a section. The pushed `entry_value` is the sum of the current field on the data field bus + the bit/byte/field counter value (depending on the ”Length mode”). If the current field is a length field, the sum represents the length of the following section + the blob data that has passed through the system so far. **Hence, this value represents the expected amount of bits/bytes/fields counted *after* the section has passed through the system.** (Section 3.4.3 provides an example of this check in practice)
  - **11: While length check mode** $\rightarrow$ Used for checking the length of a section using a repeat while loop. Pushes the value on the data field bus + bit/byte/field counter value, similarly to the exact length check mode.
- **Length mode** (2 bits): Specifies whether the `entry_value` (see Table 3.4) refers to the number of bits, bytes, fields, or is a manually incremented field (for counter mode).
  - **00: Custom Counter** $\rightarrow$ The length is initialised to zero and is manually incremented using the `inc` instruction (only compatible with Call mode 01: Counter mode)
  - **01: Byte Counter** $\rightarrow$ The length value indicates the number of bytes (only compatible with Call modes 10: Exact length check mode and 11: while length check mode)
  - **10: Field Counter** $\rightarrow$ The length value indicates the number of fields, i.e. the number of shifts (every time a shift is done, a new field is retrieved) (only compatible with Call modes 10 and 11)
  - **11: Bit Counter** $\rightarrow$ The length value indicates the number of bits and can be the same counter as the byte counter. (only compatible with Call modes 10 and 11)
- **Call address** (24 bits): The address the Program Counter is set to.

The pseudo notation for the call instruction is provided in the code block below:

```
// Call variation one
call
    (count) // Enables counter mode
    <call_subroutine>

// Call variation two
call
    len/whilelen // Select exact length check or while length check mode
    bits/bytes/fields // Select length mode
    <call_subroutine>

// Examples
call MySubroutine
call len bytes ProcessDERSequence
call whilelen fields ProcessArray
```

3.3.8. Return (`ret`)

This instruction pops a value from the stack and updates the PC with the return address in the popped stack entry. Depending on the call mode, various things are checked. The following list explains what happens at the return instruction depending on the call mode:

- **00: Function call mode** → Only updates the PC with the return address.
- **01: Counter mode** → Only updates the PC with the return address. Any assertion of the counter value is done beforehand using a `cmp` instruction, where the top `entry_value` on the stack containing the current number of repeats is compared to a constant containing the final repeat count value.
- **10: Exact length check mode** → The PC is updated with the return address stored in the popped stack entry. A comparison is made with the expected length value of the section, stored as the `entry_value` of the popped stack entry, with the appropriate counter value (depending on the ”length mode”), representing how much data has been counted (in either bits, bytes or fields). These should be equal if the length of the section is to be correct. This length assertion result is AND-ed to an internal `len_verdict` bit, initialised to 1. A single failed length assertion keeps the bit at 0, regardless of successful length assertions that follow. This updates the verdict at the next shift instruction, as the verdict moves with the data.
- **11: While length check mode** → The PC is updated with the return address stored in the popped stack entry. Any assertion of the length is done beforehand using a `cmp` instruction, where the top `entry_value` on the stack, containing the expected length value of the section, is compared against the appropriate counter value (depending on the ”length mode”). Their equality is the break condition of the while loop. An overshoot of the counter should invalidate the blob.

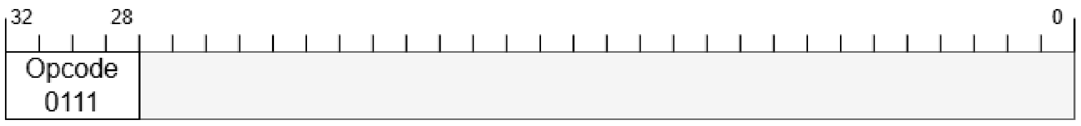

Figure 3.8: Return instruction with bit positions.

Field overview of the return instruction:

- **Opcode** (4 bits): Opcode of the instruction.

The pseudo notation for the return instruction is `ret`.

3.3.9. Increment (`inc`)

This instruction increments the top element of the stack by 1. Used for counting repeats in a fixed repeat loop.

Field overview of the increment instruction:

- **Opcode** (4 bits): Opcode of the instruction.

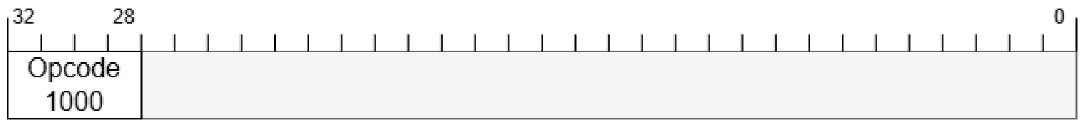

Figure 3.9: Increment instruction with bit positions.

The pseudo notation for the increment instruction is `inc`.

3.3.10. Instruction Field Width Elaboration

After the detailed instruction overview, we elaborate on why various field widths in the ISA were chosen. We assume that for smaller fields, such as configuration flags or encoded instruction modes, their widths are self-explanatory. For the following fields, the chosen width is explained:

- **Shift amount** (9 bits): The 9-bit shift amount in the `sft` instruction encodes 257 different values as an unsigned value ranging from 0 to 256, determining how many input bits to "shift" onto the data field bus. This reflects an implementation detail, as the prototype implementation supports a maximum data field size of 256 bits per shift. An implementation with a different width will also determine the shift amount field width. Furthermore, it is important that a shift of 0 can be represented by all 0s in the shift amount field. This is for two reasons. First, this can prevent conflicts when input field shifting bits are set. Second, this allows an instruction of 32 0s to be used as a no-op instruction, which can be used for no-op insertion in programs as a security technique to randomise the control flow of the system.
- **Immediate value** (26 bits): The 26 bits of the immediate value of the `ldim` instruction represent a section of the constant register which gets loaded in. Consecutive `ldim` instructions shift and load their immediate values to create the final constant. To save on instructions, this needs to be done in as few `ldim` instructions as possible. Hence, the 26 bits fill all remaining bits after the opcode and the two flags. The `ldim` instruction would benefit the most if the ISA had a larger width, as it could use this extra width to reduce the number of `ldim` instructions required to load in larger constants.
- **Branch/Jump/Call address** (24 bits): These addresses of the `brch`, `jmp`, and `call` instructions are used to update the program counter. These determine that the possible address space of the system is 24 bits wide (2^{24} 32-bit instructions = 64 MB), which we determined to be sufficient space for many realistic schemas. The 24-bit width is determined by the `call` instruction, as it is the remaining space after the opcode and mode configuration fields. Both the `brch` and `jmp` instructions were also set to 24 bits to maintain the same address space.
- **Opcode** (4 bits): The opcode is 4 bits wide, as that is the required width to encode the 9 instructions of the ISA, leaving space for 7 more instructions. If the opcode space is widened, this would first impact the widths of the immediate value field or the address fields. Another option would be to increase the ISA width to create more opcode space. For our design and implementation, we stuck to a 32-bit width, which provided the right balance between instruction compactness and instruction functionality.

3.4. From Schema to Instructions

This section explains various schema-to-pseudo notation examples below to show how the schema language gets translated to instructions. This is relevant to see how the ISA instructions achieve the desired functionality of the schema language. **NOTE:** A future compiler would directly compile the schema language to binary instructions; a conversion to pseudo notation would be a redundant intermediate step. Pseudo notation examples are used in this explanation as they are human-readable and have a clear conversion to actual instructions.

3.4.1. Field Assertions

Field declarations and assertions are converted, as shown in the code block below. On line 5, the pseudo notation can shift the input, load a constant, and compare the two in one statement. Line 6 performs the second assertion on the same field. Hence, no shifting is done.

This single example below shows how this converts into actual single instructions (we still use a human-readable notation). Starting from line 9, the pseudo notation translates to a `sft` instruction shifting in 32 bits, an `ldim` instruction loading in the constant '0', and a `cmp` instruction comparing the (unsigned) input with the loaded constant (the result is AND-ed with the accumulative `cmp_verdict` bit). The second assertion is done by loading in a new constant value of 10, and another comparison whose result gets OR-ed with the `cmp_verdict`.

```
1  //////// Schema language ////////
2  byte<4> unsigned MyField [== 0 | > 10];
3
4  //////// Pseudo notation ////////
5  sft 32 cmp unsigned inp == const=0    // Field is shifted in at this statement.
6  sft 0  cmp or unsigned inp > const=10 // No shift, only constant load and comparison.
7
8  //////// Conversion from pseudo notation to instructions ////////
9  sft 32 // Line 5, shift 32 bits
10 ldim 0 // Load in constant 0
11 cmp unsigned inp == const // Compare shifted field with constant
12 ldim 10 // Line 6, load in constant 10
13 cmp or unsigned inp > const // Compare field with constant and OR with the accumulated result.
```

3.4.2. Conditional Statements and Schema Calling

The conversion of conditional statements and schema calling is shown in the code block below. The comments explain how it works.

```
//////// Schema language ////////
schema IPv4_example {
    byte<1> unsigned Protocol;
    // Conditional statement: check which protocol follows.
    if (Protocol == 0x06) {
        IPv4_TCP_header; // Parse IPv4 remainder + TCP
    } else if (Protocol == 0x11) {
        IPv4_UDP_header; // Parse IPv4 remainder + UDP
    } else {
        fail;
    }
}

//////// Pseudo notation ////////
IPv4_example:
    // Shift in the protocol field, load a constant and compare for branching.
    sft 8 cmp brch unsigned inp == const=0x06
    // Take the branch if the branching assertion is true.
    brch if_case_1
    // Execute the branching assertion in the else if block.
    sft 0 cmp brch unsigned inp == const=0x11
    brch if_case_2
    jmp else_case // No brch assertion for the else case, hence the jmp instruction is used.

// All conditional blocks jump to this subroutine when they are finished.
IPv4_example_post_if:
    end // No code after the if statement, hence the program ends normally.

if_case_1:
    call IPv4_TCP_header // Call function
    jmp IPv4_example_post_if

if_case_2:
    call IPv4_UDP_header // Call function
    jmp IPv4_example_post_if

else_case:
    end fail

IPv4_TCP_header:
    ... // Process fields, call more schemas, etc..
    ret // Return to if_case_1

```

```
IPv4_UDP_header:
    ... // Process fields, call more schemas, etc..
    ret // Return to if_case_2
```

3.4.3. Exact length assertions

The conversion of exact length assertions is shown in the code block below. It is explained through an example. The comments provide further elaboration.

```
//////// Schema language ////////
schema MyDERSequence {
    byte<1> unsigned sequence_tag [== 0x30];
    byte<1> length bytes { // Length check of bytes
        byte<3> unsigned DER_boolean_true [== 0x0101FF];
        ...
    }
}

//////// Pseudo notation ////////
MyDERSequence:
    sft 8 cmp unsigned inp == const=0x30 // Check the tag. (byte_count is 1)
    sft 8                                // Length field with value 20. (byte_count is 2)
    call len bytes Checklen0 // The entry_value of this entry is the byte_count + length
        field value, which is 2 + 20 = 22. This is the expected value of the byte_count after
         the Checklen0 routine has finished.
    end

Checklen0:
    sft 24 cmp unsigned inp == const=0x0101FF // Process 3 bytes (byte_count is 5)
    ... // Process 17 more bytes (byte_count becomes 22)
    ret // At this instruction it is evaluated whether the entry_value (22) == byte_count
        (22). This is equal, therefore, the length assertion passes.
```

3.4.4. Repeat and repeat while loops

The conversion of a repeat loop is shown in the code block below. The comments explain how it works.

```
//////// Schema language ////////
schema MySchema {
    repeat 10 {
        check_integer;
    }
}

//////// Pseudo notation ////////
MySchema:
    call count MySchema_repeat // Counter mode enabled in MySchema_repeat
    end

MySchema_repeat:
    sft 0 cmp brch unsigned stacktop == const=10 // Check if the required number of
        iterations has been achieved.
    brch MySchema_continued
    call check_integer // Execute one iteration.
    inc // Increment the stacktop by one.
    jmp MySchema_repeat // Start next iteration.

MySchema_continued:
    ret // Return to the main subroutine.
```

The conversion of a repeat while loop is shown in the code block below. The comments explain how it works.

```
//////// Schema language ////////
schema MySchema {
    byte<2> unsigned MyLength;
    repeat MyLength bytes {
        check_integer;
    }
}
```

```

//////// Pseudo notation ////////
MySchema:
    sft 16 // Shift in the MyLength field.
    call whilelen bytes whilelen0
    end

whilelen0:
    // Repeat while loop assertion. Branch to the fail routine if the byte counter becomes
        larger than the length.
    sft 0 cmp unsigned stacktop < byte_cnt
    brch whilelen0_fail
    // Break condition if length equals the number of processed bytes.
    sft 0 cmp unsigned stacktop == byte_cnt
    brch whilelen0_post
    call check_integer // check one integer, i.e. run another iteration of the while loop.
    jmp whilelen0 // jump back to the top of the while loop.

whilelen0_fail:
    end fail

whilelen0_post:
    ret // Return to the main subroutine.
```

3.4.5. Schema Notation and Instructions Matrix

Table 3.5 provides an overview of the instructions used for each functionality in the schema language.

	shift	end	load immediate	compare	branch	jump	call	return	increment
Single-pass traversal	X								
Declaring fields	X								
Data types				X					
Assertion operations			X	X					
Multiple assertions			X	X					
Variable width fields	X								
Conditional statements			X	X	X	X			
Callable schemas							X	X	
Length assertions							X	X	
Repeat loops			X	X	X	X	X	X	X
While loops		X		X	X	X	X	X	
Explicit fails		X							
Schema restart	X	X							

Table 3.5: The schema notation to instructions matrix.

3.5. Module Design

This section discusses the modules in the system. Figure 3.10 shows the layout of the verdict engine and connections between its modules.

To explain the system structure, we start with the Input and Output systems. These are responsible for the data flow of the system. The byte-aligned input stream has the binary blob data arrive in sections.

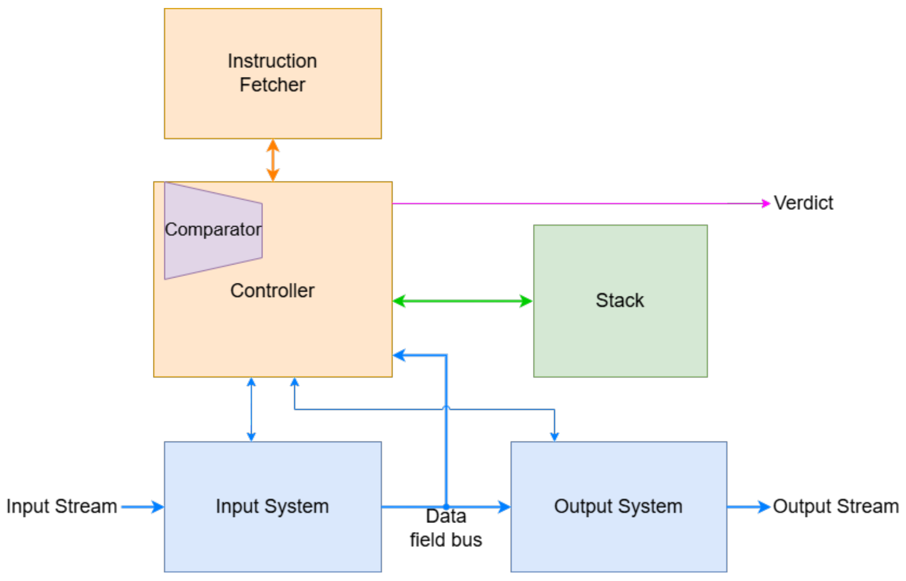

Figure 3.10: Module overview

The Input system consumes this data per section and puts, per `sft` instruction, an individual bit-aligned data field on the data field bus. The Output system will consume the data field per `sft` instruction and create the output stream, identical to the input stream. **Because of these two modules, we can assess the blob per data field and have the verdict move with the data field.**

The Controller has access to the data field bus to perform this assessment per data field. It contains a comparator to perform assertions through `cmp` instructions, can load constants for comparison, and can update the verdict based on the outcome the the field assertion, or use the outcome of an assertion to determine the control flow. The Controller receives the current instruction from the Instruction Fetcher. The Instruction Fetcher contains the expected elements of a VM-based architecture, namely the instruction memory and the program counter. Furthermore, the Instruction Fetcher handles the instruction pipelining to achieve a higher productivity within the system. This functionality is contained in a single module to keep the focus of the other modules on executing the instructions.

Finally, the Stack serves multiple purposes. It is used for calling subroutines, storing the values for checking the length of a section, or counting iterations of routines. These functions would not be possible without a stack, as they require temporary storage of a value and support for nesting these operations. For example, subroutines may call other subroutines, or sections being checked for their length can contain nested length checks within themselves.

Appendix A contains an overview of which modules are activated per instruction, showing how the described functionality relates to the instructions.

The remainder of this section performs a deep dive on the modules and is structured as follows. First, Section 3.5.1 explains both the Input and Output systems for how they handle the data flow of the system. Second, Section 3.5.2 discusses the Controller for what it functionally executes for every instruction. Third, Section 3.5.3 introduces the Instruction Fetcher and how it handles instruction pipelining. Fourth, Section 3.5.4 explains the Stack for its various functions. Finally, Section 3.5.5 handles the full design, showcasing all module interconnections.

3.5.1. Input and Output System

The Input system is responsible for converting data from the input stream to data fields, which it puts on the data field bus. The Output system converts these data fields back to an output stream, which has identical contents to the input stream. In conjunction, they also perform the task of ”flushing” data through the system, which sends any remaining data through the system as fast as possible. This is done when the blob is invalidated during processing, and the next blob needs to be accessed as soon as possible.

Input System

The Input system takes a byte-aligned input stream. This stream contains the data of the binary blob and is in big-endian order. The byte-aligned data arrives in fixed-size sections, which by themselves cannot be used for individual data field assessments. The input system instead consumes these sections, where it outputs them as an individual bit-aligned data field, suitable for assessment. This operation is initiated by a `shift_enable` signal, and the amount of bits to be shifted from these sections onto the data field bus is determined by the `shift_amount`. Multiple data fields can be located in a section, meaning that multiple `sft` instructions can operate on the same section. When the section is fully traversed, the next section of the input stream is consumed by the Input system. It is also possible that a data field is split across two or more sections of the input stream, which the Input system takes care of by consuming the required sections to create the data field.

Besides shifting in data fields by a fixed amount, the input system also supports variable shifting for TLVs or other similar formats. This essentially uses the current field on the data field bus as the shift amount for the next data field. It can be interpreted as a bit or byte amount. A byte amount will first be converted to bits, as the shift amount denotes the number of bits.

Finally, the input system counts the number of bits, bytes, or fields that have passed through the system. This is for length assertions, as these counts are used to check the length of a section. First, the counters are accessed when a length field (denoting either bits, bytes or fields) is on the data field bus. By adding the value of the appropriate counter to the length field, we get the expected counter value when that section has passed through the system. This value is pushed to the stack for later evaluation. Second, after the section has passed through the system, we pop from the stack, the popped expected counter value is then compared with the current counter value. If they match, the length assertion passes.

Figure 3.11 shows the input system as a module. A control signal which was not discussed yet is the `flush_enable`, which will be discussed further down in this section.

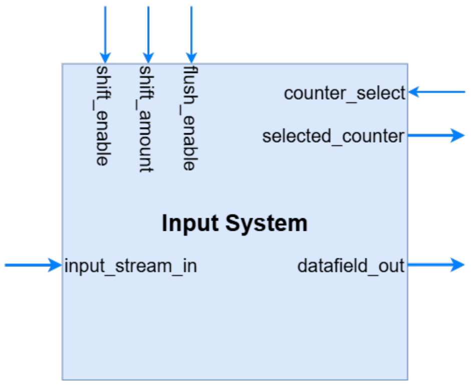

Figure 3.11: The input system module.

Output System

The output system does the reverse process compared to the input system. It takes individual data fields as input and converts them to an output stream, the contents of which are identical to the input system. It consumes data fields when the `transfer` signal is raised during the shift instruction. When the input system extracts a new data field and puts this on the bus, the output system consumes the current data field on the bus and adds this to the current section that it is building. Once the accumulated data fields have reached the section width, the section is outputted to the output stream. If a part of the data field goes over the section width, it is put in the next section.

The reason why an output system exists instead of having the input system output the original input stream is because of how the verdict travels with the data. In the case of an invalid data field, the verdict needs to become negative at the first section of the stream where the invalid data field is located. Let us assume the output system does not exist. In the case there is an invalid data field which overlaps across two sections, the first section needs to be outputted first to fetch the next section containing the remaining part of the invalid data field. This means that the previous section has already been sent to the output with a positive verdict, before the data field was even complete. Only when the second section completes the data field, assertions will invalidate the second section containing the data field. This goes against the requirement of having the verdict be in real-time with the data. This is prevented by having the output system, as the sections are only outputted once all data fields in that section have been processed, including the ones that overlap with later sections.

Figure 3.12 shows the output system as a module. A control signal which was not discussed yet is the `last_field`, which will be discussed further down in this section.

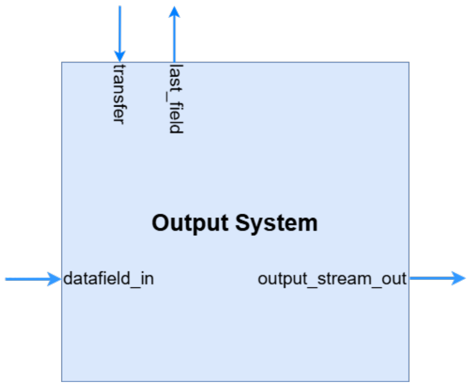

Figure 3.12: The output system module.

Flushing

Flushing is the process of having the data be passed through the input-output system as quick as possible, and can be triggered at any point when assessing the blob. Flushing happens when the verdict becomes negative, or the schema has ended but has remaining data left at the input. In both cases further processing is unnecessary, and we want to continue with the next blob as soon as possible.

Flushing happens when the `flush_enable` signal is raised for the input system. At the same time, the `transfer` signal is also raised, which causes the output system to consume the current data field and any subsequent data thereafter. First, any remaining data from the current section is outputted to the data field bus. Afterwards, the data field bus is filled with the data from each subsequent section, until the last section of that blob is consumed by the output system. From that point onwards. The output system raises a `last_field` flag, which indicates that the last section has been sent to the output. The rest of the system then knows it can reset itself for processing the next blob.

3.5.2. Controller

The controller module ensures that the instructions are properly executed. It sets the right signals for each instruction, such that the surrounding modules perform the appropriate task. It keeps the state of the verdict and three internal verdicts, namely the comparison verdict (`cmp_verdict`) for field assertions, branching verdict (`brch_verdict`), and length verdict (`len_verdict`) for length assertions. It manages the constant register and houses the comparator unit. Furthermore, it manages the state of the system, such that it is either processing or flushing. Finally, it resets the required parts of the system before processing a new blob. The list below explains what the controller performs per instruction. Appendix A highlights this functionality in terms of the module activations per instruction.

- `sft`: The controller enables the input system and output system by setting the `shift_enable` and `transfer` signals. Furthermore, it sets the shift amount either from the shift instruction or from the current data field on the bus (for variable shifting). Finally, if the `cmp_verdict` or `len_verdict` of the current data field is negative (0), the controller sets a negative verdict, such that this verdict moves with the current data field, and triggers a flush.
- `end`: In the case of a normal end, the controller checks the comparison verdict and the length verdict. If at least one of them is negative, the verdict is set to negative. Otherwise, the verdict stays positive and the blob is deemed valid. When the explicit fail flag is raised, a negative verdict is always outputted. In all cases, the flush is triggered.

