École Polytechnique Fédérale de Lausanne

School of Computer and Communication Sciences

Loopy combinational circuits in Graphiti

Master's Thesis in Computer Science
(Foundations of software)

Author:	Emily Kobler
Supervisors:	Yann Herklotz, Thomas Bourgeat
Start Date:	16.02.2026
Submission Date:	19.06.2026

Abstract

Formal hardware verification usually splits circuits into acyclic combinational logic and clocked sequential abstractions, obscuring the middle ground where gate-level delays combine with feedback loops, most notably the D-flip-flop.

This thesis develops a methodology for verifying such loopy combinational circuits within Graphiti, a Lean 4 refinement framework. We model a circuit as a state machine whose wires carry partial information refined monotonically over time, prove that each gate-level implementation refines a trusted specification, and develop reusable abstractions that make the proofs tractable. We validate the approach on two case studies fully mechanized in Lean 4, a full-adder and a D-flip-flop, showing that delays, feedback, and timing can be handled uniformly by refinement.

Contents

1 Introduction

Formal verification at the hardware level gives us a way to reason precisely about how circuits behave, and it aids in tasks such as optimization and equivalence checking.

The central problem in this thesis is the gap between combinational and sequential circuit verification. In the standard presentation, the two are treated as distinct: combinational circuits can be verified with delays as long as they are acyclic, while circuits with feedback or state are usually replaced by sequential abstractions and analysed as state machines driven by clock edges. This separation is mirrored in existing hardware-verification efforts: translations of Verilog [1], [2] and VHDL [3] into proof assistants, and Lean libraries such as circuitlib [4] and Sparkle [5], typically treat the combinational and sequential cases separately. A notable exception is Kaye [6], who develops unified denotational, operational, and algebraic semantics for digital circuits as graphs, including sequential ones. However, it does not reduce any combinational circuits into sequential ones.

In practice, the boundary is much less clean. Sequential components such as D-flip-flops are themselves built from combinational gates with feedback, and their transient behaviour, especially just after a clock edge, depends on gate-level delays and loop dynamics that sequential abstractions do not show.

Graphiti [7] is a Lean 4 framework, originally developed for verifying dataflow circuits produced by dynamically scheduled high-level synthesis [8], that models computational modules as state machines with input, output, and internal transitions connected by wires. Its central notion is *refinement* [9]: proving that a specification can simulate every behaviour of an implementation. Because Graphiti supports feedback loops and monotonic progress, it is a good fit for circuits that sit in this middle ground.

The notion of refinement is more specialized in Graphiti, but the principle remains the same.

In this work, we apply and extend Graphiti's methodology to combinational circuits with delays and feedback loops, in particular traditionally difficult ones like D-flip-flops. We develop a general pattern for translating such circuits into Graphiti modules and verifying them against high-level specifications by refinement. All proofs are mechanized in Lean 4 in this file.

Our contributions are:

- A general pattern for translating combinational circuits with delays and feedback into Graphiti modules, including reusable abstractions (buffers, speculation modules, circuit simulations) that simplify refinement proofs.
- A verified refinement proof for a full-adder circuit, which served to pin down the methodology and establish the proof patterns.
- A verified refinement proof for a D-flip-flop circuit, showing that the methodology scales to circuits with internal loops and complex timing constraints, and that the patterns from the full-adder transfer effectively.

The rest of this report is organized as follows. Section 2 covers the necessary background on Graphiti and introduces our notation. Section 3 defines the combinational module framework we use throughout. Section 4 presents the full-adder case study, and Section 5 the D-flip-flop case study. Finally, Section 6 concludes and discusses future work.

2 Background

In Graphiti, a computational module is represented as a state machine with input and output ports that can be connected by wires. Its behaviour is described by three kinds of *rules*:

- **Input rules** take the module from a state s to a new state s' while absorbing a value v from a wire at one of its input ports.
- **Output rules** take the module from s to s' while emitting a value v to a wire at one of its output ports.
- **Internal rules** are silent transitions from s to s' that neither consume nor produce a value.

Input and output rules are given by one relation per port[◦], and internal rules by a single relation over states.[◆] This model sacrifices easy termination and some simplicity for the ability to represent arbitrary computations *with* feedback loops.

We will denote accepting an input v at input port i as $s \overset{v}{\underset{i}{\rightarrowtail}} s'$, emitting an output v at output port i as $s \overset{v}{\underset{i}{\twoheadrightarrow}} s'$, and an internal transition as $s \rightsquigarrow s'$. We may omit the port when it is unambiguous.

◦ A three-way relation with the start state, the value, and the end state.

◆ Mathematically, there is a single relation for the entire input and output phase, which results from merging all input rules and output rules. Since each port can have a different type, the relation must use a dependent pair.

2.1 Refinements

The main property we want to establish is that a circuit *refines* a simpler specification: every behaviour of the circuit must also be allowed by the specification. When the specification is a high-level description that we are willing to trust, such as a simple computation or an idealized D-flip-flop, the refinement proof tells us that the gate-level implementation never steps outside that intended behaviour.

In Graphiti, we prove refinement by giving a *simulation relation* φ between implementation states and specification states. φ is an invariant stating that both states are possible to reach, and that they correspond to the same stage of computation. The refinement proof then comes down to two requirements:

- the initial states of the two modules are related by φ; and
- φ is preserved by every step of the implementation: whenever the implementation can take a transition out of a state s, the specification can take a matching sequence of transitions from any φ-related state, ending in a pair of states again related by φ.

The second requirement can be expressed more mathematically as

$$\alpha\ \varphi\ \beta \wedge \alpha \overset{i}{\underset{v}{\rightarrowtail}} \alpha' \rightarrow \quad \exists \beta', \beta_1, ..., \beta_n, \beta \overset{i}{\underset{v}{\rightarrowtail}} \beta' \wedge \beta' \rightsquigarrow \beta_1 \wedge ... \wedge \beta_{n-1} \rightsquigarrow \beta_n \wedge \alpha'\ \varphi\ \beta_n$$

$$\alpha\ \varphi\ \beta \wedge \alpha \overset{i}{\underset{v}{\twoheadrightarrow}} \alpha' \rightarrow \exists \beta', \beta_1, ..., \beta_n, \beta \rightsquigarrow \beta_1 \wedge ... \wedge \beta_{n-1} \rightsquigarrow \beta_n \wedge \beta_n \overset{i}{\underset{v}{\twoheadrightarrow}} \beta' \wedge \alpha'\ \varphi\ \beta'$$

$$\alpha\ \varphi\ \beta \wedge \alpha \rightsquigarrow \alpha' \rightarrow \exists \beta_1, ..., \beta_n, \beta \rightsquigarrow \beta_1 \wedge ... \wedge \beta_{n-1} \rightsquigarrow \beta_n \wedge \alpha'\ \varphi\ \beta_n$$

Concretely, this gives one proof obligation for each kind of implementation rule. Because this matching must succeed for *every* implementation transition, a successful proof shows that the specification simulates the implementation step by step.

Much of the rest of this report is devoted to building specifications and simulation relations for which these obligations can actually be discharged, along with the auxiliary modules, especially buffers and speculation, that make the implementation and specification line up cleanly.

3 Combinational modules

Combinational circuits (circuits built from "simple" gates) are particularly tough to formalize accurately. Real logic gates have a delay for their outputs

that cannot be ignored, as well as internal loops of logic. Graphiti lets us simulate both, if we can accurately model them.

Concretely, we want our formalisation to satisfy the following requirements:

- **Delays.** Each gate's output is delayed relative to its inputs, and these delays must be modelled faithfully rather than abstracted away.
- **Feedback loops.** Gates may be wired in cycles, so the model cannot assume an acyclic dependency order between wires.
- **Partial information.** We must be able to leave a signal's value unknown wherever the circuit's behaviour is transient or irrelevant, rather than forcing a concrete value at every instant.
- **Incremental refinement.** As information propagates through the circuit, the model should let us learn more about the internal wires one step at a time; because of cycles, we can't propagate all the information in one go.
- **Monotonic progress.** Each such step should only ever add information, which guarantees a finite number of internal steps before the maximal output is reached.

