Towards Verifying Correctness and Timing Properties for Out-of-Order Machines

by

Kosi Nwabueze

SB, Electrical Engineering and Computer Science, MIT, 2025.

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

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2026

© 2026 Kosi Nwabueze. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Kosi Nwabueze
Department of Electrical Engineering and Computer Science
May 15, 2026

Certified by: Adam Chlipala
Professor of Computer Science, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Towards Verifying Correctness and Timing Properties for Out-of-Order Machines

by

Kosi Nwabueze

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2026 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

High-performance processors rely on out-of-order execution, an instruction-scheduling algorithm that is both difficult to implement correctly and prone to timing side-channel attacks without principled defenses. This thesis develops an approach using the Rocq proof assistant for specifying and proving timing-security properties of hardware circuits. To express hardware circuits in a proof-friendly style, the thesis introduces Slate, a shallowly embedded hardware-description language built on state monads. Slate provides an imperative-programming model to describe circuits compactly in Gallina. Since Slate circuits are Gallina functions, reasoning about them in proofs is simple. The timing-security framework is exercised on a case study of a variable-latency multiplier. The thesis also presents Golemma, a deeply embedded hardware language in Rocq, to establish correctness results for key out-of-order processor submodules. Golemma provides facilities for modularity and exposes a sequential execution model that makes functional-correctness reasoning simple. The functional-correctness framework is exercised on a case study of a ring buffer. Together, these contributions provide a foundation for progression towards integrated proofs of correctness and timing security for full out-of-order processors.

Thesis supervisor: Adam Chlipala
Title: Professor of Computer Science

Acknowledgments

I would like to thank the following people for their encouragement, support, and mentorship throughout my master's thesis.

- **Adam and Mengjia**, my research advisors, for their guidance, insights, and support throughout this project. Their expert knowledge in both formal methods and computer architecture research made this work possible.
- **Shixin**, for her mentorship throughout my time as a UROP and MEng student in the MATCHA lab. I have learned an enormous amount about both computer architecture and academia from working alongside her over the past two years.
- **Stella, Elvis, and Andres**, for their collaboration on this project and many valuable technical and design discussions over the past year. Without their guidance, this project would not be where it is today.
- **Dad, Mom, Kamsi, Kamto, and Kobi**—my family—for their unconditional love and emotional encouragement throughout my time as a student at MIT.

Contents

List of Figures 6

List of Tables 8

1 Introduction 9

1.1 Background: out-of-order execution . . . 9
1.2 Goal: verifying hardware circuits . . . 11
1.3 Individual contributions . . . 11

2 Related work 13

2.1 Correctness . . . 13
2.2 Security . . . 14

3 Verifying timing security for circuits 15

3.1 State machines . . . 15
3.2 Encoding circuits with Slate . . . 17
3.2.1 Shallow and deep embeddings . . . 17
3.2.2 State monads and actions . . . 18
3.2.3 Syntax . . . 21
3.3 Specifying and proving timing security . . . 23
3.3.1 Cryptographic constant time . . . 23
3.3.2 Hardware constant time . . . 25
3.3.3 Writing proofs . . . 26

4 Verifying correctness for circuits 30

4.1 Encoding circuits with Golemma . . . 30
4.1.1 Interfaces . . . 30
4.1.2 Modules . . . 31
4.1.3 Methods . . . 31

4.1.4 Expressions 32
4.1.5 Statements 33
4.2 Extending Golemma with type classes 33
4.2.1 Synthesizable types 33
4.2.2 Structures 34
4.2.3 Enumerables 35
4.2.4 Bitflags 35
4.2.5 Metaprogramming 36
4.2.6 For loops 37
4.3 Specifying and proving correctness 37
4.3.1 Writing specifications 38
4.3.2 Writing proofs 41

5 Case studies **46**
5.1 Timing security: Variable-latency multiplier 46
5.1.1 Interface 46
5.1.2 Erasure contract 48
5.1.3 Implementation 50
5.1.4 Verification 50
5.2 Functional correctness: Ring buffer 52
5.2.1 Interface 53
5.2.2 Specification 53
5.2.3 Implementation 55
5.2.4 Verification 56

6 Evaluation **59**
6.1 Verification effort 59
6.2 Resource utilization 60

7 Conclusion **63**
7.1 Summary 63
7.2 Future work 64

References **65**

List of Figures

1.1 Out-of-order machine diagram 10

3.1 State-monad definitions 19
3.2 Guarded-state-monad definitions 20
3.3 Call operator for actions 21
3.4 GCD step, in Slate 22
3.5 GCD step, monadic notation 22
3.6 Constant-time example 24
3.7 Action refinement 27
3.8 Self-simulation and cross-simulation 28

4.1 GCD interface 31
4.2 GCD module 32
4.3 Synthesizable type class 34
4.4 Struct example 35
4.5 Enum example 35
4.6 Bitflags example 36
4.7 For-loop notation 37
4.8 GCD abstract state 38
4.9 GCD value-method specs 39
4.10 GCD load spec 40
4.11 GCD step spec 40
4.12 GCD simulation relation 43

5.1 Multiplier interface 47
5.2 Multiplier port diagram 48
5.3 Multiplier attacker view 49
5.4 Multiplier erasure functions 49
5.5 Multiplier state type 50

5.6 Shift-and-add subroutine 51
5.7 Multiplier simulation relation 52
5.8 Ring-buffer interface 53
5.9 Modular-arithmetic over naturals 54
5.10 Ring-buffer abstract state 54
5.11 Ring-buffer specifications 55
5.12 Ring-buffer implementation 56
5.13 Ring-buffer refinement relation 56

6.1 Ring-buffer gate count 61
6.2 Ring-buffer synthesis time 61
6.3 Ring-buffer resource breakdown 62

List of Tables

6.1 Case studies (LoC) 59
6.2 Personal contributions (LoC) 60

Chapter 1

Introduction

This thesis contributes an approach for proving correctness and timing properties for hardware circuits described in the Rocq proof assistant. This work is the first step toward mechanizing complete correctness and timing proofs for a realistic, modern out-of-order core.

1.1 Background: out-of-order execution

Most modern high-speed processors are based on out-of-order (OoO) execution, which allows the CPU to reorder instructions dynamically based on the availability of functional units (FUs) and input operands. The first instruction-scheduling algorithm to support out-of-order execution with precise exceptions was Tomasulo's algorithm [1], which introduced register renaming to eliminate false dependencies between instructions. Out-of-order processors combined with speculative execution are vulnerable to transient-execution attacks, which in-order processors are not. Principled defenses must be carefully designed to ensure that attackers cannot exploit programs that are safe for in-order processors to execute but unsafe for out-of-order processors.

Figure 1.1 shows a block diagram for the particular variant of the out-of-order algorithm discussed in this thesis. The algorithm is a modified design based on the MIPS R10000 (R10K) processor [2].

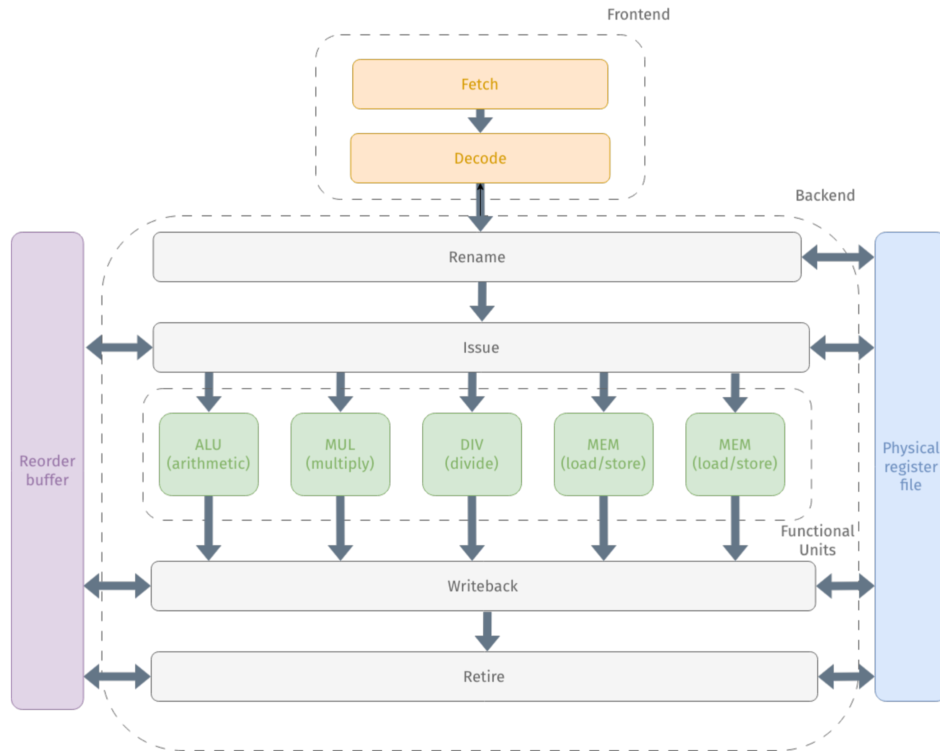

Figure 1.1: Diagram of an out-of-order machine based on the R10K processor. Arrows signify communication between submodules.

Our algorithm is split into seven separate pipeline stages: fetch, decode, rename, issue, execute, writeback, and retire. The frontend stage fetches instructions from the instruction stream and decodes them into an internal data type the machine can schedule. The renaming stage then maps architectural registers to a larger physical register file so false dependencies can be dissolved. The issue stage selects ready renamed operations and sends them to functional units that execute the actual computation. The writeback stage then writes results to the physical register file, so that downstream instructions can consume these results. The retire stage retires instructions in program order, committing their effects to the architectural state.

Our design diverges from the R10K algorithm in two key ways to reduce the invariants needed for proofs. The biggest difference is how the renaming mechanism works. The original R10K algorithm records, in the reorder buffer (ROB), the previous physical register index for each instruction's destination so that the rename map can be rolled back precisely when a younger execution path is squashed. We instead maintain two register-alias tables (RATs), avoiding the need to track a per-instruction previous physical destination in the ROB. The future table holds the most up-to-date speculative mapping from architectural registers to physical registers, while the commit table

records the mapping as of the last retired instruction. Squashing speculative state can then be performed by reverting the future back to the commit table.

Rolling back the alias tables alone would be insufficient, because speculative execution can still overwrite physical register contents before those instructions retire; restoring only the architectural-to-physical mapping would leave stale or partially updated values visible under the recovered names. Our design therefore pairs the dual RAT with explicit snapshots of register values (not just of the rename map). Enough snapshots are retained to reconstruct the physical register file as it stood at the last retired instruction, so a squash restores both the mapping from each architectural register to its physical backing and the values stored in those physical registers at that point. The maximum number of snapshots is finite, so it bounds the number of branches the machine can speculate ahead; once that budget is exhausted, the issue stage must stall until retire frees snapshot capacity.

A second major deviation concerns speculative memory accesses. To make timing security easier to prove initially, we delay all speculative loads and stores. This policy is extremely pessimistic: it foregoes the performance gains that come from speculative execution and is not meant as a proposal for a serious, production-ready design. Looking ahead, we expect that a realistic design can be achieved by pairing speculative memory execution with a dynamic taint-tracking defense, together with a mechanized proof that the resulting design still satisfies the same security property.

1.2 Goal: verifying hardware circuits

This thesis develops techniques for proving correctness and timing properties of synthesizable hardware described in Rocq. The intended workflow is to state and prove theorems against source-level circuit semantics, then relate them to the semantics of synthesizable Verilog through a verified compiler. Prior work on security proofs for hardware tends to reason about simplified ISA-level models instead of RTL. This thesis focuses on synthesizable circuits because they are consumable by downstream synthesis tools for future evaluation on FPGAs or ASICs.

1.3 Individual contributions

The content presented in this thesis was completed during an active collaboration between me, Shixin Song, Jiazheng Liu, Stella Lau, and Andres Erbsen, advised by Pro-

fessors Adam Chlipala and Mengjia Yan. Golemma, the deeply embedded hardware-description language described in Chapter 4, was developed by Jiazheng Liu. Stella contributed security formalizations and proofs for a five-stage pipelined processor with an external I/O and memory system; this work does not directly appear in this thesis, but its theoretical ideas carry over. Shixin worked on a correctness proof for the out-of-order algorithm described in Section 1.1, which is part of a larger project whose ultimate goal is a fully mechanized proof. My personal contributions include specifications, implementations, correctness proofs, and timing proofs for individual submodules of the out-of-order machine, as discussed in Chapter 5.

Chapter 2

Related work

This chapter first surveys prior work on proving functional correctness for hardware circuits (Section 2.1). Then it covers prior work on timing security, both the attacks that motivate this work and the hardware and software defenses that mitigate them (Section 2.2).

2.1 Correctness

Kami [3] is a hardware-description language, embedded in Rocq, for specifying and verifying hardware circuits in the style of Bluespec SystemVerilog. Circuits are semantically treated as labeled transition systems composed of guarded atomic rules, and correctness arguments proceed by refinement between more-abstract and more-concrete specifications.

Koîka [4] is another language for rule-based hardware design that gives Bluespec-style rules a precise sequential semantics characterized by the one-rule-at-a-time (ORAAT) property. Circuits are described as collections of rules over register state, and the language provides a scheduler that resolves rule conflicts while preserving a one-rule-at-a-time execution model. Koîka is accompanied by a verified compiler that lowers circuit descriptions to RTL, providing an end-to-end path from high-level specifications to synthesizable netlists. However, Koîka does not support a module system, so circuits must be described monolithically, and it lacks support for compositional reasoning.