- `ldim`: The controller will left shift the contents in the constant register by the amount of the immediate value in the instruction (26 bits), and load the new immediate value from the instruction. Alternatively, if the flag for the constant register reset is raised, all bits in the constant register are reset to 0 in case of an unsigned constant. All bits in the register are reset to the MSB of the immediate value in case of a signed constant, therefore handling sign extension. After the reset, the first immediate value from the instruction is loaded into the register.
- `cmp`: The controller will select the operands for the comparison based on the input_mode from the instruction and configure the comparator to perform a signed or unsigned comparison. The comparator can either output a <, =, or > comparison result bit. The assertion result is derived by the controller by comparing the set bit from the comparator output with the expected result. The assertion result is set to true if the raised bit of the comparator output matches one of the raised bits in the expected result. Either the `cmp_verdict` or the `brch_verdict` (based on the compare or branch flag) is selected for accumulation with the current assertion result. This is either AND-ed or OR-ed (based on the AND/OR flag) with the selected verdict, which is how multiple assertions are accumulated.
- `brch`: The actual branching happens in the instruction fetcher, as this module manages the program counter. (Explained in Section 3.5.3) The controller resets the `brch_verdict` to 1 after the branching has finished.
- `jmp`: The instruction fetcher executes this instruction. The controller does not perform any action.
- `call`: The controller will signal the stack to perform a push operation and select the right data for the stack entry that will be pushed into the stack. The instruction fetcher will take care of updating the PC to the call address.
- `ret`: The controller will signal the stack to perform a pop operation. In case the call mode in the popped stack entry is the exact length check mode, the length value of the stack entry is compared with the selected bit/byte/field counter. The result of this comparison is cumulatively AND-ed with the `len_verdict`. This is relevant in consecutive length assertions. All length assertions need to be true for all section lengths to be correct. Hence, the `len_verdict` is all the length assertions AND-ed together. Finally, the instruction fetcher will take care of updating the PC to the return address.
- `inc`: The controller will signal the stack to increment the `entry_value` of the top stack entry by 1.

Finally, when a flush is triggered, the controller will go into a "flush phase". This means no instructions will be executed, and the controller will stay idle until all the data has passed through the system, after which the system will reset the state for the next blob to be processed. Figure 3.13 shows the controller as a module.

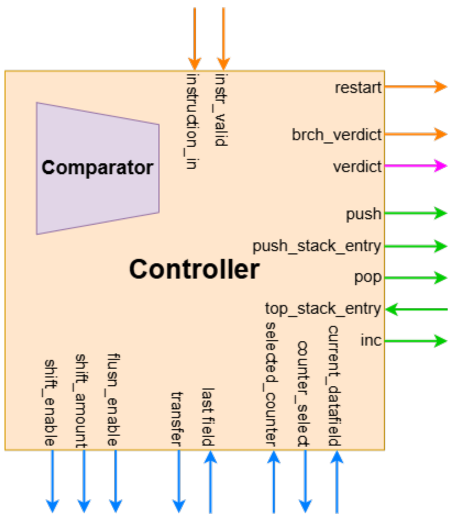

Figure 3.13: The controller module. Orange arrows interface with the instruction fetcher, green arrows interface with the stack, and blue arrows interface with the input and output system. The pink arrow is the verdict on the binary blob.

3.5.3. Instruction Fetcher

The instruction fetcher module is responsible for providing the system with instructions and managing the instruction pipeline. It contains the instruction memory and the program counter. The instruction fetcher updates the PC in case of the `brch`, `jmp`, `call` instructions. The instruction fetcher outputs the PC + 1 address such that the call instruction can use this as the return address during the `call` instruction. In the case of the `ret` instruction, the PC is updated with the return address of the popped stack entry. Figure 3.14 shows the instruction fetcher as a module.

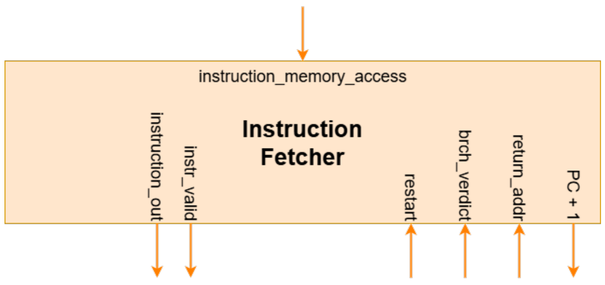

Figure 3.14: The instruction fetcher module.

Pipelining

The verdict engine has an instruction pipeline to improve system performance. In earlier versions of the system, the instructions were not pipelined, which led to every instruction taking an extra clock cycle to fetch the instruction from memory. This instruction fetch can overlap with the instruction execute, reducing the number of cycles per instruction. The system has a 2-stage pipeline with **instruction fetch** and **execute** stages. A decode, memory access, or writeback stage was not required as the system does not need to decode additional operands, like registers, nor does it need to access a memory or register file. Pipeline stalls are possible with the following instructions:

- `sft`: A data field shift causes a pipeline stall of one cycle if the data field spans across two input sections. This is dependent on consuming new sections from the input stream. Where each section takes one cycle to consume and use for the shift.
- `brch` (when the `brch_verdict` is true), `jump`, `call`, and `return`: These instructions update the PC, which introduces a one-cycle stall to fetch the correct instruction from the instruction memory.

Every cycle, the instruction fetcher will output an instruction; however, an extra `instr_valid` signal is introduced that travels with the outputted instruction. This determines whether the instruction is valid or not. Suppose the current instruction will stall the instruction pipeline for the next cycle. The instruction fetcher will then invalidate the next instruction, which allows the instruction fetch stage to fetch the correct instruction from the instruction memory for the next cycle. Figure 3.15 shows the pipeline.

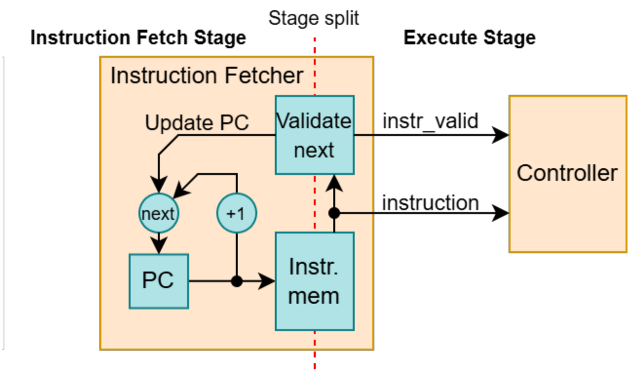

Figure 3.15: The instruction pipeline of the verdict engine. The validate next block can recognise instructions that stall. It will then invalidate the next instruction, such that the instruction fetch stage has an extra cycle to fetch the correct instruction from memory.

3.5.4. Stack

The stack module is used to store stack entries. (See stack entries in Table 3.4 and their usage in Sections 3.3.7 and 3.3.8). They are used for calling schemas, checking lengths, and managing hierarchy within data formats. It is a Last-In-First-Out (LIFO) stack, meaning that stack entries can only be accessed from the top, without random access to inner elements like in conventional processors. The stack has 3 control signals:

1. `push`: Puts a new stack entry on top of the stack.
2. `pop`: Removes to topmost entry on the stack.
3. `inc`: Increments the `entry_value` of the topmost stack entry by one. Used for repeat loops.

Figure 3.16 shows the stack as a module.

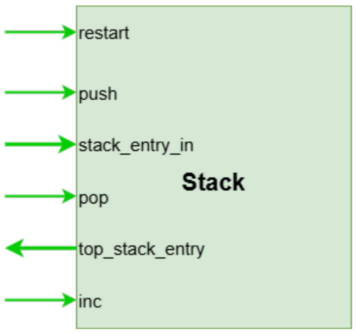

Figure 3.16: The stack module.

The system cannot perform length assertions, subroutine calling, or subroutine iterations without a stack, as these operations require the storage of a value, and these operations can be nested. A LIFO stack is the minimal data structure which can support this functionality. A random-access stack, like in conventional processors, was not chosen due to the added complexity and lack of data isolation. The complexity of a random-access stack relates to the actual implementation of the stack module, requiring more logic and error handling to support random access, but also extends to a future implementation of the compiler and ISA, requiring concepts like stack frames, liveness analysis and more instructions to ensure proper access to data on a random-access stack.

The stack will always output its topmost stack entry during system execution and is only updated during a push or pop operation. Having strict access to only the topmost element isolates access to deeper stack entries at a specific hierarchy level in a binary format. Therefore, lowering the risk of accidental or malicious data leakage across hierarchy levels.

The main advantage of a random-access stack would be to store data fields for assertions further down in the data. However, it was shown that such assertions can still be performed earlier, as shown in the ”Callable schemas” code example in Section 3.2.1. Therefore, the same control flow can be maintained, at the cost of an overhead in instructions.

3.5.5. Full Design

The overview in Figure 3.17 shows the full design, consisting of the modules and their connections. Some notable connections in this overview are:

- The data field bus is connected to the controller as it is used for multiple occasions. It is used for comparison with constants. It can be used as the shift amount in the next shift instruction, and the data field can be pushed onto the stack for length assertions.
- Next to the controller requiring a restart before processing the next blob, the restart also goes to both the instruction fetcher and the stack. The instruction fetcher needs to restart the program at the first instruction, and the stack needs to restart its stack pointer at the bottom of the stack.
- The return address is split from both the `stack_entry_in` and `top_stack_entry`. The `stack_entry_in` gets the `PC + 1` address from the instruction fetcher, and the instruction fetcher requires the return

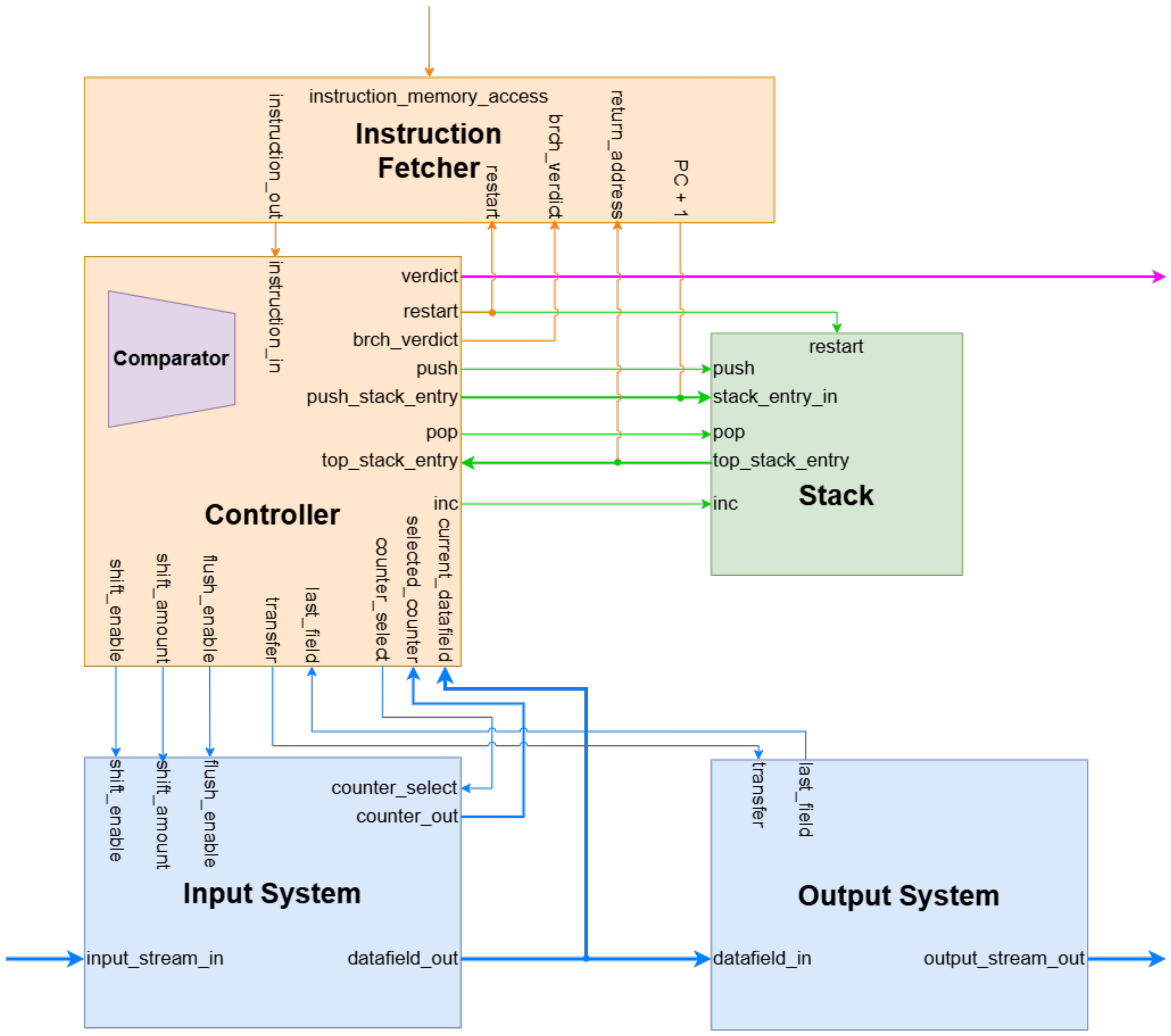

Figure 3.17: Overview of all modules and their connections.

address from the `top_stack_entry` to return to the caller subroutine.

- The `top_stack_entry` always has the topmost element available on its bus. This is in case assertions are required with the topmost `entry_value`.

3.6. Conclusion

In this chapter, we explained our design for this project. Before discussing the design, we provided the requirements of the system, as derived from the background study. These requirements were categorised into either data assessment requirements or architectural requirements. As these required much flexibility from an underlying architecture, the VM architecture was chosen as it allowed for format variability, runtime programmability, and future extension to system functionality. Furthermore, we touched on how we considered security in the design process. From the data assessment requirements, we designed a schema language which can: declare fields in the expected order of arrival; declare assertions on fields; assert the length of a section; declare fields with a variable width; declare conditional statements to change program flow; call other schemas; declare repeat loops; declare repeat while loops; and declare the end of a schema. Afterwards, the ISA was explained per instruction, and a pseudo notation was explained to denote the instructions in a human-readable format. We linked the schema language to the ISA by explaining the conversion from schema language constructs to pseudo notation. Finally, we discussed the module design. The modules execute the instructions of the system. First, the input and output systems were discussed in how they convert the input stream to data fields with a specified width, and how the data fields are converted back to an unmodified output stream. Second, the controller was explained for its function per instruction. Third, the instruction fetcher provides instructions to the system and maintains the instruction pipeline when updating the Program Counter or shifting. Fourth, the stack can push and pop stack entries used for schema calling, length assertions, repeat loops, and while loops. Lastly, the full design was provided, showing the interfaces between the modules and external interfaces.

4 Implementation

The previous chapter explained the module design, ISA, and pseudo notation for the Verdict Engine. This design is implemented in VHDL, and an assembler program is made in Python to convert the pseudo notation to binary instructions that the VHDL implementation can run. Furthermore, a functional emulator of the Verdict Engine is made in Python to test and debug schemas before deployment on an FPGA. These implementations are discussed in this chapter. At the end of this chapter, we end up with an implemented design that we evaluate in the next chapter.

This chapter is organised as follows. Section 4.1 describes the schema writing workflow that uses the implemented systems. Section 4.2 explains the Python assembler program used to convert the schemas in pseudo notation to the final binary instructions, which both the Python emulator and hardware implementation can use. Section 4.3 introduces the Python emulator and highlights the implementation details, and provides an overview of its operation. Section 4.4 introduces the hardware implementation in VHDL and explains how AXI4s, AXI4l, and pipelining work in the implementation. Finally, Section 4.5 concludes this chapter.

4.1. Schema Development Approach

The implemented systems in this chapter are part of a systematic and iterative workflow for developing schema programs for the verdict engine. This workflow is shown in Figure 4.1.

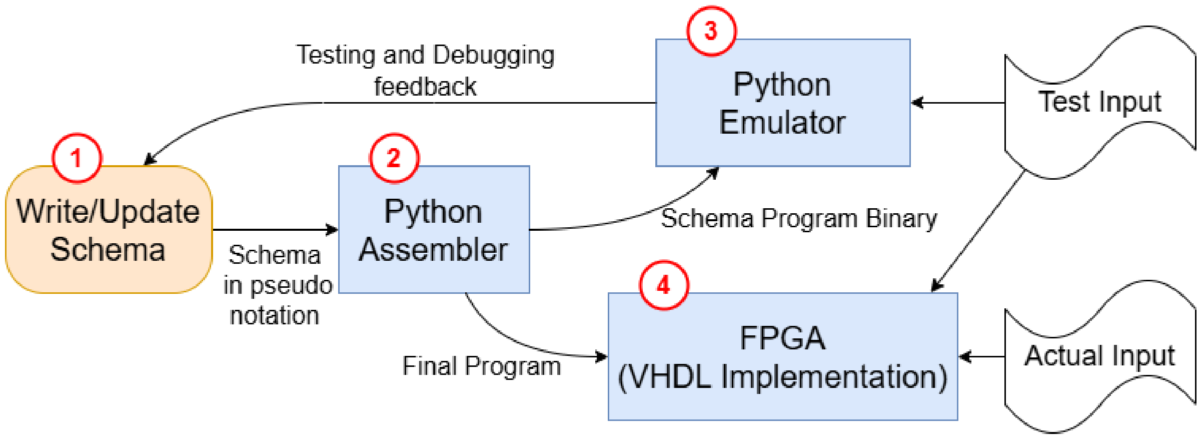

Figure 4.1: The schema development approach for new schemas.

The numbers in the figure determine the workflow order. These are:

1. **Writing a Schema**: The process begins with writing a schema for a binary format, using instructions in the pseudo notation described in Section 3.3.

2. **Python Assembler**: The written schema is passed to the Python-based assembler, which converts the instructions in pseudo notation into binary instructions that can be processed by the subsequent systems.
3. **Python Emulator**: The binary instructions are then loaded into the Python Emulator, which is a VM program that functionally emulates the verdict engine. This environment allows for testing and debugging the schema program using test input data. Any issues discovered here provide feedback for updating the schema, prompting a return to step 1.
4. **FPGA (VHDL Implementation)**: Once the schema is confirmed via the emulator to have the correct functionality, the binary program can be loaded into the VHDL implementation of the verdict engine on the FPGA. Final system-level tests can be performed to confirm the correct operation of the schema in hardware. Finally, the schema is used to process actual input data in real-time.

Each of the blue-highlighted components in this workflow will be described in detail in Sections 4.2 to 4.4.

4.2. Python Assembler

Because of the absence of a schema language compiler, the schemas have to be written as instructions. To program the system without having to write binary instructions, an assembler program is created in Python to convert the instructions written in pseudo notation to binary instructions. This allows for a more human-friendly way of programming the system. A program in pseudo notation contains instructions written inside one or more subroutines. This is processed first by the preprocessor, which turns the pseudo notation into a single list of individual pseudo instructions. This then enters the converter, which converts them to the final binary instructions.

4.2.1. Preprocessor

The preprocessor will start by removing comments, whitespaces, and leading/trailing spaces. Next, `ldim` instructions are inserted before `cmp` statements that contain an input-constant comparison. In case a shift is included in the same `cmp` statement, this is also inserted before. The code block below showcases this process with an example.

```
/* Pseudo notation. */
sft 8 cmp unsigned inp == const=1

/* Intermediate representation after the preprocessor. */
ldim 134217729 // ldim instr, the number represents the reset const flag and immediate value
    in binary.
sft 8          // Shift in 8 bits.
cmp unsigned inp == const // Compare with current data field and constant register.
```

Furthermore, the preprocessor derives the start addresses of each subroutine and creates a map from the subroutine label to the start address. This is used for all the `brch`, `jmp`, and `call` instructions to replace the label of the target subroutine by the starting address of that subroutine. Afterwards, all instructions are put into a list from all subroutines, and the program is now a list of preprocessed instructions.

Finally, the preprocessor also checks for faulty decimal, binary or hexadecimal numbers that are used in a pseudo notation. Furthermore, it is detected when a non-existent subroutine is referred to (e.g. due to a typo.) The preprocessor will throw an error and stop the program so the programmer can fix the mistake.

4.2.2. Converter

The converter converts all the preprocessed instructions to the final binary instructions. Based on the instruction keyword, the opcode is set, and the various keywords, numbers, or addresses that follow to configure the instruction are set to the right final bit values. Any instruction which has the wrong keywords, wrong values or wrong order will be detected. Every faulty notation detected has a specific error message, such that there is a degree of feedback when writing the schema, which makes it easier to program and debug. The converter outputs a list of binary instructions, which can be used by both the Python emulator and VHDL implementation of the verdict engine.

4.3. Python Emulator

The Python emulator is a Python virtual machine program which executes the instructions from the ISA. Instructions are executed in the emulator according to their functional description in Section 3.3. This Python emulator was made to help evaluate whether the ISA can meet all the functionality required to assess various binary formats. This was done before starting with the VHDL implementation, as it was deemed a harder platform to test the instructions on whether they contain the right functionality. Furthermore, the emulator makes the testing and debugging of schemas more accessible, as a flexible software environment with a Python debugger provides adequate feedback when testing a schema program. On the other hand, testing a VHDL implementation often requires manual implementation of various debugging or feedback capabilities.

4.3.1. Differences with Module Design

Although the instructions executed perform the functionality described by the ISA, the implementation has some differences concerning the module design. This is because the implementation is in software, and the use case of the emulator is schema writing and debugging. This led to different input and output handling, instruction fetching and execution.

Input and Output Handling

The software environment led to the implemented system having a different approach to the input and output stream, as well as the verdict. As an input and output, the emulator takes in binary files. Every blob is put in a single binary file to indicate the boundaries of each blob. All the data at the output is written to a single binary file containing the data of all the blobs. This was done to save the order in which the blobs were processed. The verdicts of each blob are written together to a separate file. A verdict is displayed as `true` or `false` in the file. Each verdict is written in the order in which the blobs are outputted in the output file.

When shifting in data fields, the input system reads the input file per byte. In case the data field is not byte-aligned, the remaining bits in the last fetched byte are saved for the next shift instruction. Similarly, the outgoing data field is also written to the file per byte by the output system, and any remaining bits in the last byte will be saved until the next outgoing data field needs to be written to the output file.

Finally, the maximum data field width which the input system can shift is 256 bits. This means that the comparisons are also 256-bit integer comparisons, constants can be loaded in up to 256 bits, and the entry values in the stack entries are also 256 bits wide.

Instruction Fetcher and Controller

The fetcher and controller have their logic merged, which differs compared to the module design. In hardware, this separation makes more sense as instruction fetching and execution are different stages in the instruction pipeline (Section 4.4.2 - Instruction Fetcher and Pipelining will explain this in more detail). In the emulator, this separation does not exist, which led to the logic being merged. The emulator keeps the instructions in a Python list, with the program counter being the index for this list. Once the instruction is accessed, it is decoded by a decoder object, which creates an `Instruction` instance. This instance contains the opcode of the instruction and a list containing every field in the instruction. Afterwards, the program continues with the execution of that instruction.

Stack

The stack in the Python emulator is implemented as a class. The stack is a Python list containing stack entries. The stack is manipulated through `push()`, `pop()`, and `peek()` functions.

4.3.2. Emulator Operation

The schema binary and the input files are loaded into the emulator. The emulator has a `execute_instr()` function, which executes one instruction per call. This function is continuously called in an infinite while loop and stops the program once the output is complete, or if an error is thrown during execution.