We motivate and address each of these in turn below.

Fundamentally, when simulating a circuit C, we want to obtain a *waveform* of each output, given the waveforms of the inputs. In general, we do not actually require each value of the output waveform to be known at *every* point. For example, in a D-flip-flop right after a clock rise, it is expected that the output value changes quickly in chaotic ways, so it is easier to simply model these values as "unknown".

Mathematically, we want to create a relation $R_C \subseteq \mathcal{D}^n \times \mathcal{D}^m$ (where $\mathcal{D}$ is some type representing a waveform) s.t. $(i_1, i_2, ..., i_n)\ R_C\ (v_1, v_2, ..., v_m)$ if and only if $(v_1, v_2, ..., v_m)$ is a valid output of C with the inputs $(i_1, i_2, ..., i_n)$. The relation is also trivially splittable into m relations noted $R_C[i]$ for each output.

One might be tempted to model each gate as a module taking in the signal at a single instant and outputting its computation with some delay. Unfortunately, to accurately simulate a circuit, each gate would need to operate simultaneously, which Graphiti is not very amenable to. Forcing it in would also make it very difficult to reason globally over the entire circuit.

Moreover, it is very common to know the behaviour of a signal up to an instant T_1 (and nothing after), then later knowing it until $T_2 > T_1$ (effectively, "learning information" about the signal). It is convenient to be able to assert and explain how this new information affects our output, as well as some level of consistency about it all.

For these purposes, we will now define some operators and common behaviours about our combinational modules. We will only work with discrete signals here, but all the ideas presented here generalize relatively intuitively to continuous signals.■

■ Some implementation aspects don't translate directly, but the concepts do.

A discrete signal S of type $\mathcal{D}$ is a function $\mathbb{N} \to I$ taking in a timestamp and returning information about that timestep, where $I \subseteq \mathcal{P}(V)$ with V the concrete possible values of that signal, and $\mathcal{P}(V)$ its power set. A timestep mapped to all of V is one about which we know nothing; a timestep mapped to a singleton is one whose value is known exactly. Since we will only work with binary signals, $V = \mathbb{F}_2$.

We define the operations $\trianglelefteq$ ("less defined than") and $\triangleleft$ ("strictly less defined than")as follows:

$$\forall x, y : \mathbb{N} \to I, \quad x \trianglelefteq y := \forall i : \mathbb{N}, y(i) \subseteq x(i)$$

$$\forall x, y : \mathbb{N} \to I, \quad x \triangleleft y := x \trianglelefteq y \wedge x \neq y$$

Effectively, $x \trianglelefteq y$ means that y is more information, and $x \triangleleft y$ means that y is progress over x. This information ordering, together with the monotonic progress we require below, mirrors the way streams are ordered in the semantics of dataflow networks [10].

In our code, we work with signals based on lists of booleans. We can think of it as a signal; for any value beyond the list's length, the signal is $\mathbb{F}_2$ (all values are possible). For every index in the list, we know the corresponding instant's value for certain. This definition allows us to simplify the implementations of $\trianglelefteq$ to the prefix relation, which exists in the Lean standard library and is simple to reason about. Since information generally only progresses forwards in time, this model is sufficient for our purposes.□

□ Unfortunately, it does not permit us to model uninitialized wires. However, the proof could relatively easily be adapted to use them.

From here on, $\mathcal{D}$ will represent this waveform representation. Additionally, any operation $f : \mathbb{F}_2^n \to \mathbb{F}_2$ can also be used (when unambiguous) as $f : \mathcal{D}^n \to \mathcal{D}$ defined as f applied over every tuple over time.▲

▲ Partially-undefined tuples are invalid, so the output's length is the minimum of the lengths of its inputs.

Recall that a gate is simply a relation $C \subseteq \mathcal{D}^n \times \mathcal{D}^m$. When linking gates together into circuits, internal loops mean that we cannot simply resolve a full relation over the entire circuit directly; we need to iteratively learn more information about the *internal* wires in our circuit. Graphiti provides exactly this!

Inside our circuit, each gate outputs as much information as it can, which improves the information other gates have for their inputs, etc. We effectively refine our knowledge as much as possible, and output our conclusions. These iterative refinements will be the internal transitions of our full circuit.

It'd be nice if our internal transitions were guaranteed to progress, too; it guarantees a finite number of internal steps, which should usually guarantee the maximum output information once we run out of internal steps.△

△ With monotonicity, because our information is discrete and our inputs finite, there must be a maximum value after which it cannot progress.

These requirements combined lead us to translate our gates as follows:

Given a gate $C \subseteq \mathcal{D}^n \times \mathcal{D}^m$, we create a Graphiti module with an internal state $(x_1, ..., x_n) : \mathcal{D}^n$ corresponding to the most information we've received for each port. For any port i, $(x_1, ..., x_i, ..., x_n) \overset{v}{\underset{i}{\rightarrowtail}} (x_1, ..., v, ..., x_n) \iff x_i \lhd v$. The output rule for port i is $s \overset{v}{\twoheadrightarrow} s \iff s\ C[i]\ v$, i.e. an output must respect the gate's relation, and must not change the state. There are no internal rules for this kind of module.

For consistency reasons, it is required that the output relation is *monotonic*, meaning

$$\text{Given } (x_1, ..., x_n)\ C\ x \text{ and } (y_1, ..., y_n)\ C\ y, \text{then } \forall i \leq n, x_i \trianglelefteq y_i \rightarrow x \trianglelefteq y$$

This is generally a straightforward property; it simply means that as we add input information, the output becomes better, and always remains compatible with the previous output.

Many simple gates are simple functions from their inputs to their outputs. For example, the gate $\texttt{AND}_1$ (a logical AND with a delay of 1)♥, can be thought of as relating $(\alpha, \beta)\ \texttt{AND}_1\ d_1(\alpha \wedge \beta)$ exclusively (where d_i delays a signal by i).

♥ The delay is not a real quantity of time; it is some minimum value that lets every gate have an integer delay.

As shorthand for exactly this translation pattern, we will write $\mathfrak{M}((x_1, ..., x_n) \mapsto f(x_1, ..., x_n))$ for the Graphiti module with state $\mathcal{D}^n$, input transitions $(x_1, ..., x_i, ...x_n) \overset{v}{\underset{i}{\rightarrowtail}} (x_1, ..., v, ..., x_n)$, the output transition $(x_1, ..., x_n) \overset{f(x_1,...,x_n)}{\twoheadrightarrow} (x_1, ..., x_n)$, and no internal transitions.

Since delaying is a very common operation, we will also write $\mathfrak{M}_1(\alpha, \beta \mapsto \alpha \wedge \beta)$ for $\mathfrak{M}(\alpha, \beta \mapsto d_1(\alpha, \beta))$ or, even more succinctly, $\mathfrak{M}_1(\wedge)$.

4 Full-adder example

In this first example, we will verify a very simple refinement in order to understand what we need for more complex proofs. We will create a circuit

representing a full adder, and prove that it is equivalent to a simple computation of the sum.

A full adder takes in 3 values a, b, c and computes their sum (the $\oplus$ of all 3 inputs) and carry (whether at least 2 values were high). It's composed of two half-adders (which are themselves an AND and XOR gate) and an OR gate, as shown in Figure 1.

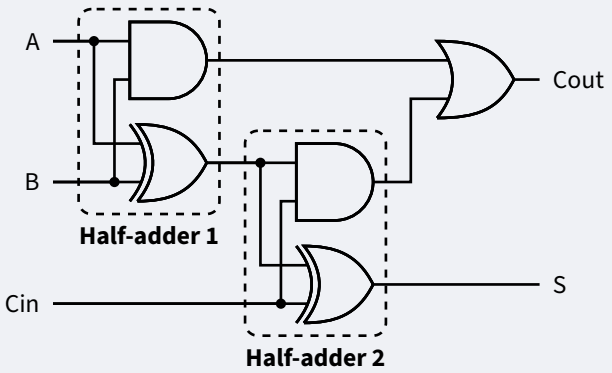

Figure 1: The circuit for a full-adder as modeled.