2.2 Security

The Meltdown [5] and Spectre [6] attacks, disclosed in 2018, demonstrated that speculative execution, while critical for performance, introduces new side-channel vulnerabilities absent in in-order processors. Both attacks exploit the fact that transiently executed instructions leave measurable traces in microarchitectural state, such as cache-residency patterns, that an attacker can recover. These attacks motivated a large body of research [7, 8, 9, 10, 11] on transient-execution attacks.

Hardware-software contracts [12] formalize hardware defenses by specifying, for a particular design, both what information is leaked and which speculative paths are explored. An analysis of an abstract out-of-order processor protected by speculative taint tracking [13] shows that the defense satisfies a contract in which leakage events are branch targets, addresses of loads and stores, and operands of variable-latency instructions. The key limitation of this line of work is that it reasons about abstract processor models rather than concrete designs expressible in a hardware-description language; this thesis targets the gap to synthesizable RTL.

At the software level, prior work on constant-time verification for compilers [14] developed techniques for proving constant-time properties for nondeterministic programs. Using a formulation of constant time as a single-copy property, the work proves end-to-end preservation of constant-time routines compiled from Bedrock2 [15] source to RISC-V machine code. The single-copy formulation of constant time used in this thesis is taken from that work, repurposing the approach from software compilers to hardware verification.

SecSep [16] addresses the gap between source-level constant-time guarantees and speculative execution on real hardware through an assembly-level transformation framework. SecSep targets processors similar to ProSpeCT [17], which partitions memory into regions whose speculative loads are delayed until non-speculative and regions whose loads are not, provably enforcing the constant-time policy without relying on fencing. SecSep rewrites compiled assembly programs so that secret and public data are partitioned on the stack, using a new typed assembly language called Octal to guide the transformation. The approach correctly handles secrets on the stack and achieves a low additional runtime overhead on cryptographic programs.

Prior work on hardware timing security [18] builds off the Kami framework to reason about timing-security properties of hardware circuits, showing that the same embedding used to verify functional correctness can also support security arguments. More recent work [19] formally specifies and verifies strong timing isolation for hardware enclaves.

Chapter 3

Verifying timing security for circuits

This chapter presents a framework for timing-security verification of hardware circuits. It begins by giving a formalism of state machines to model hardware circuits (Section 3.1) and then introduces Slate, a shallowly embedded language in Rocq for specifying hardware circuits (Section 3.2). Finally, it defines hardware timing security and develops the simulation-based proof structure used to establish this security property (Section 3.3).

3.1 State machines

A state machine is an abstraction used to describe the behavior of a computation by defining its possible states and the rules for transitioning between them. Hardware modules function as physical state machines. Registers serve as the memory that holds the current state, while combinational logic describes the transition function that determines both the module's outputs and its next state as a function of the current state. We view hardware modules as interacting through well-defined interfaces composed of methods rather than raw, unstructured signals. Each discrete state transition in the system corresponds to the execution of a method, supplied with arguments that represent the transition's data. This approach is consistent with prior work [3, 4], which views hardware modules as high-level transition systems whose transitions lie on external method boundaries. We adopt the following formal definition of a state machine.

Definition 3.1.1 (State machine). A state machine $\mathcal{M}$ is a 5-tuple (S, s_0, I, O, δ) where S is a set of states, $s_0 \in S$ is the initial state, I is a set of inputs, O is a set of outputs, and

$$\delta : S \times I \rightharpoonup S \times O$$

is a partial transition function. Here $\rightharpoonup$ denotes that certain methods can abort and produce no state change or output.

We model hardware circuits as such machines. A state $s \in S$ captures the current values of the sequential logic relevant to the module. Each input in I represents a method invocation: a variant type carrying a method tag together with the argument values for that call. The transition function δ is partial on $S \times I$ to reflect that the model may not produce a well-defined successor state and output when the call is aborted by a guard (for example, dequeuing from an empty buffer). An aborted call leaves the circuit in the same configuration. This convention simplifies trace definitions and simulation proofs: each input sequence determines one run, and the optional output indicates whether the call produced a visible event in the output trace.

For the purpose of reasoning about security with this framework, we additionally restrict attention to circuits whose behavior is deterministic in the sense above:

$$\begin{aligned} &\forall s \in S.\ \forall \iota \in I.\ \forall s_1, s_2 \in S.\ \forall o_1, o_2 \in O. \\ &\quad \delta(s, \iota) = (s_1, o_1) \wedge \delta(s, \iota) = (s_2, o_2) \\ &\quad \implies s_1 = s_2 \wedge o_1 = o_2. \end{aligned}$$

In words, if a state and input can step to two results, then those results must be identical. This restriction is convenient for reasoning about the proof obligations in Section 3.3, where we compare runs under related inputs and want equality of traces without existentially quantifying over implementation-defined resolution of nondeterminism in proofs. Determinism is a design choice of convenience, not necessity; as discussed in Chapter 4, the framework for deeply embedded circuits can use nondeterministic semantics (in particular, omnisemantics [20]) to describe specifications.

Given an input sequence $(\iota_1, \iota_2, \ldots, \iota_n) \in I^*$, repeatedly applying δ from the initial state s_0 defines a run of the machine. The corresponding sequence of per-step outputs is called the machine's observation trace. Concretely, the machine $\mathcal{M}$ produces a sequence of outputs by repeatedly applying δ from s_0 for each input $\iota_k \in I$:

$$s_0 \xrightarrow{\iota_1/o_1} s_1 \xrightarrow{\iota_2/o_2} s_2 \ \cdots \ \xrightarrow{\iota_n/o_n} s_n, \qquad o_i \in O,\ s_i \in S.$$

The observation trace of the run is the list:

$$\mathcal{T}(\mathcal{M}, \vec{\iota}) \ \triangleq\ (o_1, o_2, \ldots, o_n).$$

For timing security, an attacker interprets elements of O (modulo erasure, described in Section 3.3) as leaked information. Aborted transitions do not produce outputs in the observation trace.

3.2 Encoding circuits with Slate

This section introduces Slate, a language for describing hardware as executable descriptions in Rocq. It first compares shallow and deep embeddings and explains why Slate is implemented as a shallow embedding (Section 3.2.1). It then discusses guarded state monads as a way to model imperative programs shallowly in Rocq (Section 3.2.2). Finally, it presents a small illustrative example of using the Slate language (Section 3.2.3).

3.2.1 Shallow and deep embeddings

A shallow embedding encodes a source language directly as terms of the host language. In this style, the host's type system, reduction rules, and terms are the semantics of the object language. In Rocq, shallow embeddings are represented as ordinary Gallina functions rather than abstract syntax trees requiring a separate interpreter. By contrast, a deep embedding defines explicit syntax via inductive datatypes for commands and expressions. Since syntax trees do not execute by reducing as Gallina terms, deep embeddings must come with separate semantics, such as an interpreter or operational semantics.

Each style has distinct advantages and costs. Shallow embeddings are typically easier for verification work, because they directly leverage Rocq's facilities for Gallina; many proof steps reduce by unfolding and simplification rather than by reasoning about an interpreter or operational semantics. The main drawback is that source-level program transformations, such as compiler passes, are tricky to implement, since syntax is not explicit. These tasks often require reification and metaprogramming support to recover a syntax tree.

Deep embeddings make syntax explicit from the beginning, so compiler construction and optimization passes are easier to express. However, they impose additional verification overhead: one must define and reason about an interpreter (or step relation), and prove that this semantics aligns with the intended meaning. As a result, proofs on deep embeddings are more involved than similar theorems on shallow embeddings. Slate, the surface language used for timing-security verification in this thesis, is a shallow em-

bedding. Its core semantics work as an imperative language with mutable references. We introduce the state-monad structure that underlies Slate method definitions.

3.2.2 State monads and actions

State monads For a fixed state type s, the state monad is the type family that maps each result type t to

$$\mathrm{State}_s(t) \triangleq s \to (s * t),$$

where an element of $\mathrm{State}_s(t)$ is a stateful computation that takes an input state and returns a new state together with a result. This representation is powerful because computations can be sequenced with the monadic bind operation. Mutable references do not need to be primitively supported by the metalanguage (i.e., Gallina); instead, stateful programs are ordinary functions that transform an explicit state argument. As a consequence, one can describe sequential hardware behavior in a pure functional setting and still recover the usual intuition of sequential logic, by reading registers at the start of the cycle, performing some combinational function on those values, and updating the registers at the end of the cycle.

Figure 3.1 shows the state monad encoded in Rocq together with its basic operations. The constructor *ret* lifts a pure value into the monad without changing the state, while *bind* sequences two computations by feeding the result and resulting state of the first computation into the second. The infix operator $>>=$ is another way to express *bind*. State monads also provide state-specific primitives, notably *get* and *put*: *get* returns the current state as a value, and *put* replaces the current state with a new one.

```
Definition State s t := s -> (s * t).

Definition ret {s t} (x : t) : State s t :=
  fun st => (st, x).

Definition bind {s t1 t2} (m : State s t1) (f : t1 -> State s t2) :=
  fun st =>
    let '(st', x) := m st in
    f x st'.

Definition get {s} : State s s :=
  fun st => (st, st).

Definition put {s} (x : s) : State s unit :=
  fun _ => (x, tt).
```

Figure 3.1: State-monad definitions in Rocq.

Guarded state monads For a fixed state type s, the guarded state monad or action is type family that maps each result type t to

$$\text{Action}_s\ (t)\ \triangleq\ s \longrightarrow \mathit{option}\ (s * t),$$

that is, a function that takes an input state and either returns a new state together with a result or aborts. The framework uses actions to model method calls that are enabled only when their guards hold. The *Some* case represents a successful step that returns an updated state and a value. The *None* case represents an aborted step (for example, when a guard is false). Figure 3.2 shows the guarded state monad encoded in Rocq together with its basic operations. *ret*, *bind*, *get*, and *put* work similarly to their state-monad analogues.

```
Definition Action s t := s -> option (s * t).

Definition ret {s t} (x : t) : Action s t :=
  fun st => Some (st, x).

Definition bind {s t1 t2} (m : Action s t1) (f : t1 -> Action s t2) :=
  fun r =>
    match m r with
    | Some (s, x) => f x s
    | None => None
    end.

Definition get {s} : Action s s :=
  fun st => Some (st, st).

Definition put {s} (x : s) : Action s unit :=
  fun _ => Some (x, tt).

Definition modify {s} (f : s -> s) : Action s unit :=
  fun st => Some (f st, tt).

Definition fail {s t} : Action s t :=
  fun _ => None.
```

Figure 3.2: Guarded-state-monad definitions in Rocq.

Modify *modify* updates the state by applying a function f. It can be used as a derived write primitive: read the current state, transform it, then commit the result. In hardware, this operation is a natural fit for register updates where the next state is expressed as a pure function of the current state. Proof obligations involving *modify* usually reduce to equalities between functions applied to the same input state.

Guard failure *fail* returns *None* for any input state. In hardware-description semantics, a *None* result represents an aborted method call (for example, violating a ready condition). At the state-machine level, guard failure is normalized into a step with no

observable output and no state changes. Introducing failure in the monad itself keeps reasoning about guards local to method definitions.

Submodule calls The operator *call* implements modular method calls over nested state. It uses *getter* to project a substate, runs an *Action* on that substate, and then reintegrates the updated substate using *setter* as shown in Figure 3.3. If the callee action fails, the caller fails. Submodule calls with guarded state monads mirror the calling semantics of Bluespec SystemVerilog, where a method call that fails to fire causes the caller to also fail to fire.

```
Definition call {s a b}
  (getter : s -> a)
  (setter : a -> s -> s)
  (action : Action a b) : Action s b :=
  fun st =>
    match action (getter st) with
    | Some (st', x) => Some (setter st' st, x)
    | None => None
    end.
```

Figure 3.3: *call* operator lifts the effects of a submodule action to a larger state.

3.2.3 Syntax

The syntax for Slate is implemented using the extensible parser in Rocq. Crucially, the parser elaborates Slate syntax directly into actions as described above; it does not introduce a syntax tree. Consider Figure 3.4, which implements the *step* method of a GCD module. The state type *GCDState* holds two registers x and y. The method reads both registers into let bindings, aborts immediately when $y = 0$ (the computation is already finished), and otherwise subtracts the smaller value from the larger. This style reads like imperative code while denoting a Gallina term.

```
Record GCDState := { x : Bit#(16); y : Bit#(16) }.

Definition step : Action GCDState unit :=
  begin
    let xv = `x`read();
    let yv = `y`read();
    when (yv == 0)
      abort;
    if (xv >= yv) begin
      `x`write(xv - yv)
    end else begin
      `y`write(yv - xv)
    end
  end.
```

Figure 3.4: The *step* method of a GCD module using Slate.

The same method can be written directly with monadic operators (Figure 3.5). This version makes sequencing explicit through >>=, but is noticeably less readable for method bodies of this length. In practice, both forms are definitionally equal after notation elaboration.

```
Definition step : Action GCDState unit :=
  get >>= fun st =>
  (if st.(y) =? 0 then fail else ret tt) >>= fun _ =>
  if st.(x) >=? st.(y) then
    modify (fun s => {| x := s.(x) - s.(y); y := s.(y) |}) >>= fun _ =>
    ret tt
  else
    modify (fun s => {| x := s.(x); y := s.(y) - s.(x) |}) >>= fun _ =>
    ret tt.
```

Figure 3.5: The *step* method of a GCD module using inline monadic notation.