The `execute_instr()` function fetches the next instruction from the instruction array and decodes the instruction fields based on the opcode. Afterwards, the instruction is executed. The instruction logic is located in a large Python match case statement. Based on the opcode, the individual instruction is executed in the case block. After the execution, the PC is either incremented or updated to the target address in the case of `brch`, `jmp`, `call`, and `ret` instructions. Flushing happens during a shift instruction when the outgoing data field is invalid, or when an explicit fail call is made in the `end` instruction. After flushing, all the output is written to the output file, and the program execution is stopped.

4.4. VHDL Implementation

This section discusses the VHDL Implementation of the verdict engine. First, Section 4.4.1 explains the AXI4-Stream and AXI4-Lite protocols and their use throughout the system. Second, Section 4.4.2 provides an overview of the implemented modules.

4.4.1. AXI4-Stream and AXI4-Lite

AXI4-Stream[30] and AXI4-Lite[31], or how we for short refer to them: AXI4s and AXI4l, are synchronous communication bus protocols specified by ARM as part of the Advanced Microcontroller Bus Architecture (AMBA) specification. AXI4s is a high-speed streaming protocol and is used for the input and output stream, as well as the data field bus. AXI4l is used within the system to access control and status registers and write to the instruction memory (More on the control and status registers in Section 5.1.1 - Serial Connection and Registers).

Handshake signals

Both AXI4s and AXI4l rely on a handshaking mechanism to ensure reliable data transfer between a source and destination. The handshake involves two signals, namely the READY and VALID signals. The source sets the VALID signal to indicate the data bus has valid data is available. The destination sets the READY signal to indicate that it is ready to consume the data. A data transfer completes when, in a clock cycle, both signals are high. A READY signal being low provides back-pressure to the source, and a VALID signal being low can signal that there is data throttling to the destination. This is relevant when multiple AXI4 buses are used between modules in a pipelined manner. Each stage can signal when it is ready to provide or consume data. This throttling or back-pressure can propagate through the pipeline, ensuring there is no data loss or data overflow when it moves through the system.

AXI4-Stream

The AXI4s protocol is a high-speed data transfer protocol that operates unidirectionally without addressing. In the system, it is used to send binary blob data to the input system, and for the output system to send the checked binary blob to any receiving module. Each binary blob consists of one or more AXI4s transfers, each transfer containing 256-bit sections. We will from now on refer to the signals of an AXI4s transfer as an AXI4s **packet**. Figure 4.2 shows how data is transferred over an AXI4s bus.

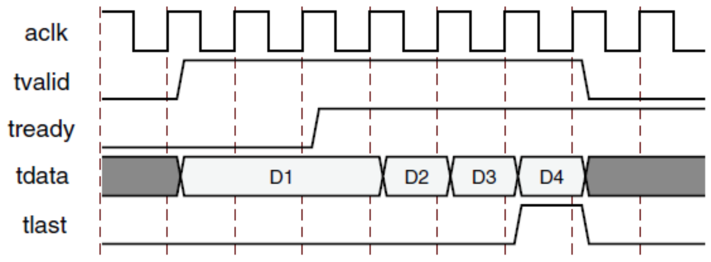

Figure 4.2: How AXI4s packets are transferred [32]. When both the valid and ready are high, a packet is transferred in that clock cycle. The last indicates the last packet of the data.

The signals in AXI4s that were used during the implementation of the input and output stream are:

- **VALID** (1 bit): Handshake signal. Indicates that the source is providing valid data on the bus.
- **READY** (1 bit): Handshake signal. Indicates that the destination is ready to accept the data.
- **DATA** (256 bits): Contains the data of the binary blob. In case the binary blob is larger than 256 bits, multiple AXI4s packets are used to send the blob.
- **LAST** (1 bit): Marks the last packet of the binary blob. This is used as a delimiter between binary blobs. The next packet after the last packet will therefore be from the next binary blob.

- **KEEP** (32 bits): Specifies which bytes of DATA are actual data. This is a 32-bit bitfield indicating which bytes (from MSB to LSB) in the DATA bus are valid for use. In case a binary blob does not have a size as a multiple of 256 bits, the last packet will have fewer KEEP flags set. All earlier packets of the same blob have all their KEEP flags set to 1.

If a binary blob has a size of 800 bits, i.e. 32 + 32 + 32 + 4 bytes, the binary blob will be sent in four AXI4s packets. The first 3 packets will have KEEP values of 11111111111111111111111111111111, indicating that all 32 bytes of the DATA bus contain data. The final packet will have the LAST signal set and will have a KEEP value of 11110000000000000000000000000000, indicating that the first 4 bytes on the DATA bus contain the remaining binary blob data.

Between the input and output systems, there is an AXI4s bus meant for presenting bit-aligned data fields to the rest of the verdict engine. Every AXI4s packet on this bus is a single data field from the binary blob. This bus has been modified from the AXI4s specification. The DATA signal on this bus is a 256-bit wide signal that holds data fields up to a maximum size of 256 bits. The LAST and KEEP signals are only meant for the output system to reconstruct the input stream packets. They are set on all the data fields that were originally located in the last AXI4s packet of the binary blob. Hence, it is possible that multiple data fields have their LAST and KEEP signals set, although they belong to the same binary blob. The KEEP signal is therefore not used to indicate the width of the data field; rather, a SIZE field is added to indicate the bit width of the data field. This is both important for comparison and for stream reconstruction.

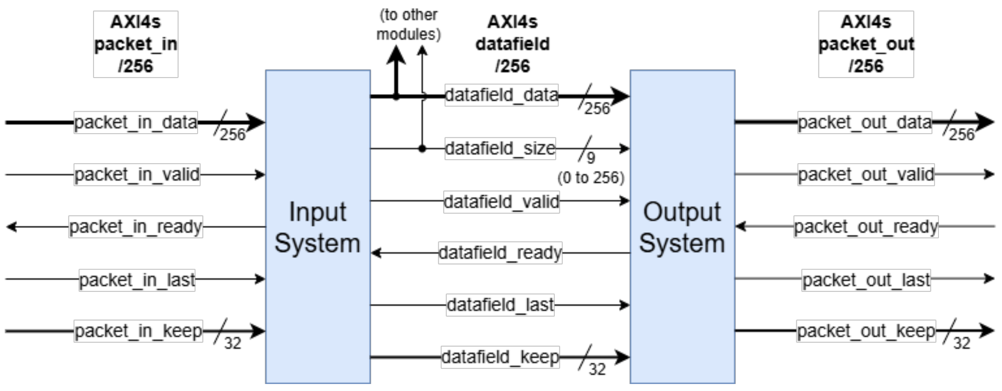

Figure 4.3: All AXI4s buses in the verdict engine. From left to right: the input stream, data field bus, and the output stream.

The 256-bit wide bus is deemed large enough for binary formats. Larger data fields are possible, but these are split into multiple packets. This data field then loses the ability to be a proper comparison operand because of this split, however, exact binary matches are still possible.

AXI4-Lite

The AXI4l protocol is a subset of the AXI4 protocol, meant for simpler register-style control and data exchange between an initiator and target. It is a bidirectional protocol, but achieves this through unidirectional channels, which have either the initiator or the target send data. It provides 5 different channels: **Write address** (from initiator), **Write data** (from initiator), **Write response** (from target), **Read address** (from initiator) and **Read data** (from target), each with their handshake signals. The data width used in the implementation per transfer is 32 bits, where it is used for memory-mapped control and status register interfacing, as well as writing in multiple transfers to the instruction memory.

4.4.2. Module Implementation

This section explains the implementation of each module in the system.

Input System

The input system converts the AXI4s input stream to data fields and outputs these to the `datafield_data` and `datafield_size` signals of the AXI4s data field bus. (See Figure 4.3. We will refer to these signals more often in this section.) Both the input stream and data field bus have a 256-bit DATA bus. The AXI4s input stream sends binary blobs to the input system. When there is data available at the input stream, the input system will handshake with the data source to receive an AXI4s packet containing a segment of the binary blob and store its `packet_in_data` in a `std_logic_vector(255 downto 0)` for traversal. The input system will not be ready for the next AXI4s packet until the current packet data has been fully traversed. For traversal, we introduce a counter, called the `header`, starting at the topmost bit position (255). This keeps track of which bit positions have not been traversed yet in the vector. Once the `header` reaches 0, the next packet is consumed from the input stream.

The Input System is implemented as an FSM with 3 states: `standby`, `shift`, and `flush`. In `standby`, the system idles until a shift or flush is initiated.

A shift is initiated through the `shift_enable`. This has the FSM transition to the `shift` state. To extract the data, the upper and lower bit positions of the data field within the stored packet data must be calculated. The `header` counter already indicates the upper bit position of this data field. The lower bit position, called the `footer`, is calculated by subtracting the `shift_amount` from the header, plus 1. With the range of bits determined, the data field is extracted from the current packet and placed on the `datafield_data` bus. The `datafield_size` is set to the `shift_amount`. To be ready for the next shift instruction, the `header` is updated by subtracting the `shift_amount` from it. The `datafield_valid` signal is then set to high, and we return to the `standby` state. This process happens in a single clock cycle. Figure 4.4 shows how the input stream packets are converted to individual data fields.

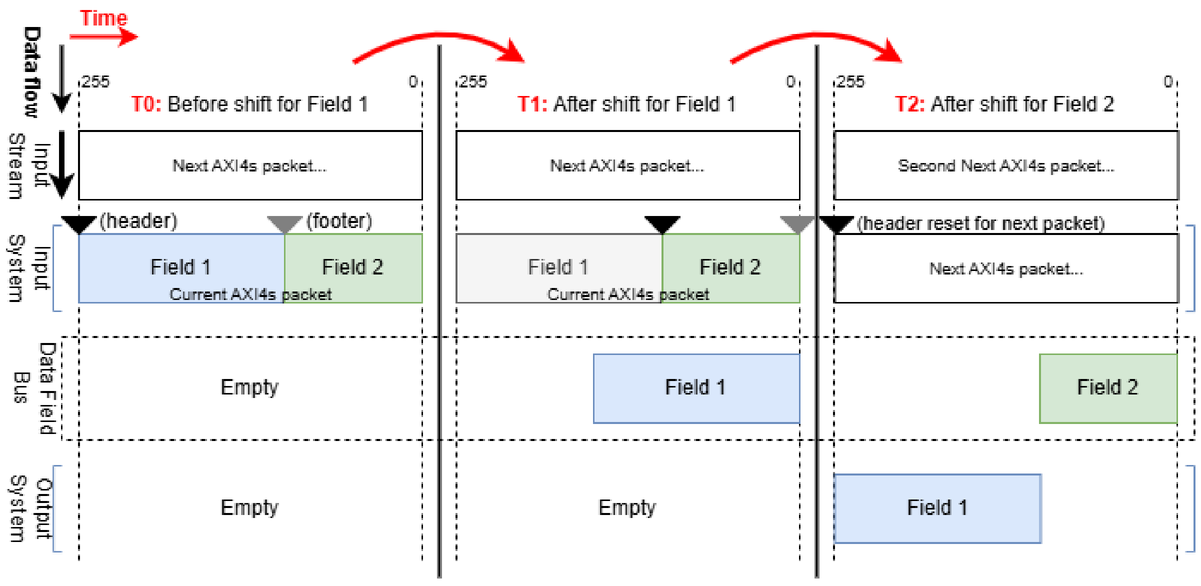

Figure 4.4: How AXI4s packets from the input stream are converted to individual data fields.

A shift can also take two clock cycles, this is the case when the `shift_amount` is larger than the current `header` value. Meaning that the next data field spans over two AXI4s packets. During the first cycle, the remaining data from the current packet is put on the data field bus, but the `datafield_valid` signal is kept low. The `header` is reset to 255, and another packet is requested from the input stream. This becomes available in the second cycle. During the second cycle, the remaining amount of bits is put on the data field bus, the `datafield_valid` signal is set to high, and we return to the `standby` state. Figure 4.5 shows how a split data field is put on the data field bus.

A flush is initiated through the `flush_enable`. The FSM transitions to the `flush` state. For the first cycle, the remaining data of the current packet is outputted. The remaining data, with a bit range of `header` down to 0, is put on the data field bus, and the VALID is set. If this was not the last packet, the next packet is requested from the input stream. In every subsequent cycle, the packet DATA, LAST and KEEP signals are copied onto the data field bus, the size is set to 256, the VALID signal is set, and the next packet is requested. The flushing ends when the last packet has been put onto the data field bus. The FSM transitions back to the `standby` state.

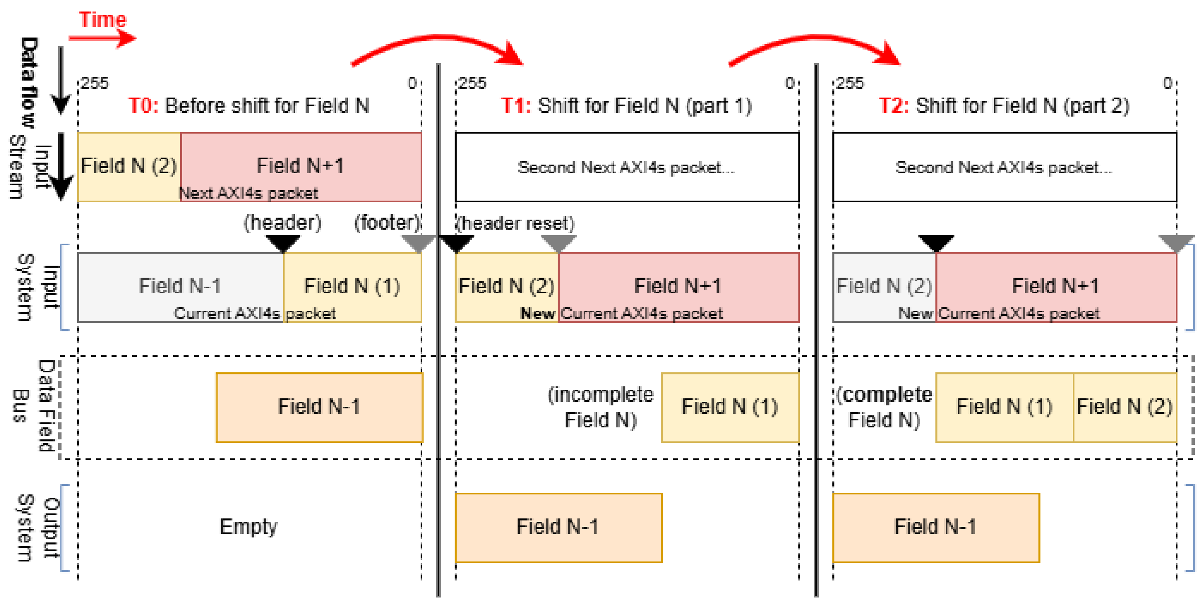

Figure 4.5: How a data field split across two AXI4s packets is put on the data field bus.

Output System

The output system converts the data fields received from the data field bus back to binary blob packets. The handshake signals of the data field bus in the output system are AND-ed with the `transfer` signal. Therefore, a data field is blocked from an AXI transaction until the `transfer` signal is raised. This happens during a shift, where the outgoing data field enters the output system and allows a new data field to be set on the bus. Similarly to the input system, the output system works with a `header` counter, this time keeping track of the bit position where the next incoming data field can be placed in the output packet. The `footer` of the output packet can be calculated from the `header` by subtracting the `datafield_size`, plus one. This determines the range where the data field is placed. The `header` is updated for the next data field transfer by subtracting the data field size from it. In case the packet is filled, the LAST and KEEP signals are copied from the data field bus and the VALID signal of the output packet is raised, the packet is outputted, and the `header` counter is reset to 255. This process happens in a single clock cycle.

The output system can also take two clock cycles when an incoming data field spans over two output packets. In the first cycle, the part of the data field is placed in the remaining part of the output packet. This output packet is made valid and outputted. In the second cycle, the remaining data of the data field is placed at the top of the new output packet.

There is no flushing state in the output system, since the input system provides the size of the data placed on the data field bus. Therefore, regardless of whether it is an actual data field or data from the flush, the output system places the incoming data into AXI4s packets and outputs these once they are filled. The output system will raise a `last_field_in` signal once the last data field has been transferred to the output system.

Controller

The implementation of the functionality per instruction is split between combinatorial logic and sequential processes. This has to do with achieving instruction execution on one clock cycle. The signals which are used to interface with modules (input and output systems and stack) are set combinatorially, such that the sequential processes in those modules can finish their operation in the same clock cycle. Combinatorial signals that trigger modules (`shift_enable`, `transfer`, `push`, `pop`, and `inc`) check for the validity of the instruction and the right opcode, and other instruction-specific checks before being raised. Execution in a single clock cycle is not possible for every instruction, such as in some `sft` instructions or instructions that update the PC. These will be explained in the instruction fetcher.

Signals that are set within the controller, such as updating the constant register or updating verdicts, happen within the sequential process. The sequential process is structured as a VHDL case statement on the opcode. This case statement only executes when the system is not in the flush phase. The following list explains where each instruction is implemented.

- **sft**: The shift_enable, shift_amt, and transfer signals are set combinatorially. In the sequential process, it is checked whether cmp_verdict or len_verdict is false (i.e. the current field is invalid), such that the controller transitions to the flush phase.
- **end**: In the sequential process, if either the cmp_verdict or len_verdict is false, or the fail flag is set, the controller transitions to the flushing phase.
- **ldim**: The constant register (i.e. std_logic_vector) is updated in the sequential process.
- **cmp**: The comparison, as explained in Section 3.5.2 is performed in the sequential process. A difference between the design description and the implementation is that a comparator module does not do the comparison. Instead, the selected operands are compared using the ”<” and ”=” operators in VHDL. (The ”>” operator is equal to both ”<” and ”=” being false, hence it is not used.)
- **brch**: The controller resets the brch_verdict to 1 in the sequential process.
- **jmp**: The instruction fetcher executes this instruction. The controller does not perform any action.
- **call**: The push signal and all the fields for the stack entry are set combinatorially, such that the stack module can write the data to the stack in the same cycle.
- **ret**: The pop signal is set combinatorially, such that the stack module can remove the data from the stack in the same cycle. The length check for a section and len_verdict update in the exact length mode are performed in the sequential process.
- **inc**: The increment (inc) signal is set combinatorially, such that the stack module can increment the top stack value in the same cycle.

During the flush phase, the controller only sets the flush_enable and transfer signals. Other signals cannot be set as the flush phase invalidates all incoming instructions. Once the output system signals the last_field_in signal, the flush phase in the controller ends, and the controller sets the restart signal. This is a different signal than the hardware reset signal, which is a global reset of every module. The restart signal is used to bring the system to a state where it is ready to process the next binary blob. It resets signals in the controller, instruction fetcher and stack modules, such that operation starts with reset verdicts, at the first instruction, with a clean stack.

Instruction Fetcher and Pipelining

The instruction fetcher contains the instruction memory and PC (implemented as a VHDL unsigned signal). The instruction memory is a dual-port RAM, one port is always reading instructions, the other port is connected to an AXI4l interface, meant for writing/reading instructions. The PC is connected to the address port of the read port, and the instruction being outputted is connected to the data port of the read port. An instruction read operation at the PC address requires 1 cycle before the read data is available.

A sequential process controls the instruction fetcher. This increments the PC by 1 every cycle, and checks for every valid instruction whether it is going to create a pipeline stall next cycle. The instructions which (optionally) stall the instruction pipeline are:

- sft (optional): Data field shifts cause a pipeline stall of one cycle if the data field spans across two input packets. This is because the second input packet is taken from the stream, which takes an extra cycle for receiving this data and extracting the remaining part of the data field. The instruction fetcher can predict this stall by checking if the shift_amt in the current instruction is bigger than the number of bits remaining in the current AXI4s packet. This is named the available_bits. In case of a stall, the PC is not incremented for that cycle.
- brch (when the brch_verdict is true), jmp, call or ret: During a branch, jump, call, or return instruction, the PC is updated to the brch/jump/call/return address. This introduces a one-cycle delay to fetch the correct instruction from the instruction memory, and the instruction during this delay cycle is invalidated. When the branch is not taken in the case of a branch instruction, there is no stall, and the next instruction is executed in the next cycle.

A catch with this implementation of handling pipeline stalls for the shift instruction is that the input system must have data available at the next cycle once requested. **This means the implementation currently only supports operation at maximum throughput.** Any data throttling or pushback can introduce errors in pipeline stalling when the shift instruction is executed. For the purpose of evaluating this prototype system, this is not an issue, as we want to run the system at maximum throughput for evaluating the maximum achievable performance of the system. How we provide the data to the system at maximum throughput during benchmarking is discussed in Section 5.1.1 - Packet Relay and Section 5.1.2.

Stack

The stack is implemented using a dual-port memory and several address pointers. These address pointers are `stack_push_addr`, `stack_top_addr`, and `stack_pop_addr`. The stack has the `push`, `pop`, and `inc` signals that are set by the controller, which initiate a stack operation. Per operation, the following happens:

- `push`: The new stack entry provided by the controller is set at the write data port of the memory and is written to the address that the `stack_push_addr` is pointing to. These are set combinatorially. The 3 address pointers are all incremented by one in the sequential process of the stack.
- `pop`: The topmost entry of the stack is always set on the bus. The controller sets the pop signal and uses the topmost entry in the same cycle. Furthermore, the read address for the stack top is set to the `stack_pop_addr` such that the second-to-top entry is read at the next cycle. The 3 address pointers are all decremented by one in the sequential process of the stack.
- `inc`: A `incremented_top` value stores the value of the topmost stack entry, where the `entry_value` is incremented by one. This is written to the address in the `stack_top_addr`. Because memory reads and writes introduce a clock cycle delay, this needs to be taken into account if there is any push or pop or increment operation before the current increment. In case in the previous clock cycle there was a push, the `incremented_top` is set in that clock cycle to the new stack entry value, incremented by one. In case in the previous cycle there was a pop, the `incremented_top` is set in that clock cycle to the second-to-top value, incremented by one (the second-to-top value is always being read out at the second port of the memory). Finally, in case in the previous clock cycle there was another increment, the `incremented_top` set during that clock cycle is again incremented by one in the current clock cycle.

A precedence order of commands is determined to prevent multiple commands from trying to access the stack within a single clock cycle. Pushing data takes precedence over popping, and popping takes precedence over incrementing the top.

Full Implemented Design

Figure 4.6 shows the implemented design of the verdict engine. This implemented design further details the design from Section 3.5.5 and includes the AXI4s and AXI4l buses, and the `available_bits` signal for pipelining.