For simplicity, we will model each half-adder as a single module with two outputs: the sum and the carry $\mathbf{HA} := (\mathfrak{M}_1(\oplus), \mathfrak{M}_1(\wedge))$. We will also not refine the carry-out, though we will still model it.

Although it has two functions, they share their input states. Without making it a single module, we'd have to add a "fork" module, which is just noisy.

Recall that all of these gates have one piece of state per input. The resulting circuit is shown in Figure 2.

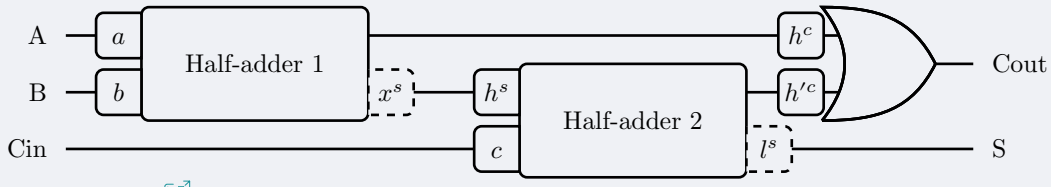

Figure 2: The full-adder circuit annotated with the names we use for each piece of state (solid boxes) and for the intermediate values we still refer to (dashed boxes).

Each full square is a piece of state and the name we will use. Each dashed square is a value that isn't part of the state, but that is still useful to name.

To ignore the carry-out, we must add a module: a *sink*. It has no state, accepts any values fed into it, and has no output. It's trivial to work with, so we will not mention it further.

Before stating the specification, we need to account for several complications caused by delays and feedback in the circuit:

- **Delayed output.** The circuit's outputs lag its inputs by a few ticks, so a naïve instant-by-instant comparison against $a \oplus b \oplus c$ does not hold.

- **Transient incorrect values.** As the inputs change, the circuit may briefly emit incorrect values because signals do not propagate at the same speed through every path; the specification must be able to ignore these periods.
- **Longer streams.** Because of the delays from gates, the implementation produces more values than a delay-free specification would, so the two output streams do not have the same length.
- **Partial information.** During internal transitions, the Graphiti modules may not yet have propagated all the information through every path, so intermediate states carry incomplete information. The outputs are therefore not always as defined as they could eventually become, and the specification must be able to represent that.

4.1 Definitions

Our goal is to prove that this circuit computes $a \oplus b \oplus c$. However, due to delays and transient incorrect values, our specification cannot require equality with $a \oplus b \oplus c$ at every instant. Instead, it must check the output only once the circuit has settled, and allow the behaviour in the transient period to remain unspecified.

In practice, we will create a module **SPEC**, which should behave as closely as possible to $\mathfrak{M}(\alpha, \beta, \gamma \mapsto \alpha \oplus \beta \oplus \gamma)$.

From testing a Verilog simulation of the full-adder, we find that the circuit matches the expectation once more than 3 ticks have passed since the last change in any input. Right after a change, the output is undefined.

In other words, the output of the spec matches the relation (a, b, c) **SPEC** s if and only if $\forall i \geq 3, (\forall j \in \mathbb{N}, x \in \{a, b, c\}, i - 3 \leq j < i, x(j) = x(i)) \rightarrow s(i) = a(i) \oplus b(i) \oplus c(i)$.

Effectively, whether a, b, or c have changed in the last 3 ticks acts as a *filter*.

4.1.1 Stability filter

For simplicity, we will write $\mathcal{S}_3(x)$ for the signal such that $\forall i : \mathbb{N}, \mathcal{S}_3(x)(i) = 1 \iff i \geq 3 \wedge (\forall j, i - 3 \leq j < i \rightarrow x(j) = x(i))$, called the *stability filter*. It is high exactly at the instants where x has held the same value for the previous 3 ticks.

We will also write $F \vdash x = y$ for $\forall i : \mathbb{N}, F(i) = 1 \to x(i) = y(i)$, pronounced "$x$ is equal to y under filter F". Similarly, we may also use $F \vdash x \trianglelefteq y$ for $\forall i : \mathbb{N}, F(i) = 1 \to y(i) \subseteq x(i)$.

4.1.2 The specification module

Our relation now becomes

$$(a, b, c)\ \mathbf{SPEC}\ s \triangleq (\mathcal{S}_3(a) \wedge \mathcal{S}_3(b) \wedge \mathcal{S}_3(c)) \vdash s = a \oplus b \oplus c$$

And we will use **SPEC** to denote the module derived from this output relation (with the corresponding state), along with the combined filter $F_{\mathbf{SPEC}}(a, b, c) \triangleq \mathcal{S}_3(a) \wedge \mathcal{S}_3(b) \wedge \mathcal{S}_3(c)$, where a, b, and c are the circuit's inputs (the same as in the relation above). We leave these arguments implicit and write simply $F_{\mathbf{SPEC}}$ when they are clear from context.

We're not quite done yet, though, as this cannot replicate every behaviour of the original circuit.

4.1.3 Speculation

The first issue we can notice is that the implementation locks in some information that **SPEC** cannot know yet, by virtue of delays. A single gate with a delay of k with an input of n will output $n + k$ values, while the same gate but as a filtered-instant module will output n.

However, we know that these unknown values will either be filtered in the future and therefore will not matter, or will be equal to the real values. Additionally, the extra values are always bounded in number.

To solve this, we create a *speculation* module $\mathbf{FS}_k$ and affix it to the output of **SPEC**. This module has one piece of state, and guarantees monotonicity on its input:

- $s \overset{v}{\rightarrowtail} v \iff s \lhd v$
- $s \overset{v}{\twoheadrightarrow} s \iff \exists x, v = s + x \wedge |x| \leq k$

In the full adder, we have $k = 3$, as that is the maximum delay experienced by the adder.

This module solves this problem completely and only adds manageable complexity. It does make the refined circuit have internal transitions, but we only need to deal with those when doing outputs.

In this definition, we use $s + x$ (concatenation) and $|x|$ (length, or "amount of information"). These are only intuitive on our lists. A more complex signal modelling would use something like $v \trianglelefteq s \wedge |v| + k \geq |s|$, with a different definition of $|\cdot|$ (that still means "amount of information").

4.1.4 Buffers

Notice that in the original circuit, we can do the following set of transitions:

$$s_1 \overset{v_a}{\underset{a}{\rightarrowtail}} s_2 \overset{v_b}{\underset{b}{\rightarrowtail}} s_3 \overset{v_c}{\underset{c}{\rightarrowtail}} s_4 \overset{d_1(c \oplus h^s)}{\twoheadrightarrow} s_4$$

.

In this path, we updated a, b, and c in the circuit's state, but since we didn't do an internal transition to update h^s, the second half-adder only had partial information and didn't output the full desired value.

In effect, this is the opposite problem; the **SPEC** has too *much* information. The solution is to add *buffers* to the inputs.◆ These buffers output partial information to whatever level desired, but must output more and more information each time, guaranteeing progress. Their state is a tuple (i, o) where i is the complete input, and o is the last output. Its transitions look like this:

◆It is possible to add the buffers at the end, and even combine them with the speculation module, but it requires us to modify our signal type, which was deemed too big a change.

$$(i, o) \overset{v}{\rightarrowtail} (v, o) \iff i \lhd v$$

$$(i, o) \overset{v}{\twoheadrightarrow} (i, v) \iff o \lhd v \land v \trianglelefteq i$$

Our final circuit is shown in Figure 3.