3.3 Specifying and proving timing security

Establishing that a hardware circuit is free from timing side channels requires both a precise definition of what it means to be secure and a methodology for proving that a given circuit satisfies that definition. This section addresses both concerns. It first introduces cryptographic constant time (CCT) in the software setting and then extends the idea to hardware circuits as hardware constant time (HCT), stated in terms of erasure contracts that model the attacker's observational capabilities (Sections 3.3.1 and 3.3.2). It then develops the simulation-based proof strategy used to establish HCT: using lock-step simulation with trace-equivalence lemmas that connects simulations to the HCT definition (Section 3.3.3).

3.3.1 Cryptographic constant time

Cryptographic software often cares about the cryptographic-constant-time (CCT) property: attackers should not observe leakage events such as branch direction, memory-access patterns, or variable-latency instructions influenced by secret inputs. This principle has been formalized and mechanically verified for software programs in multiple prior works [12, 14, 21]. Figure 3.6 illustrates the key intuition with two small routines: one has a secret-dependent branch and is not constant time, while the other computes through bit masking and avoids control-flow dependent on secret data.

More formally, constant-time security is stated as an observational equality under varying secrets: if two executions agree on public inputs, then they must be indistinguishable with respect to the attacker-observable trace. The next definition makes constant time precise in the traditional two-copy (hyperproperty) style [22], before introducing an equivalent one-copy formulation.

```c
// Not constant-time
uint32_t f_bad(x, y, secret) {
  uint32_t x;
  uint32_t y;
  uint8_t secret;

  if (secret) return x;
  else return y;
}
```

```c
// Constant-time
uint32_t f_ct(x, y, secret) {
  uint32_t x;
  uint32_t y;
  uint8_t secret;

  uint32_t m = 0u - (secret & 1u);
  return (x & m) | (y & ~m);
}
```

Figure 3.6: Constant-time example: secret-dependent branch (left) versus branch-free masked selection (right).

Definition 3.3.1 (Cryptographic constant time as a two-copy hyperproperty)**.** A state machine $\mathcal{M} = (S, s_0, \mathcal{I}, O, \delta)$ is cryptographic-constant-time (CCT) if given partitions

$$s = (s^{\text{pub}}, s^{\text{sec}}), \quad i = (i^{\text{pub}}, i^{\text{sec}}), \quad o = (o^{\text{pub}}, o^{\text{sec}})$$

of states, inputs, and outputs into public and secret components, such that for any two states $s_1, s_2 \in S$ with $s_1^{\text{pub}} = s_2^{\text{pub}}$, and for any two input sequences $(i_1, i_2, \ldots)$ and $(i'_1, i'_2, \ldots)$ in $\mathcal{I}$ with $i_k^{\text{pub}} = i_k'^{\text{pub}}$, the public output components agree at every step k, $o_k^{\text{pub}} = o_k'^{\text{pub}}$.

This definition is relational: it compares two universally quantified executions and is therefore a hyperproperty. Next, we introduce a one-copy formalism of constant time that is logically equivalent to Definition 3.3.1.

Definition 3.3.2 (Cryptographic constant time as a one-copy property)**.** A state machine $\mathcal{M} = (S, s_0, \mathcal{I}, O, \delta)$ is cryptographic-constant-time (CCT) if given partitions of states, inputs, and outputs

$$s = (s^{\text{pub}}, s^{\text{sec}}), \quad i = (i^{\text{pub}}, i^{\text{sec}}), \quad o = (o^{\text{pub}}, o^{\text{sec}}),$$

there exists a predictor function

$$f_{\text{predict}} : S^{\text{pub}} \times (\mathcal{I}^{\text{pub}})^* \to (O^{\text{pub}})^*,$$

where pub_inputs($\vec{i}$) is the elementwise projection of an input sequence to public components, and pub_outputs($\mathcal{M}, s_0, \vec{i}$) is the corresponding projected observation trace.

Then, for every initial state $s_0 \in S$ and input sequence $\vec{i} \in I^*$,

$$\text{pub_outputs}(\mathcal{M}, s_0, \vec{i}) = f_{\text{predict}}\big(s_0^{\text{pub}}, \text{pub_inputs}(\vec{i})\big).$$

This definition originally comes from work on retrofitting the Bedrock2 compiler to prove preservation of cryptographic constant time across compiler passes [14]. Intuitively, the observable behavior is fully determined by the public projection of the input and the initial state. The definition above is a one-copy specification style: each run is characterized independently without needing to relate two separate sequences. This definition is useful for certain proof techniques that can existentially construct a nice predictor function (f_{predict}). The hardware development in this thesis uses the same one-copy perspective but restricted to state-machine traces with explicit input/output-erasure functions that capture the notion of partitioning state into secret and public components.

3.3.2 Hardware constant time

Hardware constant time is motivated by hardware-software contracts [12]: the security guarantee is stated relative to an explicit attacker observation model. In that line of work, the contract ties together what information the attacker can observe (leakage events) and what information the system must hide as a function of secret inputs. In this thesis, we adopt the same pattern: we first make the attacker's observational capabilities explicit via erasure contracts (Definition 3.3.3) and then define hardware constant time as an invariance of the erased observation when secret (erased) inputs vary.

Definition 3.3.3 (Erasure contract)**.** For a state machine $\mathcal{M} = (S, s_0, I, O, \delta)$, an erasure contract is a pair $\mathcal{E} = (\text{erase}_{\text{in}}, \text{erase}_{\text{out}})$ where erase_{in} is a function that erases secret components of inputs $\iota \in I$, and $\text{erase}_{\text{out}}$ is a function that erases secret components of outputs $o \in O$ from the observational trace.

Definition 3.3.4 (Erased observation trace)**.** Given $\mathcal{M}$, $\mathcal{E}$, and an input sequence $\vec{\iota} \in I^*$, we define $\mathcal{T}_{\mathcal{E}}(\mathcal{M}, \vec{\iota})$ as the erased observation trace, which is the observation trace $\mathcal{T}(\mathcal{M}, \vec{\iota})$ with private output components deleted.

Definition 3.3.5 (Erased input sequence)**.** Given $\mathcal{M}$, $\mathcal{E}$, and an input sequence $\vec{\iota} \in I^*$, we define $\vec{\iota}_{\mathcal{E}}$ as the erased input sequence, which is the input sequence with private input components deleted.

Definition 3.3.6 (Hardware constant time)**.** A state machine $\mathcal{M}$ is hardware-constant-time (HCT) with respect to an erasure contract $\mathcal{E}$ when

$$\forall \vec{\iota} \in I^*.\ \ \mathcal{T}_{\mathcal{E}}(\mathcal{M}, \vec{\iota}) = \mathcal{T}_{\mathcal{E}}(\mathcal{M}, \vec{\iota}_{\mathcal{E}}).$$

An erasure contract can be interpreted as a threat model, a description of the observational capabilities of an attacker. It states which wires of a module's interface are observable to an attacker and which parts are treated as secret. Concretely, erase_{in} controls observability on input wires: it preserves attacker-visible components and erases the rest. Similarly, $\text{erase}_{\text{out}}$ controls observability on output wires: it keeps attacker-visible outputs and removes or abstracts outputs that are outside the attacker's view. Under this interpretation, HCT says that varying secret (erased) inputs cannot change the attacker-visible output trace.

3.3.3 Writing proofs

The hardware-constant-time property (Definition 3.3.6) reasons about entire runs of state machines rather than isolated steps. Reasoning over entire runs permits proof techniques that are not limited to step-by-step analysis; in the future, this could include simulation-based proofs that skip steps or take multiple steps at once. We restrict ourselves to reasoning only about lockstep simulations, which disallow skipping or multiple steps. Figure 3.8 demonstrates the two different proof techniques introduced in this section: first, a machine compared against itself under erased inputs (self-simulation); and second, an implementation machine compared against an abstract specification (cross-simulation).

Setup Fix state machines $\mathcal{M}_1$ and $\mathcal{M}_2$ and erasure contract $\mathcal{E}$ as the following

$$\mathcal{M}_1 = (S_1, s_{0,1}, I, O, \delta_1), \qquad \mathcal{M}_2 = (S_2, s_{0,2}, I, O, \delta_2),$$
$$\mathcal{E} = (\text{erase}_{\text{in}}, \text{erase}_{\text{out}}).$$

Fix relation $R \subseteq S_1 \times S_2$. Let a_1 and a_2 be actions over S_1 and S_2, respectively, sharing output type O. For related starting states $(s_1, s_2) \in R$, suppose each action steps to a

successor state and output:

$$a_1(s_1) = (s'_1, o_1), \qquad a_2(s_2) = (s'_2, o_2).$$

Definition 3.3.7 (Erased-result relation). Given a relation $R \subseteq S_1 \times S_2$ and erasure contract $\mathcal{E}$, we say $R_{\mathcal{E}} \subseteq S_1 \times O \times S_2 \times O$ is the erased-result relation when

$$R_{\mathcal{E}}(s_1, o_1, s_2, o_2) \triangleq \text{erase}_{\text{out}}(o_1) = \text{erase}_{\text{out}}(o_2) \wedge R(s_1, s_2).$$

Definition 3.3.8 (Action simulation). We say $a_1 \sim a_2$ (read simulates) under $(R, \mathcal{E})$ at (s_1, s_2) when

$$R(s_1, s_2) \Rightarrow R_{\mathcal{E}}(s'_1, o_1, s'_2, o_2).$$

This condition is depicted as a commutative diagram in Figure 3.7.

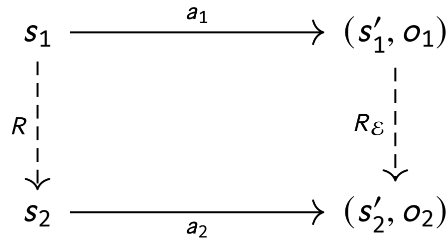

Figure 3.7: Action refinement for a_1 and a_2 as a commutative diagram.

Next, machine simulation instantiates Definition 3.3.8 on every method invocation $\iota \in I$ where the abstract machine observes $\text{erase}_{\text{in}}(\iota)$ wherever the implementation machine observes ι.

Definition 3.3.9 (Machine simulation). $\mathcal{M}_1 \sim \mathcal{M}_2$ (read simulates) under $(R, \mathcal{E})$ when, for all related states $(s_1, s_2) \in R$, the following input-indexed action simulation holds:

$$\forall \iota \in I.\ \left(\lambda s.\ \delta_1(s, \iota)\right) \sim \left(\lambda s.\ \delta_2(s, \text{erase}_{\text{in}}(\iota))\right) \text{ under } (R, \mathcal{E}) \text{ at } (s_1, s_2).$$

Self-simulation and cross-simulation The self-simulation and cross-simulation instances of this definition are shown in Figure 3.8. In self-simulation, the same machine appears on both sides: one side consumes the original input ι, while the other consumes the erased input $\text{erase}_{\text{in}}(\iota)$. Self-simulation directly matches the shape of hardware constant time, where we compare a machine against itself under secret variation. In cross-simulation, the left side is a concrete implementation, and the right side is a simpler

abstract specification that already enforces the intended security behavior; the proof then factors through simulation to transfer that behavior back to the implementation.

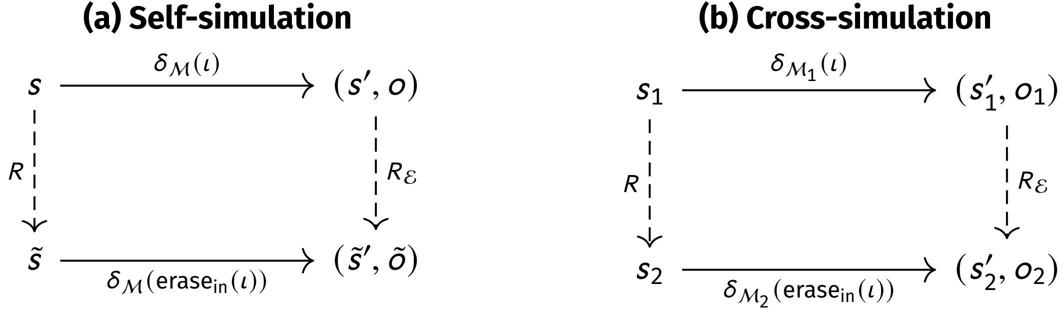

Figure 3.8: Lockstep simulation diagrams. Panel **(a)** reuses $\delta_{\mathcal{M}}$ whereas panel **(b)** contrasts $\mathcal{M}_1$ with $\mathcal{M}_2$.

Next, we turn the stepwise lockstep obligations from Figures 3.7 and 3.8 into trace consequences. We first define output-erased trace equivalence and show that machine simulation lifts to equality of erased traces. We then use these lemmas to derive the two proof patterns for hardware constant time: direct self-simulation and cross-simulation via a trace-equivalent specification.

