Formally Verifying a Programmable Network Switch

by

Jiazheng Liu

B.S., University of California, San Diego (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Authored by:	Jiazheng Liu Department of Electrical Engineering and Computer Science May 17, 2024
Certified by:	Adam Chlipala Professor of Electrical Engineering and Computer Science Thesis Supervisor
Certified by:	Arvind Johnson Professor of Electrical Engineering and Computer Science Thesis Supervisor
Accepted by:	Leslie A. Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Formally Verifying a Programmable Network Switch

by

Jiazheng Liu

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

ABSTRACT

Programmable network switches are complex pieces of hardware that leverage nonobvious optimizations such as pipelining to offer flexible configuration interfaces. In this thesis, we propose a novel formal-verification methodology aimed at establishing strong correctness theorems for synthesizable hardware designs for network functionality, demonstrated through a case-study analysis of a Tofino-like programmable switch that we call VeriSwit. Our approach hinges on *modularity*, whereby the system is split into interconnected units, each equipped with its specification and proof, oblivious to the internals of other units. We conduct VeriSwit's modular verification in the Coq theorem prover. Experiments with synthesis for both FPGA and ASIC targets, combined with simulation, show that 100 GB/s line rate is easily achieved.

Thesis supervisor: Adam Chlipala
Title: Professor of Electrical Engineering and Computer Science

Thesis supervisor: Arvind
Title: Johnson Professor of Electrical Engineering and Computer Science

Acknowledgments

Much of the content of this thesis is taken from a joint conference submission, co-authored with Prof. Thomas Bourgeat, Prof. Manya Ghobadi, and Prof. Adam Chlipala.

Contents

List of Figures

Chapter 1

Introduction

Programmable network switches are complex digital hardware designs, prone to bugs, many of which are difficult to catch with testing alone. We demonstrate for the first time that *comprehensive mathematical correctness proof of programmable switch implementations is feasible.* A mathematical proof removes the need for testing or code auditing to find bugs. Instead, one proof can cover infinitely many workloads.

Our techniques apply generally to a variety of networking hardware, but we chose to focus evaluation on one case study: a programmable switch in the style of Tofino, which we call VeriSwit. Figure 1.1 illustrates the hardware microarchitecture, which we implemented ourselves with provability in mind. The microarchitectures of Tofino and VeriSwit are highly concurrent and require performing intricate size and offset arithmetic. It is very easy to introduce bugs that are hard to catch with testing, or even with certain formal-verification techniques like bounded model checking. One hardware-industry study [1] reports that verification and validation easily account for over 25% of the costs of developing a chip, suggesting major cost-saving opportunities from streamlining the process. Common categories of bugs (see §4.1 for more discussion) include coordination errors between pipeline stages, mistakes in scheduling parallel workers, fencepost errors in buffer addressing, and lurking mistakes in infrequently exercised error-handling logic.

A bug in a switch can invalidate all of the guarantees that formal verification of e.g. P4 programs had provided. Testing and bounded model checking can help build some confidence that a switch behaves correctly in a variety of scenarios. However, it may always turn out that the test suite was insufficient or the bound to model checking was too low. As a result, *unbounded proofs of correctness* are appealing. Such a mathematical proof, ideally checkable by the proof-checking algorithm of an off-the-shelf theorem prover, establishes that an artifact behaves correctly *under any input of any length.* The feasibility of such techniques has been demonstrated before for a variety of concurrent hardware (e.g. processors [2]) and

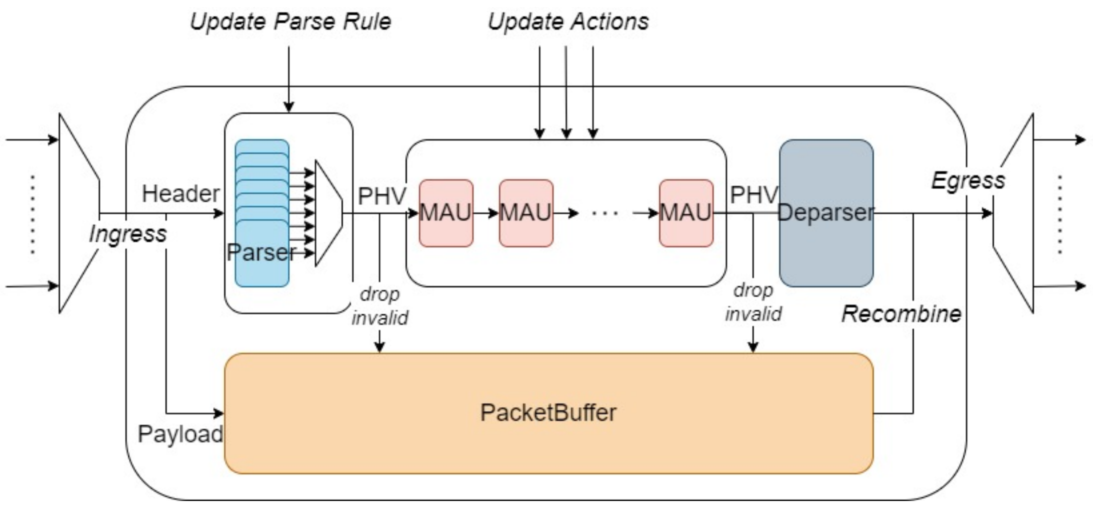


Figure 1.1: High-level illustration of VeriSwit. The rounded-corner units represent hardware components that may execute concurrently.

software (e.g. OS kernels [3], [4]). However, to the best of our knowledge, no prior work has demonstrated unbounded proof of networking hardware, just networking software (e.g. [5]–[10]).

Our unbounded proof of correctness for VeriSwit is accepted by the Coq theorem prover, which is a very general platform used for applications from pure math to systems software and hardware. It supports automatic and trustworthy checking of intricate proofs written by developers in a special source-code language. In applying formal methods to VeriSwit, we had to address two key questions.

First, *what is the right kind of formal correctness statement (specification) for a programmable switch?* We developed a specification style in terms of an *unoptimized reference implementation of switch functionality*: an alternative hardware design free of optimizations, hence much easier to audit for compatibility with expectations. While the code of VeriSwit itself is dominated by concurrency and intricate offset arithmetic, the VeriSwit specification includes very little of those aspects. We prove that any behavior of VeriSwit can be mimicked by the simpler specification.

Second, *how should the proof be divided into manageable pieces, to control proof-engineering effort?* We adopted the Fjfj framework [11] for Coq proof of hardware designs, which provides a module system for not just breaking hardware designs into hierarchies of components but also mimicking that hierarchy in proof structure as well, so that each component can have its own specification and proof, uncomplicated by the rest of the hierarchy. The largest contribution of our work is *a specific hierarchical proof structure that tames the complexities of an optimized switch.* That is, we show how to break the overwhelming overall correctness proof into lemmas that tackle one optimization at a time. (Figure 4.2 diagrams the proof structure, with more detail in section 4.3.)

We created the VeriSwit hardware design from scratch in the Bluespec hardware language, aiming for sufficient realism while controlling complexity to simplify our exploration of the design space of formal-methods techniques. To evaluate VeriSwit's realism, we synthesized for both VCU118 FPGAs and the FreePDK ASIC platform, giving us clock-frequency information. Our experiments then largely rely on cycle-accurate Verilog simulation, whose results can be scaled using the frequencies confirmed with synthesis. The upshot is that an ASIC version of this design should easily sustain line rate of 100 Gb/s in throughput.

Chapter 2

Related Work

2.1 Hardware Verification

Beyond the hardware-verification tools we adapt for VeriSwit, there is a long tradition of verification, mostly focused on processors. Combinational circuits are hardware implementations of Boolean functions. Over the years, research has produced impressive techniques to solve many practical problems in their verification [12], [13]. Those tools and techniques, very effective for combinational circuits, are also at the root of most mainstream formal-verification techniques in use today for verification of sequential machines [14]–[16]. While they can search impressively large spaces, these techniques get hit by combinatorial-explosion problems, and even the most advanced automatic-verification techniques have trouble handling processors that have more than a couple instructions in flight; we expect the same obstacles to arise for programmable networking hardware. Other approaches require specification of explicit *refinement maps* that connect state of simpler and more optimized implementations. Here some pioneering work was by Hunt et al. [17], [18], tackling industrial designs, often in the ACL2 theorem-proving system. Quite a lot of work [19]–[25] has been done in other frameworks (for example, in UCLID5 [26] or in SMV [27]), using various levels of automation and tackling custom models expressed at various levels of abstraction over synthesizable designs, trading off for complexity of the architectural schemes being proven. In general, we are not aware of past work with hardware formal verification applied to network switches.

2.2 Formal Methods in Networking

There have been many impactful applications of formal methods in networking, including different styles of programmability in networks, like software-defined networking (SDN). Zhang and Malik [5] demonstrated that SAT solvers can be used to check that SDN configurations satisfy properties like lack of forwarding loops. The configurations they dealt with are described as networks' *data planes*. Another data-plane-testing framework PTA [28] can assert properties of a given switch such as correctness and performance by using hardware to generate test packets. Guha, Reitblatt, and Foster [6], on the other hand, showed that the *control plane* is also feasible for formal verification, as they carried out a Coq proof of correctness for an SDN-language compiler and the runtime support needed for orchestrating configuration changes. Similar results have been demonstrated for the new breed of programmability around the P4 language, including in verification of individual P4 programs [7] and first steps toward verification of compilers and other tools that manipulate P4 [8]. Gauntlet [29] applies translation validation to formally compare P4 IRs through P4C (P4 compiler) passes to find semantic bugs in the compiler. With an additional model-based testing module, it is able to detect bugs in the closed-source Tofino backend compiler. In addition to employing formal analysis, Minesweeper [30] offers a verification framework with extensive network design and data plane coverage across a wide range of network protocols, topologies, and potential data planes emerging from the control plane. Its follow-on work [31] improves the developer interface and feedback on test suites. P6 [32] integrates reinforcement-learning techniques with fuzzing to guide the generator, enabling it to detect, localize, and patch P4 bugs. Network verification tools have even seen significant industry adoption, as with Batfish [9] and VeriFlow [10]. This whole research area covers an impressive span of programmable-networking stacks, but as far as we are aware, no prior work has verified the *hardware* that acts on data-plane configurations, hence missed hardware bugs could invalidate all properties being proved.

Chapter 3

Methodology and Techniques

In this chapter, we present in detail the background of the tools we utilized to carry out the proof, our definition for correctness and some key techniques we used in proving correctness.

3.1 Rule-Based Languages and Simulation Relations

Our work VeriSwit is coded in Bluespec [33], a relatively high-level hardware description language that is an alternative to the more common Verilog or VHDL. Bluespec offers high-level, software-style programming abstractions: designs may be split across modules in object-oriented programming style, with private state and public methods. The Bluespec compiler produces RTL ("the assembly language of digital hardware"), automatically coordinating the concurrent execution of different modules. Both BSV [33] and Fjfj [11] are *rule-based HDLs*, centered around modules that consist of private states, expose public *methods* as interfaces to interact with, and maintain a set of atomic internal state transitions called *rules* within the module. BSV is used to write synthesizable code and deploy on FPGA/ASIC, while Fjfj is used to model the formal behavior of the implementation code, manually translated from BSV code line-by-line, to allow us to reason about correctness. We introduce the related terminology by simplifying the settings in each to have a unified model that covers both cases.

On a high level, each module $M = (S, V, A, R)$ in rule-based languages has a set of internal states S, a list of *value methods* V where each element is represented as a ternary relation of the form $N \times S \times N$ (we take N to be the set of all possible data in the system. In BSV, N equals bitstream or tuple of bitstreams of given length; in Fjfj, $N = \mathbb{N}$, the set of all natural numbers), a list of *action methods* A where each element is represented as a ternary relation of the form $N \times S \times S$, and a list of *rules* R where each element is represented as a binary relation of the form $S \times S$. For any index n, the ternary tuple $v = (a, s, r)$ at the n^{th} index in V means that the n^{th} value method of M returns r given the input a and the current state $s \in S$. Similarly, $A[n] = (a, s, s')$ means that executing the n^{th} action value of M with argument a would transform state s to s', and the same goes for rules except rules do not take any arguments.