4.5. Conclusion

In this chapter, we discussed the implemented module design in VHDL and the Python emulator and assembler programs. An overview was provided first on a workflow with the implemented systems for developing schemas. The assembler and emulator programs are used to debug and test schemas before they are deployed onto the VHDL implementation. Second, the working of the Python assembler was explained, where a schema in pseudo notation is first preprocessed to an intermediate representation, such that there is a one-to-one conversion to the final binary instruction. Third, the differences between the functional emulator of the verdict engine and the module design were discussed, and its general operation was explained. Finally, the VHDL implementation of the verdict engine was discussed. We first explained the AXI4-Stream protocol and how we used it in the input and output system of the verdict engine. We briefly touched upon the working of the AXI4-Lite protocol and where it was used in our system. Afterwards, the implementation of the modules was explained. First, the implementation of the input system was explained. Its interfaces had been made concrete as AXI4s buses, and the internal processing from AXI4s input packets to AXI4s data field packets during both shifting and

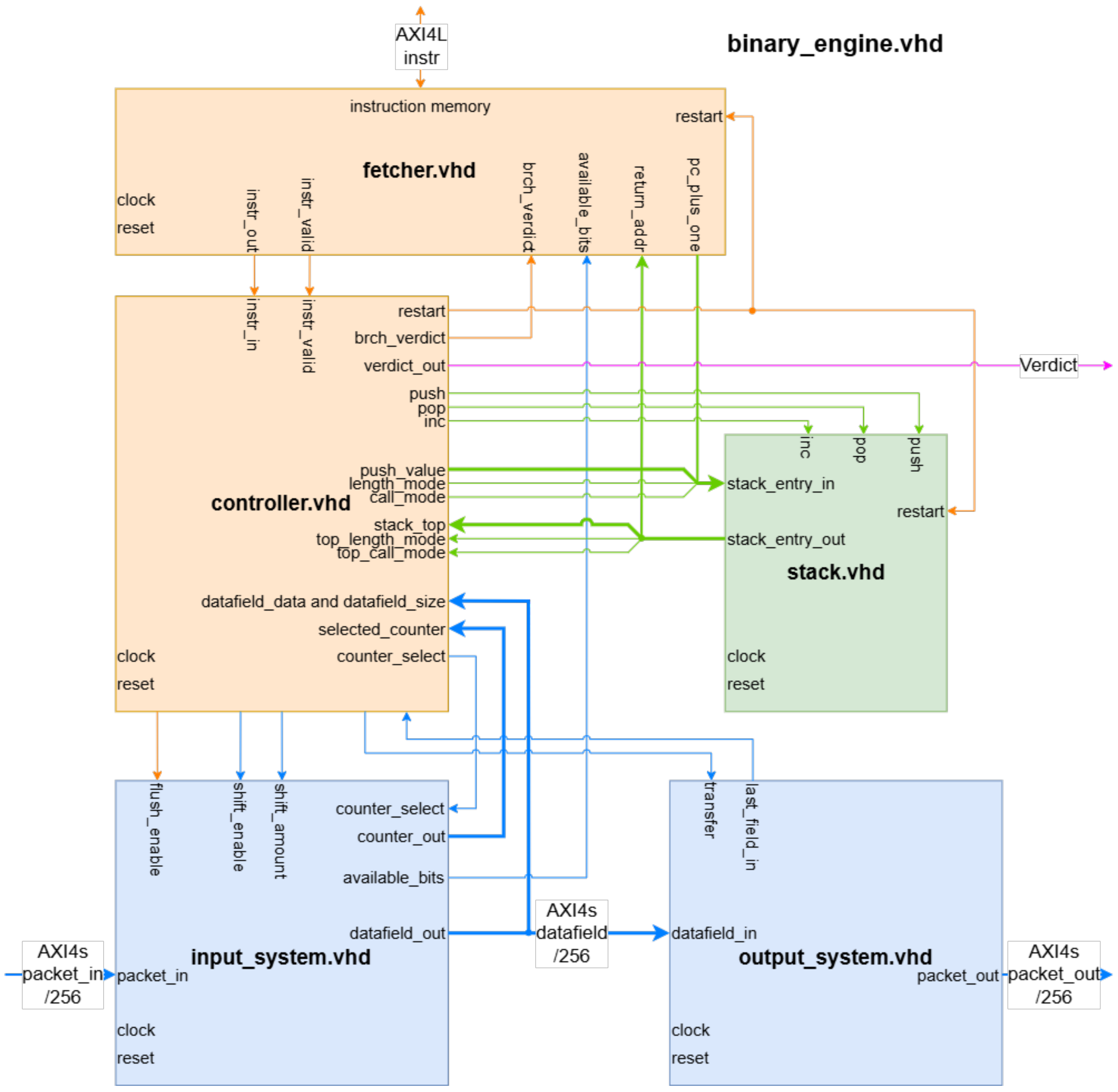

Figure 4.6: The fully implemented binary verdict engine design

flushing was explained. Second, the output system was explained for how it converts the AXI4s data field packets to the AXI4s output packets. Third, the controller was explained on how it implements the required functionality per instruction. Fourth, the instruction fetcher and its handling of the instruction pipeline were explained. The instruction fetcher needs to manage a two-stage pipeline and needs to be able to handle 1-cycle hazards during various instructions. An important catch with this pipeline was highlighted, which was that input data must always be available upon request. Fifth, the stack implementation was explained, and attention was given to how the next incremented `entry_value` was set. Lastly, the full design was shown, with the updated interfaces for the input and output systems and updated signals for the instruction pipeline.

5 Benchmarking and Results

In this chapter, the implemented design discussed in Chapter 4 is benchmarked. The system is benchmarked in several compatible use cases to showcase its flexibility. Furthermore, the benchmarks measure the performance to serve as a basis for comparison with related systems or as a reference point for future performance improvements of the system. We discuss an FPGA testbed and a VHDL simulation testbed, and how various metrics during benchmarking are measured. The performed benchmarks are explained, and their results are analysed. This evaluation of flexibility and performance fulfils the final goal to answer the research questions, which we conclude in the next and final chapter.

This chapter is organised as follows. First, Section 5.1 explains the setups created for benchmarking the verdict engine. Second, Section 5.2 displays the synthesis results of the design on the FPGA. Third, Section 5.3 explains how the metrics are collected during benchmarks and how specific metrics are derived afterwards. Fourth, Section 5.4 showcases the results from the performed benchmarks. Fifth, Section 5.5 discusses these results. Finally, Section 5.6 concludes this chapter.

5.1. Benchmarking Setups

This section will explain the setups created for benchmarking the verdict engine. Section 5.1.1 explains the setup on the FPGA and interfacing with the verdict engine module on the FPGA. Section 5.1.2 discusses the setup for benchmarking in the VHDL sim.

5.1.1. Hardware Setup

The implementation described in Chapter 4 is implemented on an FPGA to evaluate the design on physical hardware and to run benchmarks. The FPGA used is the Polarfire MPF300TS from Microchip Technology. Running on an MPF300-EVAL-KIT. The hardware setup is depicted in Figure 5.1.

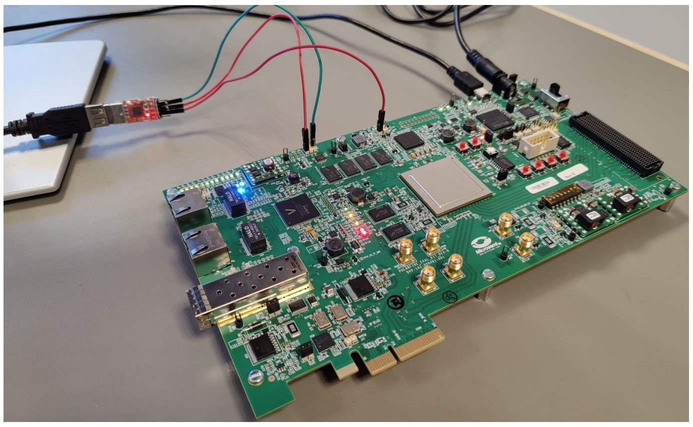

Figure 5.1: The evaluation kit with serial connection.

The testbed on the FPGA consists of the top-level verdict engine module and a packet relay module, as depicted in Figure 5.2. The verdict engine module has a 256-bit AXI4s packet input and output for the binary data. Furthermore, its AXI4l interface allows a computer to read and write to the instruction memory and status and control registers. The packet relay module matches the 256-bit AXI4s buses to send and receive the binary data. It also has an AXI4l interface for writing input data, reading output data, and writing to a control register containing the start signal.

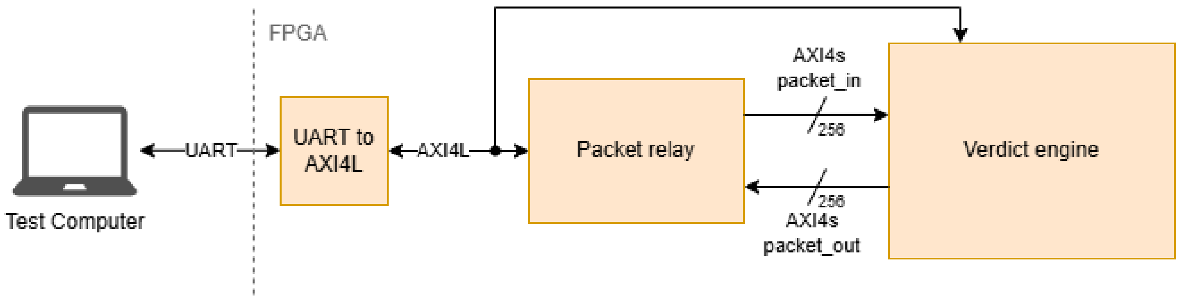

Figure 5.2: An overview of the modules and interfaces in the FPGA testbed.

Serial Connection and Registers

The test computer has a serial connection to interact with the system. To explain what this connection is used for and how it is established, we start with registers. Registers are used to interact with the testbed modules. Technolution has an in-house register description language (RDL) where registers can be declared and mapped. From an RDL description, a hardware module is generated where, on one side, the hardware can access the values of the declared registers, and, on the other side, the AXI4l bus master can read or write the values. Secondly, a Python class is generated for the test computer that contains the register values and mappings. A Python AXI4l master uses this class to manage these registers. We can use the AXI4l master to manually write to these registers via a console or use a testing script to run more automated tests and benchmarks. The Python AXI4l master connects to a serial port, performing its AXI4l reads and writes over UART with a 115200 baud rate. On the FPGA, this serial signal passes through a module that converts it from UART to AXI4l, thus establishing the AXI4l connection that can read and write to registers.

Control and status registers were implemented for the verdict engine module for interfacing over AXI4L. The control registers contain signals which the user can write to give a command to the system. The status registers contain read-only signals and counters which the user can access to see the state of the system.

The first control register contains a `schema_loaded` signal, which the user should set after programming the system. It is internally AND-ed with the READY signal of the AXI4s packet input, meaning that the system cannot consume data unless a schema is loaded. Furthermore, a `flush_then_restart` signal is used to force flush and reset the system, in case of a stuck program. The second control register is the `blob_count`, and contains the number of blobs that need to be processed before various metric counters are written to the status registers.

Various status registers were implemented. The first register contains various verdicts. It contains the real-time verdict, as well as a `sticky_verdict`, `sticky_cmp_verdict`, and `sticky_len_verdict`, which contain the verdict and the two internal verdicts of the last processed packet. The second register contains error signals. Both a `stack_underflow` and `stack_overflow` signal can be set by the stack. Next, a `shift_amt_too_high` signal will be raised if the shift amount is too big. This can happen if the current data field is used for shifting and contains a value larger than 256 bits (or >32 bytes).

Finally, multiple metric count registers are implemented, which contain the number of clock cycles counted, instructions executed, data fields shifted, and AXI4s packets traversed. These are updated once the number of blobs in the `blob_count` has passed through the system.

Packet Relay

The packet relay module is implemented for two reasons: First, it provides an interface between the computer and the verdict engine module for the binary input. It translates the AXI4l data we sent to AXI4s data, which it then sends to the verdict engine. Second, it functions as a data buffer that buffers the data until it is complete. The packet relay will always have data ready at the input when a start signal is given. This enables us to measure the system at full throughput, as the serial connection from the computer to the system cannot provide data at a sufficient rate.

The functionality of the packet relay is described by illustrating how data flows through it. The test computer divides one or more binary blobs into 32-bit chunks, as the AXI4l bus has a maximum data width of 32 bits. The packet relay modules have registers implemented for interfacing over AXI4l. The input data can be written per chunk to an input register, along with first, last, and keep signals, enabling the packet relay to interpret the data as a 32-bit AXI4s packet. An adapter module converts multiple 32-bit packets into a single 256-bit packet. The 256-bit packets are buffered in a FIFO. When the data is complete, the start signal can be activated, allowing the 256-bit packets to be consumed by the verdict engine. On the receiving end of the packet relay, the process is done in reverse: the processed packet is stored in a FIFO, after which it enters a packet adapter that converts the 256-bit packet back into multiple 32-bit packets. The computer can then consume the packets via an output register and receive the processed blobs.

5.1.2. Simulation Setup

Besides the hardware setup, A VHDL simulation environment is used for benchmarking. The simulation platform used is Riviera-PRO[33] and works in conjunction with cocotb[34]. Cocotb is a testbench environment in Python. It runs the Riviera-PRO simulator to simulate the VHDL, but the stimulus to the inputs and the monitoring of outputs are managed in the Python cocotb environment. Through this environment, bus functional models (BFMs) from Technolution simulate the AXI4s and AXI4l buses that are directly connected to the verdict engine top-level module. This enables us to do benchmarks as we can load schemas, provide the verdict engine with input blobs, consume the blob at the output, and access registers through the BFMs. Figure 5.3 provides an overview of the VHDL sim benchmarking setup.

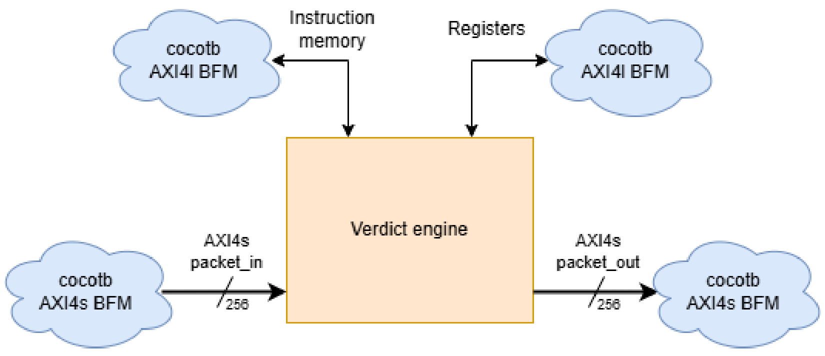

Figure 5.3: An overview of the VHDL sim benchmarking setup.

The simulation setup is used for benchmarks which require a bigger data set for metric collection. This is relevant for schemas which have variable timing for processing binary blobs, resulting in different measurements per blob. This can be because of variable lengths or many branches in the schema. Therefore, a larger set of blobs is required to evaluate the performance characteristics of the system when using the schema. However, running many blobs through the FPGA setup is a slow process due to the serial connection. With the VHDL sim, we can simulate the hardware at the maximum throughput of the data by using the BFMs, identical to how the packet relay can feed data to the system in hardware. Although VHDL sim cannot simulate the hardware at the speed of the FPGA, it is much quicker than having to send data over the serial connection.

5.2. Synthesis Results

This section shows and discusses the synthesis results of the implemented design. The synthesis tool used is Libero version 2021.2. Timing, Resource usage and Energy consumption are discussed.

5.2.1. Timing

The system achieves a **clock frequency of 100 MHz** (period of 10 ns). The synthesis tool achieved this timing constraint and reported a **worst slack of 0.818 ns**. The critical path is related to moving data fields from the input to the output system. This is, in general, where the longest paths reside, as well as in the 256-bit comparison. Knowing the maximum possible frequency of the system allows us to derive our results for throughput and latency.

5.2.2. Resource Usage

The resources used by the implemented design in the FPGA fabric consist of 4-input lookup tables (4LUT), D Flip-flops (DFF), micro SRAM (uSRAM) and large SRAM (LSRAM). uSRAM and LSRAM are memory blocks unique to the Polarfire FPGA family [35]. uSRAMs are 768-bit RAM blocks, containing one read and one write port. LSRAMs are 20K-bit true dual-port RAM blocks. The total available resources on the MPF300TS FPGA as reported by Libero are: 299544 4LUT and DFF (together they make one Logic Element), 2772 uSRAM units and 952 LSRAM units.

Table 5.1 shows the resource usage of the synthesised design. The indentation in the first column indicates that the module is a submodule. The total resources used include the verdict engine, packet relay, and other modules, e.g. for clock synchronisation and the serial connection. Libero reports the resource usage of the verdict engine module and its modules. The Input System is the largest module in terms of logic, which is in line with the functional complexity of this module. On the other hand, the Fetcher is the smallest but includes memory blocks for the instruction memory.

(Sub)modules	4LUT	DFF	uSRAM	LSRAM
Total	48295 (16.12%)	8431 (2.81%)	140 (5.05%)	3 (0.32%)
↳ Verdict Engine	43151 (14.41%)	3974 (1.33%)	46 (1.66%)	3 (0.32%)
↳ Controller	2992 (1.00%)	263 (0.09%)	0	0
↳ Fetcher	248 (0.08%)	180 (0.06%)	0	3 (0.32%)
↳ Input System	19311 (6.45%)	1163 (0.39%)	0	0
↳ Output System	16261 (5.43%)	615 (0.21%)	0	0
↳ Stack	3997 (1.33%)	1388 (0.46%)	46 (1.66%)	0

Table 5.1: Resource Usage Results for the total design, Verdict Engine module, and individual modules.

5.2.3. Energy Consumption

Table 5.2 shows the energy consumption from the Libero energy consumption report of the synthesised design. The results include the extra hardware required for the testbed, instead of solely the verdict engine module.

	Energy Consumption (mW)
Dynamic	181.195
Static	103.320
Total	284.515

Table 5.2: Static, Dynamic and total energy consumption of the system

5.3. Metrics

This section explains how the metrics are counted and derived. Section 5.3.1 explains the window during the hardware execution in which various metrics are counted. Subsequently, Section 5.3.2 explains how average throughput and latency are derived from the counted metrics.

5.3.1. The Measurement Window

Thus far, we have created benchmarking environments on the FPGA and in simulation, which can buffer data and, at a given start signal, provide the data to the verdict engine without any delays. This gives us a window of data processing at the maximum possible throughput, during which we count various metrics. This window starts the moment the first AXI4s packet handshake occurs at the input. This is the first point in time where the verdict engine gets access to the binary blob that we want to assess. Figure 5.4 shows where the measurement window starts.

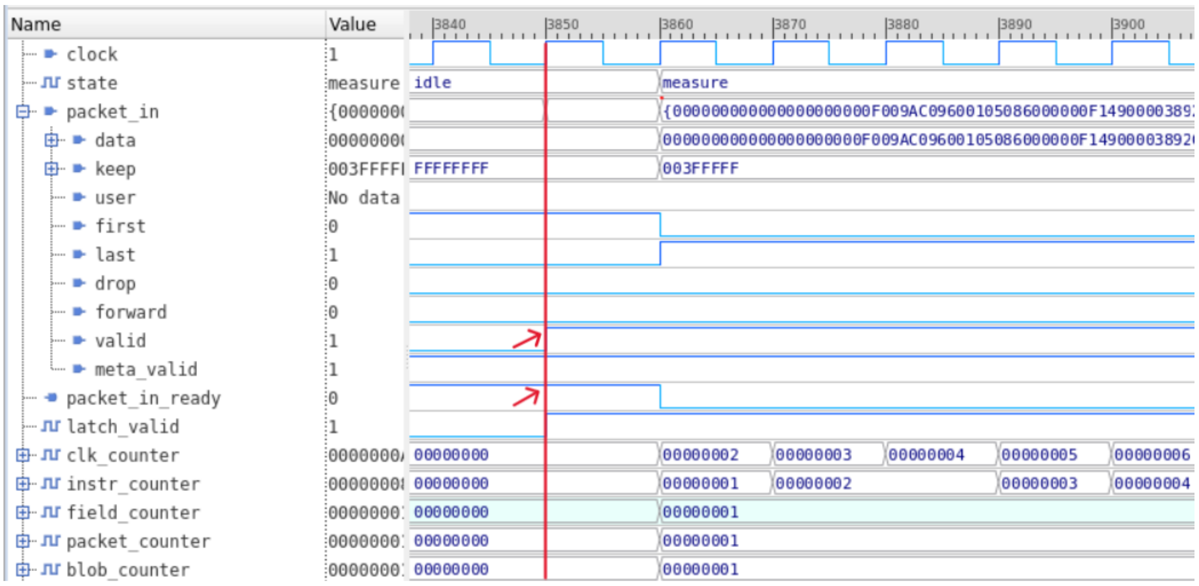

Figure 5.4: VHDL simulation showing where the measurement window starts. The red arrows highlight the AXI4s handshake at the input.

Inside the measurement window, the system counts clock cycles, instructions, data fields, AXI4s packets, and processed blobs (in case multiple blobs were buffered in the packet relay). The following list explains under which conditions a metric gets incremented in the measurement window:

- **Clock cycles**: The clock counter is incremented at every clock cycle.
- **Instructions**: The instruction counter is incremented every time the `instr_valid` signal of the instruction fetcher is high.
- **Data fields**: The field counter is incremented every time an AXI4s handshake occurs between the input and output system, indicating a new field has been placed on the data field bus for processing.
- **AXI4s packets**: The packet counter is incremented every time a packet handshake occurs at the output of the verdict engine, indicating a 256-bit AXI4s packet has been processed by the engine.
- **Processed blobs**: The processed blob counter increments at the same condition as the AXI4s packets, with the additional condition that the ”packet last” signal is high, indicating that the last AXI4s packet of a blob has been processed.

The measurement window ends when the last AXI4s packet of the last blob in the buffer has its handshake with the output. This is when the processed blobs counter is equal to the `blob_count` register. Subsequently, the metric counters are written to the status registers, after which the test computer can retrieve them. Figure 5.5 shows where the measurement window ends. In case we want to measure more blobs that cannot fit in a single window, the measurements from multiple windows can be added together. This is not a problem as two measurement windows with a break between blobs would count the same number of clock cycles as if combined.

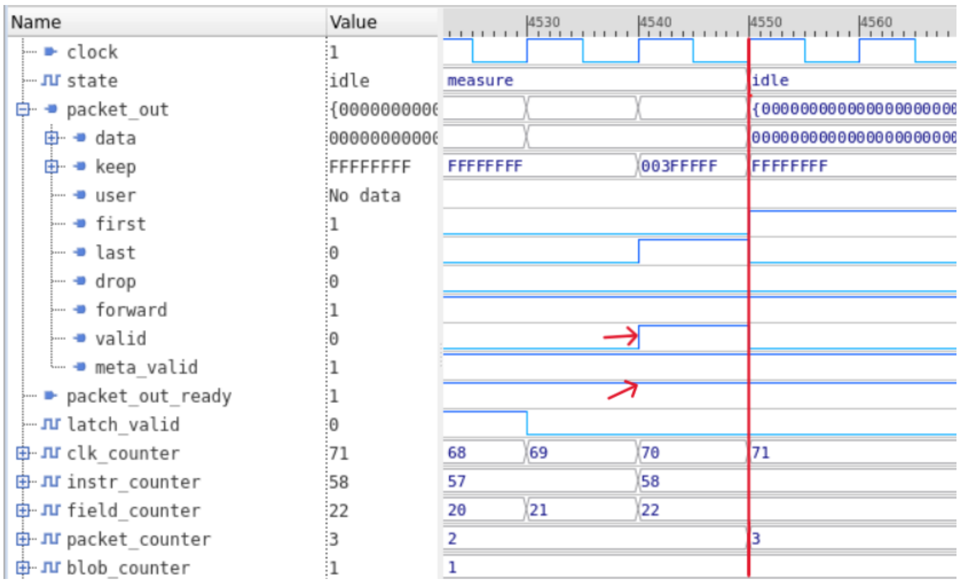

Figure 5.5: VHDL simulation showing where the measurement window ends. The red arrows highlight the AXI4s handshake with the output stream.

5.3.2. Deriving Average Throughput and Average Latency

The average throughput is the number of bits passed through the system divided by the number of counted clock cycles converted to seconds. More specifically, it is the average throughput of the system during the measurement window. If a schema has variable timing per blob, we can measure the average throughput for a larger set of blobs to get a representative average throughput for the schema. The formula below shows the calculation for the average throughput.

$$average\_throughput = \frac{\#bits\_processed}{\#clock\_cycles * 1e^{-8}}$$

To calculate the average latency, we need to calculate the average number of clock cycles required for an AXI4s packet at the input to go to the output. For every AXI4s packet, we count the clock cycles required for processing. These counts need to be summed up to average them. As every packet follows after another, the clock cycle count of the measurement window fulfils the task of counting the clock cycles for processing and summing them up. By dividing the clock cycle count by the number of AXI4s packets processed and multiplying this by 10 (1 cycle = 10 ns), we get the average latency in nanoseconds as a result. The formula below shows the calculation for the average latency.

$$average\_latency = \frac{\#clock\_cycles}{\#AXI4s\_packets} * 10$$

5.4. Benchmarking Results

This section shows the benchmarking results of the Binary Verdict Engine. Sections 5.4.1 to 5.4.6 discuss multiple benchmarks that show the performance of the verdict engine in various realistic use cases. Furthermore, Sections 5.4.7 and 5.4.8 show the timing characteristics of the engine related to schema growth and schema flexibility.