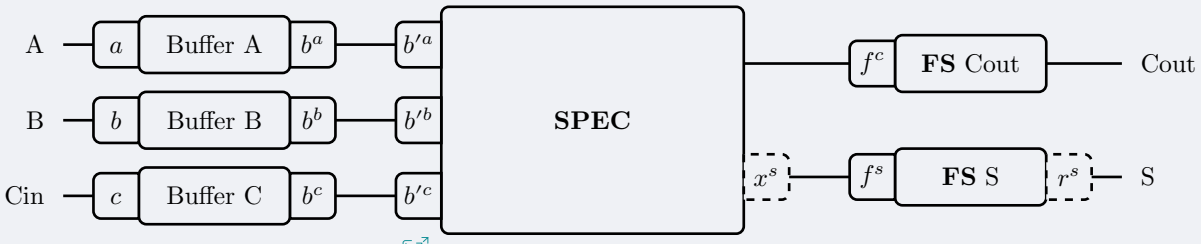

Figure 3: The full specification circuit, with a buffer on each input and a speculation module on each output.

4.2 Proving refinement

From here on, we will be working with a specific implementation circuit and a specific specification circuit. We will refer to them as **Imp** and **Spec** respectively.

Whenever we talk about the "old" and "new" values from the state, we will use the subscripts α and β.

4.2.1 Phi

When proving refinement, we need an invariant φ providing three groups of facts: that **Imp** is *consistent* (in a state reachable from initialization, with no rules broken along the way), that **Spec** is consistent, and that **Imp** and **Spec** represent the same computation state. We name the individual clauses below

to make the structure of the invariant explicit; each transition proof then works through these same clauses when re-establishing φ on the new state.

Imp consistency follows from the monotonicity of the gates in **Imp**:

- **(Imp-gate)** each gate output is bounded by the delayed function of its inputs. For example, in the second half-adder $h^s_\alpha = x^s_\alpha = d_1(a_\alpha \oplus b_\alpha)$, and since $a_\alpha \trianglelefteq a_\beta$ and $b_\alpha \trianglelefteq b_\beta$, monotonicity gives $h^s \trianglelefteq d_1(a \oplus b)$ at all times. Similarly, $h^c \trianglelefteq d_1(a \wedge b)$ and $h'^c \trianglelefteq d_1(h^s \wedge c)$.◇

◇ These last two are not necessary in our simplified proof, so we can ignore them.

Spec consistency consists of three clauses:

- **(Buffer-pass)** the spec receives exactly what each buffer emits: $b'^a = b^a$, $b'^b = b^b$, and $b'^c = b^c$.
- **(Buffer-bound)** each buffer output is no more defined than its corresponding input: $b^a \trianglelefteq a$, $b^b \trianglelefteq b$, and $b^c \trianglelefteq c$.
- **(Spec-out)** the speculated output respects **SPEC**'s relation under the filter: $F_{\mathbf{SPEC}} \vdash f^s \trianglelefteq b'^a \oplus b'^b \oplus b'^c$.

Correspondence between **Imp** and **Spec** adds three further clauses:

- **(Inputs-agree)** the two modules see the same inputs (their as, bs, and cs are equal).
- **(Not-ahead) Spec** cannot run ahead of **Imp**: $|b^a| \leq |l^s|$, $|b^b| \leq |l^s|$, and $|b^c| \leq |l^s|$, where $|l^s| = \min(|h^s|, |c|) + 1$. This lets us guarantee that **SPEC** can output the right amount of data when required.■
- **(Locked-in)** the not-yet-revealed speculated values agree with the implementation: $f^s \trianglelefteq d_1(h^s \oplus c)$. Recall that **SPEC** may output any values at the timesteps where $F_{\mathbf{SPEC}}$ is low; this clause guarantees that those "locked-in" unknown values match **Imp**.

■ Technically, we really only need $\min(|b^a|, |b^b|, |b^c|) \leq |l^s|$. However, this slightly stronger invariant is just as easy to maintain, and easier to work with.

4.2.2 Input transitions

There are 3 input transitions in **Imp**: $\{a : a, ...\} \overset{v}{\underset{A}{\rightarrowtail}} \{a : v, ...\}$, $\{b : b, ...\} \overset{v}{\underset{B}{\rightarrowtail}} \{b : v, ...\}$, and $\{c : c, ...\} \overset{v}{\underset{C}{\rightarrowtail}} \{c : v, ...\}$.

The proofs are essentially identical, so we will only talk about a.

$\{a : a, ...\} \overset{v}{\underset{A}{\rightarrowtail}} \{a : v, ...\}$ means that $a \lhd v$.

The equivalent **Spec** transition is also $\{a : a, ...\} \overset{v}{\rightarrowtail} \{a : v, ...\}$.

Given that $\mathbf{Imp}_\alpha \ \varphi \ \mathbf{Spec}_\alpha$ (1), we must now prove that $\mathbf{Imp}_\beta \ \varphi \ \mathbf{Spec}_\beta$.

$\mathbf{Imp}_\beta$'s consistency requires that $h^s \trianglelefteq d_1(v \oplus b)$. We know that $h^s \trianglelefteq d_1(a \oplus b)$ via (1) and $a \trianglelefteq v$. This is sufficient, as both $\oplus$ and d_1 are monotonic.

$\mathbf{Spec}_\beta$'s consistency requires a few things that come from $\mathbf{Spec}_\alpha$'s consistency (things that a doesn't touch), as well as $b^c \trianglelefteq v$. This is given by $b^c \trianglelefteq a$ and $a \trianglelefteq v$.

Finally, $\mathbf{Imp}_\beta$ and $\mathbf{Spec}_\beta$'s correspondence requires that $v = v$ (the rest is given by $\mathbf{Imp}_\alpha \varphi \mathbf{Spec}_\alpha$). This is trivial.

4.2.3 Internal transitions

Internal transitions on **Imp** do not require any updates from **Spec**.

All of the internal transitions can be solved quite mechanically by using monotonicity and other properties.

For example, the internal transition $\{a := a, b := b, h^s := h^s_\alpha, ...\} \rightsquigarrow \left\{a := a, b := b, h^s := h^s_\beta, ...\right\}$ has the property $h^s_\beta = d_1(a \oplus b)$.

- **Imp** consistency requires that $h^s_\beta \trianglelefteq d_1(a \oplus b)$, given by reflexivity.
- **Spec** consistency is a given, since **Spec** doesn't change states
- **Imp** and **Spec** correspondence requires properties on $l^s_\beta = d_1\left(h^s_\beta \oplus c\right)$. By monotonicity, $d_1(h^s_\alpha \oplus c) \trianglelefteq d_1\left(h^s_\beta \oplus c\right)$. All of the correspondence properties can just use transitivity with this fact.

4.2.4 Output transitions

This is the hardest part of the proof, and requires a few lemmas that we will introduce over time.

We are given a transition $s \overset{v}{\twoheadrightarrow} s$ in the **Imp**.

We know that $v = d_1(h^s \oplus c)$. We must find a set of internal transitions followed by an output transition in **Spec** that outputs exactly v, and prove φ on the new state.

The relevant circuit is **Spec**, shown in Figure 3: each input passes through a buffer into **SPEC**, whose speculated output f^s then passes through a speculation module to become the final output r^s.

The output r^s is given by f^s_β, itself given by b^a_β, b^b_β, and b^c_β. Generally, we cannot assume that any of these have the "right" value to output v, so we will assume they do not (if they do, we can skip an internal transition).

In this part, we must often think about our signals as lists. A very useful operation is the $\texttt{take}(x, s)$ operation, a $\mathbb{N} \times D \to D$ function that takes the first x values from s.

It has the following property: $\texttt{take}(x, s) \trianglelefteq s \wedge |\texttt{take}(x, s)| = \min(x, |s|)$.

Additionally, $s_1 \trianglelefteq s_2 \Leftrightarrow s_1 = \texttt{take}(|s_1|, s_2)$, and $\forall i < |x|, \texttt{take}(x, s)(i) = s(i)$.

We will use this function to fast-forward our buffers to the exact right length, then have the specification output a value as close to v as possible. The speculation module will then close the remaining gap, if any.