The correctness terminology relies on the notion of *simulation relation*. This property implies that for any behavior of the implementation as a result of a sequence of interactions with it, the exact same behavior of the specification must also be observable following the same sequence of interactions. Following from above, we are able to reduce the complexity of the implementation to a minimum as presented in its specification. We present our formal definition of simulation relation in the following.

Definition 1 (State Simulation) *Formally, given two modules $M_i = (S_i, V_i, A_i, R_i), M_s = (S_s, V_s, A_s, R_s)$ exposing the same interface (i.e. V_i, V_s have the same cardinality, and A_i, A_s have the same cardinality), a state $s \in S_s$ simulates a state $i \in S_i$ (or s is* indistinguishable *from i, denoted $i \; {}_{M_i}\!\prec_{M_s} s$, or $i \prec s$ if both M_i, M_s are unambiguous) if*

1. *for any argument and return value a, r along with the current implementation state i, if (a, i, r) is related by $V_i[n]$ for some n, then (a, s, r) must be related by $V_s[n]$ (any value readily observed by a value method of the implementation is also readily observed by the same value method of the specification with the same argument);*
2. *for any argument a and a new implementation state $i' \in S_i$, if (a, i, i') is related by $A_i[n]$ for some n, then there exists an $s' \in S_s$ such that (a, s, s') is related by $A_s[n]$ (any action method of the implementation that is able to take a step must also be able to take a step in the specification with the same argument).*

In addition, with the state simulation definition, we may define Module Simulation.

Definition 2 (Module Simulation) *Given an additional binary relation φ on $S_i \times S_s$ (often called a* refinement mapping*), we say that module M_s simulates module M_i along φ, denoted $M_i \sqsubseteq_\varphi M_s$, if for every initial state $i \in S_i$, there is a state $s \in S_s$ such that $\varphi(i, s)$ and $i \prec s$ hold, and additionally, both φ and $\prec$ (*indistinguishbility *property) are invariants of the state transitions of corresponding actions in A_i and A_s and rules in R_i, up to the transitive closure of R_s. This means that for any states $i \in S_i$ and $s \in S_s$, if $i \prec s$ and $\varphi(i, s)$, we have*

1. *for any argument a and a new implementation state $i' \in S_i$, if (a, i, i') is related by $A_i[n]$ for some n, then there exists an $s_0 \in S_s$ such that (a, s, s_0) is related by $A_s[n]$, and a (possibly empty) sequence of rules $r_1, \ldots, r_k$ of specification and sequence of states*

$s_1, \ldots, s_k \in S_s$ such that for all $0 \le \ell < k$,

$$(s_\ell, s_{\ell+1}) \in R_s[r_{\ell+1}]$$

and both $\varphi(i', s_k)$ and $i' \prec s_k$ still hold (Figure 3.1a);

2. *for any new implementation state $i' \in S_i$, if (i, i') is related by $R_i[n]$ for some n, then there exists a (possibly empty) sequence of rules $r_1, \ldots, r_k$ of specification and sequence of states $s_1, \ldots, s_k \in S_s$ such that for all $0 \le \ell < k$ (write $s_0 = s$),*

$$(s_\ell, s_{\ell+1}) \in R_s[r_{\ell+1}]$$

and both $\varphi(i', s_k)$ and $i' \prec s_k$ still holds (Figure 3.1b).

$$\begin{tikzcd}
i \arrow[dd, "{A_i[n](a)}"'] & \sim & s \arrow[d, "{A_s[n](a)}"] \\
 & & s_0 \arrow[d, dashed, "R_s^*"] \\
i' & \sim & s_n
\end{tikzcd}$$

(a) Action refinement condition.

$$\begin{tikzcd}
i \arrow[d, "{R_i[n]}"'] & \sim & s_0 \arrow[d, dashed, "R_s^*"] \\
i' & \sim & s_n
\end{tikzcd}$$

(b) Rule refinement condition

Figure 3.1: Inductive Refinement Conditions. $\sim$ represents the conjunction of φ and $\prec$. $A_i[n](a)$ represents the transformations of implementation states by the n^{th} action method with a as arguments. Same goes for spec

The above simulation relation has a couple of properties that allow such a verification methodology to scale. First, the simulation relation is transitive. Namely, for all modules M_1, M_2, M_3, if $M_1 \sqsubseteq M_2$ and $M_2 \sqsubseteq M_3$, then $M_1 \sqsubseteq M_3$ (we omit mentioning refinement mappings to put an emphasis on the refinement relations, which automatically implies the existence of the map). This theorem allows us to prove the refinements separately, tackling one verification challenge at a time and leaving others untouched. Then we can connect the proofs together via the transitivity property. The next theorem is more involving in terms of the details of the implementation; readers who are interested should look at the project of Fjfj [11]. In short, this theorem allows us to replace a "submodule" in Fjfj with its specification, and the refinement relation can be lifted to the parent modules as well.

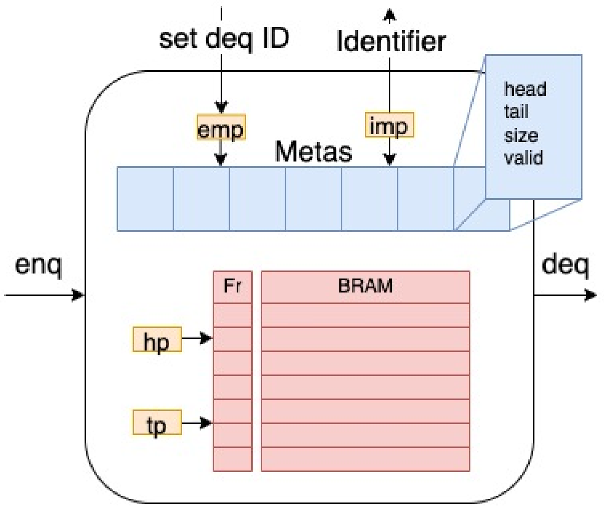


Figure 3.2: Packet-buffer implementation

3.2 Anatomy of a Module Proof

We illustrate here with the example of the packet buffer, a relatively simple module that mimics the real implementation in the work but still involves delicate optimizations (and indeed, as we cover in section 4.6, there are further optimizations to be proved in an additional proof step), to demonstrate how the proof of a refinement relation is done.

```
Record Pkt :=
{
  data : N;
  size : N;
}.
Record PacketBufferState :=
{
  mem : N -> option Pkt;
  currPkt : Pkt;
  idx : N;
  didx : N;
}.
```

Listing 3.1: Packet-buffer specification

In the optimized version (Figure 3.2), the packet buffer has 8 submodules: a BRAM[1] that stores the bodies of packets in consecutive 1024-bits chunks with a vector of Booleans in-

[1]This acronym stands for “block RAM,” the highest-capacity RAM typically made available in FPGA programming.

dicating slot availability, a vector that stores packet metadata (e.g. start and end address in the BRAM, size of actual packet in the last word) with a vector of Booleans indicating the availability of each entry, one egress meta pointer (*emp* in Figure 3.2), one ingress meta pointer (*imp*), one ingress BRAM pointer (*tp*), and a BRAM pointer that holds the head pointer (*hp*) to the current ingress packet.

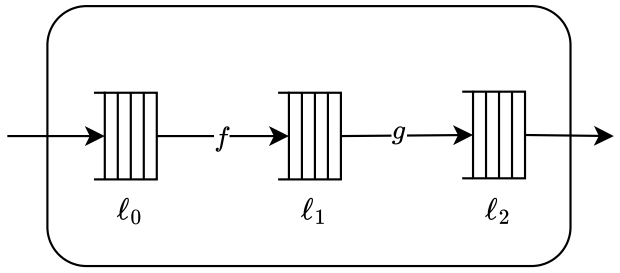
In the specification, the packet buffer (Listing 3.1) is represented as a function from numbers (addresses) to packet records, each of which holds the whole packet and its size, one packet record that stores the current ingress packet and one egress pointer and one ingress pointer, just as in the optimized version.

As an example, our proof of the packet buffer uses a simulation relation summarized in English like so:

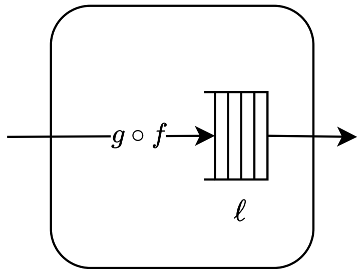
- For every valid (nonfree) meta entry, all words in BRAM between start and end pointers (both inclusive) must be valid.
- For every invalid (free) meta entry M with index n, the specification-level packet storage `mem` has no entry for n.
- For every valid (nonfree) meta entry M with index n, `mem` maps n to a packet, whose data equals the concatenation of all words of the BRAM from the start pointer b of M to the end pointer e of M (both inclusive), and the total packet size (in bytes) equals 128 multiplied by $e - b$ plus the size in the last word stored in M.
- The data of `currPkt` equals the concatenation of all words of the BRAM from the packet head pointer to the BRAM tail pointer (exclusive), and the total size (in bytes) equals 128 multiplied by $e - b$.
- Remaining index fields agree between specification and implementation.
- Other assertions of upper bounds (e.g. `Tail` $< 2^8$).

3.3 Verification Technique: Pipeline Flushing

It is quite technical to design and correct the invariants used in proving the simulation relation as we see above. Luckily, there are existing standard techniques when proving designs with certain patterns, one of which is the pipeline design. Consider an implementation design with a couple of internal pipeline queues where each deals with one arbitrary combinational function (denoted $f, g : N \to N$, where again N represents the set of data within the given system) (Figure 3.3a), and we want to prove its correctness against a specification that only has an output queue that buffers the result of $g \circ f$ upon any input (Figure 3.3b). Both modules expose two action methods, one to enqueue an element (called *enq*) and the other to dequeue (called *deq*), and a value method *first* to fetch the content at the head of the dequeue buffer. Such simplification of designs appears everywhere in hardware verification including in VeriSwit.



(a) Pipeline Implementation



(b) Pipeline Specification

Figure 3.3: An Example of Pipeline Refinement

The naive way of constructing a refinement mapping is to assert the content of ℓ in the specification to be equal to the concatenation of lists $\text{map}(g \circ f, \ell_0), \text{map}(g, \ell_1), \ell_2$, where map is a higher-order function that takes the function in the first argument and applies it to every element in the list. However, this approach becomes very tedious as the pipeline grows in size and complexity, and it is challenging to engineer good proof-automation tools for such an easily extractable pipeline pattern, hence difficult to scale to larger pipeline designs. An alternative can be adopted once an important fact is observed from the pipeline implementation: the outside world may not interact with the system further after enqueuing to observe the result, unless the enqueued element has moved across all the internal buffers and reached the output buffer. This property can be rephrased as "the position or the content of the enqueued element is indistinguishable or unknown from the outside world unless it is in the final output buffer".

With this observation, we may relate a set of possible implementation states in the refinement mapping with one configuration of the specification state. The set is given by the transitive closure of the inverses of rule relations, and it is called the *flushing relation.* Namely,

Definition 3 *A pair of implementation state (ℓ_0, ℓ_1, ℓ_2) and specification state ℓ is related by the flushing relation if*

1. Base Rule*: all the internal buffers except the output buffer ℓ_2 are empty, and the content of ℓ_2 precisely equals the content of ℓ;*
2. Inductive Rule*: for all rules R of the implementation (in this case the rules that dequeue from the previous buffers, execute the combinational circuit f or g and put it to the next buffers), if R relates (ℓ_0, ℓ_1, ℓ_2) and $(\ell'_0, \ell'_1, \ell'_2)$ for some new buffer $\ell'_0, \ell'_1, \ell'_2$ of the implementation, then $(\ell'_0, \ell'_1, \ell'_2)$ and ℓ must be related by the* flushing relation*.*

In other words, starting from the base rule that relates (ℓ_0, ℓ_1, ℓ_2) and ℓ, any state of the implementation that can be a starting state of executing any sequence of rules and end up with (ℓ_0, ℓ_1, ℓ_2) is related by the flushing relation to ℓ as well.