5.4.1. Benchmark: Packet Header Parsing

The first benchmark has the verdict engine parse internet packet headers. This benchmark enables the system to be compared to FPGA packet parsing systems discussed in the related work (Section 2.4). Various notable works related to FPGA-based header parsers implement parsers that implement two parse graphs. The first graph is **Ethernet, 2x VLAN, 2x MPLS, IPv4/IPv6, TCP/UDP, ICMP/ICMPv6**. The second graph is a smaller variation of the first: **Ethernet, IPv4/IPv6, TCP/UDP, ICMP/ICMPv6**. The graphs are displayed in Figure 5.6.

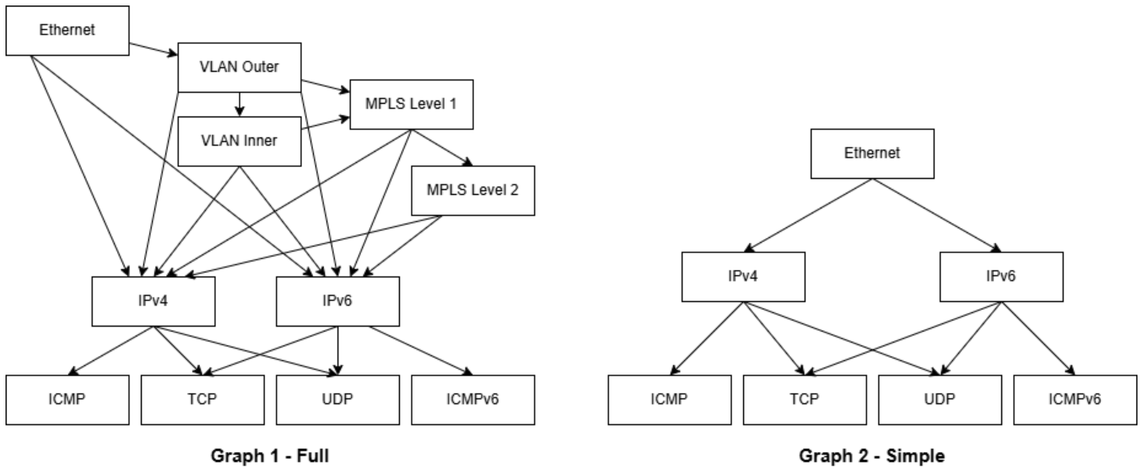

Figure 5.6: The two parse graphs which the benchmark schemas adhere to.

Four schemas are made, which can be found in Appendix B.1, each handling a different parsing case:

1. From the first graph, the classical quintuple of header fields is extracted: **IP source, IP destination, Protocol, Source Port and Destination Port**. (Appendix B.1.1)
2. From the first graph, all header fields are extracted. (Appendix B.1.2)
3. From the second graph, the classical quintuple of header fields is extracted. (Appendix B.1.3)
4. From the second graph, all header fields are extracted. (Appendix B.1.4)

A catch with this benchmark is that the verdict engine cannot provide an output of the parsed fields. Instead, the schema localises the appropriate field by putting it on the data field bus. The schemas mainly consist of shift instructions, which shift the appropriate fields onto the data field bus or shift through fields which can be skipped. Furthermore, it checks whether the data adheres to the parse graphs by checking the appropriate ”next header” fields in each header. Therefore, it can also invalidate any header stack which does not adhere to the parse graphs.

For benchmarking these schemas, we report the worst-case average throughput and latency. For this, an input is required, which takes the longest to complete. In the case of packet header parsing, this is a packet header with the most fields. For the first graph, the longest input we can provide is a header stack containing **Ethernet, 2x VLAN, 2x MPLS, IPv4**, and **TCP** headers. For the second graph, this is a header stack containing the **Ethernet, IPv4** and **TCP** headers. The verdict engine only needs a single blob to derive the worst-case timing, as the schema has a fixed timing for this input. The results are displayed in Table 5.3.

	1st graph 5 tuple	1st graph all	2nd graph 5 tuple	2nd graph all
Schema size (#instrs)	85	142	51	103
Clock cycle count	53	71	26	40
Instr count	42	61	19	33
Field count	9	25	17	38
Processed bits count	592	592	432	432
AXI4s packet count	3	3	2	2
Avg throughput (Mbps)	1116.98	833.80	1661.54	1080.00
Avg latency (ns)	176.67	236.67	130	200

Table 5.3: Benchmark results for packet header parsing

Other works achieve higher performance with packet parsing compared to the system. The work of Benacek et al.[5] achieves 100 Gbps throughput for extracting the classical 5-tuple in both graphs with a latency of around 46.1 ns. Similarly, the research of Santiago Da Silva et al.[7] achieves 100 Gbps throughput for both graphs but has a lower latency of around 25.6 ns. Finally, the work of Mashreghi-Moghadam et al.[8] achieved a throughput of over 1 Tbps for all 4 cases, with their latency staying under 15 ns.

The gap in performance of the verdict engine compared to these systems is expected, as the verdict engine uses a run-to-completion model and handles data fields sequentially. These systems gain higher performance through parallel extraction of fields, pipelining multiple packets and their hardware being custom-generated to only parse packets. Our system instead shows its strength in flexibility, as shown in Section 5.4.8. Being able to load in new schemas within a few seconds means that changing existing parse graphs or loading in new parse graphs is much quicker than resynthesizing a newly generated parser.

5.4.2. Benchmark: Packet Header Validation

The second benchmark has the verdict engine validate Internet packet headers for structure and content. A packet filtering system can subsequently use the verdict to filter the packet. A schema has been created for ICMP that only validates an ICMP ”Echo Request” or ”Echo Reply”. The use case would be to ping a host to test the reachability of that host within a network. The verdict engine does not count other ICMP types as valid to minimise the attack surface of ICMP attacks.

The schema checks that data consists of an Ethernet header, followed by a base IPv4 header and finally, ICMP. It checks for the correct EtherType and Protocol field, as well as checks whether the total length of the IPv4 packet does not exceed the maximum transmission unit. In the ICMP header, only the Echo Request and Echo Reply types are allowed. The schema can be found in Appendix B.2.

The benchmark data is a single blob containing a valid Ethernet-IPv4-ICMP packet header. The timing of the schema is fixed as the header formats do not specify variable length. For one or more correct blobs, the average throughput and latency stay the same. The benchmark results are displayed in Table 5.4.

	ICMP validation
Schema size (#instrs)	29
Clock cycle count	35
Instr count	29
Field count	10
Processed bits count	336
AXI4s packet count	2
Avg throughput (Mbps)	959.99
Avg latency (ns)	175.00

Table 5.4: Benchmark results for ICMP packet validation

The ICMP validation schema is a representative case of assessing an externally defined binary format. The schema checks for adherence to the expected order of packet headers and validates important fields to minimise the potential attack surface from improper packets. The average throughput of roughly 960 Mbps for the header shows that the system with the schema shows suitable speeds for gigabit networks that require real-time packet validation. The added benefit of the system’s flexibility allows for quick adjustments of the schema in dynamic networking conditions.

5.4.3. Benchmark: Checking the Structure of X.509 Certificates

The largest benchmark has the verdict engine check for the structure of ASN.1 DER encoded X.509 public key certificates. In the domain of secure networking, public key certificates are the basis for a public key infrastructure (PKI), which allows two parties to establish trust. Once this trust is established, it enables a secure exchange of information over a network. An example is browsing the web through HTTPS, where Transport Layer Security (TLS) uses these certificates to encrypt data. Another example would be a PKI for IoT devices to ensure trusted exchanges of data. For trust to be valid, the certificate needs to be correct. Incorrect certificates can be a way to exploit a system, introducing possible attacks around these certificates, such as [36][37][38]. To this end, a schema is made to check for the correct structure of these certificates.

The schema checks the structure of an ASN.1 DER encoded version 3 X.509 certificate according to the specification provided by the ITU [39]. It follows the public key certificate definition from the `AuthenticationFramework.asn` ASN.1 module. And checks whether the data is specifically DER-encoded, instead of other ASN.1 binary encoding rules. The schema can be found in Appendix B.3. It should be noted that no claims are made about the correctness of this schema, as a way to prove the correctness of a schema is currently outside the scope of the project. For now, the responsibility is for the programmer to check whether the schema checks the format against the appropriate standard.

The benchmark is done in the VHDL sim, as the timing of the schema is variable per certificate blob, and certificates can vary in size. To obtain a realistic performance evaluation of the schema, a large dataset was required, containing random, unique certificates. To this end, a public dataset has been found called the x509-cert-testcorpus [40]. This corpus consists of around 1.74 million X.509 certificates, which have been collected from TLS servers. This dataset was downloaded from GitHub and consists of many SQLite3 database files. These were merged using Python into one large database. However, this amount was too much to run through the VHDL sim. Hence, the size was reduced to the first 10000 entries of this merged database, making this our benchmark dataset. The results are shown in Table 5.5.

	X.509 structure checking
Schema size (#instrs)	227
Clock cycle count	69799183
Instr count	55452436
Field count	13736635
Processed bits count	127331256
AXI4s packet count	502251
Avg throughput (Mbps)	182.43
Avg latency (ns)	1389.73

Table 5.5: Benchmark results for X.509 structure checking

The results show a lower average throughput and latency relative to the other performed benchmarks. This is expected as the schema shows a fair amount of complexity compared to other benchmarked schemas. Translating the clock cycles from our benchmark to the time in seconds results in the 10000 certificates being checked within 0.7 seconds. Within the related research, there were no comparable hardware-based systems found that could parse and check X.509 certificates. One software-based solution from Ni et al. [41] has built an ASN.1 DER parser. Focusing on the correctness of parsing ASN.1 DER. They evaluated their system by parsing 10138 X.509 certificates, which their system could parse in 198 seconds. However, they do not state what hardware was used to run their parser, and they used a different dataset from ours. (We made attempts to obtain the same dataset, but they were unsuccessful.) Therefore, a comparison between their system and the verdict engine is not possible.

5.4.4. Benchmark: Weather Station Data Validation

To showcase support for other binary formats, the following 3 benchmarks use formats which were not specifically taken into account during the design phase, but are still partially/fully supported by the system. The first of these new format benchmarks has the system validate weather data blobs at a weather station. A use case was created where weather data blobs are created from IoT weather sensors and are encoded using MessagePack. These are then sent to a weather station that checks if the fields within the blobs are within the correct bounds. This is relevant for minimising exploitation through falsely encoded blob data, as well as valid data collection, as out-of-bound sensor values get detected and can therefore be discarded.

This schema assesses data fields containing information about the sensed data (e.g. temperature and humidity) and metadata concerning the IoT sensing device (e.g. device ID and battery level). The blob passes the schema if all datatypes are correct and values fall within the right ranges. The schema can be found in Appendix B.4.

The benchmark is done in the VHDL sim, as the schema has a variable timing. This is because MessagePack encodes fields with a variable type and length. For instance, a field containing a sensor value can have a range for which the smaller values can be encoded in an 8-bit field, but larger values require 16 bits or more to encode the data. As a variable number of instructions are executed to evaluate the data type and data length, the timing of the schema is different per blob. Therefore, a dataset of correct blobs is required to evaluate the schema's realistic performance. To this end, we created a data generator in Python which generates a dataset containing 10000 correct blobs and randomly generated values, resulting in various data types being used to encode such a value across different blobs. The benchmark results are displayed in Table 5.6.

	Weather data filter
Schema size (#instrs)	95
Clock cycle count	2525806
Instr count	2175806
Field count	355681
Processed bits count	2304976
AXI4s packet count	10000
Avg throughput (Mbps)	91.26
Avg latency (ns)	2525.81

Table 5.6: Benchmark results for weather data filtering

The performance of this schema is the lowest relative to the other benchmarked schemas. The complexity of the schema is reflected in the average latency, which reports the most clock cycles (and therefore instructions) per unit of data compared to other schemas. The schema has many value comparisons, branching, and subroutine calling relative to the amount of data that needs to be checked. This was mainly due to almost every field being checked for its correct range, as well as for determining which number encoding was used for each field. To better understand the timing characteristics when a schema scales in complexity, the schema timing growth is investigated in Section 5.4.7.

5.4.5. Benchmark: MPEG TS Video Stream Validation

This benchmark assesses MPEG Transport Stream (TS) packets. This is a binary format used in the storage and transmission of audio and video data. The packet size is 188 bytes and consists of a header and payload data. Validating these packets has, for instance, the use case of filtering a video stream.

The schema checks for several flags and fields to check whether packets are not corrupted and belong to a specific packet ID to invalidate other streams. The payload for this stream consists of a Packetized Elementary Stream (PES) packet, of which the header is also checked. The schema ends by flushing the payload data through the system. Furthermore, the schema reduces its instruction count by checking multiple fields in one comparison. This is possible in the case that multiple consecutive fields need one value to be exactly matched on. Therefore, we can construct a constant which concatenates the constants required for the two individual fields. This results in only one comparison with one constant, saving on instruction execution. The schema can be found in Appendix B.5.

As the schema has a fixed timing, the provided data to the benchmark is a single blob containing the MPEG TS header, PES header, and payload. The benchmark results are displayed in Table 5.7.

The average throughput and latency are high for this schema and can be attributed to the payload data being flushed through the system. Additionally, the complexity of this schema is not high due to the headers not containing many fields. Furthermore, the schema is less complex as some parts of the format could not be checked, as state is required between blobs. This format has, for example, a ”continuity counter”, which increments per blob. Moreover, the format has a ”Program Association Table” payload when the packet ID is 0. This table lists available programs with their linked packet IDs. If those IDs can be saved, validation of packets can be state-specific, which is useful for this format.

	Video stream filter
Schema size (#instrs)	17
Clock cycle count	26
Instr count	18
Field count	6
Processed bits count	1504
AXI4s packet count	6
Avg throughput (Mbps)	5784.62
Avg latency (ns)	43.33

Table 5.7: Benchmark results for video stream filtering

Considering the current functionality of the system, a format can only be checked for its fields which do not require state saving between blobs. But it is shown that an extension to the verdict engine is required for properly checking stateful binary formats.

5.4.6. Benchmark: XDR Drone Command Validation

The final schema benchmark has the system validate a custom binary command message for a drone, encoded in the External Data Representation (XDR) format. This has its use case in operational drones, which require real-time commands for information on where to fly. Validation for these commands is important as they are mission-critical. If a command is invalid, it can lead to drones flying to the wrong target location, or worse, potentially crashing.

The schema checks whether the command fields, such as priority level, target latitude, target altitude, etc, have the right values or fall within the correct ranges. Various fields in the blob are encoded as a fixed-point number. Which are checked accordingly to the constants by offsetting them. The schema can be found in Appendix B.6.

The data provided to the benchmark is a single correct command blob. As the command format is externally described and does not have a variable length, the schema also has a fixed timing. Therefore, the performance characteristics will be the same for all blobs. The results are shown in Table 5.8.

	Drone command filter
Schema size (#instrs)	49
Clock cycle count	57
Instr count	50
Field count	16
Processed bits count	928
AXI4s packet count	4
Avg throughput (Mbps)	1628.07
Avg latency (ns)	142.5

Table 5.8: Benchmark results for drone command filtering

This benchmark of this schema shows how the system can check command messages in real-time. With an average throughput of around 1630 Mbps, the system can keep a consistent and adequate throughput for processing valid messages. For communication channels in use today in drones such as TCDL [42], with data rates around 1 to 10 Mbps, the system will not be a bottleneck regarding throughput for checking mission-critical command data, and it helps minimise any attacks related to tampering with these messages.

5.4.7. Schema Timing Growth

From the benchmarks, it has been shown that schemas that contain many comparisons with constants per unit of data take a hit in performance. This is reflected by the latency metric, which becomes higher once the clock cycles increase due to comparisons, whilst the amount of data (expressed in AXI4s packets) stays the same. A synthetic benchmark was created to profile this performance characteristic.

The benchmark consists of 4 schemas. In the first schema a dummy field of a byte is put on the data field bus, after which a comparison is made with a constant. For the second, third and fourth schemas, this same comparison is made 10, 50 and 100 times respectively. Having the same comparison being done multiple times on the same field provides insight into the impact on performance and how that impact varies per schema.

Table 5.9 shows the results for this benchmark.

	1 constant	10 constants	50 constants	100 constants
Clock cycle count	8	26	106	206
Instr count	5	23	103	203
Field count	1	1	1	1
Processed bits count	8	8	8	8
AXI4s packet count	1	1	1	1
Avg throughput (Mbps)	100	30.77	7.55	3.88
Avg latency (ns)	80	260	1060	2060

Table 5.9: Benchmark results for schema growth

This benchmark confirms, based on the clock cycle count and latency, that there is a linear increase in the time taken to perform more comparisons on the data. For the average throughput, this becomes a reciprocal function based on the clock cycles. The more clock cycles used per bit, the further the throughput converges to 0.

Larger constants also influence the slope of the linear time increase. Currently, a comparison is made up of 2 instructions (`ldim` + `cmp`), but bigger constants increase the number of `ldim` instructions for loading the constant, increasing the instruction count per comparison.

The properties of this behaviour lead to what can be implemented for future improvement. Optimisations lie in eliminating the linearity in time caused by comparison operations being performed sequentially. Or reducing the slope of the linear increase per comparison by reducing the amount of instructions required for a comparison.

5.4.8. Schema Flexibility

To see how adaptable the system is to changing input data conditions, we benchmarked the flexibility of changing schemas during the runtime of the system. We measured the time required to program the system during its runtime.

For loading each schema from the previous sections into memory, we performed two measurements. The first measurement is done in the VHDL simulator, where the clock cycles are counted from the first AXI4l transaction with the instruction memory until the `schema_loaded` flag in the control register is set. This provides the maximum speed at which each schema can be loaded into the system. The second measurement uses the FPGA and the serial connection, where a schema program is serially written to the FPGA via a serial bus. The Python program that runs the AXI4l master then measures the time taken to write the schema program to the FPGA. The second measurement is not as strict, as it can contain more overhead from the measurement process, the OS, the serial connection, etc. However, it does provide a realistic environment in which schemas can be loaded into the system.

Table 5.10 shows the benchmark results, which are ordered from smallest schema to largest schema.

The benchmark results show a linear increase in both measurements as the schema size increases. This aligns with the sequential writing to the instruction memory, where every instruction contributes to a longer writing time. The smaller schemas, such as the MPEG TS and ICMP validation, take a longer time to write per instruction. This can be attributed to overhead. In the first measurement, this overhead can be attributed to setting the `schema_loaded` flag high. In the second case, the overhead is more random, as is apparent from the cases where a bigger schema takes more time per instruction write compared to the previous smaller schema.

	MPEG TS	ICMP Validation	XDR Drone	Parsing 5tup simple	Parsing 5tup full	MsgPack weather	Parsing all simple	Parsing all full	x509 cert
Instructions	17	29	49	51	85	95	103	142	227
Clock cycles (VHDL sim)	57	93	153	159	261	291	315	432	687
Clock cycles per instr	3.35	3.21	3.12	3.12	3.07	3.06	3.06	3.04	3.03
Milliseconds (serial - FPGA)	61	97.75	149.33	154.85	263.08	272.73	296.07	405.02	655.47
Milliseconds per instr	3.59	3.37	3.05	3.04	3.10	2.87	2.87	2.85	2.89

Table 5.10: Benchmark results for schema flexibility

These results highlight the flexibility of the verdict engine, as these schema loads can be performed during runtime without having to change the underlying hardware, with a low downtime during writing. In contrast, other related hardware-based parsing/filtering systems require resynthesis to integrate schema changes, or are limited in what changes can be made in the format during runtime. This makes the verdict engine suitable for different use cases with dynamic workloads.

5.5. Results Discussion

The benchmarking results showcase the performance characteristics across various schemas and, importantly, showcase the flexibility of both schema diversity and schema loading of the verdict engine.

The system supports a broader range of binary formats beyond the initial design scope of validating the base IPv4 header and ASN.1 DER encoded data. This is shown through the additional benchmarks of ICMP packet validation, MessagePack weather station data validation, MPEG TS video stream validation, and XDR drone command validation.

Across the benchmarked schemas, the system can successfully check for adherence to the data, emphasising the verdict engine's ability to support various use cases. The flexibility comes at the cost of lower performance compared to fully custom FPGA designs. This trade-off is an inherent result of the VM architecture and a run-to-completion model, but allows us to prioritise implementation of diverse schemas, complex assertions, and runtime schema updates, instead of aiming for high performance with a pipelined model.

The timing characteristics of schemas were analysed using a synthetic benchmark. The results confirmed that an increase in the number of comparisons on the same data linearly influences the increase in execution time and average latency, and has the average throughput decrease reciprocally. This is also seen in the more complex schemas, such as the weather data validation case, where there is a higher latency on average due to a large number of field assertions and branching per data field. We reason that with optimisation of these aspects, through parallel comparisons or instruction efficiency improvements, performance can be increased.

Finally, the verdict engine is flexible in how quickly it can change from checking one format to another. The benchmark results show that new schemas can be loaded into the system with minimal downtime. Compared to other related hardware-based systems, which usually require a lengthy resynthesis process to update to a new format, the verdict engine does not have to change the underlying hardware at all. This ability benefits various use cases with dynamic scenarios that often experience changing conditions and require minimal downtime, such as network security, streaming and mission-critical systems.

The benchmarking results highlight the flexibility of the verdict engine and report adequate performance characteristics for various use cases, establishing its use as a flexible validation system for diverse and dynamic use cases.

5.6. Conclusion

In this chapter, we evaluated the implemented design and benchmarked the system for its performance and flexibility. To be able to test and benchmark the verdict engine on an FPGA, a hardware setup was determined. For the FPGA testbed, a packet relay module was implemented to be able to send data to the verdict engine at maximum throughput over a serial connection with the test computer. Besides the FPGA setup, a VHDL simulation setup was explained for benchmarks with larger datasets, where BFMs send multiple packets in succession and provide them at the maximum throughput of the system. Afterwards, the synthesis results were discussed for timing, resource usage, and energy consumption. The design's obtained clock frequency of 100 MHz was used to convert clock cycles to time during the benchmarks. Subsequently, the measurement window was explained, in which the system counts various metrics during a benchmark. Finally, the system was evaluated through several benchmarks, consisting of: Internet packet parsing, Internet header validation, X.509 certificate validation, weather station MessagePack data validation, MPEG TS video stream validation, and XDR drone command validation. This showed that the system is compatible with binary formats that go beyond the initial formats for which the system was designed. This showcased flexibility in the various formats that the system can assess. The measured performance varied per use case due to the varying number of required checks in the data and the complexity of branching. To profile this, a synthetic benchmark was run, measuring the timing effects of the number of comparisons on the same data. This led to a linear increase in clock cycles and a reciprocal decrease in the average throughput when the number of comparisons increased. Finally, the timing regarding schema switching was benchmarked. It was shown that downtimes of the system were minimal. This, together with not having to manage a lengthy and complex synthesis process, made the system a much more flexible option compared to other systems. This evaluation showed that the binary verdict engine is a flexible validation system that can be used in the validation of diverse and dynamic use cases.