The full set of transitions **Spec** will do is

$$\{b^a := b^a_\alpha, b^b := b^b_\alpha, b^c := b^c_\alpha, f^s := f^s_\alpha, ...\}$$

$$\rightsquigarrow \left\{b^a := b^a_\beta, ...\right\} \qquad (1)$$

$$\rightsquigarrow \left\{b^b := b^b_\beta, ...\right\} \qquad (2)$$

$$\rightsquigarrow \left\{b^c := b^c_\beta, ...\right\} \qquad (3)$$

$$\rightsquigarrow \left\{f^s := f^s_\beta, ...\right\} \qquad (4)$$

$$\overset{v}{\twoheadrightarrow} \left\{b^a := b^a_\beta, b^b := b^b_\beta, b^c := b^c_\beta, f^s := f^s_\beta, ...\right\} \qquad (5)$$

Recall that, from φ, we have $f^s_\alpha \trianglelefteq v$, $|b^a_\alpha|, |b^b_\alpha|, |b^c_\alpha| \leq |v|$, and $F_{\mathbf{SPEC}} \vdash f^s_\alpha = b^a_\alpha \oplus b^b_\alpha \oplus b^c_\alpha$.

Steps (1)-(3) require that $b^a_\alpha \triangleleft b^a_\beta$ and $b^a_\beta \trianglelefteq a$ (with analogous rules for b and c).

We will pick $b^a_\beta = \texttt{take}(|v|, a)$, $b^b_\beta = \texttt{take}(|v|, b)$ and $b^c_\beta = \texttt{take}(|v|, c)$.

Definedness ordering.

Let s_1, s_2, s be three signals such that $s_1, s_2 \trianglelefteq s$. Suppose $|s_1| \leq |s_2|$. Then $s_1 \trianglelefteq s_2$.

Proof: This is a Lean standard library lemma, `List.prefix_of_prefix_length_le`.

We have that $|b^a_\alpha| \leq |v|, |a|$. We have $|b^a_\beta| = \min(|v|, |a|)$. Thus $|b^a_\alpha| \leq |b^a_\beta|$. By definedness ordering, $b^a_\alpha \trianglelefteq b^a_\beta$. We assumed that $b^a_\alpha \neq b^a_\beta$ (otherwise we can skip step (1)), therefore $b^a_\alpha \triangleleft b^a_\beta$. By `take`'s main property, $b^a_\beta \trianglelefteq a$.

The proof is identical for b^b_β and b^c_β, and is branched off into a lemma.

We pick $f^s_\beta = \texttt{take}(\min(|a|, |b|, |c|), v)$.

In order to do step (4), we must have $F_{\mathbf{SPEC}} \vdash f^s_\beta = b^a_\beta \oplus b^b_\beta \oplus b^c_\beta$ and $f^s_\alpha \triangleleft f^s_\beta$.

In order to do step (5), we must have $f^s_\beta \trianglelefteq v$ and $|v| - |f^s_\beta| \leq 3$.

$f^s_\beta \trianglelefteq v$ is given by `take`'s main property.

Since $F_{\mathbf{SPEC}} \vdash f^s_\alpha = b^a_\alpha \oplus b^b_\alpha \oplus b^c_\alpha$, $|f^s_\alpha| = \min\big(|b^a_\alpha|, |b^b_\alpha|, |b^c_\alpha|\big)$. We know that $|f^s_\beta| = \min(|a|, |b|, |c|, |v|)$. It is thus quite easy to show that $|f^s_\alpha| \leq |f^s_\beta|$. By definedness ordering, $f^s_\alpha \trianglelefteq f^s_\beta$, and since we assumed $f^s_\alpha \neq f^s_\beta$, $f^s_\alpha \triangleleft f^s_\beta$.

We now have two properties left to prove: $F_{\mathbf{SPEC}} \vdash f^s_\beta = b^a_\beta \oplus b^b_\beta \oplus b^c_\beta$, and $|v| - |f^s_\beta| \leq 3$. Both of these are relatively complex and require their own lemmas.

For the first, we first apply our definitions and simplify things:

$$
\begin{aligned}
& F_{\mathbf{SPEC}} \vdash f^s_\beta = b^a_\beta \oplus b^b_\beta \oplus b^c_\beta \\
\iff \quad & |f^s_\beta| = |b^a_\beta \oplus b^b_\beta \oplus b^c_\beta| \\
& \wedge \Big(\forall i < |f^s_\beta|, F_{\mathbf{SPEC}}(i) = 1 \to f^s_\beta(i) = \Big(b^a_\beta \oplus b^b_\beta \oplus b^c_\beta\Big)(i)\Big) \\
\iff \quad & \min(|a|, |b|, |c|, |v|) = \min(\min(|v|, |a|), \min(|v|, |b|), \min(|v|, |c|)) \\
& \wedge (\forall i < \min(|a|, |b|, |c|, |v|), F_{\mathbf{SPEC}}(i) = 1 \to v(i) = a(i) \oplus b(i) \oplus c(i)) \\
\iff \quad & \forall i < \min(|a|, |b|, |c|, |v|), \mathcal{S}_3(a)(i) = 1 \wedge \mathcal{S}_3(b)(i) = 1 \wedge \mathcal{S}_3(c)(i) = 1 \\
& \to v(i) = a(i) \oplus b(i) \oplus c(i)
\end{aligned}
$$

We use a lemma to prove this.

Adder correctness.

Given an **Imp** state and **Imp** correctness, for all $i < \min(|a|, |b|, |c|, |l^s|)$ where $\mathcal{S}_3(a)(i) = 1$, $\mathcal{S}_3(b)(i) = 1$, and $\mathcal{S}_3(c)(i) = 1$, we have $d_1(h^s \oplus c)(i) = a(i) \oplus b(i) \oplus c(i)$.

The second property, $|v| - |f^s_\beta| \leq 3$, is equivalent to $|v| \leq \min(|a|, |b|, |c|) + 3$. It is also a lemma, proven using **Imp** correctness. It is simple enough to be proven by calculation.

4.3 Failed approaches

Initially, we didn't have speculation modules, and only had one buffer on the output. This approach worked when the adder's output was maximized, but when the adder is behind, the computation module requires us to set all unknown values, which the buffer then requires to maintain the same. Since we can't easily know these values in advance, we would have to change the signal type to account for these unknowns and have the buffer's output realize the unknown values.

However, we then realized we needed speculation modules. Adding them to the buffer as well was deemed overcomplicated, and we instead moved the buffers to the inputs.

Before landing on the speculation module design, we initially thought we could determine in advance what the unknown values would be, by duplicating the last few values of our perfect adder. The unknown values are (nearly) entirely caused by the delay, and so must match our perfect adder at some point in the past. This approach was abandoned due to its fragility; transient incorrect values are minimal in the adder, but much more frequent in other circuits.

There were also several designs for the gate-to-Graphiti translation. The main disadvantage of our solution is the excessive state; every gate must have state that verifies that the inputs are monotonic. Unfortunately, designs with less state were deemed too fragile, as they required careful thought about where values exist.

We also considered using linear temporal logic [11] for defining the behaviour of most components. This approach has potential, but was not used due to its initial higher complexity and more abstract nature.

5 D-flip-flop example

5.1 Setting up

Now we will verify a more complex refinement, including loops and relying on delay properties to work.

A **D-flip-flop** is a basic form of volatile memory. It has a data input and a clock input, one output, and one bit of state. For a perfect D-flip-flop, the output is always equal to the state, and whenever the clock rises, the state is instantly set to the current data value.

Conceptually, there is only one bit of state. However, the concrete implementation has significantly more.

Our D-flip-flop is implemented as shown in Figure 4.

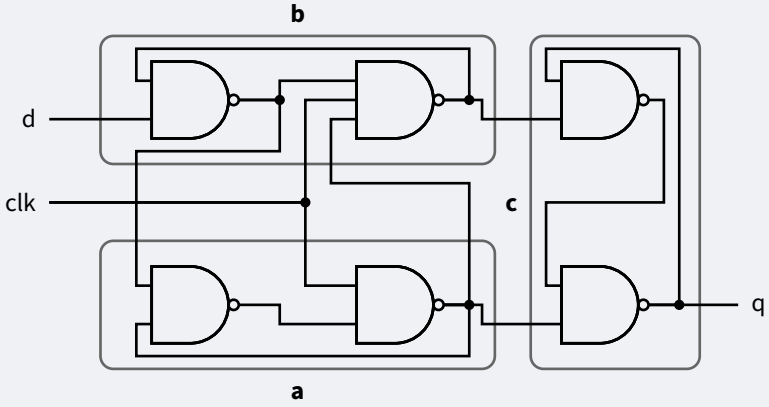

Figure 4: The D-flip-flop circuit, as modeled.