To use the flushing relation as the refinement mapping and prove the refinement according to definition 2, we use induction on the rules of the flushing relation. But first, it helps to show that the flushing relation as φ implies indistinguishability, or $\prec$. In the base case, we know that $\ell_2 = \ell$, so it is trivial to see that observing using *first* would either be blocked if ℓ is also empty or give the same return value in both modules. Similarly, *deq* is able to proceed in the specification assuming it may proceed in the implementation. Since we are assuming an infinite buffer size, *enq* will always be ready. It also holds that both ℓ_0 and ℓ_1 are empty in the base case, so none of the rules in the implementation is ready because they cannot dequeue from the previous buffer. We leave the proof for the inductive case for the implication of indistinguishability after we prove that φ is an invariant of the system, as it exhibits a very similar but simpler proof structure.

After we showed that φ implies indistinguishability, it is now sufficient to prove that φ by itself is an invariant of the system as stated in definition 2. We demonstrate how the rule case in definition 2 is done and the action method case would follow from the same proof. The proof requires to show that given the initial state $i = (\ell_0, \ell_1, \ell_2)$ and ℓ being related by the flushing relation, for any index to the rules list, after executing the rule method $R_i[n]$ on (ℓ_0, ℓ_1, ℓ_2) that results in new state $(\ell_0', \ell_1', \ell_2')$, the flushing relation still holds on $(\ell_0', \ell_1', \ell_2')$ and ℓ (note there is no internal rule in the specification so we omit the transitive closure from the proof statement). In the base case, note as every internal buffer is empty, it prevents any internal rule from firing hence vacuously holds. In the inductive case for one of the flushing rules $R_i[m]$, the inductive hypothesis states that the implementation state i_h' as a result of executing the rule sequence $R_i[m]$ and then $R_i[n]$ from initial state i is related by flushing with s. With the inductive hypothesis, it is required to show that starting with the same initial state i, the implementation state i'' as a result of executing the rule sequence $R_i[n]$ and then $R_i[m]$ is related by flushing with s (and of course it is possible to proceed with the rules), as shown in Figure 3.4a.

When $n = m$, it is trivial to conclude that $i_h = i'$ and $i_h' = i''$ as both f and g are functional and have unique output given the same input, making the rules deterministic. When $n \neq m$, one needs to reduce the proof to Figure 3.4b and show the commutativity of rules $R_i[n]$ and $R_i[m]$ hence $i'' = i_h'$ in the diagram on the left, ultimately utilizing the inductive hypothesis to finish the proof. The commutativity follows from the fact that buffers touched by different rules do not overlap or block each other. More specifically, either the two distinct rules would dequeue and enqueue to 4 distinct buffers (in a pipeline system with

$$\begin{tikzcd}
i \arrow[r, "R_i[m]"'] \arrow[d, "A_i[n](a)/R_i[n]"'] & i_h \arrow[d, "A_i[n](a)/R_i[n]"] \\
i' \arrow[d, "R_i[m]"'] & i'_h \arrow[d, "F"] \\
i'' \arrow[r, dashed, "F"] & s
\end{tikzcd}$$

(a) Inductive Proof

$$\begin{tikzcd}
i \arrow[r, "R_i[m]"'] \arrow[d, "A_i[n](a)/R_i[n]"'] & i_h \arrow[d, "A_i[n](a)/R_i[n]"] \\
i' \arrow[r, dashed, "R_i[m]"'] & i''
\end{tikzcd}$$

(b) Commutativity of Rules/Actions with flushing Rules

Figure 3.4: Proof for refinement relation for flushing. F represents the Flushing Relation. Dashed line represents the final proposition to be proved

more stages), or one of the buffers overlaps but one rule consumes while the other supplies, preventing any inconsistent behavior in this form of race condition. It is worth noticing that every pair of rules needs to be proved to commute, so the total complexity is on the order of N^2 if there are N rules. Nevertheless, such a commutativity proof only needs to look at the behavior or the semantics of the rules, with no other complex proof required, providing a good opportunity to engineer one proof tactic that can handle all possible forms of the pipeline. This would finish the proof.

As we remarked before, the case for the action method in definition 2 is similar to the above. One important difference is in the base case for *enq*, where all the internal buffers are empty and $\ell_2 = \ell$, we want to show that after enqueuing in both implementation and specification modules the states are still related by the flushing relation. To do this, we manually flush the enqueued element through the pipeline by applying the inductive rule of flushing. Since each internal buffer is empty, we may do so consecutively until the element is pushed to the output buffer, resulting in an equality of the new states $\ell_2' = \ell'$ and consequentially satisfying the base rule again.

In the experience of proving using the flushing relation, we noticed that it is not necessary to include every rule of the implementation in the inductive rules for flushing. For a system with multiple streams of pipelines or even orthogonal pipelines, it may be possible to split different pipelines into different proofs and connect at last by the transitivity of module refinement, only flushing the corresponding rules involved in a given pipeline. This reduces the total number of commutativity proofs as well as other complexities involved in the base rule. It is also possible to apply flushing to multiple-staged sequential execution that may use stage counters instead of pipeline buffers.

Sometimes the pipeline may not always be in order or even deterministic. When dealing with out-of-order nondeterministic pipelines, the standard flushing definition and its proof may fail due to insufficient inductive hypothesis. For example, rules are no longer commutativity as nondeterminism could lead to non-unique elements dequeued from certain buffers. Instead, it is required to relax both the flushing relation definition and the module simulation definition to allow arbitrary permutation (or constrained permutation up to some equivalence relation depending on the level of nondeterminism given by the pipeline) of the internal buffers prior to any rule execution. This generalizes the inductive hypothesis to include a family of possible initial states, allowing one to prove a strong commutativity theorem that is valid up to arbitrary permutations of the initial lists.

Chapter 4

Case Study: VeriSwit and its Specification

The heart of formal verification is showing that a complex artifact behaves like a simpler specification. We begin introducing VeriSwit by explaining both its real microarchitecture and its simpler specification.

4.1 VeriSwit Microarchitecture

We primarily follow the RMT (Reconfigurable Match Tables) model, also known as PISA (Protocol Independent Switch Architecture), proposed in previous work [34]. The two most important components that provide programmability are *programmable parsers* and *reconfigurable match-action units* (MAUs). By embedding high-performance hardware key-value storage called TCAM (ternary content-addressable memory), both the parsers and MAUs perform wildcard matching at line rate. At the top level, multiple MAUs are pipelined to maintain line rate throughout, and multiple parsers are parallelized together to process different packets at the same time to match the high throughput of the MAU pipeline. Figure 1.1 shows our primary components, which also include a deparser and a packet buffer.

Recall that VeriSwit is coded in Bluespec [33], where itself is a Bluespec module exposing eight public methods (which wind up compiled to input and output wires). There is *enq* to enqueue a chunk of ingress packet, *deq* to dequeue a chunk of egress packet, and six methods for reprogramming the switch: one to update parsing rules and five to update aspects of MAU behavior. (A production implementation would include a compiler from P4 to the table formats used by VeriSwit, but so far we have tested VeriSwit by producing those tables manually.) The *enq* method can take up to 1024 bits of packet content in each cycle, represented as a 1024-bit data value, an 8-bit length, an 8-bit input port, and a Boolean value that indicates if the current chunk of 1024 bits contains the last bytes of the current packet.

When a packet tries to enter VeriSwit, its first 1k data bits are enqueued by calling *enq*, at which point that chunk of data is immediately sent to the parallelized parsers to start parsing, in parallel to storing the data in the packet buffer (where they will be ignored until the packet makes its way to egress). The rest of the packet would then be considered as part of the packet body and stored inside the packet buffer without sending it to parsers.

Parsing proceeds header-by-header, extracting fields into a PHV (packet header vector). A parser may also mark a packet as invalid (e.g., when a table lookup fails, a header is too long, etc.), at which point the corresponding packet content in the packet buffer is also deleted. If a PHV is parsed successfully, it is passed to the MAUs, which may also mark a packet invalid. Last, valid PHVs would proceed to the deparser, where the egress header is assembled according to the PHV and combined with the original body and sent as switch output.

We selected a realistic subset of switch functionality that falls short of full Tofino but presents enough of the core representative challenges for verification.

- At the center of a typical Tofino flow diagram is a *traffic manager*, which can employ queues to schedule the flow of packets, replicate packets, and so forth. We implemented a simpler architecture where every ingress packet is either dropped or passed on ASAP as one egress packet. Architectural consequences include not distinguishing MAU activity for ingress vs. egress; we instead have one MAU pipeline.
- We use TCAMs to support both exact matching and ternary matching, instead of having dedicated SRAM-based cuckoo hash tables for exact matching as in the PISA model.
- We only support *stateless* network policies, where decisions on dropping or rewriting packets are only made on the basis of ingress packet contents, not other stateful data (e.g. flow rate, packet count, or previous packets in general).

We manually transcribed our implementation into the Coq theorem prover, using the Fjfj framework [11] for verifying Bluespec-style code. Compared to the Bluespec version, the Fjfj version and proof are abbreviated in considering some simple combinational-logic circuits to be given and assumed correct, but we explicitly represent and prove all stateful elements and any logic that spans multiple clock cycles.

Opportunities for Bugs

A few categories of bugs are difficult to catch in switches.

Pipelines. Switches often boost throughput by breaking expensive computations into stages that run concurrently, with queues connecting the stages to each other. Simple cases look like linear pipelines, but more complex topologies are also useful (as we will see for MAUs in section 4.5). Coordination errors between subsystems can lead to wrong answers in relatively rare race conditions. As an example, consider a parser that repeatedly consults a TCAM to determine the proper handling of each header field, pushing requests into later pipeline stages as appropriate to extract and store each relevant header into the right numbered slot of a PHV. It would be natural to code logic dictating that when the TCAM returns a flag indicating that no more headers should be processed, the PHV should be passed on to the next stage of the switch. Perhaps this logic is even correct for specific rule sets that include several unused fields before a header's end, giving the rest of the stages enough time to extract. However, in general, pending work might still exist in the pipeline that should block the release of the PHV. An early release would lead to MAUs running on incomplete PHV values. It is nontrivial to design a test suite that we can be sure would exercise any such potential coordination problem.

Dynamic multiplexing. It is also natural for switches to include multiple copies of certain functional blocks, which run in parallel and need to be scheduled dynamically. Parsers are the core examples in our switch. We need multiple parsing "threads" to run simultaneously (on different packets) to keep up with the throughput of MAU pipelines. Hardware logic must route each arriving packet to an available parser. That logic might very well include a bug that only manifests when all parsers are busy, for instance, assigning a packet to a parser that is already in use, causing confusing mixing of their headers. We expect this bug to show up only under high load, and the specific standard of "high" depends on the implementation details of the switch, so it would be easy to write a test suite that accidentally omits any cases to trigger such a bug in certain designs. Furthermore, such cases are challenging even for some styles of formal methods, like bounded model-checking, which can guarantee correctness up to some bound on a number of system execution steps. The reason is that bugs that only show up under high load may require many steps to appear, quite likely above practical bounds for model-checking. Bugs of this kind have been discovered in real systems like NetFPGA and reported in previous work [28].