Definition 3.3.10 (Erased-trace equivalence)**.** Call $\mathcal{M}_2$ erased-trace equivalent to $\mathcal{M}_1$ under $\mathcal{E}$ when, for every $\vec{\iota} \in I^*$, the erased-input histories $\vec{\iota}_{\mathcal{E}}$ induce the same erased observation trace:

$$\mathcal{T}_{\mathcal{E}}(\mathcal{M}_1, \vec{\iota}_{\mathcal{E}}) = \mathcal{T}_{\mathcal{E}}(\mathcal{M}_2, \vec{\iota}_{\mathcal{E}}).$$

Lemma 3.3.1 (Simulation implies erased-trace equivalence)**.** *If $\mathcal{M}_1$ simulates $\mathcal{M}_2$ under $(R, \mathcal{E})$ in the sense of Definition 3.3.9 and $R(s_{0,1}, s_{0,2})$, then for every input history $\vec{\iota} \in I^*$,*

$$\mathcal{T}_{\mathcal{E}}(\mathcal{M}_1, \vec{\iota}) = \mathcal{T}_{\mathcal{E}}(\mathcal{M}_2, \vec{\iota}_{\mathcal{E}}).$$

Proof. Induct on $\vec{\iota}$; the base case on the empty list is immediate. The inductive case first applies Definition 3.3.9 to the head of the trace before appealing to the induction hypothesis on the tails. □

Lemma 3.3.2 (Self-simulation implies hardware constant time)**.** *If $\mathcal{M}$ simulates itself under $(R, \mathcal{E})$ and $R(s_0, s_0)$, then $\mathcal{M}$ satisfies hardware constant time (Definition 3.3.6) with respect to $\mathcal{E}$.*

Proof. Instantiate Lemma 3.3.1 with the specification and implementation both taken to be $\mathcal{M}$; the resulting equation is Definition 3.3.6. □

Lemma 3.3.3 (Cross-simulation with trace equivalence implies hardware constant time)**.** *Suppose $\mathcal{M}_\mathrm{I}$ simulates $\mathcal{M}_\mathrm{S}$ under $(R, \mathcal{E})$, that $R(s_{0,\mathrm{I}}, s_{0,\mathrm{S}})$, and that $\mathcal{M}_\mathrm{S}$ is erased-trace equivalent to $\mathcal{M}_\mathrm{I}$, as stipulated in Definition 3.3.10. Then $\mathcal{M}_\mathrm{I}$ satisfies hardware constant time (Definition 3.3.6) with respect to $\mathcal{E}$.*

Proof. Instantiate Lemma 3.3.1 with the implementation taken to be $\mathcal{M}_\mathrm{I}$ and the specification taken to be $\mathcal{M}_\mathrm{S}$. Proof obligations to show $\mathcal{M}_\mathrm{I}$ simulates $\mathcal{M}_\mathrm{S}$ under $(R, \mathcal{E})$ and that $R(s_{0,\mathrm{I}}, s_{0,\mathrm{S}})$ are provided as assumptions of the lemma. □

Chapter 4

Verifying correctness for circuits

This chapter presents a framework for verifying functional correctness on synthesizable hardware circuits. It first introduces the deeply embedded Golemma language, which has similarities to Bluespec SystemVerilog (Section 4.1). Then, it covers language extensions that enable the synthesis of Gallina types and for loops into Golemma (Section 4.2). Finally, it discusses how functional specifications are written and develops the proof strategy used to show modular refinement (Section 4.3).

4.1 Encoding circuits with Golemma

This section introduces the key constructs of Golemma through an example circuit: a module that computes the greatest common divisor (GCD) of two 16-bit numbers using the Euclidean algorithm. The interface and module definition are shown in Figures 4.1 and 4.2, respectively.

4.1.1 Interfaces

An interface in Golemma is a typed list of method signatures that a module presents to its environment. Interfaces serve as contracts in two complementary senses. First, interfaces define the compositional boundary between a module and its environment, playing the same role as the port list of a Verilog module. Second, they constrain what can be verified, since any specification must be stated in terms of the same method signatures that the implementation exposes.

The interface for the GCD module (Figure 4.1) declares four methods. Two are action methods: *load* and *step*, which can write registers and may abort if their guards are not

met; their return type is *Action*. Two are value methods: *result* and *finished*, which are purely observational and cannot modify registers.

```
interface GCD;
  method  Action      load  <Bit#(16) a, Bit#(16) b>;
  method  Action      step  <>;
  method  Bit#(16)    result <>;
  method  Bit#(1)     finished  <>;
endinterface.
```

Figure 4.1: The GCD interface with two action methods and two value methods.

4.1.2 Modules

A module instantiates an interface. Modules contain submodules with which they interact through method invocations defined in interfaces. Registers are implemented as submodules: a register submodule holds state across clock cycles and exposes methods for reading and updating that state. Figure 4.2 shows the implementation of the GCD module. The header declares that the module implements the *GCD* interface. The two register declarations *x* and *y* create the persistent state. Submodule interactions, including register reads and writes, use backtick notation, as in $x.read()$ and $x.write(v)$. Unlike most register-transfer-level languages, Golemma does not have the restriction that every read within a method body sees the value the register held at the start of the method call; it sees the last write that occurred.

4.1.3 Methods

Every module method is declared with a return type, a name, and an optional parameter list enclosed in angle brackets. Methods whose return type are *Action* or $ActionValue\#(t)$ may call submodule action methods; they can also abort, leaving state unchanged. Methods with any other return types are value methods: they may only read state and return pure values, and they never abort. This distinction is enforced syntactically: only action methods may contain write or abort statements, so the separation between observation and mutation is visible directly in the interface signature.

```
module mkGCD#(GCD);
  Reg#(Bit#(16))  x  <- mkReg();
  Reg#(Bit#(16))  y  <- mkReg();

  method Action load <Bit#(16) a, Bit#(16) b>;
    begin
      `x`write(a);
      `y`write(b);
    end.
  endmethod.

  method Action step <>;
    begin
      let xv = `x`read();
      let yv = `y`read();
      if (yv == 16'd0)  abort;
      if (xv >= yv)     `x`write(xv - yv);
      else              `y`write(yv - xv);
    end.
  endmethod.

  method Bit#(16) result <>;
    begin return `x`read(); end.
  endmethod.

  method Bit#(1) finished <>;
    begin return (`y`read() == 16'd0); end.
  endmethod.
endmodule.
```

Figure 4.2: A GCD module written in Golemma.

4.1.4 Expressions

Expressions are pure terms of a given type. The primitive types in Golemma are bitvectors ($Bit\#(n)$ for a n-bit word), fixed-size vectors of types ($Vector\#(n,t)$ for a fixed-size array of t with n items), structs, and enums. An expression can be a constant literal such as $16'd0$, a local variable bound by a prior *let* statement, a field access $e.f$, a primitive operator such as addition or comparison, or a submodule call $sub.met(args)$. Value method calls read current register state inside a submodule and return a result without modifying it; they appear as expressions because their effect are purely observational.

4.1.5 Statements

Statements sequence effects inside a method body. The *let* $x = e$ form binds a fresh variable x to the value of expression e in the rest of the block. A typed form $T\#(t)\ x = e$ binds x with an explicit type annotation. Register state is updated via submodule action method calls: $reg.write(e)$ calls the *write* action on register *reg* with argument e. The result of calling an action-method can be captured with *let* $x \leftarrow mod.met(args)$. Conditional branching uses *if* $(c)\ \{\ldots\}$ *else* $\{\ldots\}$, where c is an expression of type $Bit\#(1)$; the *else* branch is optional and defaults to a no-op. A statement may be terminated early by *abort*, which causes the entire method to fail and appear as if it never exited. No special synthesis is needed to implement *abort*, but it induces a precondition on the method. The method will only be called in states where execution cannot reach the *abort*. During verification, callers must prove that this precondition holds. Statements are sequenced by separating them with semicolons inside a *begin* ... *end* block. A block may end with a *return* e expression that specifies the method's output value; methods of type *Action* whose return type is *unit* may omit *return* and close with *end* directly.

4.2 Extending Golemma with type classes

This section discusses a language extension that enables the reification of finite Gallina types and terms into Golemma (Section 4.2.1). It also discusses how this extension enables other language features such as structs (Section 4.2.2), enums (Section 4.2.3), bitflags (Section 4.2.4), and for loops (Section 4.2.6).

4.2.1 Synthesizable types

Golemma programs cannot directly use standard Rocq types such as plain records and inductive types; instead, they must use the types provided by Golemma's type system. Writing functional specifications and invariants that reason directly over Golemma types is possible but cumbersome. Proofs would need to reason about raw bitvectors rather than natural Gallina types, which are the most natural to work with and perform best with Rocq's type checker A better approach is to allow the developer to use familiar Gallina types directly in Golemma circuits, so that specifications and proofs can be stated at the Gallina level and then connected to the hardware representation transparently.

The Synthesizable type class formalizes this connection. Instances of the *Synthesizable* type-class witness that a Gallina type α has a corresponding Golemma hardware type τ

and provide the two functions that convert between them. Figure 4.3 shows the definition.

```
Class Synthesizable (α : Type) := {
  (* [type] is the type of embedded Golemma types *)
  τ           : type;
  pack        : α -> type_denote τ;
  unpack      : type_denote τ -> α;
  pack_unpack : forall x, unpack (pack x) = x;
}.
```

Figure 4.3: Definition of the *Synthesizable* type class.

The field τ is the Golemma type representing α in hardware. The function *pack* is an injection from α into the hardware representation: *unpack* is its left inverse, recovering the original Gallina value from the encoding, and *pack_unpack* proves that *pack* is indeed an injection. Every native Golemma type is trivially synthesizable with *pack* = *unpack* = *id*. Beyond Golemma types, several finite Gallina types have built-in instances: bitvectors *bv n*, bounded naturals *fin n*, Booleans *bool*, and *unit*. Three additional type classes extend *Synthesizable* to structured Gallina types: Struct, Enum, and Bitflags. Each provides a Synthesizable instance automatically during type-class resolution.

4.2.2 Structures

The Struct type class handles Gallina records whose fields are all synthesizable. An instance maps the record to a Golemma deeply embedded struct type, packing and unpacking each field individually. Figure 4.4 shows a simple example. The tactic *derive_struct* inspects the definition of the record using Ltac2 metaprogramming (Section 4.2.5), extracts the field names and types, constructs the packing and unpacking mechanism, and discharges all proof obligations automatically.

```
Record TwoBits := {
  bit0 : bool;
  bit1 : bool;
}.

Instance two_bits_struct : Struct TwoBits. derive_struct. Defined.
```

Figure 4.4: Deriving a *Struct* instance for a simple record.

4.2.3 Enumerables

The Enum type class handles Gallina inductive types with only nullary constructors, that is, simple variant types with no associated data. An instance maps each constructor to a distinct element of a Golemma *enum_t*, with the representation width chosen as $\lceil \log_2 n \rceil$ bits for a type with n constructors. Figure 4.5 shows a three-constructor color type. The tactic *derive_enum* automatically computes the constructor names, bitvector representations, and all proof obligations.

```
Variant Color : Set := Red | Green | Blue.

Instance color_enum : Enum Color. derive_enum. Defined.
```

Figure 4.5: Deriving an *Enum* instance for a Rocq variant type.

4.2.4 Bitflags

The Bitflags type class handles Gallina variant types where each constructor represents an independent flag in a bitset. Each constructor is mapped to a distinct bit position in a fixed-width bitvector, and the associated *Bitset* type, which is formalized as a finite set using set-algebra instances from stdpp. Proofs over bitsets can therefore use the *set_solver* tactic and the full stdpp set lemma library. Figure 4.6 shows a hardware-flag example with five variants. The *Bitset* type synthesizes to a *Bit#*(n) for a type with n constructors.

```
Variant Permission := Read | Write | Execute | User | Valid.

Instance permission_flags : Bitflags Permission. derive_bitflags. Defined.

Definition user_rx : Bitset Permission := {[Valid; Read; Execute; User]}.

Example user_rx_has_read : elem_of Read user_rx. Proof. set_solver. Qed.
```

Figure 4.6: Deriving a *Bitflags* instance and using stdpp set reasoning.

4.2.5 Metaprogramming

In proof assistants such as Rocq, tactics are used as scripts to construct proof terms concisely. Ltac1 is Rocq's original and most commonly used tactic language. Ltac2 is a newer, typed tactic language with better support for inspecting and constructing Rocq terms programmatically, making it a better fit for metaprogramming purposes.

Deriving instances for *Struct*, *Enum*, and *Bitflags* requires inspecting inductive type definitions at elaboration time. Rather than requiring these type-class instances to be derived by hand, the framework uses Ltac2 to perform this metaprogramming. The derivation tactics (*derive_struct*, *derive_enum*, *derive_bitflags*) each reflect a Rocq inductive into a Golemma type signature and discharge any resulting proof obligations using Ltac1.

When a Golemma expression contains an identifier, the language frontend uses a sophisticated type-class-resolution mechanism to decide if it refers to a local method parameter, a local let binding, or a submodule. To add support for referring to record projections or Gallina constructors, this system was extended to recognize identifiers related to synthesizable types. This resolution is handled by a hint in the type-class resolution database that fires whenever the elaborator encounters a particular type-class goal that resolves identifiers. The hint, written in Ltac2, recovers the identifier name as a Rocq string, searches first among local hypotheses and then the global environment for a binding with that name, and, if found, wraps the value in $Cst\ (pack\ \cdot)$: the syntax tree for a constant node in Golemma. The result is that any Gallina constant or local variable of a synthesizable type can be dropped into a Golemma expression directly by name, without any explicit coercion syntax.

4.2.6 For loops

Hardware circuits commonly need to operate over fixed-size arrays of registers, for example, to implement content-addressable memories (CAMs). Golemma did not provide native syntax for for-loop iteration at the time this thesis was written. The for constructor in the syntax tree takes a bound m and a body function from $Fin\ m$ to a block of actions, executing the body once for each index in $0, 1, \ldots, m - 1$.

The key step is converting the index i of type $Fin\ m$ into a bitvector of size $\lceil \log_2 m \rceil$ so that the body can use it in Golemma expressions directly. This conversion process happens automatically via the type-class-resolution procedure described in Sections 4.2.1 and 4.2.5. A custom notation then exposes a familiar loop syntax as a Golemma statement (Figure 4.7).

```
for (let i = 0 to hi) begin
  (* body, i : bv hi *)
end
```

Figure 4.7: For-loop notation in Golemma.