6 Conclusion

6.1. Summary

In Chapter 2, we discussed the required background for the thesis project. We set the project scope on binary formats. A binary format classification was created to describe the binary formats that needed to be supported, where we identified that formats could be externally described or self-describing (or a hybrid), and flat or hierarchical. We described how binary formats hold data fields with various data types and how these types can be checked, such as exact matching on strings and comparison on integers. Afterwards, the binary parsing process was explained, specifically the advantages and disadvantages of single-pass binary parsing. We discussed how binary parsing relies on assertions to check for the format structure and content, and why a DSL was required to provide parsing configurations. Finally, for the general background, we discussed how FPGAs provided advantages over CPUs related to better data streaming, bit-aligned operations, and purpose-built hardware. We also briefly touched upon why FPGAs were preferred over ASICs for this project. Afterwards, the related work was discussed and provided an exploration of various categories of configurable hardware-based packet parsing systems. The first category, generated HDL parsers and filters, generated hardware based on a parsing/processing description. They provide high performance for field extraction but are inflexible to change. The second category, templated generic parsers, is more flexible as it supports changes in the format description during runtime through generic hardware stages, but has challenges related to supporting assertions and data hierarchy. The third category of systems, match-action systems, are hardware-based systems with wide functionality for executing network functions. It also remained a challenge for this category to support assertions and data hierarchy. The final category, hardware-based VMs, are programmable systems which are very flexible, at the cost of high-performance parsing. Lastly, we considered what concepts several packet parsing DSLs had in common in terms of notation. The showcase of related research provided the knowledge to make a design direction for the system.

In Chapter 3, we explained our design for this project. Before discussing the design, we provided the requirements of the system, as derived from the background study. These requirements were categorised into either data assessment requirements or architectural requirements. As these required much flexibility from an underlying architecture, the VM architecture was chosen as it allowed for format variability, runtime programmability, and future extension to system functionality. Furthermore, we touched on how we considered security in the design process. From the data assessment requirements, we designed a schema language which can: declare fields in the expected order of arrival; declare assertions on fields; assert the length of a section; declare fields with a variable width; declare conditional statements to change program flow; call other schemas; declare repeat loops; declare repeat while loops; and declare the end of a schema. Afterwards, the ISA was explained per instruction, and a pseudo notation was explained to denote the instructions in a human-readable format. We linked the schema language to the ISA by explaining the conversion from schema language constructs to pseudo notation. Finally, we discussed the module design. The modules execute the instructions of the system. First, the input and output systems were discussed in how they convert the input stream to data fields with a specified width, and how the data fields are converted back to an unmodified output stream. Second, the controller was explained for its function per instruction. Third, the instruction fetcher provides instructions to the system and maintains the instruction pipeline when updating the Program Counter or shifting. Fourth, the stack can push and pop stack entries used for schema calling, length assertions, repeat loops, and while loops. Lastly, the full design was provided, showing the interfaces between the modules and external interfaces.

In Chapter 4, we discussed the implemented module design in VHDL and the Python emulator and assembler programs. An overview was provided first on a workflow with the implemented systems for developing schemas. The assembler and emulator programs are used to debug and test schemas before they are deployed onto the VHDL implementation. Second, the working of the Python assembler was explained, where a schema in pseudo notation is first preprocessed to an intermediate representation, such that there is a one-to-one conversion to the final binary instruction. Third, the differences between the functional emulator of the verdict engine and the module design were discussed, and its general operation was explained. Finally, the VHDL implementation of the verdict engine was discussed. We first explained the AXI4-Stream protocol and how we used it in the input and output system of the verdict engine. We briefly touched upon the working of the AXI4-Lite protocol and where it was used in our system. Afterwards, the implementation of the modules was explained. First, the implementation of the input system was explained. Its interfaces had been made concrete as AXI4s buses, and the internal processing from AXI4s input packets to AXI4s data field packets during both shifting and flushing was explained. Second, the output system was explained for how it converts the AXI4s data field packets to the AXI4s output packets. Third, the controller was explained on how it implements the required functionality per instruction. Fourth, the instruction fetcher and its handling of the instruction pipeline were explained. The instruction fetcher needs to manage a two-stage pipeline and needs to be able to handle 1-cycle hazards during various instructions. An important catch with this pipeline was highlighted, which was that input data must always be available upon request. Fifth, the stack implementation was explained, and attention was given to how the next incremented `entry_value` was set. Lastly, the full design was shown, with the updated interfaces for the input and output systems and updated signals for the instruction pipeline.

In Chapter 5, we evaluated the implemented design and benchmarked the system for its performance and flexibility. To be able to test and benchmark the verdict engine on an FPGA, a hardware setup was determined. For the FPGA testbed, a packet relay module was implemented to be able to send data to the verdict engine at maximum throughput over a serial connection with the test computer. Besides the FPGA setup, a VHDL simulation setup was explained for benchmarks with larger datasets, where BFMs send multiple packets in succession and provide them at the maximum throughput of the system. Afterwards, the synthesis results were discussed for timing, resource usage, and energy consumption. The design's obtained clock frequency of 100 MHz was used to convert clock cycles to time during the benchmarks. Subsequently, the measurement window was explained, in which the system counts various metrics during a benchmark. Finally, the system was evaluated through several benchmarks, consisting of: Internet packet parsing, Internet header validation, X.509 certificate validation, weather station MessagePack data validation, MPEG TS video stream validation, and XDR drone command validation. This showed that the system is compatible with binary formats that go beyond the initial formats for which the system was designed. This showcased flexibility in the various formats that the system can assess. The measured performance varied per use case due to the varying number of required checks in the data and the complexity of branching. To profile this, a synthetic benchmark was run, measuring the timing effects of the number of comparisons on the same data. This led to a linear increase in clock cycles and a reciprocal decrease in the average throughput when the number of comparisons increased. Finally, the timing regarding schema switching was benchmarked. It was shown that downtimes of the system were minimal. This, together with not having to manage a lengthy and complex synthesis process, made the system a much more flexible option compared to other systems. This evaluation showed that the binary verdict engine is a flexible validation system that can be used in the validation of diverse and dynamic use cases.

6.2. Contributions

This section answers the research questions and lists the contributions of this work.

6.2.1. Answering the Research Questions

The questions from Section 1.1 can now be answered. Starting with the research question:

> *How can we build a reconfigure-once, FPGA-based verdict system to assess structured binary data across different formats?*

We built the binary verdict engine as a hardware-based VM, using schema programs (i.e. instructions) to traverse the binary data stream per data field, assess data fields and binary sections, and change control flows to other instruction routines. We assess the data through assertions, of which we have created two variants in the system: field assertions that check the data field with a constant, and length assertions that check if a section of the data is of the correct length. We defined a schema DSL with which we can create schema programs to specify how the data adheres to a specific binary format. Schema programs can be interchanged by uploading a new program to the instruction memory. We achieved a reconfigure-once design with the binary verdict engine, as the system can be reprogrammed to support assessment of various binary formats without having to alter the underlying hardware design. We implemented the design on an FPGA, and we evaluated the implementation for its flexibility and performance. We showed the flexibility of the design in various use cases containing different formats and in applying schema changes, and evaluated the performance in use cases and schema timing characteristics in synthetic benchmarks.

The first additional question:

> *What hardware architecture should be chosen for this verdict system?*

We identified multiple architectural categories from researching the related work. We chose a Virtual Machine architecture for the verdict system for its flexibility, as it supports wide variability in the formats that it can parse. Because of its inherent programmability, it is also flexible in its ability to quickly change schemas during the system's runtime. Furthermore, the VM architecture is flexible as it can be extended in functionality through the implementation of more instructions. We found that other architectures take a more parallelised approach, but lack flexibility as they can only support a maximum size per blob, are limited to a fixed amount of operations as the data goes through the system, and are limited in the formats that they can support.

The second additional question:

> *How can we describe binary data such that the verdict system can check adherence to this description?*

We discussed various DSLs from the related work that demonstrated similar methods of a binary format description. We used this as a basis to create the schema language to describe data encoded in a binary format per data field. We can give data fields a width in bytes or bits, and they can be interpreted as an unsigned or signed integer. We can declare one or more assertions per data field, or we can check sections of data fields for their combined length using length assertions. We can determine the control flow of the program using conditional statements and calling functions. We showed conversions from the schema language to instructions to show how a future compiler can compile the schema language to the program binary that the verdict engine can execute.

6.2.2. Main Contributions

The main contributions this work has provided are:

1. Compiled a list of requirements for the design of a verdict system for binary data.
2. Designed a schema language for describing and assessing binary formats.
3. Provided a design for the binary verdict engine to assess binary data against a schema.
4. Implemented an FPGA-based binary verdict engine on an FPGA.
5. Implemented a Python Emulator of the binary verdict Engine.
6. Evaluated the binary verdict engine for its flexibility and performance characteristics.

6.2.3. Other Contributions

Other contributions made in this project are:

1. Created an assembly-like pseudo notation with which schemas can be written in a human-readable format.
2. Implemented a Python Assembler that converts Schemas in pseudo instruction notation to binary instructions.
3. Presented and demonstrated the design within Technolution and one of its clients, which was met with positive reception.

6.3. Future work

This final section discusses various proposals for future work on the system.

6.3.1. Schema Language Compiler

The first proposal is the implementation of a compiler for the schema language. This compiler provides a higher-level abstraction to the system user, as schemas can now be implemented using the schema language. Compared to the pseudo notation, the schema language looks like a more conventional programming language and shows a clearer structure of what the data is expected to look like. Furthermore, the compiler compiles fields and other constructs to multiple instructions, so there is less room for human error as instructions do not need to be written manually. Finally, a compiler can create an abstraction for comparisons bigger than the maximum data field size into smaller comparisons. For example, the following code block showcases a string field with an assertion which goes beyond the 256-bit/32-byte maximum data field size, but is two string fields under the hood.

```
// This big string of 45 bytes
byte<45> unsigned MyBigString [== "The quick brown fox jumped over the lazy dog."]
// is an abstraction for:
byte<32> unsigned MySmallString1 [== "The quick brown fox jumped over "]
byte<13> unsigned MySmallString2 [== "the lazy dog."]
```

6.3.2. New Data Type Support

To broaden the support for binary format assessment, other data types can be supported for comparison operations, such as IEEE floats, or date and time formats. For these data types, it is expected that they require new hardware with a new or adjusted comparison instruction to support these comparisons. Furthermore, as the system now only supports formats that encode their data in big-endian order, we can also extend support to formats which use a little-endian encoding for their data fields. Data type support can also be implemented more generically, though the implementation of an ”offload” comparison instruction. This can send the current data field to another external module, which performs the comparison, after which it returns an assertion result which can be processed by the verdict engine.

6.3.3. Smaller System Architectures Evaluation

The verdict engine is designed to be modular regarding its input and output stream data widths and data field bus width. However, the current prototype implementation has 256-bit-wide buses. This affects the rest of the system, as it must account for this width when comparing 256-bit wide data fields to constants, storing these wide data fields to the stack, or incrementing them on the stack. This increases the hardware resources required for the system. Depending on the use case, the assessed binary formats may not contain 256-bit data fields and can be adequately assessed using, for instance, a smaller 64-bit wide verdict engine. Implementing a smaller system will require less hardware and memory, resulting in different timing characteristics, resource usage, and energy consumption.

It is expected that systems with a smaller data field bus width can achieve higher clock speeds, allowing them to assess binary formats more quickly. Evaluating the performance of these systems across various bus widths can provide us with a profile of the verdict engine's performance relative to the bus width. If possible, we can recommend an optimal configuration of the verdict engine which balances performance, area usage and bus width.

References

[1] *Rfc 791 - internet protocol*, [Online; accessed 2025-05-04]. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc791.

[2] *X.690: Information technology - asn.1 encoding rules: Specification of basic encoding rules (ber), canonical encoding rules (cer) and distinguished encoding rules (der)*, [Online; accessed 2025-05-04]. [Online]. Available: https://www.itu.int/rec/T-REC-X.690-202102-I/en.

[3] J. C. Viotti and M. Kinderkhedia, "A survey of json-compatible binary serialization specifications," no. arXiv:2201.02089, Jan. 2022, arXiv:2201.02089 [cs]. DOI: 10.48550/arXiv.2201.02089. [Online]. Available: http://arxiv.org/abs/2201.02089.

[4] E. Maltsev, O. Muliarevych, and A. Razzaque, "Classifying serialization formats for inter-service communication in distributed systems," 2024. [Online]. Available: https://science.lpnu.ua/sites/default/files/journal-paper/2024/dec/37009/vsedoi-87-92_0.pdf.

[5] P. Benácek, V. Pu, and H. Kubátová, "P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 148–155. DOI: 10.1109/FCCM.2016.46. (visited on 07/15/2024).

[6] J. Cabal, P. Benáček, L. Kekely, M. Kekely, V. Puš, and J. Kořenek, "Configurable FPGA Packet Parser for Terabit Networks with Guaranteed Wire-Speed Throughput," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Monterey CALIFORNIA USA: ACM, Feb. 2018, pp. 249–258, ISBN: 978-1-4503-5614-5. DOI: 10.1145/3174243.3174250. (visited on 07/15/2024).

[7] J. Santiago da Silva, F.-R. Boyer, and J. P. Langlois, "P4-Compatible High-Level Synthesis of Low Latency 100 Gb/s Streaming Packet Parsers in FPGAs," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18, New York, NY, USA: Association for Computing Machinery, Feb. 2018, pp. 147–152, ISBN: 978-1-4503-5614-5. DOI: 10.1145/3174243.3174270. (visited on 07/15/2024).

[8] P. Mashreghi-Moghadam, T. Ould-Bachir, and Y. Savaria, "A Templated VHDL Architecture for Terabit/s P4-programmable FPGA-based Packet Parsing," in *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2022, pp. 672–676. DOI: 10.1109/ISCAS48785.2022.9937607. (visited on 06/06/2024).

[9] B. Li, K. Tan, L. (Luo, *et al.*, "ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, New York, NY, USA: Association for Computing Machinery, Aug. 2016, pp. 1–14, ISBN: 978-1-4503-4193-6. DOI: 10.1145/2934872.2934897. (visited on 07/18/2024).

[10] H. Wang, R. Soulé, H. T. Dang, *et al.*, "P4FPGA: A Rapid Prototyping Framework for P4," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17, New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 122–135, ISBN: 978-1-4503-4947-5. DOI: 10.1145/3050220.3050234. (visited on 07/15/2024).

[11] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4->NetFPGA Workflow for Line-Rate Packet Processing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19, New York, NY, USA: Association for Computing Machinery, Feb. 2019, pp. 1–9, ISBN: 978-1-4503-6137-8. DOI: 10.1145/3289602.3293924. (visited on 06/25/2024).

[12] A. Yazdinejad, R. M. Parizi, A. Bohlooli, A. Dehghantanha, and K.-K. R. Choo, "A high-performance framework for a network programmable packet processor using P4 and FPGA," *Journal of Network and Computer Applications*, vol. 156, p. 102 564, Apr. 2020, ISSN: 1084-8045. DOI: 10.1016/j.jnca.2020.102564. (visited on 06/03/2024).

[13] A. Fiessler, S. Hager, B. Scheuermann, and A. W. Moore, “HyPaFilter: A Versatile Hybrid FPGA Packet Filter,” in *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, Santa Clara California USA: ACM, Mar. 2016, pp. 25–36, ISBN: 978-1-4503-4183-7. DOI: 10.1145/2881025.2881033. (visited on 07/16/2024).

[14] M. B. Anwer, M. Motiwala, M. bin Tariq, and N. Feamster, “SwitchBlade: A platform for rapid deployment of network protocols on programmable hardware,” *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 183–194, Aug. 2010, ISSN: 0146-4833. DOI: 10.1145/1851275.1851206. (visited on 05/30/2024).

[15] V. Puš, L. Kekely, and J. Kořenek, “Design methodology of configurable high performance packet parser for FPGA,” in *17th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, Apr. 2014, pp. 189–194. DOI: 10.1109/DDECS.2014.6868788. (visited on 07/01/2024).

[16] M. Attig and G. Brebner, “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*, ser. ANCS ’11, USA: IEEE Computer Society, Oct. 2011, pp. 12–23, ISBN: 978-0-7695-4521-9. DOI: 10.1109/ANCS.2011.12. (visited on 05/23/2024).

[17] M. Lixin, L. Qingrang, and W. Xin, “Software-Defined Protocol Independent Parser based on FPGA,” in *Proceedings of the International Conference on Industrial Control Network and System Engineering Research*, ser. ICNSER2019, New York, NY, USA: Association for Computing Machinery, Mar. 2019, pp. 42–46, ISBN: 978-1-4503-6627-4. DOI: 10.1145/3333581.3333591. (visited on 05/23/2024).

[18] Y. Sun and Z. Guo, “The Design of a Dynamic Configurable Packet Parser Based on FPGA,” *Micromachines*, vol. 14, no. 8, p. 1560, Aug. 2023, ISSN: 2072-666X. DOI: 10.3390/mi14081560. (visited on 05/29/2024).

[19] P. Bosshart, D. Daly, G. Gibb, *et al.*, “P4: Programming protocol-independent packet processors,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014, ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. (visited on 07/15/2024).

[20] P. Bosshart, G. Gibb, H.-S. Kim, *et al.*, “Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN,” in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM ’13, New York, NY, USA: Association for Computing Machinery, Aug. 2013, pp. 99–110, ISBN: 978-1-4503-2056-6. DOI: 10.1145/2486001.2486011. (visited on 07/15/2024).

[21] T. Luinaud, T. Stimpfling, J. S. da Silva, Y. Savaria, and J. P. Langlois, “Bridging the Gap: FPGAs as Programmable Switches,” in *2020 IEEE 21st International Conference on High Performance Switching and Routing (HPSR)*, May 2020, pp. 1–7. DOI: 10.1109/HPSR48589.2020.9098978. (visited on 08/01/2024).

[22] J. Santiago da Silva, T. Stimpfling, T. Luinaud, B. Fradj, and B. Boughzala, “One for All, All for One: A Heterogeneous Data Plane for Flexible P4 Processing,” in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, Sep. 2018, pp. 440–441. DOI: 10.1109/ICNP.2018.00063. (visited on 07/01/2024).

[23] S. Pontarelli, R. Bifulco, M. Bonola, *et al.*, “{FlowBlaze}: Stateful Packet Processing in Hardware,” in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 531–548, ISBN: 978-1-931971-49-2. (visited on 07/16/2024).

[24] H. Zolfaghari, D. Rossi, and J. Nurmi, “A custom processor for protocol-independent packet parsing,” *Microprocessors and Microsystems*, vol. 72, p. 102 910, Feb. 2020, ISSN: 0141-9331. DOI: 10.1016/j.micpro.2019.102910. (visited on 06/03/2024).

[25] M. S. Brunella, G. Belocchi, M. Bonola, *et al.*, “hXDP: Efficient software packet processing on FPGA NICs,” *Commun. ACM*, vol. 65, no. 8, pp. 92–100, Jul. 2022, ISSN: 0001-0782. DOI: 10.1145/3543668. (visited on 07/18/2024).

[26] “Xdp.” en-US. (), [Online]. Available: https://www.iovisor.org/technology/xdp.

[27] “Ebpf - introduction, tutorials & community resources.” en. (), [Online]. Available: https://ebpf.io.

[28] R. D. G. Pacífico, L. F. S. Duarte, L. F. M. Vieira, B. Raghavan, J. A. M. Nacif, and M. A. M. Vieira, "eBPFlow: A Hardware/Software Platform to Seamlessly Offload Network Functions Leveraging eBPF," *IEEE/ACM Transactions on Networking*, vol. 32, no. 2, pp. 1319–1332, Apr. 2024, ISSN: 1558-2566. DOI: 10.1109/TNET.2023.3318251. (visited on 07/18/2024).

[29] H. Zolfaghari, H. Mustafa, and J. Nurmi, "Run-to-Completion versus Pipelined: The Case of 100 Gbps Packet Parsing," in *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*, Jun. 2021, pp. 1–6. DOI: 10.1109/HPSR52026.2021.9481797. (visited on 07/18/2024).

[30] Arm, *Amba axi-stream protocol specification*, [Online; accessed 2025-04-22], Jun. 2023. [Online]. Available: https://developer.arm.com/documentation/ihi0051/latest/.

[31] Arm, *Amba axi and ace protocol specification version h.c*, [Online; accessed 2025-04-22], Jan. 2021. [Online]. Available: https://developer.arm.com/documentation/ihi0022/hc/?lang=en.

[32] *Axi4-stream interface • soft-decision fec integrated block logicore ip product guide (pg256) • reader • amd technical information portal*, [Online; accessed 2025-04-23]. [Online]. Available: https://docs.amd.com/r/en-US/pg256-sdfec-integrated-block/AXI4-Stream-Interface.

[33] "Riviera-pro - advanced verification platform." [Online; accessed 2025-04-02]. (), [Online]. Available: https://www.aldec.com/en/products/functional_verification/riviera-pro.

[34] *Cocotb 1.9.2 documentation*, [Online; accessed 2025-04-02], 2025. [Online]. Available: https://docs.cocotb.org/en/stable/.

[35] *Polarfire family fabric user guide*, [Online; accessed 2025-03-28], 2023. [Online]. Available: https://ww1.microchip.com/downloads/aemDocuments/documents/FPGA/ProductDocuments/UserGuides/PolarFire_FPGA_PolarFire_SoC_FPGA_Fabric_UG_VD.pdf.

[36] *Nvd - cve-2023-30588*, [Online; accessed 2025-05-04]. [Online]. Available: https://nvd.nist.gov/vuln/detail/cve-2023-30588.

[37] *Nvd - cve-2022-3602*, [Online; accessed 2025-05-04]. [Online]. Available: https://nvd.nist.gov/vuln/detail/cve-2022-3602.

[38] *Nvd - cve-2022-3786*, [Online; accessed 2025-05-04]. [Online]. Available: https://nvd.nist.gov/vuln/detail/cve-2022-3786.

[39] *X.509: Information technology - open systems interconnection - the directory: Public-key and attribute certificate frameworks*, [Online; accessed 2025-05-04]. [Online]. Available: https://www.itu.int/rec/T-REC-X.509-201910-I/en.

[40] johndoe31415, *Github - johndoe31415/x509-cert-testcorpus: X.509 certificate test corpus that was scraped from public tls servers*, [Online; accessed 2025-04-07]. [Online]. Available: https://github.com/johndoe31415/x509-cert-testcorpus.

[41] H. Ni, A. Delignat-Lavaud, C. Fournet, T. Ramananandro, and N. Swamy, "ASN1*: Provably Correct, Non-malleable Parsing for ASN.1 DER," in *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2023, New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 275–289, ISBN: 9798400700262. DOI: 10.1145/3573105.3575684. (visited on 05/30/2024).

[42] C. to Wikimedia projects, *Common data link - wikipedia*, [Online; accessed 2025-04-28], Jun. 2007. [Online]. Available: https://en.wikipedia.org/wiki/Common_Data_Link.

A
Module Activation per Instruction

This appendix describes which modules from the module design in Section 3.5 get activated per instruction from the ISA in Section 3.3. Each figure shows the module activation of an instruction. Green modules or buses (arrows) represent module activation or usage of the bus. Yellow modules or buses represent optional module activation or optional bus usage.