Fencepost errors. Some classic programming headaches also arise here, including unusual cases where numeric parameters line up just right to make trouble. Take the example of a packet buffer, which stores packet contents while headers are being analyzed and rewritten. What happens if a 1024-bit chunk of packet data is being written concurrently with an attempt by another part of the system to dequeue that chunk (after header-rewriting finished)? Such a race condition might lead to that chunk being missed in the generated egress packet. It requires precise knowledge of the switch's "microarchitecture" to craft a test case to trigger the bug.

Handling error conditions. Another familiar programming challenge is error-handling code, which may be triggered relatively infrequently. For instance, one MAU in a pipeline may detect that a PHV is invalid. It does not drop the packet immediately, as some coordination needs to be done to free the buffer that stores the corresponding packet body. Instead, the packet is pushed onward through the pipeline, with a bit set to indicate that it should be dropped at the end. (Motivations for this implementation choice include saving on duplicate circuitry for full error handling across many MAUs.) If this process is not properly scheduled in the logic throughout MAU stages, the switch may, for example, apply an arithmetic operation intended for a valid packet to this doomed invalid packet instead, even if correct behavior always arises for any flow of valid packets. Again, to trigger this behavior with tests, it does not suffice merely to realize that error handling is worth exercising; test cases must trigger errors with just the right timing, even relative timing across packets.

4.2 VeriSwit Specification

Our goal with exhaustive verification is to prove that a design is free of whole categories of bugs, as sketched above. In formal methods in general, a *specification* is a (system-specific) logical characterization of correct behavior – and here we mean logical formulas written out completely rigorously in text files and ingested by appropriate software formal-verification tools. Any bug category that we are worried about should be ruled out by the specification. The basic strategy is to *design a specification that does not include the complexities that allow those bug categories to arise*, then prove that the implementation mimics the specification on any possible system input (of any length). The greatest source of simplification in our specification is that it will keep concurrency to a minimum, including just what is fundamental to the switch's interface and not anything motivated only by performance considerations. More specifically, the specification we settled on includes queues for ingress and egress packets, but *any packet is processed atomically.* There is no chance for concurrent interference from other packets wending their way through the switch.

We want to emphasize a crucial upshot of this methodology: *the switch may employ sophisticated concurrency, but we prove that it behaves the same as a switch with minimal concurrency.*

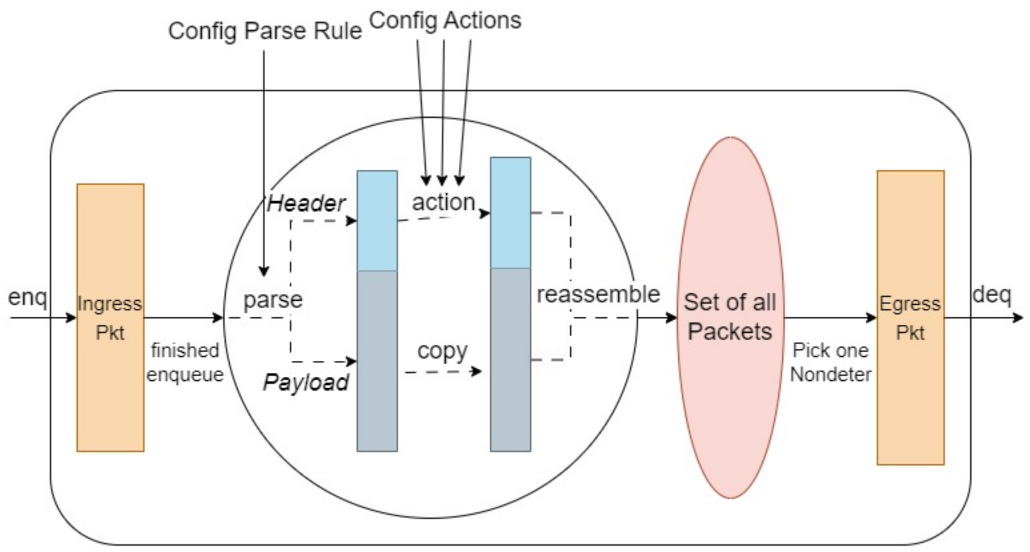


Figure 4.1: VeriSwit's specification, where the ovals delineate the only units of concurrency.

The top-level specification necessarily exposes the same interface of methods as the real switch, but its internal state is greatly simplified (Figure 4.1). The specification holds the configuration tables for both parsers and MAUs, in addition to current ingress and egress packets, plus a list of all other processed packets, waiting to be dequeued. When a packet tries to enter the specification, the enqueued content is concatenated to the end of an unbounded ingress buffer. When the last chunk is enqueued, meaning that the whole ingress packet has entered the switch, those buffer contents are sent to be processed with the policy represented by the tables, in one single step. Invalid packets are discarded immediately and silently. Hence, all concurrency is in maintaining data structures of packets waiting to be processed and packets waiting to be output by the system. The actual complex logic to analyze and rewrite packets is pushed into a single routine that runs uninterrupted on a full packet.

These questions of concurrency granularity are so important, and we will return to them so often, that a visual convention in diagrams will help. Recall Figure 1.1, which shows the optimized VeriSwit design. *Every rounded-corner rectangle in that figure there denotes a kind of concurrent thread that runs alongside the others*, with opportunities for race conditions and other classic concurrency bugs. In contrast, in the specification diagram of Figure 4.1, the only two threads are those drawn within ovals. Most importantly, *all of the complex logic is within the largest oval, where it is able to process a whole packet sequentially, with no interference by other threads.* If VeriSwit is proved to mimic the behavior of its much-less-concurrent specification, then there must not be any lurking concurrency bugs.

Our final theorem covers only the *safety* property that all egress packets have been constructed using the proper logic. We do not yet consider *liveness* properties, like that any

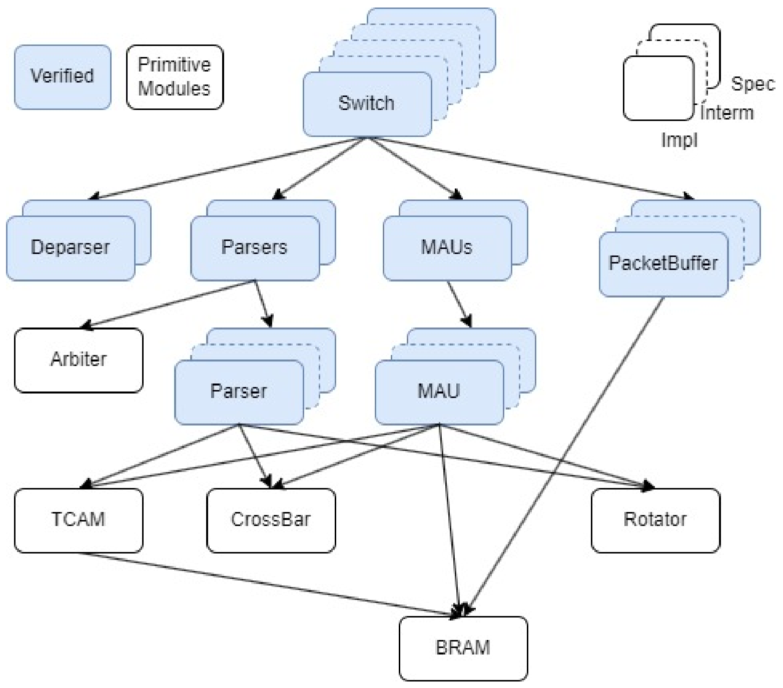


Figure 4.2: Hierarchical decomposition of VeriSwit's proof. "Stacks of boxes coming out of the page" indicate proofs by transitivity through multiple intermediate module designs.

valid ingress packet eventually exits. We also do not prove (only show empirically) that the system meets any particular throughput target. These additional properties seem feasible for future work.

4.3 Structure of VeriSwit's Proof

In theory, one could write out a monolithic simulation relation, connecting the state elements of the top-level specification and the VeriSwit microarchitecture. However, a direct proof against that relation would be extremely intricate. Instead, we break the proof down in two main ways, following the reasoning tools that Fjfj provides.

Recall the transitive property of the refinement relation. Given a sequence of modules, each of which is proved to simulate the next, this sequence may be collapsed with *transitive* reasoning: the first module simulates the last. How does this technical construction relate to the implementation challenges of switches? The first module would be the fully optimized version, and the last module is the specification. Each intermediate module represents a hybrid design including some but not all optimizations. This way, the proof can introduce optimizations gradually, and each simulation relation need only explain why a single optimization is sound.

The second method is more complex to describe, but it is also more central to taming sophisticated designs. We want to decompose proof effort *hierarchically*. For instance, VeriSwit depends on subsystems for parsing, match-action units, deparsing, and buffer management. Each subsystem should have its own specification, proved separately. The crucial modular proof principle of Fjfj says *if VeriSwit is correct when the subsystems are replaced with their own specifications, then the switch with the true subsystem implementations is also correct*. That way, switch-level reasoning is protected from complications of the optimizations within the subsystems. A switch-level refinement relation may be written just in terms of specification-level state of subsystems.

In fact, this hierarchy principle applies throughout levels of our system design. Figure 4.2 shows the decomposition we follow, with hierarchy three levels deep. The promise of this technique is that each box in the diagram may be assigned to a different engineer, who is free to implement new optimizations, which require new proof only at the level of the module directly affected.

Now let us explain the most interesting modules.

4.4 Parser

4.4.1 Parser Implementation

A parser in the PISA model allows the switch to support any arbitrary protocol stack. It does so by acting like a finite state machine, with configurable transition functions from state to state encoded as entries in a TCAM. Starting with a default state, it computes the next state by looking up the value of a specific header within the TCAM. Based on the TCAM response, it puts the fields of the current header at specific locations in the final PHV and advances pointers within the packet and parsing state. This process continues until either the TCAM responds with a flag indicating the end of parsing or some error occurs.

Figure 4.3 diagrams VeriSwit's optimized parser. Submodules with square corners represent register-like interfaces, i.e., read and write operations that are executed instantly. Submodules with rounded corners represent server-like interfaces, where requests flow in and responses flow out, potentially after significant and unpredictable delays. A BRAM is a canonical example of a server-like interface, given practical latencies for memories of nontrivial size. The parser's TCAM is the reconfigurable storage unit that provides programmability, with a method *updateParseTable* available to install changes. However, we do not want to allow the parser to be reprogrammed while it is processing a packet, nor do we want one parser to work on two packets at once. Hence, using the *guard* feature of Bluespec,

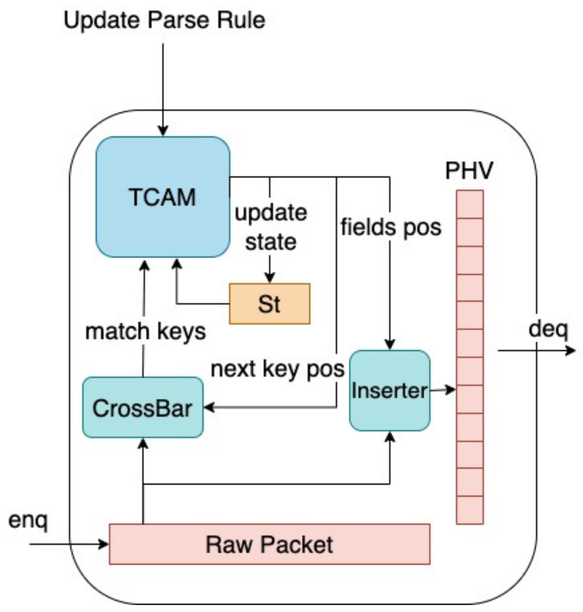


Figure 4.3: VeriSwit parser implementation.

updateParseTable and the input method *enq* are written to *abort* safely, if called while the parsing state machine is active. When the parser is ready to process a packet, the first 1024 bits may be *enq*ed. The rest of the job is pipelined across *rules processTableResponse*, *issueTableRequest*, and *rotate*. Rules are one of the distinctive features of Bluespec, which act somewhat like parallel threads in software. Once the parsing has finished, *parseState* is changed to again permit new packets or reconfiguration.