The loop bound *hi* must be a natural number fixed at elaboration time, so the loop always runs a statically known number of iterations, as required for synthesizability. During verification, for-loop semantics reduce to a loop rule that requires an explicit loop invariant.

4.3 Specifying and proving correctness

Proving that a Golemma module is correct requires a way to express what the module is supposed to do and a way to connect that description to the module's behavior. This section describes both. It first introduces how to write relational specifications for Golemma methods and rules that are then composed into a modular specification (Section 4.3.1). Finally, it discusses the proof strategy needed to show that the implementation refines the specification (Section 4.3.2).

4.3.1 Writing specifications

A specification describes the set of valid behaviors a module may exhibit. Specifications do not need to be executable; they are relational, and it is reasonable for them to use existential quantifiers, which are expressive and enable nondeterminism naturally. The only constraint is that a specification must cover the same interface as the implementation it is meant to describe. The interface acts as the specification boundary, and both the concrete module and its specification provide an implementation of every method declared there, with matching argument and return types. The rest of this subsection develops a specification for the greatest-common-divisor module from Figure 4.1, using it to illustrate the key features of this specification style.

Abstract state. Every specification module has an associated abstract state, which is an ordinary Gallina type chosen to model the module's behavior at the desired level of abstraction. For the GCD module, the abstract state records both the original inputs and the current state of the Euclidean algorithm. The fields a and b are the original operands loaded into the module, while a' and b' are the current operands after some number of *step* calls. The representation invariant (*wf*) states that the current operands still have the same greatest common divisor as the original inputs. Figure 4.8 shows the record.

```
Record SpecState := mkState {
  a  : nat;
  b  : nat;
  a' : nat;
  b' : nat;
  wf : gcd a b = gcd a' b';
}.
```

Figure 4.8: Abstract state record for the GCD specification.

The abstract state lives entirely in Gallina; it is only the implementation that requires the use of the hardware-description language. This specification style is advantageous: proofs, invariants, and lemmas about the specification are stated in terms of natural-number arithmetic and the standard GCD function rather than in terms of bitvectors.

Predicate-transformer semantics. Each method specification is written in a continuation-passing style. Rather than producing an output value directly, a specification takes a postcondition P and is responsible for establishing that P holds of its outputs. This continuation-passing convention gives a uniform way to state both what the method requires of its inputs (preconditions) and what it guarantees about its outputs (postconditions). Golemma provides two specification types. Value-method specifications have the following type:

$$(\mathit{args} \times \mathit{State}) \rightarrow (\mathit{ret} \rightarrow \mathrm{Prop}) \rightarrow \mathrm{Prop}$$

Action-method specifications have the following type, where the postcondition additionally receives the next state:

$$(\mathit{args} \times \mathit{State}) \rightarrow ((\mathit{ret} \times \mathit{State}) \rightarrow \mathrm{Prop}) \rightarrow \mathrm{Prop}$$

In both cases the outermost proposition is the body of the specification.

Value methods. The *result* method returns the correct result from the GCD computation and *finished* returns whether the current second operand b' is zero. Each specification simply calls P on the appropriate return value (Figure 4.9).

```
Definition result : MRVSpec State (mkMetSig <> Bit#(16)) :=
  fun '(_, st) P =>
    st.(b') = 0 /\ P (nat_to_bv (gcd st.(a) st.(b))).

Definition finished : MRVSpec State (mkMetSig <> Bit#(1)) :=
  fun '(_, st) P =>
    P (if (st.(b') == 0) then 1%bv else 0%bv).
```

Figure 4.9: Value-method specifications for *result* and *finished*.

The postcondition is stated by choosing what term to apply P to. Different terms produce stronger or weaker guarantees; the specification's job is to apply P to exactly the value the method is guaranteed to return.

Action methods and preconditions. Action-method specifications additionally describe how the abstract state evolves. The *load* method sets the two operands to the supplied arguments (Figure 4.10).

```
Definition load : MRASpec State (mkMetSig <Bit#(16) a, Bit#(16) b> Bit#(0)) :=
  fun '(args, _) P =>
    let a := bv_to_nat args[[@0]] in
    let b := bv_to_nat args[[@1]] in
      P (0%bv, mkState a b a b _).
```

Figure 4.10: Action-method specification for *load*.

The successor state stores the loaded operands twice: once as the fixed original inputs and once as the current operands. The proof witness *pf* establishes $gcd\ a\ b = gcd\ a\ b$, so the invariant holds immediately after loading.

The *step* method is more interesting because the implementation aborts when the current second operand is zero. A precondition on the method is expressed as a conjunct placed outside of P, before any existential quantification (Figure 4.11).

```
Definition step : MRASpec State (mkMetSig <> Bit#(0)) :=
  fun '(_, st) P =>
    st.(b') <> 0 /\
    let next :=
      if decide (st.(a') >= st.(b')) then
        mkState st.(a) st.(b) (st.(a') - st.(b')) st.(b') _
      else
        mkState st.(a) st.(b) st.(a') (st.(b') - st.(a')) _ in
      P (0%bv, next).
```

Figure 4.11: Action-method specification for *step*, with precondition outside P.

Since $st.(b') \neq 0$ sits outside P, it is a delegated obligation that callers of this submodule must discharge. The refinement proof must show that the implementation never invokes *step* in a state where the abort path can be reached. Postconditions, by contrast, are captured by constructing the next abstract state passed to P; here that state preserves the original a and b fields while updating only a' and b'.

Representation invariants. Not every tuple of natural numbers is a valid intermediate state for a GCD computation. The *SpecState* record shown earlier encodes this constraint through the *wf* field: a Rocq proof that the current pair (a', b') has the same GCD as the original pair (a, b). This proof field is a representation invariant that every abstract state must carry.

Since *wf* is a field of the record, Rocq's type system ensures it is present in every *SpecState* value. The type system disallows constructing an abstract state that violates it. Action-method specifications that modify the state have a proof obligation to discharge the invariant. For example, the *step* specification must supply a proof that the updated current pair still has GCD $gcd\ \ st.(a)\ \ st.(b)$ whenever it constructs the next *SpecState*.

Connection to abstract data types. This specification style is directly analogous to the theory of abstract data types (ADTs). In an ADT, an abstraction function (AF) maps concrete states to abstract values, and each operation is specified by its effect on that abstract model. The same structure appears here: the *SpecState* record is the abstract model, and each method specification describes the contract that any correct implementation must satisfy. The predicate-transformer style mirrors Hoare-logic-style method contracts. A precondition is a proposition outside P that constrains what the method requires of its inputs; a postcondition is a proposition inside P that describes what the method guarantees about its outputs. Similar to ADT specifications, these contracts are relational. Specifications are allowed to use arbitrary Gallina propositions, including existential quantification, so they can express nondeterminism cleanly. An implementation that can nondeterministically return any element of a set satisfying some predicate is expressed by quantifying existentially over the return value. The specification makes no commitment about which element is chosen. This feature is particularly useful when specifying the issue logic of an out-of-order processor scheduler, where the choice of which ready instruction to issue is left unspecified. The abstraction function appears explicitly as the simulation relation between the concrete register state of the implementation and the abstract state, when establishing refinement.

4.3.2 Writing proofs

A functional-correctness proof in Golemma is a proof of module refinement: every behavior the implementation can exhibit is permitted by the specification. The proof is organized around a simulation relation R that pairs implementation states with their

abstract counterparts. The proof proceeds method-by-method, showing that each implementation method satisfies the postcondition demanded by its specification whenever R holds of the current state.

Refinement. A simulation relation $R : \mathcal{I} \to \mathcal{S} \to \text{Prop}$ relates each implementation state $i \in \mathcal{I}$ to an abstract specification state $s \in \mathcal{S}$. Throughout the definitions below, φ denotes a predicate transformer for the implementation, and ψ denotes the corresponding predicate transformer for the specification, where each is a predicate transformer of the appropriate type from Section 4.3.1. Given such a relation, refinement comes in three variants: value-method refinement, action-method refinement, and rule refinement.

Definition 4.3.1 (Value-method refinement)**.** Let φ and ψ be value-method predicate transformers over states $\mathcal{I}$ and $\mathcal{S}$, respectively. φ refines ψ under R when, for every related pair $(i, s) \in R$, every argument α, and every postcondition $P : \text{ret} \to \text{Prop}$,

$$\psi(\alpha,\ s,\ P)\ \Rightarrow\ \varphi(\alpha,\ i,\ P).$$

Value-method refinement is straightforward: if the specification establishes postcondition P, so must the implementation. Action methods will update state, so their refinement condition must also account for how the two states evolve:

Definition 4.3.2 (Action-method refinement)**.** Let φ and ψ be action-method predicate transformers over states $\mathcal{I}$ and $\mathcal{S}$, respectively. φ refines ψ under $R : \mathcal{I} \to \mathcal{S} \to \text{Prop}$ when, for every $(i, s) \in R$, every argument α, and every postcondition $P : (\text{ret} \times \mathcal{S}) \to \text{Prop}$,

$$\psi(\alpha,\ s,\ P)\ \Rightarrow\ \varphi(\alpha,\ i,\ \lambda\,(r,\ i').\ \exists\, s'.\ P(r,\ s') \wedge R(i',\ s')).$$

Rather than demanding a unique successor state, the existential allows the implementation to reach any state i' for which there exists a specification successor s' still in R and satisfying P. Rules carry no arguments and produce no return values, but they do update state. Rule refinement follows the same shape as action-method refinement, with the added flexibility of a possibly different post-relation R' for relating successor states:

Definition 4.3.3 (Rule refinement)**.** Let φ and ψ be rule predicate transformers over states $\mathcal{I}$ and S, respectively. φ refines ψ under $(R, R' : \mathcal{I} \to S \to \text{Prop})$ when, for every $(i, s) \in R$ and every postcondition $P : S \to \text{Prop}$,

$$\psi(s, P) \;\Rightarrow\; \varphi(i, \lambda\, i'.\, \exists\, s'.\, P(s') \wedge R'(i', s')).$$

The use of R' as the post-relation separates the invariant needed to fire a rule from the invariant that holds once it has fired, which is useful when rules have side-conditions that are established by other rules in the same module.

Module refinement packages all of the method-level and rule-level obligations together with a proof that the reset states are related:

Definition 4.3.4 (Module refinement)**.** A module $\mathcal{M}_{\mathcal{I}}$ refines a module $\mathcal{M}_S$ when there exist simulation relations $R, R' : \mathcal{I} \to S \to \text{Prop}$ and initial states $i_0 \in \mathcal{I}$, $s_0 \in S$ such that:

1. $R(i_0, s_0)$ (the reset states are related);
2. every external method of $\mathcal{M}_{\mathcal{I}}$ refines the corresponding method of $\mathcal{M}_S$ under R; and
3. every rule of $\mathcal{M}_{\mathcal{I}}$ refines the corresponding rule of $\mathcal{M}_S$ under (R, R').

Example. We walk through a complete functional-correctness proof for the GCD module as a worked example. The proof obligations reduce to a collection of per-method-refinement lemmas. The most important design decision for the proof is the choice of simulation relation R. The relation asserts that each register value, converted from a bitvector to a natural number, agrees with the corresponding abstract field. Figure 4.12 shows the Rocq definition.

```
Definition R (ist : ImplState) (sst : SpecState) : Prop :=
  bv_to_nat ist.(reg_a) = sst.(a') /\ bv_to_nat ist.(reg_b) = sst.(b').
```

Figure 4.12: Simulation relation for the GCD module.

With R fixed, each method proof follows the same pattern: use the equalities in R to rewrite the implementation's register values as their abstract counterparts, then

show the resulting expression matches the postcondition P of the specification. Action-method proofs require exhibiting a successor abstract state and showing that R is restored after the step. The lemmas below demonstrate an example proof for each method of the GCD module.

Lemma 4.3.1. *For all a, b, x, if* $\gcd(a, b) = \gcd(x, 0)$ *then* $x = \gcd(a, b)$.

Lemma 4.3.2. *For all a, b, if $b \geq a$ then* $\gcd(a, b) = \gcd(b, b - a)$.

Lemma 4.3.3. *For all a, b,* $\gcd(a, b) = \gcd(b, a)$.

Lemma 4.3.4. *finished refines its method specification under R.*

Proof. Assume $(i, s) \in R$. The implementation checks whether $i.b$ is zero; the specification checks whether the abstract field $s.b'$ is zero. Since R equates the two (modulo type conversion), both checks yield the same Boolean, so the implementation returns exactly the value the specification prescribes. □

Lemma 4.3.5. *result refines its specification under R, provided $s.b' = 0$. Moreover, the value returned by result equals* $\gcd(s.a, s.b)$.