Shift Instruction

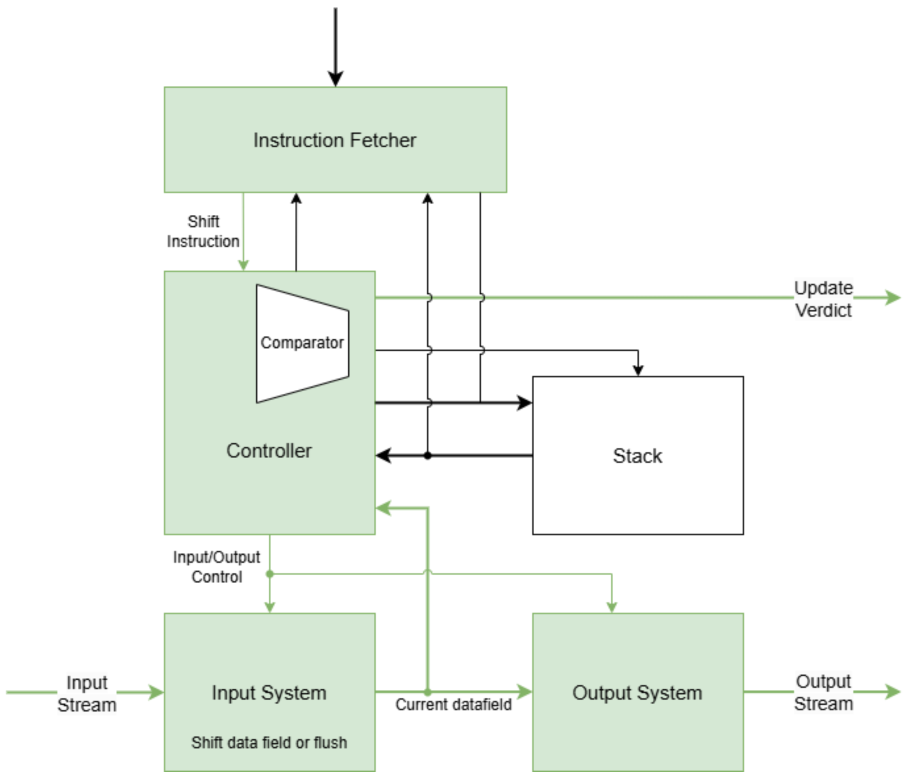

Figure A.1: Module activation during the Shift instruction.

End Instruction

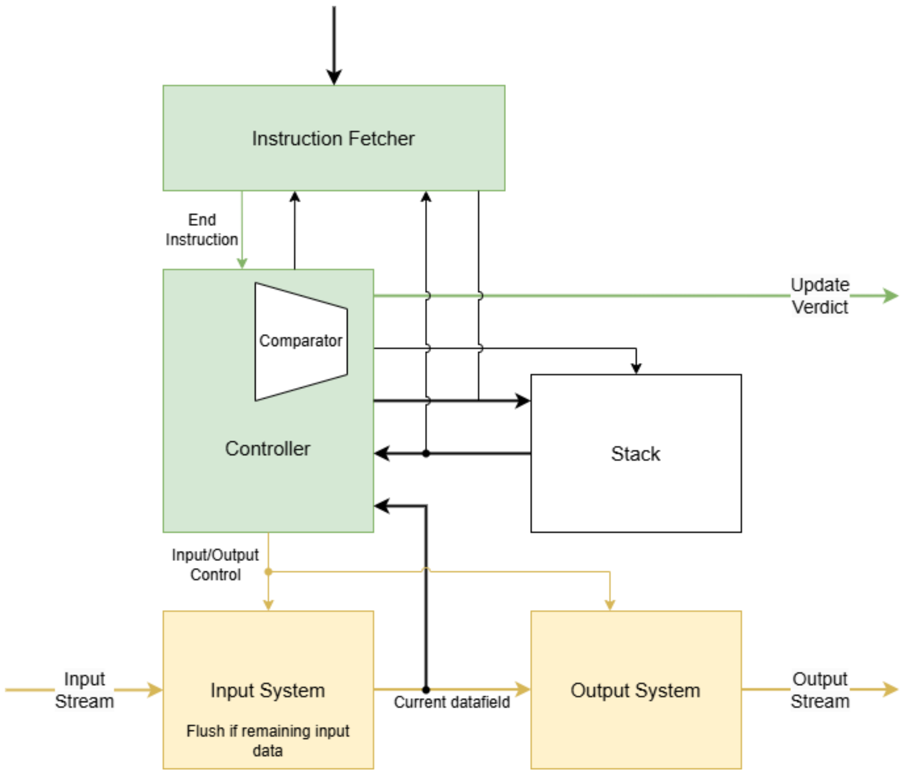

Figure A.2: Module activation during the End instruction.

Load Immediate Instruction

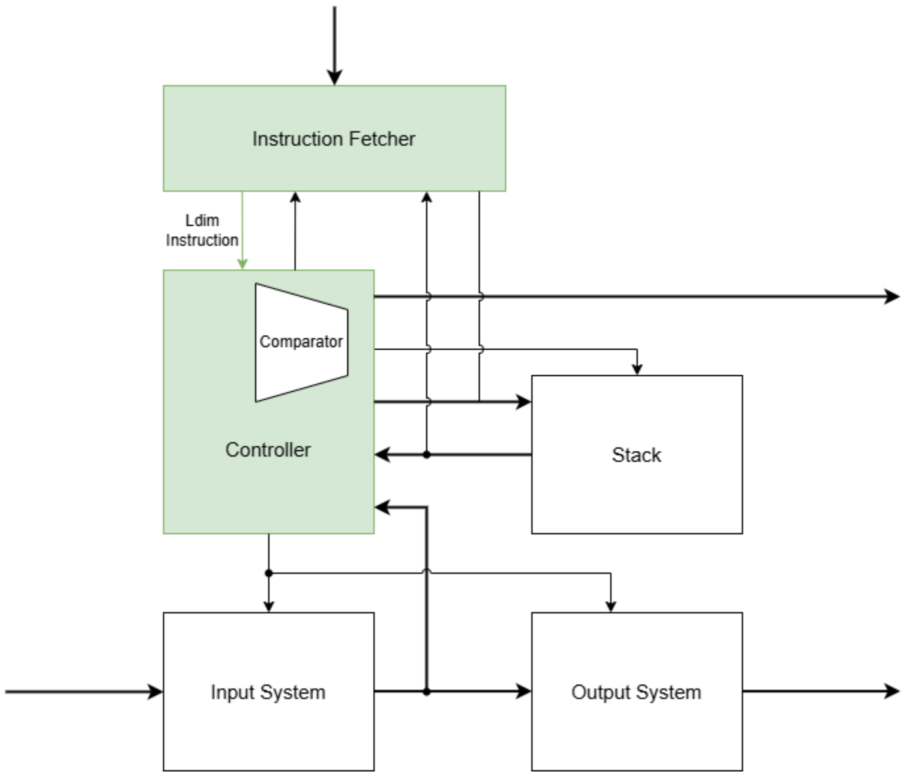

Figure A.3: Module Activation during the Load Immediate instruction.

Compare Instruction

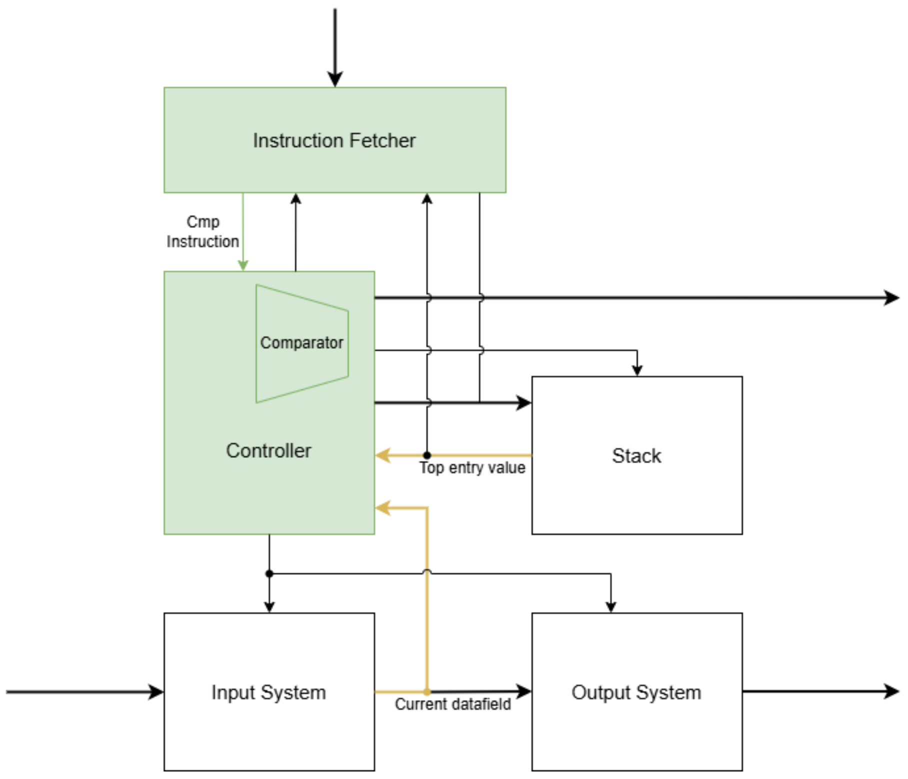

Figure A.4: Module activation during the Compare instruction.

Branch Instruction

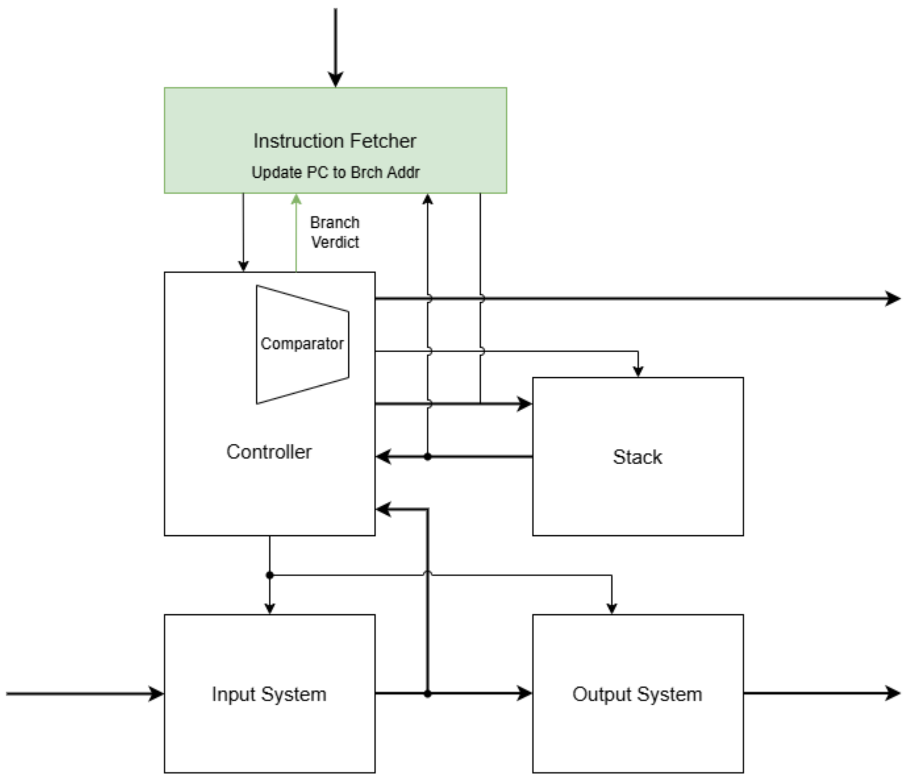

Figure A.5: Module activation during the Branch instruction.

Jump Instruction

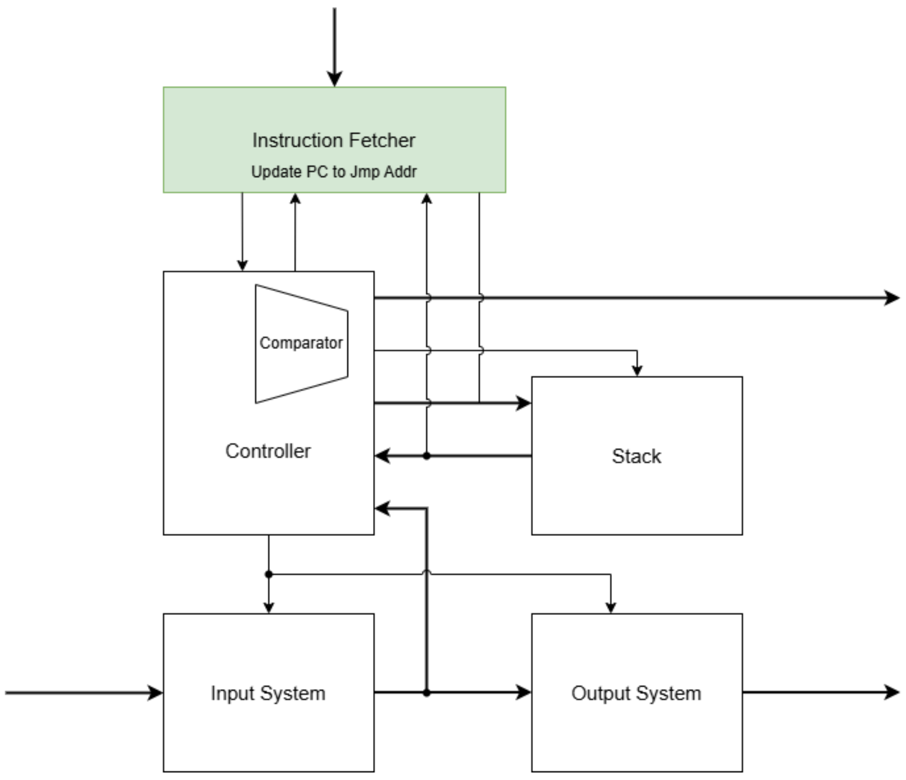

Figure A.6: Module activation during the Jump instruction.

Call Instruction

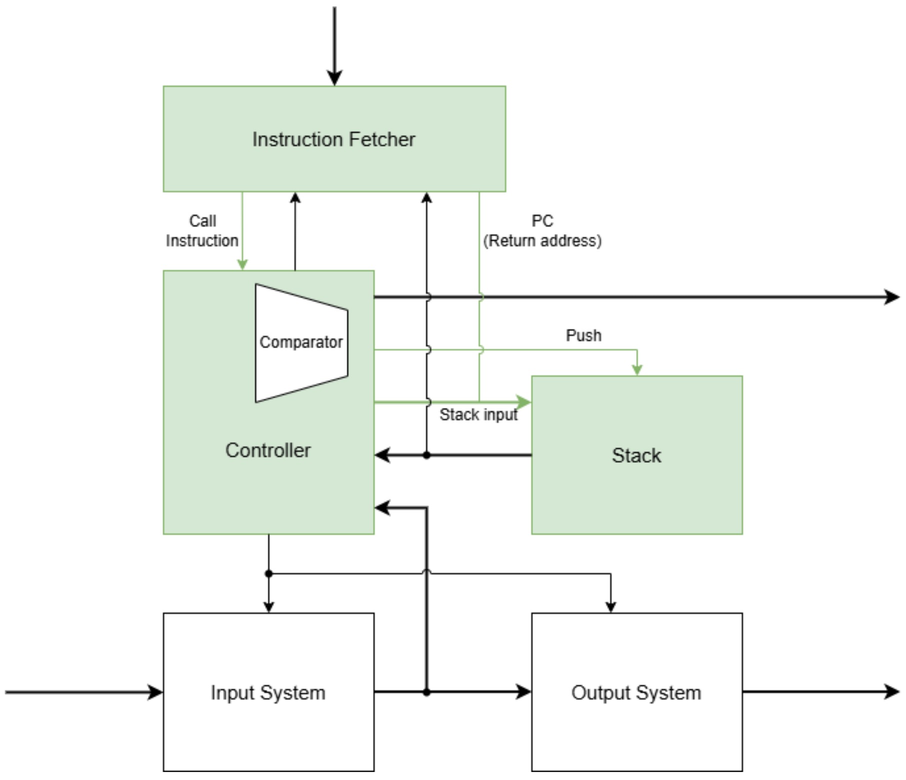

Figure A.7: Module activation during the Call instruction.

Return Instruction

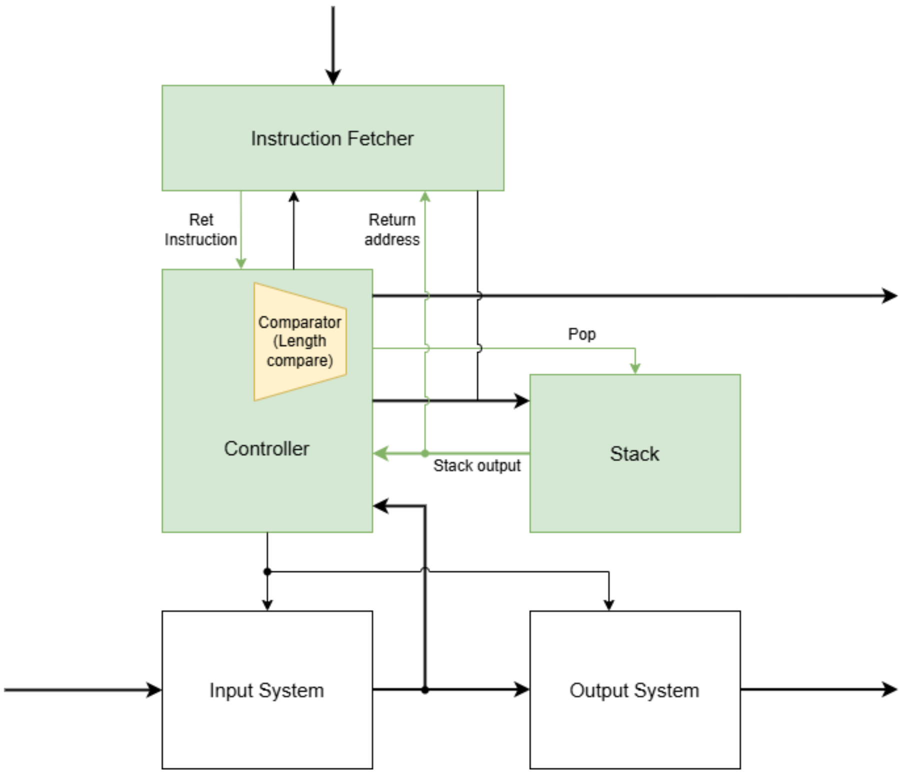

Figure A.8: Module activation during the Return instruction.

Increment Instruction

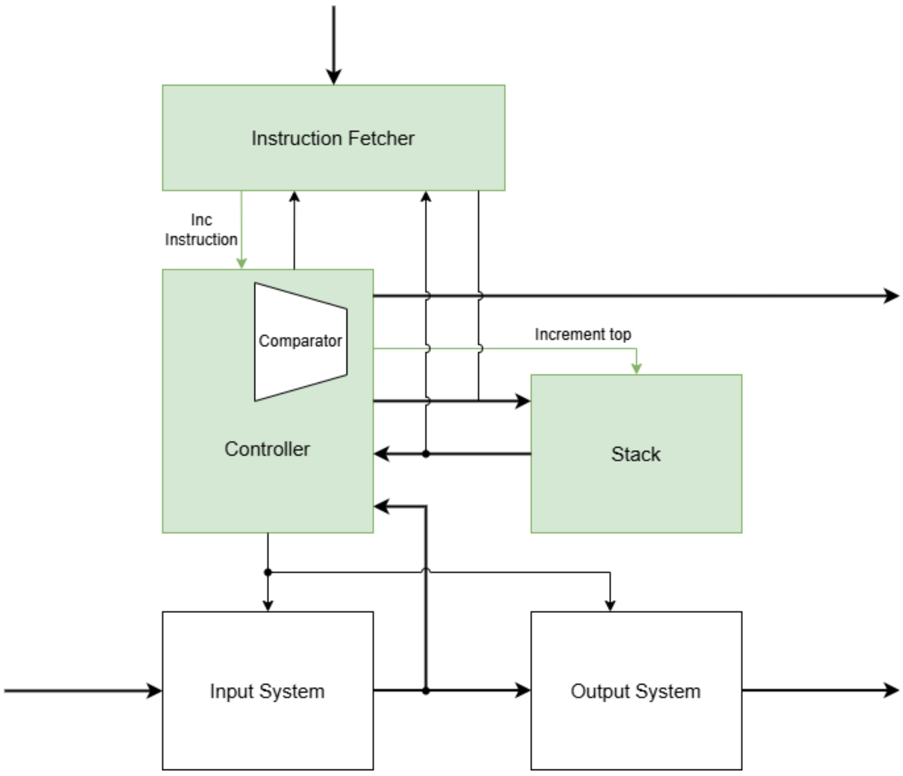

Figure A.9: Module activation during the Increment instruction.

B Benchmarking Schemas

This appendix shows all schemas that were used during benchmarking. Schemas start their execution from the topmost routine.