Note that every wire branching requires an extra module to model in Graphiti. These "fork" modules are simply $\mathfrak{M}(\mathrm{id})$ with 2 or 3 outputs, so they do not add any logical complexity. However, they add 5 extra values of state and a good amount of bookkeeping/added proof complexity. They have been omitted from this report for clarity.

This D-flip-flop, being imperfect, has the following conditions attached to its good behaviour:

- A clock rise will set the state correctly **if and only if** the data does not change for a certain number of timesteps before and after the clock rise, called the **setup** and **hold** times respectively. If this is not respected, the output becomes unpredictable until the next clock rise, a manifestation of metastability [12].▲
- If the clock changes too quickly, the flip-flop might break in a difficult-to-recover manner, which we model simply as breaking "forever".▵
- Right after a clock rise, there is a **delay** where the output value is unknown.

▲ Note that the state isn't simply unknown; it's possible that the output is broken and oscillates until the clock rise.

▵ Importantly, a clock fall too quickly after a rise *also* breaks the state

From testing a Verilog simulation, we determine that in this implementation, the setup time is 2, the hold time is 1, the delay is 3, and the minimum time between clock changes is 3.

5.2 Definitions

Exactly like the full-adder, we again need buffers and speculative modules (this time with 4 ticks of speculation). Our **SPEC** module is slightly more complex. The conditions we listed previously correspond well to the filtering pattern, but it isn't as simple to define everything.

5.2.1 A perfect D-flip-flop

We are looking for a function $f : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ that follows the intuitive definition of a D-flip-flop.

We will define $c \overline{\uparrow} i \triangleq i > 0 \wedge c(i) = 1 \wedge c(i-1) = 0$, the property "$c$ has a rising edge at i".

We need $i > 0$, because Lean's subtraction saturates and having this as part of our hypotheses helps reasoning.

Mathematically, we want

$$\begin{aligned} f(c, d) \mapsto v \text{ such that } \forall i \in \mathbb{N}, \\ v(i) = 1 \iff \exists t : \mathbb{N}, (\forall t' \in]t, i], \neg(c \overline{\uparrow} t')) \wedge \\ c \overline{\uparrow} t \wedge \\ d(t) = 1 \end{aligned}$$

In plain English, "there exists a most-recent rising edge at which the data value was 1".

This definition has useful properties, but it is difficult to determine what value fulfills this easily; we could prove uniqueness and existence of this value, but it isn't "computationally intuitive".

Instead, we give a directly computational definition using a variant of a left scan. Given a function $f : (\alpha, \beta) \to \alpha$ and an initial state $s : \alpha$, we define $\mathcal{M}[f][s] : \texttt{List}\ \beta \to \texttt{List}\ \alpha$ as

$$\begin{aligned} \mathcal{M}[f][s] : v \mapsto q \text{ such that } q(0) = f(s, v(0)) \wedge \\ q(n) = f(q(n-1), v(n)) \end{aligned}$$

Concretely, this is Lean's `scanl` followed by a `drop` of the initial state.

Our perfect D-flip-flop is then

$$\mathcal{M}[(c_0, s), c_1, d \mapsto \text{if } c_0 = 0 \wedge c_1 = 1 \text{ then } (c_1, d) \text{ else } (c_1, s)][(0, 0)]$$

The threaded state is a pair of the previous clock value c_0 (used to detect rising edges) and the stored data bit s, while the per-timestep inputs are the current clock c_1 and data d. At each timestep, the stored bit is overwritten with the current data exactly when the clock does a rising edge and is left unchanged otherwise.

We need the drop, because `scanl` also includes the initial state, which we don't want.

By taking the data state of each element of the resulting list (with a `map`), we obtain our perfect D-flip-flop's function, which we name Q.

We can prove our mathematical definition on this computational definition via induction.

5.2.2 Filters

The three cases where our real D-flip-flop does correspond to the perfect D-flip-flop can be split into three filters, which we then combine into one.

Delay filter.

The delay filter is the simplest. It must be low for k ticks after a rising edge:

$$D(c)(i) = 0 \iff \exists t : \mathbb{N}, t \leq i < t + k \wedge c \overline{\uparrow} t$$

We choose the equivalent property

$$D(c)(i) = 1 \iff \forall r \leq i, c \overline{\uparrow} r \to r + k \leq i$$

which instead states "the delay filter is high if and only if any rising edges before this point were at least k timesteps away". This property is easier to work with, since it is defined on 1s.

Once again, we must define it more computationally:

$$\mathcal{M}[(c_0, n), c_1 \mapsto \text{if } c_0 = 0 \wedge c_1 = 1 \text{ then } (c_1, 0) \text{ else } (c_1, n + 1)][(0, 0)]$$

This function computes the number of timesteps since the last rising edge. By filtering the timesteps where this value is $\geq k$, we obtain our delay filter.

Setup-hold filter.

The setup-hold filter is more complex:

$$S(c, d)(i) = 1 \iff \exists t : \mathbb{N}, \quad c \overline{\uparrow} t \wedge (\forall t' \in]t, i], \neg(c \overline{\uparrow} t')) \wedge$$
$$t \geq s \wedge (\forall k \in [t - s, t + h[, d(k) = d(t))$$

where s is the setup time and h is the hold time. In plain English, "the setup-hold filter is high if and only if, at the most recent rising edge, the data doesn't change for s ticks before and h ticks after".

This expression is quite complex. However, we have a simplifying element: in our implementation, $h = 1$. This means that we can simplify our second condition to simply "the data hasn't changed in $s + 1$ ticks at the rising edge".

Luckily, we already have a function that provides this condition: the stability filter $\mathcal{S}$.

Our computational equivalent is

$$\mathcal{M}[(c_0, s), c_1, d_s \mapsto \text{if } c_0 = 0 \wedge c_1 = 1 \text{ then } (c_1, d_s) \text{ else } (c_1, s)][(0, 0)](c, \mathcal{S}_2(d))$$

The state bit is the value of the filter, and this scan takes in the clock as well as the stability filter of the data.

Clock correctness filter.

The clock correctness filter states, for a stability k, that

$$\mathsf{C}(c)(i) = 1 \iff \forall j \in]0, i], c(j-1) \neq c(j) \to (\forall j' \in [j-k, j[, c(j') = c(j-1))$$

In plain English, "the clock correctness filter is high if and only if every clock change in the past was preceded by at least k identical values".

It could also be expressed as "there is no other change within $[j-k, j[$", but this definition is more useful in the proofs.

The computational equivalent is

$$\begin{aligned}\mathcal{M}[(c_0, v, c), c_1 \mapsto \text{if} \quad & c_0 == c_1 \text{ then } (c_1, v, c) \\ \text{else if } & c < k \text{ then } (c_1, 0, 1) \\ \text{else} \quad & (c_1, v, 1)] \\ [(0, 1, 0)]\end{aligned}$$

The second value of the state tracks whether the condition has ever been false, and the third is the number of consecutive identical values.

By combining these three filters into one, we obtain $F_{\mathbf{SPEC}}$, a global filter that is only high whenever the implementation matches the perfect D-flip-flop.

Note a convenience to this definition; the initial state of the D-flip-flop is undefined (due to the setup-hold filter starting low), until the first clock rise, which must be after enough time has passed for the setup to be possible (otherwise, clock stability fails and the filter stays low forever). This isn't strictly necessary, but it is more accurate to a real circuit which starts undefined.

5.3 Skeleton of the refinement proof

5.3.1 Phi

Recall that a φ has three parts; **Imp** consistency, **Spec** consistency, and correspondence.