Proof. Assume $(i, s) \in R$. The specification gives the bitvector encoding of $s.a'$. Since R equates $i.a$ with $s.a'$, the implementation's return value is equal to what the specification states. The representation invariant gives $\gcd(s.a, s.b) = \gcd(s.a', 0)$, and Lemma 4.3.1 then concludes $i.a = s.a' = \gcd(s.a, s.b)$. □

Lemma 4.3.6. *load refines its specification under R.*

Proof. Assume $(i, s) \in R$ and arguments a_{val}, b_{val}. The specification produces a successor state in which all four fields are set to those argument values (the original inputs a and b and the current operands a' and b' are all initialized simultaneously). The implementation writes the same two values into the corresponding registers. Taking the successor abstract state to be the one the specification constructs, R holds immediately from the register-write semantics, and the representation invariant $\gcd(a, b) = \gcd(a', b')$ reduces to $\gcd(a_{val}, b_{val}) = \gcd(a_{val}, b_{val})$, which is trivially true. □

Lemma 4.3.7. *step refines its specification under R, provided $s.b' \neq 0$.*

Proof. Assume $(i, s) \in R$. Since R equates $s.b'$ with $i.b$, $i.b$ is also nonzero, so the implementation does not reach the *abort* path. Both the specification and the implementation perform the same conditional subtraction, and since the register values equal

the abstract fields by R, the two conditionals take the same branch. After the update, the new register values equal the new abstract fields, so R is restored.

Now we must show that $\gcd(a, b) = \gcd(a'', b'')$ holds for the successor state (a'', b''). In the first branch, $s.a' \geq s.b'$, so $(a'', b'') = (s.a' - s.b',\ s.b')$. The invariant $\gcd(a, b) = \gcd(s.a', s.b')$ and Lemma 4.3.2 give $\gcd(s.a', s.b') = \gcd(s.a' - s.b', s.b')$, establishing the invariant. In the second branch, $s.a' < s.b'$, so $(a'', b'') = (s.a',\ s.b' - s.a')$. Applying Lemma 4.3.3 gives $\gcd(s.a', s.b') = \gcd(s.b', s.a')$, then Lemma 4.3.2 gives $\gcd(s.b', s.a') = \gcd(s.b' - s.a', s.a')$, and a final application of Lemma 4.3.3 yields $\gcd(s.a', s.b' - s.a')$, establishing the invariant. □

Theorem 4.3.1. *The GCD implementation module refines the GCD specification module.*

Proof. Apply Definition 4.3.4: the reset states are related by construction, and Lemmas 4.3.4 to 4.3.7 discharge the per-method obligations. □

Chapter 5

Case studies

This chapter demonstrates the expressivity of the two verification frameworks covered in Chapters 3 and 4 by presenting case studies. First, it discusses a proof of timing security for a variable-latency multiplier using the Slate framework (Section 5.1). Then, it uses the Golemma framework to present a correctness proof for a ring buffer (Section 5.2).

5.1 Timing security: Variable-latency multiplier

This section presents the timing-security verification of a variable-latency integer multiplier using Slate. The multiplier computes the product of two 8-bit operands using a shift-and-add algorithm. Since the number of cycles depends on the first operand and not the second, the second operand can be treated as a secret without affecting the timing behavior. The goal is to establish hardware constant time with respect to an erasure contract that hides the second operand and the final product.

5.1.1 Interface

The multiplier exposes six methods that are summarized in Figure 5.1 as a Golemma interface. Three operations are stateful: *enq* loads a pair of operands and starts the computation, *step* advances the shift-and-add algorithm by one step, and *deq* retrieves the 16-bit product once it is ready. Three additional operations are observational: *empty*, *busy*, and *full*.

```
Definition Multiplier : ModSig :=
  interface
    method  Bit#(1)   empty;
    method  Bit#(1)   busy;
    method  Bit#(1)   full;
    method  Action    enq  <Bit#(8) a, Bit#(8) b>;
    method  Action    step;
    method  Bit#(16)  deq;
  endinterface.
```

Figure 5.1: Interface definition for the multiplier. Action methods (*enq*, *step*) mutate state; value methods return a result without modifying state.

Figure 5.2 shows what the port-level signals of the multiplier might look like at the RTL abstraction. Each method is represented as a group of signals: an enable input, driven by the caller to invoke the method; and a ready output, driven by the module to indicate that the method's guard holds and the call will succeed. These signals form a handshaking protocol that ensures that the method will only execute if both signals are latched high on the same cycle. Stateful methods may also carry data signals: *enq* takes the two operands *a* and *b* as inputs, while *deq* produces the 16-bit product *c* as an output. The observational methods (*empty*, *busy*, *full*) each return a single Boolean result and do not require handshaking.

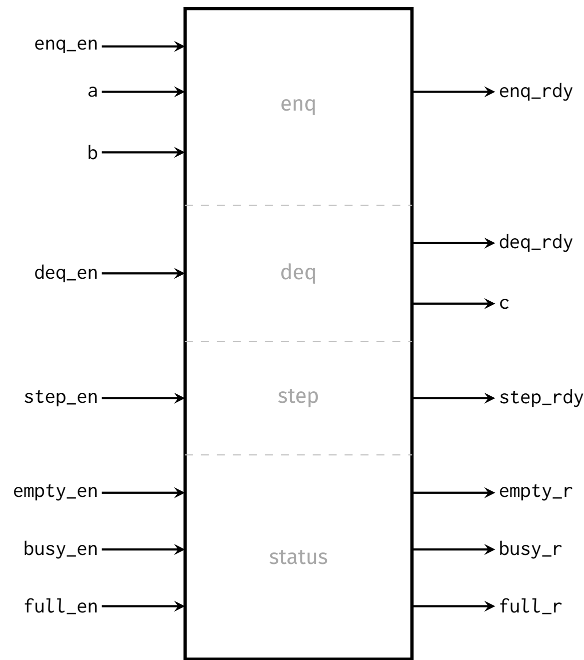

Figure 5.2: Port-level block diagram of the variable-latency multiplier. Each method exposes an enable input (asserted by the caller) and a ready output (asserted by the module when the guard holds).

5.1.2 Erasure contract

The secrets are the second operand *b* of *enq* and the product *c* of *deq*. The attacker's observation model is as follows: the attacker can see when any method is called, but not the values of *b* or *c* being transferred. Figure 5.3 highlights the attacker-visible signals under this threat model.

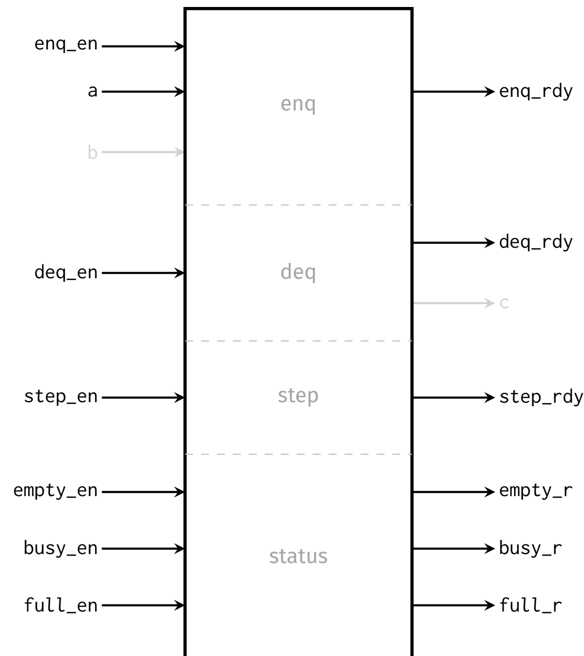

Figure 5.3: Attacker-visible signals of the multiplier under the erasure contract. Faded signals are erased: *b* is replaced by zero and the product *c* is mapped to *None*.

The erasure contract $\mathcal{E} = (erase_inp, erase_out)$ formalizes this model in two functions shown in Figure 5.4. *erase_inp* replaces the second operand with zero on every *enq* and leaves all other inputs unchanged. *erase_out* replaces the product in *deq* with zero, and maps all remaining outputs to observable.

```
Definition erase_inp (m : I) : I :=
  match m with
  | InpEnq a _ => InpEnq a 0%bv
  | other      => other
  end.

Definition erase_out (out : option O) : option O :=
  match out with
  | Some (OutDeq _)  => Some (OutDeq 0%bv)
  | other            => other
  end.
```

Figure 5.4: Erasure functions for the multiplier. The second operand and the product are both hidden.

5.1.3 Implementation

The multiplier holds five registers: *src*1 and *src*2 (the two 8-bit operands), *dst* (the 16-bit accumulator), *count* (a 3-bit iteration counter), and *phase*. Figure 5.5 shows the state record.

```
Variant Phase := Empty | Busy | Full.

Record State := {
  src1  : Bit#(8);
  src2  : Bit#(8);
  dst   : Bit#(16);
  count : Bit#(3);
  phase : Phase;
}.
```

Figure 5.5: State type and phases of the multiplier.

Each method call to *step* invokes the *shift_and_add* subroutine defined in Figure 5.6. If *src*1 is already zero then *shift_and_add* is a no-op. If the least-significant bit of *src*1 is one, the subroutine adds *src*2 shifted left by *count* to *dst*. If the least-significant bit is instead zero, nothing is added to *dst*. In both cases, *src*1 is right-shifted by one and *count* is incremented. Once *src*1 reaches zero, the outer *step* transitions the phase to *Full*, making the result available for dequeue.

5.1.4 Verification

The central observation is that the timing of the multiplier depends entirely on *src*1: the number of steps before the phase transitions to *Full* is exactly the position of the most-significant set bit of *src*1. The value of *src*2 affects only the accumulator *dst* and therefore does not influence any externally observable signal. This observation is captured by the simulation relation in Figure 5.7, which equates the *phase*, *count*, and *src*1 registers of two states but makes no requirement on *src*2 or *dst*.

The proof uses self-simulation (Lemma 3.3.2): the implementation is simulated against itself with the erased arguments and results. It suffices to show that R is preserved by each operation.

```
Definition shift_and_add :=
  begin
    let src1'  = `src1`read();
    let src2'  = `src2`read();
    let count' = `count`read();
    let dst'   = `dst`read();
    if (src1' == 0%bv)
      pass
    else begin
      let t =
        if (truncate1 src1' == 1)%bv then
          zero_extend src2'
        else
          (16`0x00)%bv;
      `dst`write((dst' + bv_shiftl t (zero_extend count'))%bv);
      `count`write((count' + 1)%bv);
      `src1`write(bv_shiftr src1' 1)
    end
  end.
```

Figure 5.6: The *shift_and_add* subroutine. On each call it processes one bit of *src*1, conditionally adding a shifted copy of *src*2 to the accumulator.

Lemma 5.1.1. *enq self-simulates under* $(R, \mathcal{E})$.

Proof. Assume $(s_1, s_2) \in R$. The guard checks $s_1.phase = Empty$; since R equates the phases, $s_2.phase = Empty$ as well, so both copies take the same branch. When the guard succeeds, both copies set $src1 := a$, $dst := 0$, $count := 0$, and $phase := Busy$. The first copy additionally sets $src2 := b$ while the erased copy sets $src2 := 0$; since R imposes no constraint on *src*2 or *dst*, the successor states remain related. Both copies return void, and $\mathcal{E}$ erases void to void, so the observed outputs agree. □

Lemma 5.1.2. *deq self-simulates under* $(R, \mathcal{E})$.

Proof. Assume $(s_1, s_2) \in R$. The guard checks $s_1.phase = Full$; since R equates the phases, both copies take the same branch. Both copies set $phase := Empty$ and return *dst*. The products $s_1.dst$ and $s_2.dst$ may differ, but $\mathcal{E}$ erases the output by replacing the product with zero, so the observed outputs are equal. In the successor states, *phase*, *count*, and *src*1 are unchanged, so R is preserved. □

```
Definition R (st st' : State) : Prop :=
  st.(phase) = st'.(phase) /\
  st.(count) = st'.(count) /\
  st.(src1)  = st'.(src1).
```

Figure 5.7: Simulation relation for the multiplier. Two states are related when they agree on the timing-relevant registers.

Lemma 5.1.3. *step self-simulates under* $(R, \mathcal{E})$.

Proof. Assume $(s_1, s_2) \in R$. The guard checks $s_1.phase = Busy$; since R equates the phases, both copies take the same branch. Inside the guard, both copies check whether *src*1 is zero; since R equates *src*1, both take the same inner branch. If $s_1.src1 = 0$, both copies set $phase := Full$ and leave the remaining registers unchanged, so R holds for the successor states. Otherwise, both right-shift *src*1 by one and increment *count* identically, preserving the *src*1 and *count* components of R. The accumulation into *dst* may differ since $s_1.src2$ and $s_2.src2$ may differ, but *dst* is not tracked by R, so R is preserved in both branches. □

Lemma 5.1.4. *empty, busy, and full each self-simulate under* $(R, \mathcal{E})$.

Proof. Assume $(s_1, s_2) \in R$. Each status query reads only *phase*, compares it against a fixed constant, and returns a Boolean. Since R equates the two phases, both copies return the same Boolean, leave the state unchanged, and produce equal outputs. □

Theorem 5.1.1. *The multiplier satisfies hardware constant time with respect to the erasure contract* $\mathcal{E}$.

Proof. Apply Lemma 3.3.2 using simulation relation R. Lemmas 5.1.1 to 5.1.4 discharge the resulting proof obligations. The reset state satisfies R trivially since both copies of the reset record are identical. □

5.2 Functional correctness: Ring buffer

This section presents the proof of correctness for a ring buffer. The ring buffer holds entries of parameterized type ε and supports a capacity of 2^N slots. Beyond the standard enqueue and dequeue operations, the ring buffer also exposes a squash method that retracts the tail pointer to an earlier position. These extra operations are motivated

by their use in an out-of-order processor's reorder buffer, where instructions that are resident in the buffer may need to be squashed upon a branch misprediction.