B.1. Packet Header Parsing

B.1.1. Full Graph - 5 tuple fields

```
packet_parser_5tup_full:
    // Start with Ethernet header
    sft 96
    // 802.1ad double tagging
    sft 16 cmp brch unsigned inp == const=0x88A8
    brch VLAN_Outer
    // 802.1Q
    sft 0 cmp brch unsigned inp == const=0x8100
    brch VLAN_Inner
    jmp ethernet_2

ethernet_2:
    // MPLS Unicast or Multicast
    sft 0 cmp brch unsigned inp == const=0x8847
    sft 0 cmp brch or unsigned inp == const=0x8848
    brch MPLS_Level_1
    jmp ethernet_3

ethernet_3:
    // IPv4
    sft 0 cmp brch unsigned inp == const=0x0800
    brch IPv4_header
    // IPv6
    sft 0 cmp brch unsigned inp == const=0x86DD
    brch IPv6_header
    end fail

VLAN_Outer:
    sft 48
    // MPLS
    sft 16
    jmp ethernet_2

VLAN_Inner:
    sft 16
    // MPLS
    sft 16
    jmp ethernet_2

MPLS_Level_1:
    sft 23
    sft 1 cmp brch unsigned inp == const=0b0
    brch MPLS_Level_2
    sft 8
    sft 16
    jmp ethernet_3

MPLS_Level_2:
    sft 31
    sft 1 cmp unsigned inp == const=0b1
    sft 8
    sft 16
    jmp ethernet_3

IPv4_header:
    sft 72
    // Protocol
    sft 8 cmp brch unsigned inp == const=0x06
    sft 0 cmp brch or unsigned inp == const=0x11
    brch IPv4_TCP_or_UDP_header
    sft 0 cmp brch unsigned inp == const=0x01
    brch ICMP4
    end fail

IPv4_TCP_or_UDP_header:
    sft 16
    // IP Source Address
    sft 32
    // IP Dest Address
    sft 32
    // TCP/UDP Source Port
    sft 16
    // TCP/UDP Dest Port
    sft 16
    // Remainder is not relevant for extraction in this schema, thus we flush
    end

ICMP4:
    sft 16
    // IP Source Address
    sft 32
    // IP Dest Address
    sft 32
    // Remainder ICMP4 is not relevant for extraction in this schema, thus we flush
    end

IPv6_header:
    sft 48
    // Next Header field
    sft 8 cmp brch unsigned inp == const=0x06
    sft 0 cmp brch or unsigned inp == const=0x11
    brch IPv6_TCP_or_UDP_header
    sft 0 cmp brch unsigned inp == const=0x3A
    brch ICMP6
    end fail

IPv6_TCP_or_UDP_header:
    sft 8
    // IP Source address
    sft 128
    // IP Dest address
    sft 128
    // TCP/UDP Source Port
    sft 16
    // TCP/UDP Dest Port
    sft 16
    // Remainder is not relevant for extraction in this schema, thus we flush
    end

ICMP6:
    sft 8
    // IP Source Address
    sft 128
    // IP Dest Address
    sft 128
    // Remainder ICMP4 is not relevant for extraction in this schema, thus we flush
    end
```

B.1.2. Full Graph - All fields

```
packet_parser_5tup_full:
    // Start with Ethernet header
    sft 48 // MAC Source
    sft 48 // MAC Dest
    // 802.1ad double tagging
    sft 16 cmp brch unsigned inp == const=0x88A8
    brch VLAN_Outer
    // 802.1Q
    sft 0 cmp brch unsigned inp == const=0x8100
    brch VLAN_Inner
    jmp ethernet_2

ethernet_2:
    // MPLS Unicast or Multicast
    sft 0 cmp brch unsigned inp == const=0x8847
    sft 0 cmp brch or unsigned inp == const=0x8848
    brch MPLS_Level_1
    jmp ethernet_3

ethernet_3:
    // IPv4
    sft 0 cmp brch unsigned inp == const=0x0800
    brch IPv4_header
    // IPv6
    sft 0 cmp brch unsigned inp == const=0x86DD
    brch IPv6_header
    end fail

VLAN_Outer:
    sft 16 // TCI Outer
    sft 16 // TPID Inner
    sft 16 // TCI Inner
    sft 16 // Ethertype
    jmp ethernet_2

VLAN_Inner:
    sft 16 // TCI
    sft 16 // Ethertype
    jmp ethernet_2

MPLS_Level_1:
    sft 20 // Label
    sft 3 // Traffic Class
    sft 1 cmp brch unsigned inp == const=0b0
    brch MPLS_Level_2
    sft 8 // TTL
    sft 16 // Ethertype
    jmp ethernet_3

MPLS_Level_2:
    sft 8 // TTL from level 1
    sft 20 // Label
    sft 3 // Traffic Class
    sft 1 cmp unsigned inp == const=0b1
    sft 8 // TTL
    sft 16 // Ethertype
    jmp ethernet_3

IPv4_header:
    sft 4 // Version
    sft 4 // IHL
    sft 8 // TOS
    sft 16 // Total Length
    sft 16 // Identificiation
    sft 3 // Flags
    sft 13 // Fragment Offset
    sft 8 // TTL
    // Protocol
    sft 8 cmp brch unsigned inp == const=0x06
    brch IPv4_TCP
    sft 0 cmp brch unsigned inp == const=0x11
    brch IPv4_UDP
    sft 0 cmp brch unsigned inp == const=0x01
    brch ICMP
    end fail

IPv4_TCP:
    sft 16 // Header Checksum
    sft 32 // IP Source Address
    sft 32 // IP Dest Address
    // TCP
    sft 16 // TCP Source Port
    sft 16 // TCP Dest Port
    sft 32 // Sequence Number
    sft 32 // Acknowledge Number
    sft 4 // Data Offset
    sft 4 // Reserved
    sft 8 // Flags
    sft 16 // Window
    sft 16 // Checksum
    sft 16 // Urgent Pointer
    end

IPv4_UDP:
    sft 16 // Header Checksum
    sft 32 // IP Source Address
    sft 32 // IP Dest Address
    // UDP
    sft 16 // UDP Source Port
    sft 16 // UDP Dest Port
    sft 16 // Length
    sft 16 // Checksum
    end

ICMP:
    sft 16 // Header Checksum
    sft 32 // IP Source Address
    sft 32 // IP Dest Address
    // ICMP Header
    sft 8 // Type
    sft 8 // Code
    sft 16 // Checksum
    sft 32 // Rest of Header
    end

IPv6_header:
    sft 4 // Version
    sft 8 // Traffic Class
    sft 20 // Flow Label
    sft 16 // Payload Length
    // Next Header field
    sft 8 cmp brch unsigned inp == const=0x06
    brch IPv6_TCP
    sft 0 cmp brch unsigned inp == const=0x11
    brch IPv6_UDP
    sft 0 cmp brch unsigned inp == const=0x3A
    brch ICMPv6
    end fail

IPv6_TCP:
    sft 8 // Hop Limit
    sft 128 // IP Source address
    sft 128 // IP Dest address
    // TCP Header
    sft 16 // TCP Source Port
    sft 16 // TCP Dest Port
    sft 32 // Sequence Number
    sft 32 // Acknowledge Number
    sft 4 // Data Offset
    sft 4 // Reserved
    sft 8 // Flags
    sft 16 // Window
    sft 16 // Checksum
    sft 16 // Urgent Pointer
    end

IPv6_UDP:
    sft 8 // Hop Limit
    sft 128 // IP Source address
    sft 128 // IP Dest address
    // UDP Header
    sft 16 // UDP Source Port
    sft 16 // UDP Dest Port
    sft 16 // Length
    sft 16 // Checksum
    end

ICMPv6:
    sft 8 // Hop Limit
    sft 128 // IP Source address
    sft 128 // IP Dest address
    // ICMPv6 Header
    sft 8 // Type
    sft 8 // Code
    sft 16 // Checksum
    end
```

B.1.3. Simple Graph - 5 tuple fields

```
packet_parser_5tup_simple:
    // Start with Ethernet header
    sft 96
    // IPv4 or IPv6
    sft 16 cmp brch unsigned inp == const=0x0800
    brch IPv4_header
    // IPv6
    sft 0 cmp brch unsigned inp == const=0x86DD
    brch IPv6_header
    end fail

IPv4_header:
    sft 72
    // Protocol
    sft 8 cmp brch unsigned inp == const=0x06
    sft 0 cmp brch or unsigned inp == const=0x11
    brch IPv4_TCP_or_UDP_header
    sft 0 cmp brch unsigned inp == const=0x01
    brch ICMP4
    end fail

IPv4_TCP_or_UDP_header:
    sft 16
    // IP Source Address
    sft 32
    // IP Dest Address
    sft 32
    // TCP/UDP Source Port
    sft 16
    // TCP/UDP Dest Port
    sft 16
    // Remainder is not relevant for extraction in this schema, thus we flush
    end

ICMP4:
    sft 16
    // IP Source Address
    sft 32
    // IP Dest Address
    sft 32
    // Remainder ICMP4 is not relevant for extraction in this schema, thus we flush
    end

IPv6_header:
    sft 48
    // Next Header field
    sft 8 cmp brch unsigned inp == const=0x06
    sft 0 cmp brch or unsigned inp == const=0x11
    brch IPv6_TCP_or_UDP_header
    sft 0 cmp brch unsigned inp == const=0x3A
    brch ICMP6
    end fail

IPv6_TCP_or_UDP_header:
    sft 8
    // IP Source address
    sft 128
    // IP Dest address
    sft 128
    // TCP/UDP Source Port
    sft 16
    // TCP/UDP Dest Port
    sft 16
    // Remainder is not relevant for extraction in this schema, thus we flush
    end

ICMP6:
    sft 8
    // IP Source Address
    sft 128
    // IP Dest Address
    sft 128
    // Remainder ICMP4 is not relevant for extraction in this schema, thus we flush
    end
```

B.1.4. Simple Graph - All fields

```
packet_parser_5tup_simple:
    // Start with Ethernet header
    sft 48 // MAC Source
    sft 48 // MAC Dest
    // Ethertype, check for IPv4 or IPv6
    sft 16 cmp brch unsigned inp == const=0x0800
    brch IPv4_header
    // IPv6
    sft 0 cmp brch unsigned inp == const=0x86DD
    brch IPv6_header
    end fail

IPv4_header:
    sft 4 // Version
    sft 4 // IHL
    sft 8 // TOS
    sft 16 // Total Length
    sft 16 // Identificiation
    sft 3 // Flags
    sft 13 // Fragment Offset
    sft 8 // TTL
    // Protocol
    sft 8 cmp brch unsigned inp == const=0x06
    brch IPv4_TCP
    sft 0 cmp brch unsigned inp == const=0x11
    brch IPv4_UDP
    sft 0 cmp brch unsigned inp == const=0x01
    brch ICMP
    end fail

IPv4_TCP:
    sft 16 // Header Checksum
    sft 32 // IP Source Address
    sft 32 // IP Dest Address
    // TCP
    sft 16 // TCP Source Port
    sft 16 // TCP Dest Port
    sft 32 // Sequence Number
    sft 32 // Acknowledge Number
    sft 4 // Data Offset
    sft 4 // Reserved
    sft 8 // Flags
    sft 16 // Window
    sft 16 // Checksum
    sft 16 // Urgent Pointer
    end

IPv4_UDP:
    sft 16 // Header Checksum
    sft 32 // IP Source Address
    sft 32 // IP Dest Address
    // UDP
    sft 16 // UDP Source Port
    sft 16 // UDP Dest Port
    sft 16 // Length
    sft 16 // Checksum
    end

ICMP:
    sft 16 // Header Checksum
    sft 32 // IP Source Address
    sft 32 // IP Dest Address
    // ICMP Header
    sft 8 // Type
    sft 8 // Code
    sft 16 // Checksum
    sft 32 // Rest of Header
    end

IPv6_header:
    sft 4 // Version
    sft 8 // Traffic Class
    sft 20 // Flow Label
    sft 16 // Payload Length
    // Next Header field
    sft 8 cmp brch unsigned inp == const=0x06
    brch IPv6_TCP
    sft 0 cmp brch unsigned inp == const=0x11
    brch IPv6_UDP
    sft 0 cmp brch unsigned inp == const=0x3A
    brch ICMPv6
    end fail

IPv6_TCP:
    sft 8 // Hop Limit
    sft 128 // IP Source address
    sft 128 // IP Dest address
    // TCP Header
    sft 16 // TCP Source Port
    sft 16 // TCP Dest Port
    sft 32 // Sequence Number
    sft 32 // Acknowledge Number
    sft 4 // Data Offset
    sft 4 // Reserved
    sft 8 // Flags
    sft 16 // Window
    sft 16 // Checksum
    sft 16 // Urgent Pointer
    end
```

```

IPv6_UDP:
    sft 8 // Hop Limit
    sft 128 // IP Source address
    sft 128 // IP Dest address
    // UDP Header
    sft 16 // UDP Source Port
    sft 16 // UDP Dest Port
    sft 16 // Length
    sft 16 // Checksum
    end

ICMPv6:
    sft 8 // Hop Limit
    sft 128 // IP Source address
    sft 128 // IP Dest address
    // ICMPv6 Header
    sft 8 // Type
    sft 8 // Code
    sft 16 // Checksum
    end
```

B.2. Packet Header Validation

```
ICMP_filter:
    // Start Ethernet header
    sft 96                                     // Shift through MAC Source and MAC Dest
    sft 16 cmp unsigned inp == const=0x0800    // Ethertype
    // Start IPv4 header
    sft 8 cmp unsigned inp == const=0x45       // Version and IHL
    sft 8                                      // Skip TOS
    sft 16 cmp unsigned inp >= const=28        // 28 <= Total Length <= 1500 (MTU)
    sft 0 cmp unsigned inp <= const=1500
    sft 40                                     // Skip IPv4 fields until protocol field.
    sft 8 cmp brch unsigned inp == const=0x01 // Protocol == ICMP
    brch ICMP
    end

ICMP:
    sft 80 // Shift through remaining IPv4 fields
    // Start ICMP header
    sft 8 cmp unsigned inp == const=0x0    // Type == Echo Reply
    sft 0 cmp or unsigned inp == const=0x8 // Type == Echo Request
    sft 8 cmp unsigned inp == const=0x0    // Code == 0
    // Shift through the rest
    end
```

B.3. X.509 Certificate Checking

```
Certificate:
    sft 8 cmp unsigned inp == const=0x30
    call parse_length
    call len bytes Certificate_value
    end

Certificate_value:
    // Parse TBSCertificate (SEQUENCE)
    sft 8 cmp unsigned inp == const=0x30
    call parse_length
    call len bytes TBSCertificate_value
    // Parse AlgorithmIdentifier (SEQUENCE)
    sft 8 cmp unsigned inp == const=0x30
    call parse_length
    call len bytes AlgorithmIdentifier_value
    // Parse Signature (BIT STRING)
    jmp parse_BIT_STRING

TBSCertificate_value:
    // version (Context-specific INTEGER) Only version 3 certificates supported
    sft 40 cmp unsigned inp == const=0xA003020102
    // serialNumber (INTEGER)
    sft 8 cmp unsigned inp == const=0x02
    sft 8 cmp unsigned inp <= const=32
    sft byteinp
    // signature (AlgorithmIdentifier)
    sft 8 cmp unsigned inp == const=0x30
    call parse_length
    call len bytes AlgorithmIdentifier_value
    // issuer (Name)
    sft 8 cmp unsigned inp == const=0x30
    call parse_length
    call whilelen bytes Name_value
    // validity (Validity)
    sft 8 cmp unsigned inp == const=0x30
    call parse_length
    call len bytes Validity_value
    // subject (Name)
    sft 8 cmp unsigned inp == const=0x30
    call parse_length
    call whilelen bytes Name_value
    // subjectPublicKeyInfo (SubjectPublicKeyInfo)
    sft 8 cmp unsigned inp == const=0x30
    call parse_length
    call len bytes SubjectPublicKeyInfo_value
    // issuerUniqueIdentifier (Context-specific UniqueIdentifier OPTIONAL)
    sft 0 cmp brch unsigned stacktop != byte_cnt
    sft 8 cmp brch unsigned inp == const=0xA1
    brch issuerUID_start
    // subjectUniqueIdentifier (Context-specific UniqueIdentifier OPTIONAL)
    sft 0 cmp brch unsigned stacktop != byte_cnt
    sft 0 cmp brch unsigned inp == const=0xA2
    brch subjectUID_start
    // extensions (Context-specific Extensions OPTIONAL)
    sft 0 cmp brch unsigned stacktop != byte_cnt
    sft 0 cmp brch unsigned inp == const=0xA3
    brch extensions_start
    ret

AlgorithmIdentifier_value:
    // algorithm (OID)
    sft 8 cmp unsigned inp == const=0x06
    sft 8 cmp unsigned inp <= const=32
    sft byteinp
    // parameters (ANY OPTIONAL)
    sft 0 cmp brch unsigned stacktop != byte_cnt
    brch parse_ANY
    ret

Name_value:
    // while-loop assertion. Fail if the byte counter becomes larger than the length
    sft 0 cmp brch unsigned stacktop < byte_cnt
    brch while_fail
    // break condition if length equals the amount of processed bytes
    sft 0 cmp brch unsigned stacktop == byte_cnt
    brch while_continue
    // parse one RelativeDistinguishedName in the sequence
    sft 8 cmp unsigned inp == const=0x31
    call parse_length
    call whilelen bytes RDN_value
    // jump back to the top of the while loop
    jmp Name_value

RDN_value:
    // while-loop assertion. Fail if the byte counter becomes larger than the length
    sft 0 cmp brch unsigned stacktop < byte_cnt
    brch while_fail
    // break condition if length equals the amount of processed bytes
    sft 0 cmp brch unsigned stacktop == byte_cnt
    brch while_continue
    // parse one AttributeTypeAndValue in the sequence
    sft 8 cmp unsigned inp == const=0x30
    call parse_length
    call len bytes ATV_value
    // jump back to the top of the while loop
    jmp RDN_value

ATV_value:
    //AttributeType
    sft 8 cmp unsigned inp == const=0x06
    sft 8 cmp unsigned inp <= const=32
    sft byteinp
    //AttributeValue (ANY)
    jmp parse_ANY

Validity_value:
    sft 8 cmp unsigned inp == const=0x17
    sft 0 cmp or unsigned inp == const=0x18
    sft 8 cmp unsigned inp <= const=32
    sft byteinp
    sft 8 cmp unsigned inp == const=0x17
    sft 0 cmp or unsigned inp == const=0x18
    sft 8 cmp unsigned inp <= const=32
    sft byteinp
    ret

SubjectPublicKeyInfo_value:
    // AlgorithmIdentifier
    sft 8 cmp unsigned inp == const=0x30
    call parse_length
    call len bytes AlgorithmIdentifier_value
    // Public Key (BIT STRING)
    jmp parse_BIT_STRING

issuerUID_start:
    call parse_length
    call len bytes parse_BIT_STRING
    jmp issuerUID_rest

issuerUID_rest:
    // Check if subjectUniqueIdentifier is present
    sft 0 cmp brch unsigned stacktop != byte_cnt
    sft 8 cmp brch unsigned inp == const=0xA2
    brch subjectUID_start
    // Check if extensions are present
    sft 0 cmp brch unsigned stacktop != byte_cnt
    sft 0 cmp brch unsigned inp == const=0xA3
    brch extensions_start
    ret

subjectUID_start:
    call parse_length
    call len bytes parse_BIT_STRING
    jmp subjectUID_rest

subjectUID_rest:
    // Check if extensions are present
    sft 0 cmp brch unsigned stacktop != byte_cnt
    sft 8 cmp brch unsigned inp == const=0xA3
    brch extensions_start
    ret

extensions_start:
    call parse_length
    call len bytes parse_extensions
    ret

parse_extensions:
    sft 8 cmp unsigned inp == const=0x30
    call parse_length
    call whilelen bytes Extensions_SEQUENCE_value

```

```
Extensions_SEQUENCE_value:
    // while-loop assertion. Fail if the byte counter becomes larger than the length
    sft 0 cmp brch unsigned stacktop < byte_cnt
    brch while_fail
    // break condition if length equals the amount of processed bytes
    sft 0 cmp brch unsigned stacktop == byte_cnt
    brch while_continue
    // parse one Extension in the sequence
    sft 8 cmp unsigned inp == const=0x30
    call parse_length
    call len bytes Extension_value
    // jump back to the top of the while loop
    jmp Extensions_SEQUENCE_value

Extension_value:
    // extensionID (OID)
    sft 8 cmp unsigned inp == const=0x06
    sft 8 cmp unsigned inp <= const=32
    sft byteinp
    // critical (BOOLEAN)
    sft 8 cmp brch unsigned inp == const=0x01
    brch parse_critical
    jmp Extension_value_rest

parse_critical:
    sft 16 cmp unsigned inp == const=0x01FF
    sft 8
    jmp Extension_value_rest

Extension_value_rest:
    // extensionValue (OCTET STRING)
    sft 0 cmp unsigned inp == const=0x04
    call parse_length
    sft 0 cmp brch unsigned inp > const=32
    brch big_shift
    sft byteinp
    ret

////////// HELPER ROUTINES //////////
// Helper routine to parse one TLV
parse_ANY:
    sft 8
    call parse_length
    sft 0 cmp brch unsigned inp > const=32
    brch big_shift
    sft byteinp
    ret

// Helper routine to parse a BIT STRING
parse_BIT_STRING:
    sft 8 cmp unsigned inp == const=0x03
    call parse_length
    sft 0 cmp brch unsigned inp > const=32
    brch big_shift
    sft byteinp
    ret

// Helper routines to shift a large (> 256-bit) section of data
big_shift:
    call whilelen bytes big_shift_while
    ret

big_shift_while:
    // Shift one byte and break condition if length equals the number of shifted bytes
    sft 8 cmp brch unsigned stacktop == byte_cnt
    brch while_continue
    // jump back to the top of the while loop
    jmp big_shift_while

// While-loop stop routines
while_continue:
    ret

while_fail:
    end fail

// Helper routines to parse a DER (long-)length field
parse_length:
    sft 1 cmp brch unsigned inp == const=1
    sft 7
    brch parse_long_length
    ret

parse_long_length:
    sft byteinp
    ret
```

B.4. Weather Station Data Validation

```
Start:
    sft 8 cmp unsigned inp == const=0x9b
    // sensor_id
    call extract_unsigned_data
    sft 0 cmp unsigned inp >= const=1000
    sft 0 cmp unsigned inp <= const=2000
    // battery_level
    call extract_unsigned_data
    sft 0 cmp unsigned inp <= const=100
    // timestamp
    call extract_unsigned_data
    // temperature
    call extract_data
    sft 0 cmp signed inp >= const=-12800
    sft 0 cmp signed inp <= const=12800
    // humidity
    call extract_unsigned_data
    sft 0 cmp unsigned inp >= const=10
    sft 0 cmp unsigned inp <= const=100
    // wind_speed
    call extract_data
    sft 0 cmp unsigned inp >= const=0
    sft 0 cmp unsigned inp <= const=25600 // 100.0 m/s in fixed_point with 8 bits frac
    // wind_direction
    call extract_unsigned_data
    sft 0 cmp unsigned inp <= const=360
    // pressure
    call extract_data
    sft 0 cmp unsigned inp >= const=8448
    sft 0 cmp unsigned inp <= const=28160
    // precipitation
    call extract_data
    sft 0 cmp unsigned inp >= const=0
    // solar_radiation
    call extract_data
    sft 0 cmp unsigned inp >= const=0
    sft 0 cmp unsigned inp <= const=19200
    // uv_index
    call extract_unsigned_data
    sft 0 cmp unsigned inp <= const=10
    end

extract_data:
    sft 1 cmp brch unsigned inp == const=0b0
    brch pos_fixint
    sft 2 cmp brch unsigned inp == const=0b11
    brch neg_fixint
    sft 0 cmp unsigned inp == const=0b10
    sft 5 cmp brch unsigned inp == const=0b01100
    sft 0 cmp brch or unsigned inp == const=0b10000
    brch sft_8
    sft 0 cmp brch unsigned inp == const=0b01101
    sft 0 cmp brch or unsigned inp == const=0b10001
    brch sft_16
    sft 0 cmp unsigned inp == const=0b01110
    sft 32
    ret

extract_unsigned_data:
    sft 1 cmp brch unsigned inp == const=0b0
    brch pos_fixint
    sft 7 cmp brch unsigned inp == const=0b1001100 // uint8
    brch sft_8
    sft 0 cmp brch unsigned inp == const=0b1001101 // uint16
    brch sft_16
    sft 0 cmp unsigned inp == const=0b1001110 // uint32
    sft 32
    ret

pos_fixint:
    sft 7
    ret

neg_fixint:
    sft 5
    ret

sft_8:
    sft 8
    ret

sft_16:
    sft 16
    ret
```

B.5. Video Stream Header Validation

```
Start:
    // Sync byte and TEI
    sft 9 cmp unsigned inp == const=0b010001110
    // PUSI and Transport Priority
    sft 2
    // PID, TSC and AFC
    sft 17 cmp unsigned inp == const=0x00211
    // CC
    sft 4
    // PES packet
    // Start code
    sft 32 cmp unsigned inp >= const=0x000001E0
    sft 0 cmp unsigned inp <= const=0x000001EF
    // Packet length
    sft 16 cmp unsigned inp == const=0
    // Flush the payload
    end
```

B.6. Drone Command Validation

```
Start:
    // Session ID (uint)
    sft 32
    // Command code (uint)
    sft 32 cmp unsigned inp <= const=10
    // Priority level (uint)
    sft 32 cmp unsigned inp <= const=0x02
    // Timestamp (uint)
    sft 32
    // Target speed (ufixed e2)
    sft 32 cmp unsigned inp <= const=27778
    // Target orientation (fixed e2)
    sft 32 cmp signed inp >= const=-18000
    sft 0 cmp signed inp <= const=18000
    // Target latitude (fixed e7)
    sft 32 cmp signed inp >= const=300000000
    sft 0  cmp signed inp <= const=400000000
    // Target longitude (fixed e7)
    sft 32 cmp signed inp >= const=-450000000
    sft 0  cmp signed inp <= const=-250000000
    // Target altitude (ufixed e2)
    sft 32 cmp unsigned inp <= const=2000000
    // Auth token (opaque)
    sft 32 cmp unsigned inp == const=32
    sft 256
    // origin_id (opaque)
    sft 32 cmp unsigned inp == const=1
    sft 8 cmp unsigned inp == const=1 // Only allow command from origin ID 1
    sft 24
    // command_hash (opaque)
    sft 32 cmp unsigned inp == const=32
    sft 256
    end
```