In order to prevent the parser from running into a nonterminating loop, there is also an iteration counter.

4.4.2 Parser Specification

Here we present VeriSwit's parser specification, highlighting its simplifications from the implementation. Most directly, there are simply fewer state elements in the specification: just the reprogrammable parse table, a flag indicating if parsing is finished, and a list of PHVs ready to be output. Even more importantly, the parser spec has *no internal concurrency*: every packet is parsed instantly with sequential code. The specification distinguishes 5 cases of steps by the parser, considering whether the limit on iterations has been reached, whether a TCAM lookup succeeds, a bit in the TCAM result saying whether parsing is finished, and whether enough bytes of the packet have been read to allow processing the current header.

4.4.3 Parser Proof

The proof uses transitive reasoning via an intermediate implementation that eliminates all the pipeline modules except for the TCAM, which necessarily requires some asynchronous processing due to its unpredictable latency. In the intermediate implementation, both match-key extraction and field insertion are done in one single step, leaving only one rule that fetches TCAM responses, decodes them, and acts accordingly, either ending the process or issuing a new TCAM request for the next iteration.

To reason about pipelines, we rely on a classic technique known from verification of CPUs: *flushing relations*. One could in principle write out an ad-hoc simulation relation for each pipeline. However, one general relation often suffices, when comparing an implementation with a longer pipeline to a specification with only a suffix of those pipeline stages. We say that the longer pipeline is related to the shorter one if all of the shared queues have identical contents, *after letting all additional implementation stages run fully to empty out their input queues*.

The resulting proof obligations center on showing that pieces of logic *commute* with each other. That is, two such logic units must produce identical state when run in either order. We must show that each rule ("parallel thread") of the longer pipeline commutes with its associated *deq* method, as well as with any other rule. For a manual proof collapsing n pipeline stages, it would be quite time-consuming to work through the reasoning for each of the $O(n^2)$ cases, but for a pipelined system, different stages usually touch different parts of the internal modules, hence it is often mostly trivial to show commutativity. We generally develop proof automation that can handle all of the cases using common proof structure.

Flushing can capture most of what makes a pipeline correct, but we do need to write out some parts of a simulation relation manually, such as asserting equivalence of internal state elements like the TCAM.

Applying the general technique to our parser example, we must first show refinement from the implementation to the intermediate implementation with a shorter pipeline. Instead of allowing the flushing relation to flush any rule, we only allow flushing for the rule that matches the key and the rule that inserts the fields in the PHV. Our base state equivalence forces identical response lists in the TCAMs of both modules, other modules like CrossBar and Inserter must have empty responses, plus a few other bookkeeping conditions.

Next, to show refinement from the intermediate implementation to the specification, no flushing relation is used as there is no more pipelining. Instead, the relation asserts that at any time, if we continue to parse what is currently sitting in the registers of the intermediate implementation using the parse function defined in the specification, we would end up with the exact same content as what is already completely parsed by the specification.

4.4.4 Parallizing Parsers

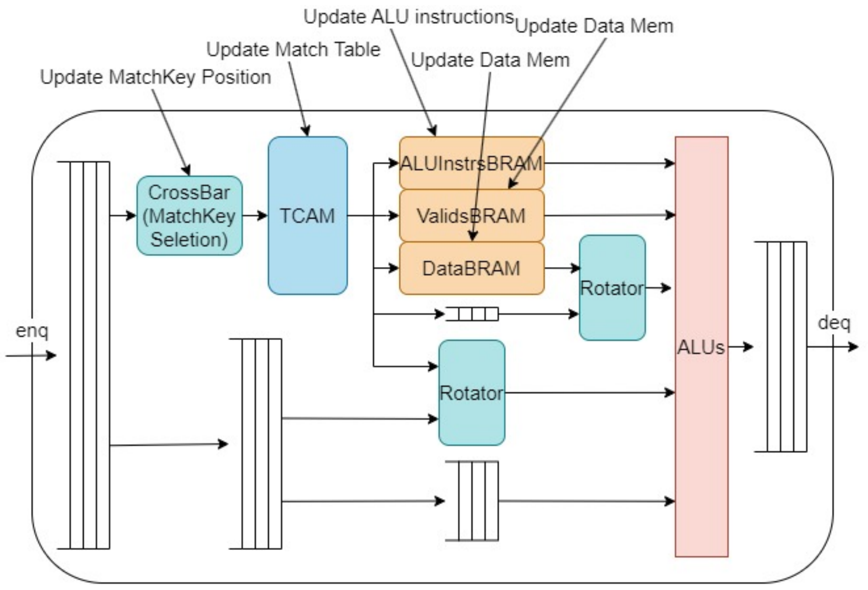
To feed the high throughput of the MAU pipeline, multiple parsers are parallelized together to form a module Parsers. In the implementation, an arbiter is used to resolve conflicts when more than one parser is ready to dequeue. The specification of Parsers is very similar to one instance of Parser, where the only difference is that the dequeue ordering is nondeterministic: any fully parsed packet may be selected to output next. The proof's simulation relation asserts that the concatenation of all the parsed PHVs across parsers in the implementation is a permutation of the parsed PHVs in the specification. Our proof relies heavily on automation specific to permutations.

4.5 Match-Action Units (MAUs)

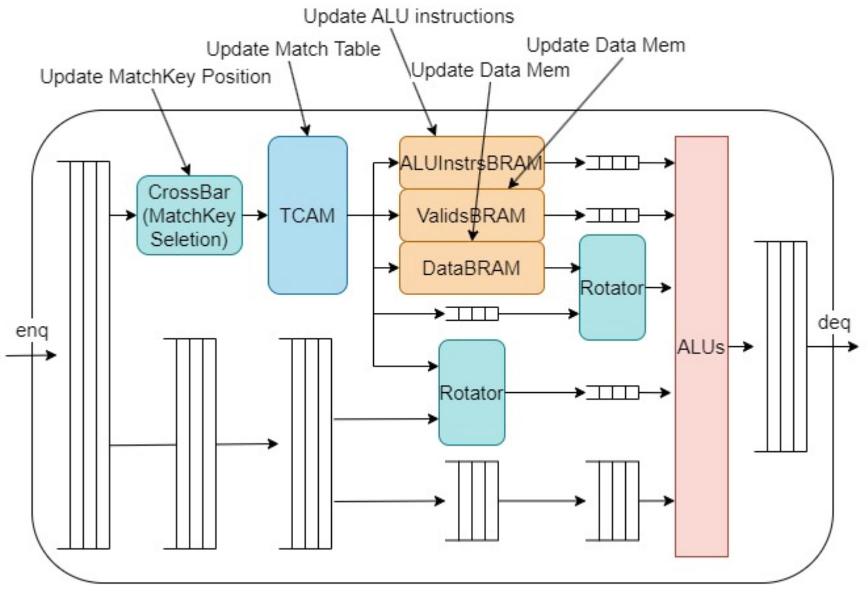
In proving MAUs and sequences of MAUs, we again make heavy use of the technique of pipeline flushing, adopting comprehensive proof automation. In addition, we include the notion of "undefined behavior" when designing the specification for MAU/MAUs, allowing us to remove unnecessary circuit logic that guards against disallowed usage, when we can prove statically that other modules would never ask for disallowed behavior from the one we are proving.

4.5.1 MAU Implementation

Our MAU implementation (Figure 4.4a) is a purely pipelined system that distributes the PHV processing across 5 stages: key extraction, matching, instruction lookup, data forwarding, and ALU. The first stage extracts preconfigured fields from the PHV. The second stage sends the extracted key to match in the TCAM. In the third stage, the TCAM response is fetched and decoded to form BRAM requests to retrieve VLIW (very long instruction word) instructions for the ALUs, PHV valid bits, preconfigured constant data and its corresponding location in the PHV, and the amount of PHV rotation for field copying. The last stage uses all of the above data to instruct the ALUs to update each register in the PHV accordingly and push the result to the output buffer. If the PHV is marked invalid at any time (either when enqueued to the MAU or with a TCAM lookup failure), the PHV is still pipelined through the MAUs in FIFO order, but no request is issued in the computational units.



(a) The real implementation, with some bypassing of pipeline stages



(b) An intermediate implementation used in the proof, with extra queues inserted to synchronize all handling of a given packet

Figure 4.4: MAU implementation

4.5.2 MAU Specification

```
Record MAUConfig := {
  ksInstrs : N;
  tableInstrs : list (N * N * N);
  validsInstrs : N -> N;
  aluInstrs : N -> N;
  memInstrs : N -> N;
}.

Record MAUState := {
  state : MAUConfig;
  resps : list N;
}.
```

Listing 4.1: MAU specification state

In our specification, we only have 5 internal state fields (Listing 4.1) and one undefined-behavior bit. According to the specification, when a valid PHV is enqueued to the MAU, keys are extracted according to *ksInstrs* and are directly fed to search in *tableInstrs* by comparing the keys with the keys stored. If there is a match, the value is used to index *tableInstrs*, *aluInstrs*, and *memInstrs*, and the final computation is done according to their values. The whole matching and computation is done in one single step. After the PHV is processed, it is placed in the output buffer *resps*. If the PHV is marked invalid at any point, it is sent directly to the output buffer while preserving the FIFO order.

Note that VeriSwit would reject any reprogramming request if there are packets in flight. Such a check is and should be handled at the outermost level by checking the contents of the packet buffer, a different module (section 4.6). It would break modularity to allow the MAU specification to mention packet-buffer state, so instead we refer to MAU-specific signs of packets in flight, with guard conditions that mention just that local state. Specifically, the specification sets a bit for undefined behavior when any table is reprogrammed while *resps* is nonempty, and thereafter (as with undefined behavior in the C programming language) *any* behavior of an MAU is legal ("the warranty has been voided"). Higher levels of the VeriSwit proof will establish that, in fact, this condition is never encountered.

4.5.3 MAU Proof

Due to the complexity of pipelining and branching in an MAU, an intermediate implementation is used to simplify the proofs. Note that in the MAU implementation, there are unaligned pipeline stages across different information flows. For example, CrossBar and TCAM serve as two pipeline stages, but there is only one PHV queue corresponding to these two stages. Unnecessary instances of queues waste registers, hence are undesirable in the real implementation. However, this creates a challenge for verification, specifically in characterizing how elements move through pipelines in slightly different flows between versions of a module. Thus in the intermediate implementation, we break every queue and server-like module into multiple stages that it might map to in the other parallel flows, and we created an MAU with aligned pipeline stages across all flows (Figure 4.4b).

In addition to breaking up stages, another "simplification" for verification is additional rules ("parallel threads"). We use this technique to let us perform a top-level case split in a proof, on whether a PHV is valid or not. When proving the correctness from implementation to the intermediate implementation, one only needs to assert that, for every queue in one module that gets broken into multiple in the other module, the concatenated contents of the expanded queues equal contents of the original. In proving each internal rule of the implementation, one would branch on the validity of the first PHV and first rules in the intermediate implementation accordingly. When trying to prove the refinement relation from the intermediate implementation to the specification, a flushing relation is again used to assert that the final queue in the intermediate implementation equals the *resps* queue in the specification, if we allow all the rules in the intermediate implementation to fire and flush all other queues.

4.6 Packet Buffer

The packet-buffer unit stores the content of each whole packet, including both the header and the payload. It assigns a unique identifier to each incoming packet, which is sent down the processing pipeline along with the header and metadata. When the pipeline finishes processing the header, a request to retrieve its payload is sent to the packet buffer with the corresponding identifier. Subsequently, all parts of the packet are returned chunk-by-chunk.