In **Imp**, for a given NAND gate with inputs α, β and output γ, consistency means that $\gamma \trianglelefteq d_1(\alpha \uparrow \beta)$. Note that γ isn't stored in the NAND gate, but in whichever gate(s) it's connected to.

For a fork with input α and outputs β, γ, it is $\alpha \trianglelefteq \beta \wedge \alpha \trianglelefteq \gamma$.

In **Spec**, consistency is nearly the same as for the adder: $b^c = b'^c \wedge b^d = b'^d$, and $b^c \trianglelefteq c \wedge b^d \trianglelefteq d$ are equivalent. However, instead of using filtered equality, we simplify it to $|f^q| = \min\big(|b'^c|, |b'^d|\big)$, as this is the only property of it we need as an invariant, and we don't want to clutter up our assertions too much more.

Correspondence is, similarly to the adder, $f^q \trianglelefteq n^5 \wedge |b'^c| \leq |n^6| \wedge |b'^d| \leq |n^6|$ plus two conditions forcing the inputs to be identical.

5.3.2 Proving input and internal rules

Similarly to the full-adder's refinement, input rules are quite straightforward to prove, requiring some simple bookkeeping and monotonicity/transitivity for some properties.

Internal rules are more complex than in the full-adder; there are 18 different internal rules, each of which is subtly different and requires us to prove all of φ on the new state. If the statements are naïvely split into all the different assertions and cases we need to prove, we end up with approximately 250 different assertions which each require a slightly different set of assumptions, which Lean does not handle very well.

The proofs themselves are luckily *relatively* straightforward as well, mainly requiring length assertions, monotonicity, and transitivity. We progressively split the cases to never end up with too many values to juggle, and try to prove as many assertions at the same time. The resulting proof works, but is unfortunately by far the slowest part to evaluate.

5.3.3 Proving the output rule

Once again, similarly to the full-adder, we have a value v that n^5 provides, and we must make the speculation module output that same value. We use the same technique; fast-forward b^c to $\texttt{take}(|v|, c)$ and b^d to $\texttt{take}(|v|, d)$, then outputting $f^q_\beta = \texttt{take}(\min(|c|, |d|), v)$ from **SPEC**.

We now have to prove that $F_{\mathbf{SPEC}} \vdash f^q_\beta = Q\left(b^c_\beta, b^d_\beta\right)$ and $|v| - |f^q_\beta| \leq 4$.

Once again, we can abstract away our definitions to end up with these two lemmas:

D-flip-flop speculation.

Given a consistent **Imp** state, with clock c, data d, output v, $|v| \leq \min(|c|, |d|) + 4$.

Proof: For any module $\mathfrak{M}_1(\uparrow)$ with inputs α, β and output γ, $|\gamma| \leq \min(|\alpha|, |\beta|) + 1$. By repeatedly using this fact on our knowledge of **Imp** consistency, we can prove this via cases on the lengths.

D-flip-flop correctness.

Given a consistent **Imp** state, with clock c, data d, output v,

$$F_{\mathbf{SPEC}} \vdash v = Q(c, d)$$

Proving D-flip-flop correctness is done indirectly. Calculating $v(i)$ is unfortunately quite complex despite **Imp** consistency, because the state values are

abstract and disconnected; we must repeatedly assert things that are intuitively trivial.

To bridge the gap, we create a *circuit simulation* that runs the circuit synchronously. Each timestep has 6 bits of state, corresponding to the 6 NANDs. At every timestep, each bit is computed from the previous step's values.

This is a function $\mathsf{F} : \mathcal{D} \times \mathcal{D} \to \texttt{List State}$ where `State` is a tuple of 6 bits. We will prove correctness of this simulation with Q, and correspondence of this simulation with **Imp**.

5.4 Core lemmas

5.4.1 Imp-to-simulation

First, we define $\mathbf{Imp} \underset{i}{\approx} s$ where $s : \texttt{State}$ as the property that, at timestep i, the output of every NAND in **Imp** is either unknown, or equal to the corresponding state bit.

Partial Imp-sim correspondence.

Given $c, d : \mathcal{D}$ clock and data signals, and **Imp** a consistent **Imp** state

$$\forall i < \min(|c|, |d|), \mathbf{Imp} \underset{i}{\approx} (\mathsf{F}(c, d))(i)$$

Proof: By induction.

For each NAND gate, if it is defined at timestep i in the **Imp**, then its inputs must be defined at $i - 1$. By the induction hypothesis, those inputs at $i - 1$ must correspond to the circuit simulation. Therefore, at time i, their output must also correspond to the circuit simulation's.

This leads directly to

Output Imp-sim correspondence.

Given $c, d : \mathcal{D}$ clock and data signals, and **Imp** a consistent **Imp** state with output v,

$$\forall i < \min(|c|, |d|, |v|), (\mathsf{F}(c, d))(i)_5 = v(i)$$

Proof: We know that v must be defined at i since $i < |v|$. By partial **Imp** -sim correspondence, we obtain the conclusion.

5.4.2 Simulation-to-computation

Recall that the $F_{\mathbf{SPEC}}$ filter is a combination of the filters D, S, C, which have their own previously-defined properties.

We combine a few high-level lemmas, which all require a timestep i such that $F_{\mathbf{SPEC}}(i) = 1$.

Filter forces a rising edge.

Given $c, d : \mathcal{D}$ clock and data signals,

$$\exists r : \mathbb{N}, c \overline{\uparrow} r \wedge (\forall r' \in]r, i], \neg(c \overline{\uparrow} r')) \wedge r + 4 \leq i$$

This provides us with the most recent rising edge, which must be a certain number of timesteps away.

Proof: Straightforward from the properties of S and D.

We will use c, d, r, and its properties in the following lemmas.

Filter forces data stability at rising edge.

$$\forall k, r - 2 \leq k \leq r \rightarrow d(k) = d(r)$$

Proof: Using S on our rising edge (which must be unique), we get $\mathcal{S}_2(d)(i) = 1$, which is equivalent to this property after expansion.

Clock values around rising edge.

$$\forall k, r - 3 \leq k < r \rightarrow c(k) = 0$$

$$\forall k, r \leq k < r + 3 \rightarrow c(k) = 1$$

$$r \geq 3$$

Proof: the first two are both specializations of the properties of C at $r - 1$ and r, though the second requires proof by contradiction.

The third property is also derived from C, but requires a property that was implicit in our definition of clock-correctness; $\forall j' \in [j - k, j[$ is only valid if $j - k$ exists! In Lean, subtraction saturates, which made our definition incomplete. Luckily, our computational definition does fulfill it as well.

We are now ready to define our main **invariant** on the circuit simulation. This invariant defines the behaviour of each NAND in the circuit simulation.

$$\begin{aligned}\mathcal{G}_r^i s := \quad & s_5 = d(r) \wedge s_6 = \neg d(r) \wedge \\ & s_2 = \neg c(i-1) + \neg d(r) \wedge \\ & s_3 = \neg c(i-1) + d(r) \wedge \\ & c(i-1) = 1 \to d(r) = 1 \to s_4 = 1 \wedge \\ & c(i-1) = 1 \to s_1 = d(r)\end{aligned}$$

These properties are relatively unintuitive, but the ones we really care about are the first two; the others describe sufficient properties for the data to propagate correctly, based on the previous clock value. This is why the clock must stay high for a period after a rising edge; the data must propagate throughout.

We can finally define our last two main lemmas, which do *not* need the filter:

Simulation correctness.

Given clock and data signals c and d, a boundary timestep i, and an r such that:

- $\forall r', r < r' \leq i \to \neg(c \overline{\uparrow} r')$
- $3 \leq r$, $r + 3 \leq i$
- $c(r-3) = c(r-2) = c(r-1) = 0$
- $c(r) = c(r+1) = c(r+2) = 1$
- $d(r-2) = d(r-1) = d(r)$

We have

$$\forall k \in [r+3, i], \mathcal{G}_r^k(\mathsf{F}(c, d))$$

Proof: By induction.

The base case $(r+3)$ is done by expanding $F(c,d)(r+3)$ by 6 steps. Since we know the value of the clock for the entire time, as well as some of the data, we can then simplify all of the computations, until we prove the invariant.