5.2.1 Interface

The ring buffer exposes nine methods, shown in Figure 5.8. Five are value methods: *empty* and *full* return the status of the buffer; *first* returns the element at the head; *tail* returns the current tail index; and *sub* returns the element at a given logical index. Four are action methods: *enq* inserts an entry at the tail and advances the tail, *deq* removes the element at the head, *squash* retracts the tail to a given earlier position discarding pending entries, and *upd* overwrites an existing entry in-place. Each action method guards against invalid calls: *enq* aborts when the buffer is full; *deq* and *first* abort when it is empty; and *sub*, *upd*, and *squash* abort when the provided index does not have a valid element there.

```
Definition RingBuffer (ε : type) (N : nat) : ModSig :=
  interface
    method  Bit#(1)        empty;
    method  Bit#(1)        full;
    method  ε              first;
    method  Bit#(N)        tail;
    method  ε              sub     <Bit#(N) idx>;
    method  Action         deq;
    method  Action         enq     <ε e>;
    method  Action         squash  <Bit#(N) tail'>;
    method  Action         upd     <Bit#(N) idx, ε e>;
  endinterface.
```

Figure 5.8: Interface definition for the ring buffer, parameterized by entry type and address width.

5.2.2 Specification

Since specifying the behavior of pointers requires modular arithmetic, both the specification and the implementation rely on a library of modular arithmetic on natural numbers. Figure 5.9 summarizes the key definitions.

```
Definition nat_wrap x  := x mod 2 ^ N.
Definition x ⊕ y := (x + y) mod 2 ^ N.
Definition x ⊖ y := (x + 2 ^ N - y mod 2 ^ N) mod 2 ^ N.
Definition x ⊗ y := (x * y) mod 2 ^ N.
Definition ⊖ x := (2 ^ N - x mod 2 ^ N) mod 2 ^ N.
Definition x ≡ y := nat_wrap x = nat_wrap y.
```

Figure 5.9: Modular arithmetic definitions over natural numbers.

These operations form a commutative ring under the equivalence $x \equiv y$, registered with Rocq's *ring* tactic, so that goals such as $(x \oplus y) \ominus y \equiv x$ close automatically. The library also proves that each operation corresponds to its bitvector counterpart under the conversion *nat_to_bv*, enabling implementation-level bitvector goals to be translated into specification-level modular-arithmetic goals during refinement proofs.

The specification represents the ring buffer abstractly as an ordered list of entries together with a physical tail index. Figure 5.10 shows the abstract state record.

```
Record State := mkState {
  entries       : list ε;
  tail          : nat;
  size          := length entries;
  head          := tail ⊖ size;
  wf            : tail < 2 ^ N /\ size <= 2 ^ N;
  full          := size = 2 ^ N;
  empty         := size = 0;
  index i       := i ⊖ head;
  valid_index i := i < 2 ^ N /\ index i < size;
}.
```

Figure 5.10: Abstract state for the ring-buffer specification.

The abstract model stores the logical contents of the buffer as the list *entries*, ordered from head to tail, along with a *tail* index. The *head* index is derived as *tail* ⊖ *size*, the ring slot that is *size* positions behind the tail. A logical index is a ring address j that falls within the live portion of the buffer: one satisfying $s.valid_index(j)$, meaning the forward distance $j \ominus s.head$ is strictly less than $s.size$. The *sub* and *upd* operations access elements by logical index, and *squash* accepts a new tail that must itself be a logical index. The representation invariant *wf* bounds *tail* and *size* to stay within 2^N. When the head equals the tail, the buffer may be full or empty; the size disambiguates the two

cases, so no explicit full flag appears in the abstract state. The method specifications are written in the predicate-transformer style described in Section 4.3.1. Figure 5.11 shows the specifications for the two core action methods.

```
Definition enq st e `{~ full st} : State :=
    mkState (entries st ++ [e]) (nat_wrap_succ (tail st)) _.

Definition enq' :=
  fun '(args, st) P =>
    exists Hfull : ~ full st,
      P (tt, enq st (args[[@0]]) Hfull).

Definition deq st `{~ empty st} : State :=
  mkState (tl (entries st)) (tail st) _.

Definition deq' :=
  fun '(_, st) P =>
    exists Hempty : ~ empty st,
      P (tt, deq st Hempty).
```

Figure 5.11: Predicate-transformer specifications for *enq* and *deq*. The existential witness encodes the precondition as a proof term passed directly to the abstract operation.

The *enq* operation on the abstract state appends the new element to *entries* and increments *tail* by one modulo 2^N. The *deq* operation drops the head of *entries* and leaves *tail* unchanged, so that subsequent indices shift down by one. The *squash* operation truncates *entries* to the first $(index\ st\ tail')$ elements and sets $tail := tail'$. The *upd* and *sub* operations perform a list update and a list lookup, respectively, at the physical offset computed by *index*.

5.2.3 Implementation

The implementation maintains a fixed-size vector of registers for entries and three control registers: *head* and *tail* as N-bit values and *full* as a one-bit flag. Figure 5.12 shows the module declaration.

The *enq* method writes the new element to $entries[tail]$, advances *tail* by one using bitvector addition (which wraps naturally at 2^N), and updates *full* by checking whether the new tail equals *head*. The validity check for *sub*, *upd*, and *squash* is computed as the comparison $(idx - head) < (tail - head)$, which effectively checks if $entries[idx]$ is a valid element. Unlike the specification, which derives the *full* condition from the size,

```
module mkRingBuffer#(RingBuffer#(ε, N));
  Vector#(2 ^ N, Reg#(ε)) entries <- replicateM(mkReg());
  Reg#(Bit#(N)) head <- mkReg();
  Reg#(Bit#(N)) tail <- mkReg();
  Reg#(Bit#(1)) full <- mkReg();
```

Figure 5.12: Module declaration for the ring-buffer implementation.

the implementation must maintain the *full* flag explicitly to disambiguate whether the queue is empty or full when the head equals the tail.

5.2.4 Verification

The refinement relation R connects implementation states to specification states as shown in Figure 5.13. The relation has four components. First, for every valid logical index i in the specification's ring buffer, the corresponding entry in the implementation's ring buffer matches. The notation xs !!! i denotes a lookup into a list xs at position i, returning a default value if the index is out of bounds. Second, the implementation *head* and *tail* registers equal the bitvector representations of the specification's head and tail. Third, the implementation *full* bit matches with the specification's full predicate. Notably, R imposes no constraint on vector slots at indices that are not valid in the specification, so the implementation is allowed to hold arbitrary data in those positions.

```
Definition R (ist : ImplState) (sst : SpecState) :=
  (forall i, Spec.valid_index sst i ->
     Impl.entries ist !!! i = Spec.sub sst i) /\
  Impl.head ist = nat_to_bv (Spec.head sst) /\
  Impl.tail ist = nat_to_bv (Spec.tail sst) /\
  Impl.full ist = Spec.full sst.
```

Figure 5.13: Refinement relation for the ring buffer. Implementation entries must agree with the specification list at every valid index, with control registers matching under bitvector conversion.

The main theorem is proved by first applying Definition 4.3.4, which produces one proof obligation per-method. We derive a suite of 52 helper lemmas that reason about combinations of methods (e.g. taking the tail of the specification buffer after a dequeue). We use these helper lemmas to rewrite proof obligations without internally inspect-

ing the components of the specification state. Reasoning about the implementation requires destructing the state and combining bitvector-arithmetic facts with modular-arithmetic lemmas.

Lemma 5.2.1. *The full, head and tail methods each refine their specifications under R.*

Proof. Each follows immediately from R, which directly relates the registers in the implementation with their counterparts in the specification. □

Lemma 5.2.2. *The empty method refines its specification under R.*

Proof. Assume $(i, s) \in R$. The implementation checks $\neg i.full \wedge (i.head = i.tail)$. By R, we are given the following:

$$(i.full = s.full) \wedge (i.head = nat_to_bv(s.head)) \wedge (i.tail = nat_to_bv(s.tail)).$$

A helper lemma establishes that $s.head = s.tail \iff s.full \vee s.empty$. Therefore the implementation's conjunction reduces to $\neg s.full \wedge (s.head = s.tail)$, which holds exactly when $s.empty$. □

Lemma 5.2.3. *The first method refines its specification under R.*

Proof. Assume $(i, s) \in R$. The nonempty guard agrees with the specification's precondition by the same reasoning as Lemma 5.2.2. A helper lemma shows that when s is non-empty, $s.head$ is a valid index. By R, we get $i.entries[s.head] = s.sub(s.head) = s.first$. □

Lemma 5.2.4. *The sub method refines its specification under R.*

Proof. Assume $(i, s) \in R$ and logical index j. The implementation's range check computes $(j - i.head) < (i.tail - i.head)$ using bitvector arithmetic. This formula is equivalent to $(j \ominus s.head) < s.size$, which is the definition of $s.valid_index(j)$. By R, we get $i.entries[j] = s.sub(j)$ given $s.valid_index(j)$. □

Lemma 5.2.5. *The enq method refines its specification under R.*

Proof. Assume $(i, s) \in R$ and entry e. The fullness check agrees by Lemma 5.2.1. After writing $i.entries[i.tail] := e$ and advancing $i.tail$ by one, let i' and $s' = s.enq(e)$ be the successor states. We must show $(i', s') \in R$. For array agreement, fix a valid index j of s'. If j was already a valid index of s, then $i'.entries[j] = i.entries[j]$, and by R we get $i.entries[j] = s.sub(j) = s'.sub(j)$. Otherwise $j = s.tail$, the newly occupied slot, and $i'.entries[j] = e = s'.sub(j)$. For the remaining conjuncts, s' updates *head*, *tail*, and *full* in the same way as i', so those components of R are reestablished. □

Lemma 5.2.6. *The deq method refines its specification under R.*

Proof. Assume $(i, s) \in R$. The empty check agrees by Lemma 5.2.2. After advancing $i.head$ by one and clearing $i.full$, let i' and $s' = s.deq()$ be the successor states. We must show $(i', s') \in R$. For array agreement, if j is valid in s' then j was valid in s and $s'.sub(j) = s.sub(j)$; by R we get $i'.entries[j] = i.entries[j] = s'.sub(j)$. For the head pointer, $s'.head = s.head \oplus 1$, so by R we get $nat_to_bv(s'.head) = i.head + 1 = i'.head$. Dequeue always produces a nonfull state, so $i'.full = 0 = s'.full$. □

Lemma 5.2.7. *The squash method refines its specification under R.*

Proof. Assume $(i, s) \in R$ and new tail index $tail'$. The range check on $tail'$ translates to $s.valid_index(tail')$ by similar logic to Lemma 5.2.4. After setting $i.tail := tail'$ and $i.full := 0$, let i' and $s' = s.squash(tail')$ be the successor states. We must show $(i', s') \in R$. For array agreement, if j is valid in s' then j was valid in s and $s'.sub(j) = s.sub(j)$; by R we get $i'.entries[j] = i.entries[j] = s'.sub(j)$. By construction, $i'.tail = tail'$ and $i'.full = 0$ match s'. □

Lemma 5.2.8. *The upd method refines its specification under R.*

Proof. Assume $(i, s) \in R$, logical index j, and entry e. The range check translates as in Lemma 5.2.4. After writing $i.entries[j] := e$, let i' and $s' = s.upd(j, e)$ be the successor states. We must show $(i', s') \in R$. For array agreement, fix a valid index k of s' (equivalently, of s, since *upd* preserves size). If $k = j$, then $i'.entries[k] = e = s'.sub(j)$. If $k \neq j$, then $i'.entries[k] = i.entries[k]$ and by R we get $i.entries[k] = s.sub(k) = s'.sub(k)$, since distinct valid indices map to distinct physical slots. The *head*, *tail*, and *full* registers are unaffected, so those components of R are preserved. □

Theorem 5.2.1. *The ring-buffer implementation refines the ring-buffer specification.*

Proof. Apply Definition 4.3.4. Lemmas 5.2.1 to 5.2.8 discharge the per-method obligations. The reset state satisfies R trivially since both the implementation and specification reset to an empty buffer with *head*, *tail*, and *full* all equal to zero. □

Chapter 6

Evaluation

This chapter evaluates the two verification frameworks developed in this thesis. Section 6.1 measures verification effort by counting lines of code across the specification, implementation, and proof components of each case study. Section 6.2 examines resource utilization by reporting the synthesis results for the hardware designs produced by the Golemma compiler.

6.1 Verification effort

Table 6.1 summarizes the lines of code (LoC) required to specify, implement, and verify each case study, measured with "`wc -l`". For the multiplier, all three components live in a single file; the breakdown reflects logical sections of that file: the circuit implementation, the erasure contract, and the proof. For the ring buffer, each component corresponds to one or more dedicated Rocq source files.

Module	Interface	Specification	Implementation	Proof	Total
Multiplier	—	44	132	86	262
Ring buffer	25	793	165	335	1,318

Table 6.1: Lines of code for each component across both case studies.

The multiplier is the more-compact case study. Its proof is straightforward and highly automated. Each of the six lemmas requires only unfolding the definitions and rewriting goals using the equalities supplied by the simulation relation. The ring-buffer case study is significantly larger. The specification file includes the abstract state and many helper lemmas used in the proof. The majority of the effort in the ring-buffer case study stems

from the large suite of helper lemmas. Despite the size of the specification, the proof-to-spec ratio is reasonable: the 335-line proof covers nine methods plus the reset-state proof obligation. Table 6.2 breaks down the total LoC across both repositories including work-in-progress modules.

Component	LoC
Slate	**990**
Framework code (Sections 3.2 and 3.3)	361
Variable-latency multiplier (Section 5.1)	264
Multi-cycle processor†	365
Golemma	**5,018**
Language extensions (Section 4.2)	1,051
Theories of natural numbers, bitvectors, etc.	1,975
Ring buffer (Section 5.2)	1,318
Reorder buffer†	674
Total	**6,008**

Table 6.2: Total lines of code across all personal contributions to this project. †Work-in-progress.