4.6.1 Packet-Buffer Implementation

The packet buffer is implemented in the style of a ring buffer. It uses SRAM to buffer the content of the packet, and it maintains a list of packet metadata such as a start and end pointer of the packet in SRAM and its total size. There are pointers to keep track of the "head" of the buffer for both SRAM and metadata, and every slot of SRAM and metadata is tagged with a free bit to indicate if the slot is free to use. When a packet tries to enter the switch, the enqueuing request for the packet buffer is issued with the packet data. The request is accepted once the availability of the heads at both SRAM and metadata list is asserted, and if either one of the heads at SRAM and metadata list is occupied, the switch is considered full, causing the incoming packet to be rejected. For every chunk of data accepted by the packet buffer, the head pointer for SRAM is incremented. Thus the whole packet is stored consecutively in the buffer. Once the whole packet finishes enqueuing, the corresponding metadata information regarding this packet (e.g. the head and tail location in the SRAM, the total size) is stored at the head of the metadata list, and the head pointer is incremented. The corresponding identifier is precisely the pointer to the metadata list for this packet.

The packet buffer supports random access to the set of packets stored in it. Reading a packet triggers an SRAM read per 1024-bit chunk of the packet. Throughput optimizations include allowing the request to read one chunk and the response returning the previous chunk to be in-flight simultaneously. Some further subtlety is connected to a queue recording packet identifiers chosen to be dropped, which eventually triggers a state machine for freeing all associated packet-buffer resources.

4.6.2 Packet-Buffer Specification

We propose a clean specification for the packet buffer that does not require breaking packets into chunks. This spec behaves like a map from the packet identifier to the full packet contents, or a null value if the slot is unoccupied. It maintains a fresh identifier that can be assigned to incoming packets and does not conflict with stored packets, plus separate state to store incoming but incomplete packets. Once the packet finishes enqueuing, it is stored in the storage map indexed by the unused identifier. The new identifier is then nondeterministically set to any unused identifier.

4.6.3 Packet-Buffer Proof

The proof for the packet buffer is split into two parts. The bottom part unifies the SRAM blocks to eliminate data segmentation, and the top part eliminates the concurrent SRAM requests and response handling.

In the bottom proof that connects the implementation to an intermediate specification, we heavily rely on an invariant assertion about the “nonoverlapping” property of the ranges specified by packet metadata that is stored in the metadata list. The notion of range also includes the case when the tail wraps around the edge of the SRAM. This invariant allows us to assert that when certain parts of the SRAM are modified, all other parts are left untouched. In addition, it is also crucial to relate the segmented packet in SRAM with its full packet represented in the intermediate specification, considering the offset of the trailing bits and the availability of the slots.

In the top proof that connects the intermediate specification with the final specification for the packet buffer, the response queue is eliminated, along with the concurrent logic that handles queue responses. It is also required to concatenate the segmented data in the SRAM's response queue and map to its full representation.

4.7 Overall Switch

Finally, we present the top-level proof for VeriSwit, with respect to the proven specifications for submodules such as parsers and MAUs. That is, in this capstone proof of the project, we may assume that each optimized component is replaced by its specification, because the simulation proof for each component justifies that reasoning. Before diving into proof details, we quickly recap the optimized design and the specification.

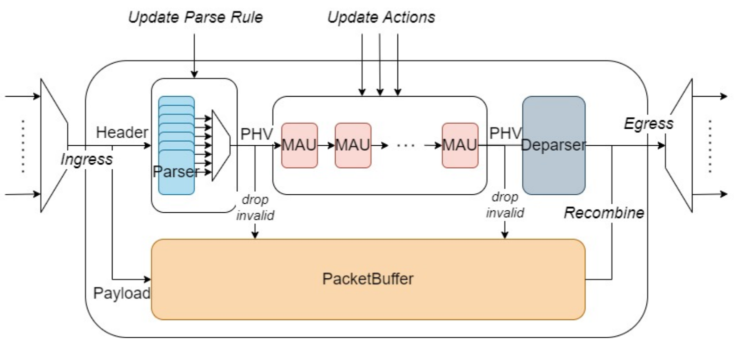
In VeriSwit (Figure 1.1), when a packet arrives, its header is separated from the payload and sent to one of the several *parsers* that run in parallel. A parser reads the raw content in the header and generates structured data to pass on to *match-action units (MAUs)*, which run as a pipeline to perform different header transformations. Afterward, the *deparser* patches the header changes into original packets. The *packet buffer* stores the original packet payload throughout its lifetime in the switch.

In the specification (Figure 4.1), the header transformations are done in one single computational step, and the payload moves along with its header in this process. When a packet finishes enqueuing, it is directly manipulated according to the configured policies, then buffered into a set of all processed packets, waiting to be dequeued. One packet is then chosen nondeterministically from the buffer to be the egress packet, outputting to the outside.

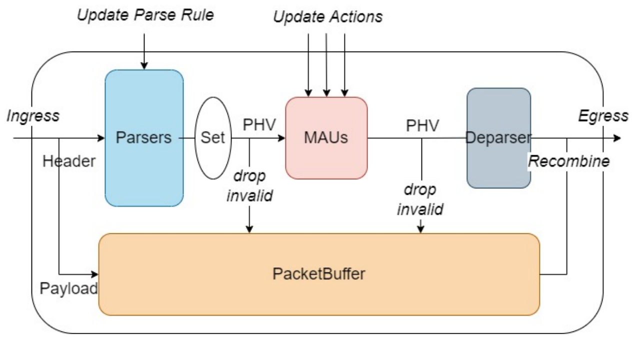
4.7.1 Top-Level Proof

This proof is split by transitivity into four steps (illustrated with Figure 4.5), where each eliminates one or more chosen architectural optimizations to ease verification later in the process. The bottom 3 layers are rewritten in Fjfj and look like synthesizable code, while the top 2 layers are specified as logical Coq functions. The transitive reasoning proceeds through the following intermediate designs.

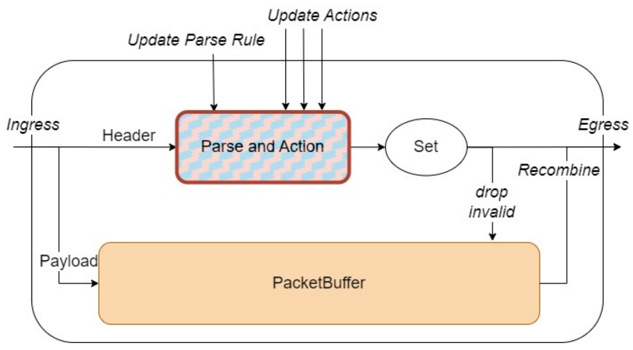
1. The first intermediate design actually combines three consequential simplifications,



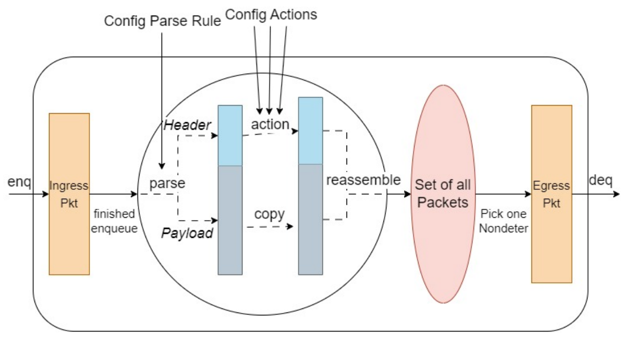
(a) Real implementation



(b) Simplify checking of when reconfiguration is permitted, simplify error handling, reduce deparser nondeterminism



(c) Collapse all handling of a packet into an atomic step



(d) Remove packet buffer, instead flowing packet bodies throughout system

Figure 4.5: The four steps of the top-level VeriSwit proof

which happen to be orthogonal enough to each other that it is roughly as easy to prove them in one step as in several.

   (a) The system must block reconfiguration requests when packets are in-flight, to avoid inconsistent processing. The necessary condition for safe reconfiguration is that all places packets might be stored within the system are empty. However, it is expensive to check all of those places on every reconfiguration request. Instead, we prove that it suffices just to check if the packet buffer is nonempty.
   (b) Our specifications allow some nondeterminism in the order of packet processing. We add an extra rule to force some nondeterminism to be resolved earlier in the deparser, dequeuing a packet to stash it in a register, from where it will later be dequeued.
   (c) To avoid duplicating error-handling logic throughout a circuit, VeriSwit continues to move packets through its pipelines after they are found invalid. Here we change that behavior, performing immediate drops of invalid packets.

2. The next intermediate design collapses the packet-processing pipeline so that all parsing, match-action application, and deparsing happens atomically for a packet, much like in the top-level specification. Due to the introduction of nondeterminism at the pipeline head by parsing, we had to apply a careful variant of the flushing technique for proving simulation, allowing the states being related to differ by permutation of queue contents.
3. Finally, our last intermediate differs from the specification only in eliminating packet-buffering logic and state, instead passing full packet contents through the process as needed. However, packet-buffer memory is retained in the design but not relied on in all the same places. The flushing-based proof of this refinement involves some care in choosing which rules to allow to fire during flushing (specifically avoiding the one that introduces nondeterminism).
4. We prove the last refinement to the top-level specification, where we must remove packet-buffer memory altogether and delay processing of any given packet until all of its data has arrived.

Chapter 5

Evaluation

In this section, we answer the following three questions about VeriSwit. The first two are to confirm that we chose a sufficiently realistic design to verify (as our research focus is on proofs, not novel switch implementation).

1. What is the *architectural* performance of VeriSwit, in gigabits per clock cycle, for different configurations and workloads (section 5.1)?
2. What clock periods can VeriSwit achieve both on FPGA and in ASIC, to get a throughput performance number in gigabits per seconds (section 5.2)?
3. How much effort was it to prove the correctness of VeriSwit (section 5.3)?

Methodology Our verified codebase of VeriSwit in Coq aligns closely with an implementation of VeriSwit in Bluespec SystemVerilog [33]. The design evaluated consists of about three thousand lines of BSV, parameterized by the numbers of MAUs and parsers. We compiled the Bluespec code into Verilog using the bsc compiler[1]. The resulting Verilog was then separately synthesized for an FPGA, processed through an ASIC flow, and simulated with Verilator [35]. Our evaluation primarily focuses on two configurations: (2P,2MAU) and (8P,8MAU), with reported results pertaining to the (8P,8MAU) configuration unless specified otherwise.

The performance metrics in our architectural performance analysis are derived from Verilator simulations. This method was chosen to eliminate potential bottlenecks associated with transmitting a generated workload to VeriSwit running on an FPGA via PCIe, which is constrained by the limited bandwidth (a few GB per second) of the Connectal framework[2]. Therefore, by using Verilator simulations, we ensure that the throughput measurements are not affected by communication bottlenecks with the FPGA.

[1]https://github.com/B-Lang-org/bsc
[2]https://github.com/cambridgehackers/connectal

5.1 Architectural Performance

The overall throughput of the switch in Gbps is a product of the throughput per cycle multiplied by the clock frequency. We first report the first factor and how it varies with different parameters for different workloads, as it is independent of the synthesis target.

Throughput Characterization To quantify the variability of the performance of VeriSwit, we first design a synthetic test suite parameterized by the depth of the parsing tree (the maximum number of headers of packets) and the size of the ingress packet. In each test iteration, we configure the parse table so that the given number of headers is read from the packet, and we set all the MAUs to pass the PHV on without modification. As our MAUs always process 1 packet per cycle, this simplification in the benchmark should not affect the throughput of the system (confirmed in the next paragraph). Then we start sending packets of the given length, recording the average throughput per cycle.