The step is done by only expanding $F(c,d)$ once, then splitting by cases on the relevant clock and data inputs. This portion also requires that there is no rising edge (given by the first requirement on r).

Computation correctness.

Given clock and data signals c, d, a timestep i, and an r such that $c \overline{\uparrow} r$ and $\forall r', r < r' \leq i \to \neg(c \overline{\uparrow} r')$ (the most recent rising edge), we have

$$Q(c, d)(i) = d(r)$$

Proof: by induction between r and i (with the base case being $i = r$). We expand the definition of Q and compute based on the knowledge that there is no rising edge at i.

We can finally derive our final major lemma!

Simulation-computation correspondence.

Given clock and data signals c, d,

$$F_{\mathbf{SPEC}} \vdash F_5(c, d) = Q(c, d)$$

Proof: For all i, $F_{\mathbf{SPEC}}(i) = 1$ gives us the requirements for simulation correctness. From $\mathcal{G}_r^i(F(c, d))$, we get $F(c, d)(i)_5 = d(r)$. From computation correctness, we get $Q(c, d)(i) = d(r)$. Therefore $F_5(c, d)(i) = Q(c, d)(i)$.

6 Conclusion

In this work, we developed and applied a general methodology for formally verifying combinational circuits with delays and feedback loops using Graphiti and Lean 4. Through two case studies, we showed that refinement proofs between gate-level implementations and high-level specifications are feasible, and that the patterns established on simpler circuits transfer well to more complex ones.

6.1 Evaluation

The complete proof development is nearly 3000 lines of Lean 4 code, with over 200 definitions and lemmas. The entire file compiles from scratch in about 2 minutes on a laptop◇, not including libraries (including Graphiti).

◇ A mid-high-range laptop, with 32GB of RAM.

6.1.1 Proof techniques

The general approach used for most proofs was a loop of simplification (either applying various rewrites or with `simp`), then apply relevant lemmas, or use `grind`.

Generally, to make using `grind` easier, we simplified the goal and hypotheses until it is either "obvious" to a human reader or extremely mechanical. In particular, obligations about list lengths, signal indices, and timing constraints have their hypotheses reduced to sufficient conditions, then grinded away.

Bookkeeping was unfortunately quite heavy and very manual; it generally involved splitting into cases then applying a relevant lemma, which cannot be automated easily.

We also experimented with using Aristotle, an LLM-powered proof creator. Although it successfully identified incorrect lemmas and succeeded at proving correct ones, the proofs it produced were extremely difficult to understand and evaluated very slowly. In the end, I kept none of the actual proof content, but the general approach it used was useful for inspiration.

Although we expected loops to require a complex induction proof, reasoning is local enough that it is not relevant when proving consistency. When proving lemmas on the simulation, `grind` meant that its loops were not hard to reason about either.

6.1.2 Defining specifications

Surprisingly, finding the right approach to defining our specifications was extremely complex. The design space was large, and it is difficult to predict which approach will result in the best tradeoff of cleanliness, intuition, and ease-of-automation. We do conclude that filters are one of the best approaches, but we do believe they should be more integrated into the signals, rather than as part of a choice whenever the Graphiti module outputs.

Unfortunately, writing correct, easy-to-work-with filters while not exposing needless complexity is difficult. The stability filters are the most obvious case; do we use a constant delay, or should we reduce it in some paths, depending on the state?

The D-flip-flop is a good case study for this. Although we could have modeled its behaviour when the clock is unstable more accurately, it would've added

large amounts of complexity for no good reason. A clock staying low and high for a certain minimum length is a reasoanble assumption.

This same filter caused issues later due to a subtle error in it making its property definitions insufficient. Likewise, the interaction between the setup-hold filter and the delay filter produced cases in which the specification was intentionally undefined while still needing to remain compatible with the implementation.

These waveforms were very helpful in choosing the right filter parameters and in checking that the specification matched the circuit's observed behaviour before moving on to the formal proofs.

On the Graphiti side, buffers and speculation modules are also necessary in some form, but the internal connections increase the amount of state and bookkeeping needed. Alternative approaches include defining buffers and speculation as module-to-module functions adding this behaviour to a module, inlining their definitions into **SPEC**, or merging them into one module on the output side of the **SPEC** module.

Module-to-module functions create a new module based on the transitions of another passed as argument, and can be *extremely* generic.

Inlining was attempted but added too much noise. Merging the entire definition might risk the same, but should compensate with the reduced amount of state and internal transitions.

6.2 Future work

6.2.1 Improvements

There's room to simplify the proofs and reduce code duplication. In particular, some definitions may be overly computational, which makes some proofs significantly more complex. Finding more declarative definitions that are easier to reason about, while still being computationally equivalent, would reduce proof burden. One avenue for this could be linear temporal logic, which suits signals well.

Genericizing some of the shared components, such as the circuit simulation, could let other refinements be more automated and simpler to write. Developing custom Lean tactics for the repetitive parts of refinement proofs (particularly internal transitions) could also significantly reduce proof size and compilation time.

It'd also be worth investigating whether the filters could be defined at a different level of abstraction, rather than at the Graphiti level, to better integrate with the rest of the proof structure.

6.2.2 Extending

Natural next steps would be creating refinements for more complex pieces of hardware, adding "bookkeeping" refinements to simplify intermediate values, and eventually, end-to-end refinements for complex modules like an ALU or a simple CPU.

Alternatively, future work could aim to use a more flexible signal definition, such as continuous signals and true sets of possibilities at each timestep.

Bibliography

[1] A. Lööw and M. O. Myreen, "A Proof-Producing Translator for Verilog Development in HOL," in *2019 IEEE/ACM 7th International Conference on Formal Methods in Software Engineering (FormaliSE)*, IEEE, May 2019, pp. 99–108. doi: 10.1109/FormaliSE.2019.00020.

[2] J. Choi, J. Kim, and J. Kang, "Artifact for "Revamping Verilog Semantics for Foundational Verification"." [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.16923443

[3] O. Sankur, B. Boyer, and F. Faissole, "Verification of Generic VHDL Designs and Their Translation to Rocq," in *Verification, Model Checking, and Abstract Interpretation*, Springer Nature Switzerland, 2026, pp. 287–308. doi: 10.1007/978-3-032-15700-3_14.

[4] M. Hunzinger, "circuitlib: A digital circuit verification library for Lean 4." [Online]. Available: https://github.com/matthunz/circuitlib

[5] Verilean, "Sparkle: A type-safe, formally verifiable HDL compiler in Lean 4." [Online]. Available: https://github.com/Verilean/sparkle

[6] G. Kaye, "Foundations of Digital Circuits: Denotation, Operational, and Algebraic Semantics." [Online]. Available: https://arxiv.org/abs/2502.08497

[7] Y. Herklotz, A. Elakhras, M. Camaioni, P. Ienne, L. Josipović, and T. Bourgeat, "Graphiti: Formally Verified Out-of-Order Execution in Dataflow Circuits," in *Proceedings of the 31st ACM International Confer-*

ence on Architectural Support for Programming Languages and Operating Systems, Volume 2, in ASPLOS '26. USA: Association for Computing Machinery, 2026, pp. 821–837. doi: 10.1145/3779212.3790166.

[8] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically Scheduled High-level Synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, in FPGA '18. ACM, Feb. 2018, pp. 127–136. doi: 10.1145/3174243.3174264.

[9] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, May 1991, doi: 10.1016/0304-3975(91)90224-P.

[10] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Information Processing 74: Proceedings of the IFIP Congress 74*, J. L. Rosenfeld, Ed., Amsterdam: North-Holland, 1974, pp. 471–475.

[11] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)*, IEEE, Sept. 1977, pp. 46–57. doi: 10.1109/SFCS.1977.32.

[12] T. J. Chaney and C. E. Molnar, "Anomalous Behavior of Synchronizer and Arbiter Circuits," *IEEE Transactions on Computers*, no. 4, pp. 421–422, Apr. 1973, doi: 10.1109/T-C.1973.223730.