6.2 Resource utilization

To evaluate synthesizability, we ran the verified Golemma compiler on the verified ring buffer and synthesized the resulting RTL code with Yosys, an open-source synthesis suite. All configurations were targeted at a 250 MHz clock and were successfully implemented on a Xilinx VCU118 FPGA. The experiment sweeps two parameters of the ring buffer: the address width N, which gives a buffer depth of 2^N entries; and the buffer width M (in bits), corresponding to the N and ε parameters in the interface (Section 5.2). We evaluated $N \in \{3, 5, 7, 9\}$ and $M \in \{32, 128, 512\}$; the configuration $(N = 7,\ M = 512)$ timed out during synthesis, and larger configurations were not attempted.

Figure 6.1 shows a heatmap of total gate count as a function of N and M. Gate usage ranges from 1.9k gates at the smallest configuration $(N = 3,\ M = 32)$ to 188.8k at the largest successful configuration $(N = 5,\ M = 512)$, with the $N = 7$ row reaching 52.1k and 186.0k for $M \in \{32, 128\}$. Both dimensions drive gate count upward, but buffer

width has the stronger effect: quadrupling M while holding N fixed roughly quadruples the gate count.

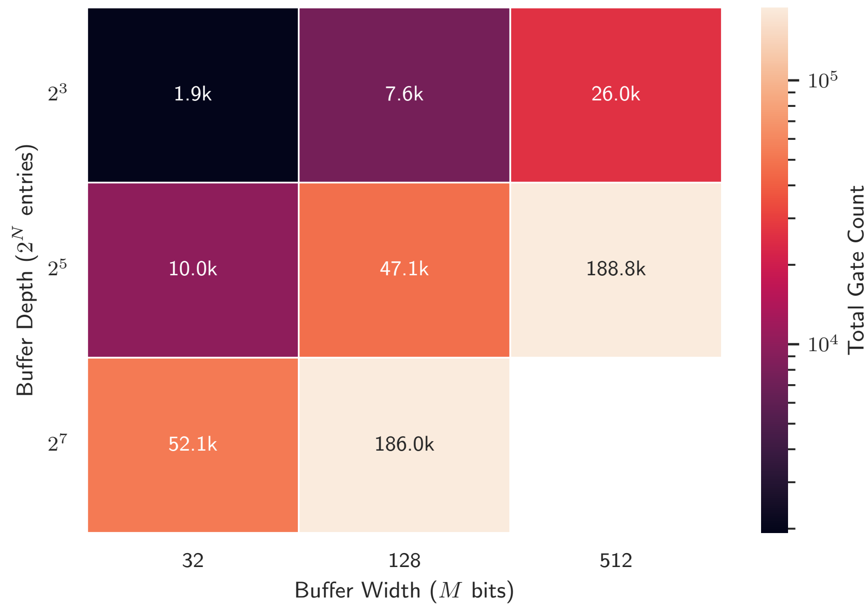

Figure 6.1: Total gate count across ring-buffer configurations. The cell at $N = 7$, $M = 512$ is absent due to a synthesis timeout.

Figure 6.2 shows synthesis time as a function of M for each depth. Synthesis time grows roughly exponentially with both parameters. The $N = 3$ designs complete in under one second across all widths, while the $N = 7$ configurations require hundreds of seconds at $M = 128$ and exceed the timeout (30 minutes) at $M = 512$.

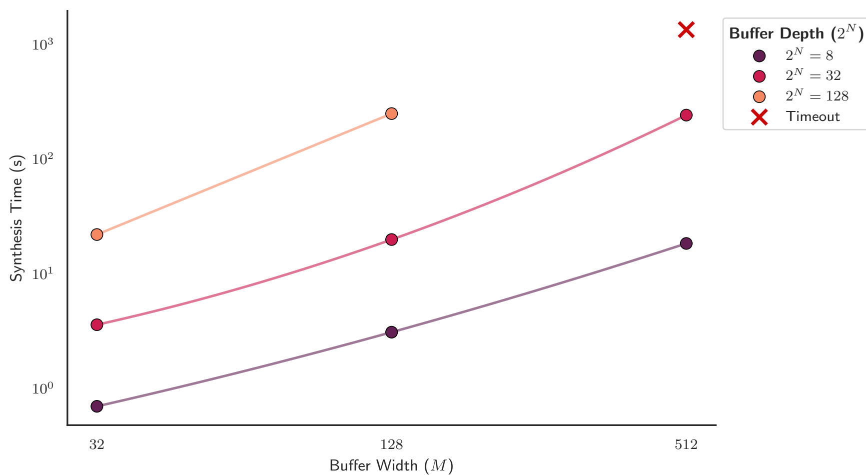

Figure 6.2: Synthesis time (log scale) versus buffer width for each buffer depth. The red cross at $N = 7$, $M = 512$ marks a timeout.

Figure 6.3 breaks down resource usage into logic gates, multiplexers, and flip-flops for each configuration. The dominant resource shifts depending on which parameter is varied: increasing buffer width M causes logic gates to claim a larger share of total resources, while increasing buffer depth N causes multiplexers to dominate, with flip-flops growing modestly as well.

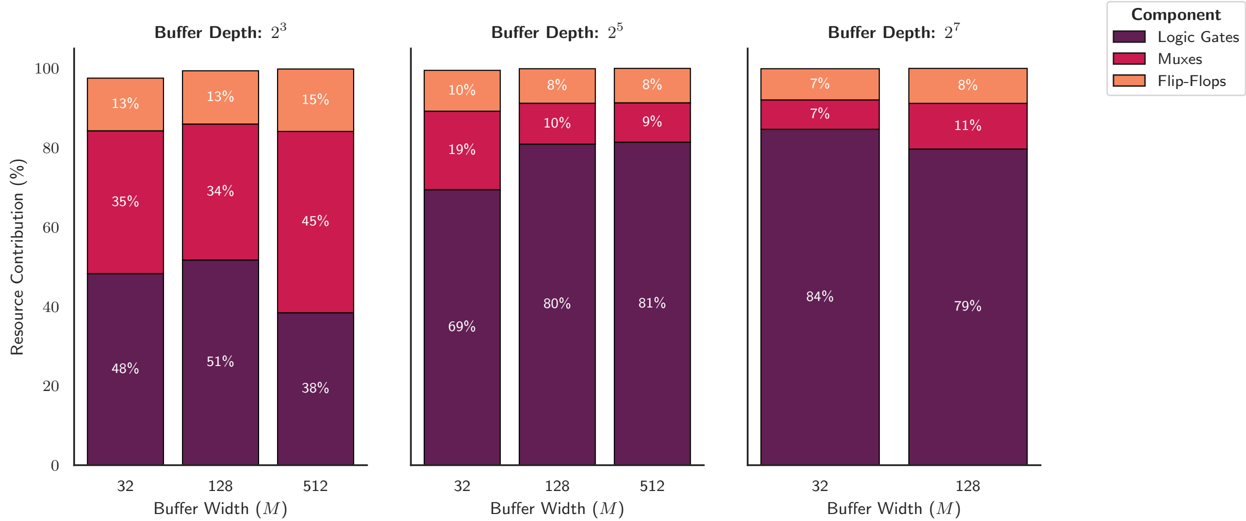

Figure 6.3: Resource breakdown (logic gates, muxes, flip-flops) as a percentage of total resources for each buffer depth. Increasing M raises the gate share; increasing N raises the mux share.

Overall, the results confirm that the Golemma-compiled ring buffer is synthesizable across a practical range of parameters and produces designs that meet timing on a real FPGA. The verified compiler provides an end-to-end path from a mechanically checked Rocq proof to hardware, and the parameterized interface allows the module to be instantiated at whatever depth and width a given application requires.

Chapter 7

Conclusion

This thesis presented two frameworks for mechanized verification of hardware circuits in Rocq. These frameworks target correctness and timing-security properties needed for a full proof of an out-of-order processor. Chapters 1 and 2 introduced the background and motivation behind this thesis. Chapter 3 introduced Slate and the hardware-constant-time property for timing security; Chapter 4 introduced Golemma and the modular-refinement property for functional correctness; and Chapter 5 applied both to case studies. Chapter 6 analyzed the amount of effort required to write proofs using these frameworks and investigated the resource utilization when synthesizing Golemma circuits.

7.1 Summary

The first framework, Slate, is a shallowly embedded language based on guarded state monads that models hardware circuits as deterministic state machines. Slate supports a proof strategy for hardware constant time, in which security reduces to a per-method simulation argument. Erasure contracts make the attacker's observation model explicit. This approach was exercised on a variable-latency multiplier, establishing that its timing behavior is independent of the secret second operand.

The second framework, Golemma, is a deeply embedded hardware-description language whose syntax is inspired by Bluespec SystemVerilog. Golemma provides a module system, typed interfaces, and a weakest-precondition semantics that supports compositional, per-method correctness proofs via modular refinement. Extensions to the language include type classes for synthesizing structs, enums, and bitflags, as well as a for-loop construct, enabling specifications to be stated directly over familiar Gallina

types." Functional correctness was demonstrated on a ring buffer whose specification is written using modular arithmetic over naturals.

Together, these contributions provide a solid foundation for scaling towards integrated proofs of correctness and timing security for a full out-of-order processor.

7.2 Future work

The immediate next step is to complete the correctness proofs for the remaining submodules of the out-of-order processor. The reorder buffer specification and implementation are already complete; the remaining work is to finish its correctness proof. After that, the renaming logic, the functional units, the physical register file, and the register-alias tables each require specifications and correctness proofs.

Given those correctness proofs and specifications, the natural target for a conference paper is a composed correctness and timing-security result for a complete out-of-order processor backend without speculation on memory operations. Looking further ahead, a natural extension is to augment the processor with a hardware defense against transient-execution attacks and to verify that the defense is sufficient. Speculative taint tracking [13] is a promising candidate, as it delays secret-dependent memory accesses until they are known to be non-speculative; adding STT to the Golemma model and proving that the resulting processor satisfies the same erasure-contract obligations would close the gap between the abstract security arguments from prior work and a concrete, synthesizable design.

Finally, an end-to-end theorem connecting the execution of secure source code to a verified processor would be another target for a conference paper. A similar end-to-end theorem has been explored for a pipelined RISC-V processor in Kami and the Bedrock2 compiler [15]. This proposed work would compose together prior work on retrofitting the Bedrock2 compiler with constant-time preservation [14] and a proof of correctness of a complex out-of-order machine in Golemma. Connecting those results would yield an end-to-end guarantee that a secret-free source program running on the verified out-of-order processor cannot leak information through timing side channels.

References

[1] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[2] N. Vasseghi, K. Yeager, E. Sarto, and M. Seddighnezhad, "200-MHz superscalar RISC microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 11, pp. 1675–1686, 1996.

[3] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: a platform for high-level parametric hardware specification and its modular verification," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, Aug. 2017.

[4] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind, "The essence of Bluespec: a core language for rule-based hardware design," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 243–257.

[5] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC'18. USA: USENIX Association, 2018, pp. 973–990.

[6] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19.

[7] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, Aug. 2018.

[8] G. Maisuradze and C. Rossow, "ret2spec: Speculative Execution using Return Stack Buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Com-*

munications Security, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2109–2122.

[9] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," 2018. [Online]. Available: https://arxiv.org/abs/1807.03757

[10] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 753–768.

[11] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. V. Bulck, D. Genkin, D. Gruss, F. Piessens, B. Sunar, and Y. Yarom, "Fallout: Reading Kernel Writes from User Space," 2019. [Online]. Available: https://arxiv.org/abs/1905.12701

[12] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-Software Contracts for Secure Speculation," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1868–1883.

[13] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-52. New York, NY, USA: Association for Computing Machinery, 2019, pp. 954–968.

[14] O. Conoly, A. Erbsen, and A. Chlipala, "Smooth, Integrated Proofs of Cryptographic Constant Time for Nondeterministic Programs and Compilers," *Proc. ACM Program. Lang.*, vol. 9, no. PLDI, Jun. 2025.

[15] A. Erbsen, S. Gruetter, J. Choi, C. Wood, and A. Chlipala, "Integration verification across software and hardware for a simple embedded system," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 604–619.

[16] S. Song, T. Dong, K. Nwabueze, J. Zanders, A. Erbsen, A. Chlipala, and M. Yan, "Securing Cryptographic Software via Typed Assembly Language," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 141–155.

[17] L.-A. Daniel, M. Bognar, J. Noorman, S. Bardin, T. Rezk, and F. Piessens, "ProSpeCT: Provably Secure Speculation for the Constant-Time Policy," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 7161–7178.

[18] F. Duxovni, "Mechanized Proofs that Hardware is Safe from Timing Attacks," Master's Thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2018, available at https://hdl.handle.net/1721.1/122053.

[19] S. Lau, T. Bourgeat, C. Pit-Claudel, and A. Chlipala, "Specification and Verification of Strong Timing Isolation of Hardware Enclaves," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1121–1135.

[20] A. Charguéraud, A. Chlipala, A. Erbsen, and S. Gruetter, "Omnisemantics: Smooth Handling of Nondeterminism," *ACM Trans. Program. Lang. Syst.*, vol. 45, no. 1, Mar. 2023.

[21] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying Constant-Time Implementations," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX, USA: USENIX Association, 2016, pp. 53–70.

[22] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, p. 1157–1210, Sep. 2010.