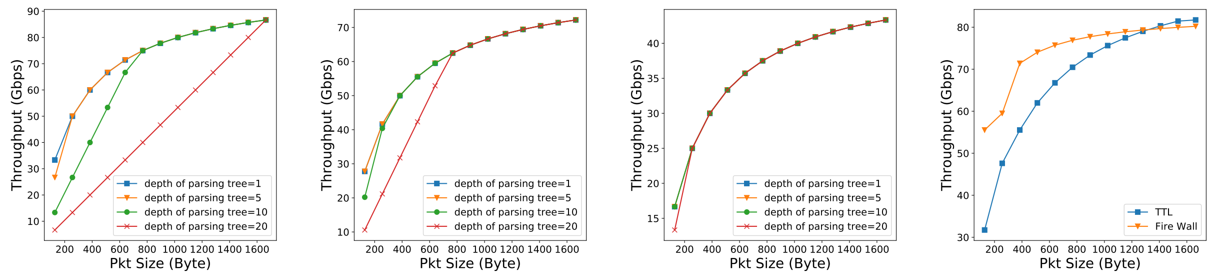
We run the above test cases against a few different design parameters and report the results in Figure 5.1a, Figure 5.1b, and Figure 5.1c. These throughput numbers are scaled with the FPGA clock rates we report on in the next section. Though our work is oriented toward eventual ASIC fabrication, we already nearly saturate 100 Gb/s line rate with the FPGA clock rate. Adding more parsers does decrease the throughput ceiling thanks to additional coordination logic, but again we believe that ceiling would be pushed back above 100 Gb/s thanks to the constant factor of performance that typically follows going from FPGAs to ASICs.

Realistic Policies For an integration test and evaluation, we put VeriSwit to the test in simulation against a set of real network policies summarized in Table 5.1. In this way, we both confirm the expressivity of VeriSwit as a programmable switch and evaluate the performance of VeriSwit on those policies. Our policy-specific throughput observations line up well with our estimates from the last section (Figure 5.1).

5.2 Synthesis

We now report on evaluating frequency and resource usage for different configurations on both FPGA and ASIC targets.

FPGA We synthesized the Verilog produced by the Bluespec compiler down to FPGA using Vivado 2019.1, targeting a Virtex UltraScale+ XCVU9P-L2FLGA2104 FPGA on a



(a) 4 Parsers (b) 8 Parsers (c) 16 Parsers (d) TTL and Firewall

Figure 5.1: Throughput graphs. Figs. (a), (b), and (c) are the results for the generic test suites with 4, 8, and 16 parsers, respectively, running in simulation. The throughput is calculated as the average number of bits ingested per cycle in simulation multiplied by the highest clock frequency achieved on FPGA. Fig (d) shows the throughput in the case of packet dropping for the TTL and Firewall policies (Policies C and F). The drop rates are 25% in C and 50% in F.

VCU118 board. For testing the design, we communicate with the host machine using Connectal. We report resource usage of the synthesized bitstreams for a few parameters in Figure 5.2. In term of resources, we highlight the heavy use of BRAMs: the (8P,8MAU) configuration requires about 3.2 MB (572*36Kb + 282*18Kb) of on-chip memory. We successfully synthesized a small configuration (2 Parsers, 2 MAU) of VeriSwit at 100MHz, and we successfully synthesized the ideal configuration design we propose (8 Parsers, 8 MAUs) at 83.3MHz. The slightly lower frequency is likely due to routing congestion as we are reaching high occupancy on the FPGA. We had access just to a smaller VCU108 FPGA, which we synthesized for separately and then ran some simple tests on, to confirm end-to-end proper behavior and performance.

(# MAUs,# Parsers)	Freq	LUT	FF	BRAMs
(2,2)	100	183K	98K	(172, 76)
(8,8)	88	462K	347K	(572, 282)

Figure 5.2: Clock frequency (MHz) and resource usage (Lookup Tables, Flipflops, and BRAMs (36Kb blocks and 18Kb blocks)) for a Xilinx VCU118 board, for several VeriSwit configurations.

ASIC To preview how VeriSwit might synthesize for silicon fabrication, we used the academic open-source Process Design Kit FreePDK 45nm [36] for ASIC logic synthesis. Due to the lack of available open SRAM macros for this PDK, we substituted the memories with black boxes. The post-synthesis logic area estimate reported is 4.3mm^2, for a total of 2.5M cells (2M combinational cells and 500K sequential cells). This is without counting the SRAM

ID	Policy Description	#instructions
A	Uncond. forwarding to a port	15
B	Forwarding based on dest. addr.	42
C	Dropping from a src. addr.	34
D	Uncond. header insertion	41
E	Uncond. header removal	23
F	TTL decr. and discard if 0	37

Table 5.1: List of policies

areas that would need to be added. The post-synthesis logic passes the timing analysis with a clock speed of 400MHz (a max throughput of about 400 Gbps), giving us confidence that with access to commercial PDKs of similar technologies and a memory compiler, we can readily surpass a total bandwidth of 100Gbps post-place and route.

5.3 Code Statistics

We summarize the size of the codebase (Table 5.2). We include 4 main components and report the number of lines of code in Bluespec implementation (synthesizable HDL code), Fjfj (embedded HDL in Coq for proof), and their specification and proofs. The "MAUs" line may stand out for including more lines of specification code than implementation; here the restatement of some code as logical formulas happens to be in a more verbose style, though such specifications are significantly easier to reason about.

Table 5.2: Lines of code per module/type

Module	BSV	Fjfj Impl.	Fjfj Spec.	FjFj Proof
Parsers	246	266	203	2575
MAUs	258	257	513	2778
PacketBuffer	172	238	163	5229
Switch	149	285	170	7985

Chapter 6

Conclusion

VeriSwit demonstrates that it is feasible to carry out an unbounded mathematical proof of correctness for a programmable switch. Our Bluespec implementation nearly achieves 100 Gb/s line rate running on an FPGA (so hitting that target in an ASIC version should be straightforward), showing an important aspect of realism for a switch. The machine-checked Coq proof then applies two main tools to structure the proof and control human effort: hierarchical decomposition following module structure and transitive proof via intermediate module designs. These ideas should generalize to other kinds of networking hardware (including more full-featured switches), which certainly make for a fruitful future-work direction.

Appendix A

Appendix

```
Record NetworkPkt := {
  PktData : N;
  PktSize : N;
  PktPort : N;
}.

Record SwitchState := {
  parseTable : list (N * N * (ParserSpec.ParseTableValue));
  matchStates : list MAUState;

  ingressPkt : NetworkPkt;
  resps : list NetworkPkt;
  egressPkt : NetworkPkt;
}.
```

Listing 1: Datatypes of Top-Level Switch Specification

```

(* synthesize *)
module mkParser(Parser#(1));
  TCAM_R#(SizeOf#(ParseTableKey), SizeOf#(ParseTableValue),
     ParseTableSz) parseTable <- mkParserTCAM;
  CrossBar#(6,2,8) ks <- mkCrossBar64To4;
  Rotater#(7, 8) rot <- mkRotater128;
  Meta meta = unpack(0);

  Reg#(ParseState) parseState <- mkReg(Ready);
```

```
Reg#(Bit#(1024)) stream <- mkReg(0);
Reg#(Bit#(8)) pktOffset <- mkReg(0);
Reg#(Bit#(8)) bodyOffset <- mkReg(0);
FIFO#(Bit#(8)) states <- mkFIFO();
Reg#(Bit#(8)) countDown <- mkReg({'1});

Reg#(Vector#(128, Bit#(8))) phv <- mkReg(unpack(0));
Reg#(Vector#(128, Bool)) valids <- mkReg(unpack(0));
Reg#(Bool) valid <- mkReg(True);
Reg#(Bit#(8)) index <- mkReg(0);

rule processTableResponse
  if (parseState == Processing);
  let resp <- parseTable.deq_resp_r();
  if(countDown == 0)
  begin
    parseState <= Done;
    valid <= False;
  end
  else
  begin
    case (resp.value) matches
    tagged Valid .rr:
    begin
      ParseTableValue r = unpack(rr);
      if(r.done)
      begin
        parseState <= Done;
      end
      else if({'0, r.size} > pktOffset)
      begin
        parseState <= Done;
        valid <= False;
      end
      else
      begin
        Bit#(1024) mask = ~({'1} >> {r.size, 3'b0});
        Bit#(128) maskV = (~({'1} >> r.size)) >> r.contOffset;
```

```

          Bit#(1024) header = stream & mask;
          rot.enq(unpack(header), (0)-unpack(truncate(r.contOffset)))
             ;

          stream <= stream << {r.size, 3'b0};
          pktOffset <= pktOffset - {'0, r.size};
          bodyOffset <= bodyOffset + {'0, r.size};
          valids <= unpack((pack(valids) | maskV));

          ks.enq(unpack(truncateLSB(header)), r.matchData);
          states.enq(r.state);
          countDown <= countDown - 1;
        end
      end
      tagged Invalid:
      begin
        valid <= False;
        parseState <= Done;
      end
      endcase
    end
  endrule

  rule issueTableRequest
    if (parseState == Processing);
    let data <- ks.deq();
    states.deq();
    ParseTableKey key = ParseTableKey {
      state: states.first,
      data: data
    };
    parseTable.enq_req_r(TCAMRequest {
      tkey: TernaryKey {
        key: pack(key),
        dc: {'1}
      },
      value: 0
    });
  endrule

  rule rotate
    if (parseState == Processing);
    let data <- rot.deq();
    phv <= unpack((pack(phv) | pack(data)));
  endrule

  method Action updateTable(ParseTableKey key, ParseTableKey dc,
     ParseTableValue value) if (parseState == Ready);
    parseTable.enq_req_w(TCAMRequest {
      tkey: TernaryKey {
        key: pack(key),
        dc: pack(dc)
      },
      value: pack(value)
    });
  endmethod

  method Action enq(ParserRequest pkt) if (parseState == Ready);
    let new_size = pkt.size <= 128 ? pkt.size : 128;
    countDown <= {'1};
    parseState <= Processing;
    pktOffset <= new_size;
    bodyOffset <= 0;
    Vector#(128, Bit#(8)) _phv = unpack(0);
    _phv[meta.inPortPos] = pkt.p;
    stream <= (pkt.pkt >> {(128 - new_size), 3'b0 }) << {(128 -
       new_size), 3'b0 };
    phv <= _phv;
    Vector#(128, Bool) _valids = unpack(0);
    // _valids[meta.inPortPos] = True;
    valids <= _valids;
    valid <= True;
    index <= pkt.index;
    parseTable.enq_req_r(TCAMRequest {
      tkey: TernaryKey {
        key: unpack(0),
        dc: {'1}
      },
      value: 0
    });
  endmethod

  method ActionValue#(PHVData) deq()
    if (parseState == Done && (!rot.notEmpty) && (!ks.notEmpty));
    // $display("(%d)deq", cnt);
    parseState <= Ready;
    return PHVData {
      phv: phv,
      valids: valids,
      valid: valid,
      index: index,
      bodyOffset: bodyOffset
    };
  endmethod

   method PHVData first()
    if (parseState == Done && (!rot.notEmpty) && (!ks.notEmpty));
    return PHVData {
      phv: phv,
      valids: valids,
      valid: valid,
      index: index,
      bodyOffset: bodyOffset
    };
  endmethod

  method Bool ready();
    return (parseState == Ready);
  endmethod

  method Bool done();
    return (parseState == Done);
  endmethod
```

```

endmodule
```

Listing 2: Parser Implementation in Bluespec

```
Module ParserSpec.
  Record ParserState :=
  {
    parseTable : list (N * N * N);
    resps : list N;
    isDone : bool;
  }.

  Notation "'NewParserState' parseTable resps done" :=
  {|
    parseTable := parseTable;
    resps := resps;
    isDone := done;
  |} (at level 100).

  Definition updateParseTable (arg : N) (st : SModule) (new_st :
     SModule) : Prop :=
    exists (st' : ParserState) (new_st' : ParserState)
    parseTable parseTable' resps done,
    st = *( st' )* /\ new_st = *( new_st' )* /\
    st' = NewParserState parseTable resps done /\
    new_st' = NewParserState parseTable' resps done /\
    dlet {key dc value} := arg in
    parseTable' = (parseTable) ++ [(dc, (N.lor key dc), value)] /\
    resps = [].
  Arguments updateParseTable arg st / new_st.

  Definition enq (arg : N) (st : SModule) (new_st : SModule) : Prop
     :=
    exists (st' : ParserState) (new_st' : ParserState)
    parseTable resps' resps done ,
    dlet {pkt size index port} := arg in
    st = *( st' )* /\ new_st = *( new_st' )* /\
    st' = NewParserState parseTable resps done /\
    new_st' = NewParserState parseTable resps' done /\
    let new_size := (set_size size) in
    let '(phv', valids', valid', bodyOffset') := parse parseTable (
       ParserSpec.correct_header_offset pkt 1024 new_size ) new_size
```

```
      port in
  resps' = resps ++ [{# phv' valids' valid' bodyOffset' index} ]
  .
Arguments enq arg st / new_st.

Definition deq_first (arg : N) (st : SModule) (new_st : SModule) (
   ret : N) : Prop :=
exists (st' : ParserState) (new_st' : ParserState)
    parseTable resps resps' resp done,
st = *( st' )* /\ new_st = *( new_st' )* /\
st' = NewParserState parseTable resps done /\
new_st' = NewParserState parseTable resps' false /\
resps = (resp :: resps') /\
ret = resp
.
Arguments deq_first arg st / new_st ret.

Definition deq (arg : N) (st : SModule) (new_st : SModule) : Prop
   :=
exists ret, deq_first arg st new_st ret.
Arguments deq arg st / new_st.

Definition first (arg : N) (st : SModule) (ret : N) : Prop :=
exists new_st, deq_first arg st new_st ret.
Arguments first arg st / ret.

Definition ready (arg : N) (st : SModule) (ret : N) : Prop :=
exists (st' : ParserState)
    parseTable resps done,
st = *( st' )* /\
st' = NewParserState parseTable resps done /\
ret = match resps with | [] => 1 | _ => 0 end.
Arguments ready arg st / ret.

Definition done (arg : N) (st : SModule) (ret : N) : Prop :=
exists (st' : ParserState)
    parseTable resps done,
st = *( st' )* /\
  st' = NewParserState parseTable resps done /\
  ret = N.b2n done.
  Arguments done arg st / ret.

  Definition updateDone (st : SModule) (new_st : SModule) : Prop :=
  exists (st' : ParserState) (new_st' : ParserState)
      parseTable resps done',
  st = *( st' )* /\
  new_st = *( new_st' )* /\
  st' = NewParserState parseTable resps false /\
  new_st' = NewParserState parseTable resps done'.
  Arguments updateDone st / new_st.

  Definition spec : spec_module_t :=
  {|
    value_spec := list_to_array unexisting_vmethod [first; ready;
       done];
    action_spec := list_to_array unexisting_amethod [updateParseTable
       ; enq; deq];
    rule_spec := [updateDone];
    subrules_spec := [];
  |}.
  Global Instance mkParser : module spec :=
  primitive_module#(rules [updateDone] vmet [first; ready; done ]
     amet [updateParseTable; enq; deq]).

End ParserSpec.
```
Listing 3: Coq example code

References

[1] M. LaPedus, "Big trouble at 3nm," *Semiconductor Engineering*, Jun. 21, 2018. URL: https://semiengineering.com/big-trouble-at-3nm/.

[2] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: A platform for high-level parametric hardware specification and its modular verification," in *ICFP'17: Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming*, Oxford, UK, Sep. 2017. URL: http://adam.chlipala.net/papers/KamiICFP17/.

[3] G. Klein, K. Elphinstone, G. Heiser, *et al.*, "SeL4: Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09, 10.1145/1629575.1629596, Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220, ISBN: 9781605587523. URL: https://ts.data61.csiro.au/publications/nicta_full_text/3783.pdf.

[4] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "CertiKOS: An extensible architecture for building certified concurrent OS kernels," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16, Savannah, GA, USA: USENIX Association, 2016, pp. 653–669, ISBN: 9781931971331. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

[5] S. Zhang and S. Malik, "SAT based verification of network data planes," in *Automated Technology for Verification and Analysis*, D. Van Hung and M. Ogawa, Eds., Cham: Springer International Publishing, 2013, pp. 496–505, ISBN: 978-3-319-02444-8.

[6] A. Guha, M. Reitblatt, and N. Foster, "Machine-verified network controllers," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13, Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 483–494, ISBN: 9781450320146. DOI: 10.1145/2491956.2462178. URL: https://doi.org/10.1145/2491956.2462178.

[7] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "P4v: Practical verification for programmable data planes," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18, Budapest, Hungary: Association for Computing Machinery, 2018, pp. 490–503, ISBN: 9781450355674. DOI: 10.1145/3230543.3230582. URL: https://doi.org/10.1145/3230543.3230582.

[8] R. Peterson, E. H. Campbell, J. Chen, N. Isak, C. Shyu, R. Doenges, P. Ataei, and N. Foster, "P4cub: A little language for big routers," in *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2023, Boston, MA, USA: Association for Computing Machinery, 2023, pp. 303–319, ISBN: 9798400700262. DOI: 10.1145/3573105.3575670. URL: https://doi.org/10.1145/3573105.3575670.

[9] M. Brown, A. Fogel, D. Halperin, V. Heorhiadi, R. Mahajan, and T. Millstein, "Lessons from the evolution of the Batfish configuration analysis tool," in *Proceedings of the ACM SIGCOMM 2023 Conference*, ser. ACM SIGCOMM '23, New York, NY, USA: Association for Computing Machinery, 2023, pp. 122–135, ISBN: 9798400702365. DOI: 10.1145/3603269.3604866. URL: https://doi.org/10.1145/3603269.3604866.

[10] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2013.

[11] T. Bourgeat, "Specification and verification of sequential machines in rule-based hardware languages," Ph.D. dissertation, Massachusetts Institute of Technology, 2023.

[12] Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986. DOI: 10.1109/TC.1986.1676819.

[13] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, ACM, 2001, pp. 530–535. DOI: 10.1145/378239.379017. URL: https://doi.org/10.1145/378239.379017.

[14] K. L. McMillan, "Interpolation and SAT-based model checking," in *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, W. A. H. Jr. and F. Somenzi, Eds., ser. Lecture Notes in Computer Science, vol. 2725, Springer, 2003, pp. 1–13. DOI: 10.1007/978-3-540-45069-6_1. URL: https://doi.org/10.1007/978-3-540-45069-6_1.

[15] A. R. Bradley, "SAT-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, R. Jhala and D. A. Schmidt, Eds., ser. Lecture Notes in Computer Science, vol. 6538, Springer, 2011, pp. 70–87. DOI: 10.1007/978-3-642-18275-4_7. URL: https://doi.org/10.1007/978-3-642-18275-4_7.

[16] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Inf. Comput.*, vol. 98, no. 2, pp. 142–170, 1992. DOI: 10.1016/0890-5401(92)90017-A. URL: https://doi.org/10.1016/0890-5401(92)90017-A.

[17] W. A. H. Jr., "Microprocessor design verification," *J. Autom. Reason.*, vol. 5, no. 4, pp. 429–460, 1989. DOI: 10.1007/BF00243132. URL: https://doi.org/10.1007/BF00243132.

[18] C. Brock and W. Hunt, "Formally specifying and mechanically verifying programs for the Motorola complex arithmetic processor DSP," in *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, 1997, pp. 31–36. DOI: 10.1109/ICCD.1997.628846.

[19] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *Computer Aided Verification*, D. L. Dill, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 68–80, ISBN: 978-3-540-48469-1.

[20] R. E. Bryant, "Formal verification of pipelined Y86-64 microprocessors with UCLID5," Technical Report CMU-CS-18-122, Tech. Rep., 2018.

[21] K. L. McMillan, "Verification of an implementation of Tomasulo's algorithm by compositional model checking," in *International Conference on Computer Aided Verification*, Springer, 1998, pp. 110–121.

[22] K. L. McMillan, "A methodology for hardware verification using compositional model checking," *Science of Computer Programming*, vol. 37, no. 1-3, pp. 279–309, 2000.

[23] B. A. Brady, R. E. Bryant, S. A. Seshia, and J. W. O'Leary, "ATLAS: Automatic term-level abstraction of RTL designs," in *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, Jul. 2010, pp. 31–40. DOI: 10.1109/MEMCOD.2010.5558624.

[24] B. A. Brady, R. E. Bryant, and S. A. Seshia, "Learning conditional abstractions," in *2011 Formal Methods in Computer-Aided Design (FMCAD)*, Oct. 2011, pp. 116–124.

[25] S. Berezin, A. Biere, E. Clarke, and Y. Zhu, "Combining symbolic model checking with uninterpreted functions for out-of-order processor verification," in *Formal Methods in Computer-Aided Design*, G. Gopalakrishnan and P. Windley, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 369–386, ISBN: 978-3-540-49519-2.

[26] S. A. Seshia and P. Subramanyan, "UCLID5: Integrating modeling, verification, synthesis and learning," in *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, 2018, pp. 1–10. DOI: 10.1109/MEMCOD.2018.8556946.

[27] K. L. McMillan, "The smv system," in *Symbolic Model Checking*. Boston, MA: Springer US, 1993, pp. 61–85, ISBN: 978-1-4615-3190-6. DOI: 10.1007/978-1-4615-3190-6_4. URL: https://doi.org/10.1007/978-1-4615-3190-6_4.

[28] P. Bressana, N. Zilberman, and R. Soulé, "Finding hard-to-find data plane bugs with a PTA," in *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '20, Barcelona, Spain: Association for Computing Machinery, 2020, pp. 218–231, ISBN: 9781450379489. DOI: 10.1145/3386367.3431313. URL: https://doi.org/10.1145/3386367.3431313.

[29] F. Ruffy, T. Wang, and A. Sivaraman, "Gauntlet: Finding bugs in compilers for programmable packet processing," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, Nov. 2020, pp. 683–699, ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/ruffy.

[30] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17, Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 155–168, ISBN: 9781450346535. DOI: 10.1145/3098822.3098834. URL: https://doi.org/10.1145/3098822.3098834.

[31] R. Beckett and R. Mahajan, "Putting network verification to good use," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, ser. HotNets '19, Princeton, NJ, USA: Association for Computing Machinery, 2019, pp. 77–84, ISBN: 9781450370202. DOI: 10.1145/3365609.3365866. URL: https://doi.org/10.1145/3365609.3365866.

[32] A. Shukla, K. Hudemann, Z. Vági, L. Hügerich, G. Smaragdakis, A. Hecker, S. Schmid, and A. Feldmann, "Fix with P6: Verifying programmable switches at runtime," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021, pp. 1–10. DOI: 10.1109/INFOCOM42981.2021.9488772.

[33] R. S. Nikhil, “Bluespec System Verilog: Efficient, correct RTL from high level specifications,” in *Proceedings of the Second ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, ser. MEMOCODE ’04, Washington, DC, USA: IEEE Computer Society, 2004, pp. 69–70, ISBN: 0-7803-8509-8. DOI: 10.1109/MEMCOD.2004.1459818. URL: https://doi.org/10.1109/MEMCOD.2004.1459818.

[34] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, “Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN,” *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 99–110, Aug. 2013, ISSN: 0146-4833. DOI: 10.1145/2534169.2486011. URL: https://doi.org/10.1145/2534169.2486011.

[35] Veripool, *Verilator*, https://www.veripool.org/wiki/verilator.

[36] N. C. S. University, *Freepdk45*, https://www.eda.ncsu.edu/wiki/FreePDK45:Contents.