Formal Semantics of Verilog, Revisited for Deductive Verification

A Coq-Embedded Verification Framework as a Use Case

JOONWON CHOI

Conventional approaches to formal verification of hardware have been biased towards model checking, with deductive verification often viewed as unsuitable especially for hardware designs described in low-level languages like Verilog, due to the substantial human effort required for manual proofs. In this paper, we aim to challenge this perception by proposing a formal semantics of Verilog tailored for deductive verification. The key idea is to design a denotational semantics so that the semantics of a module is simply a state-transition function, thus avoiding the burden of deducing state-transition relations. To demonstrate the effectiveness of our formal semantics, we have prototyped a verification framework, embedded in the Coq proof assistant, that contains verification tools to make proofs faster. On top of the framework, as a case study, we formally proved the correctness of a RISC-V pipelined processor over a single-cycle processor as the specification.

1 INTRODUCTION

Formal verification of hardware designs has been an active research area for decades. Model checking [Clarke and Emerson 1981; Clarke et al. 1986; McMillan 1993] has been a dominant approach in hardware verification, and for a long time hundreds of research projects have contributed to make model checking more effective. That said, state explosion is still the most challenging problem for model checkers; the problem becomes much harder to overcome, obviously, when dealing with hardware designs described in a low-level hardware-description languages (HDLs) such as Verilog [Sys 2018].

Deductive verification has emerged as an alternative approach to hardware verification, gaining popularity in recent years. Particularly, interactive theorem proving (ITP) has been used in the hardware world. Instead of letting the tool to explore state space and try to verify properties automatically, ITP requires a user to provide mathematical proofs manually, and the tool machine-checks if the proofs are correct or not. By leveraging high-order logic and parameterization provided by theorem provers, users can design reusable parameterized proofs for various purposes. In contrast, model checkers have been known to be effective only for the properties described in first-order logic.

However, ITP also has a scalability issue as it requires considerable human effort to provide manual proofs. In order to overcome it, in the hardware world, recent approaches have designed and verified either high-level hardware models or modules described in high-level HDLs, by leveraging the abstraction coming with the models or languages [Bourgeat et al. 2020; Choi et al. 2022, 2017; Vijayaraghavan et al. 2015]. To the best of our knowledge, no work has directly aimed to prove the correctness of nontrivial hardware designs described in register-transfer-level (RTL) languages like Verilog. Apparently, there has been a common perception that the RTL languages are too low-level, and thus it is not feasible to employ ITP to verify RTL designs in a scalable way. Nevertheless, it is undeniable that Verilog remains the most popular HDL, dominantly ussed in industry for designing high-performance hardware blocks.

The complexity of formal semantics poses a significant obstacle to achieving scalability in ITP. While Verilog already has its standard formal semantics [Sys 2018], it is observed to be excessively

Author's address: Joonwon Choi, joonwonc@alum.mit.edu.

2023. XXXX-XXXX/2023/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

intricate, making it unsuitable for direct use in conducting proofs. Verilog is a language not only for describing synthesizable hardware designs but also for modeling designs for simulation that are generally not synthesizable. In order to encompass both the synthesis and simulation aspects of Verilog, the standard has adopted "scheduling semantics," which allows for *nondeterminism* in the concurrent execution of design blocks. In this approach, each block is treated as an individual process that can generate events during its execution, subsequently triggering the concurrent execution of other processes. The scheduling semantics covers all possible outcomes resulting from such concurrent execution of processes with various interleaving cases. The presence of such nondeterminism is generally unfavorable in theorem proving as it increases the number of cases that need to be proven. Consequently, the current state of the standard semantics appears unsuitable for conducting proofs.

Our focus in this paper lies only in *practical and synthesizable* part of Verilog. We observe that nondeterminism in the scheduling semantics serves its major role in designs that are either not synthesizable or do not occur in practice. In fact, there exist coding guidelines [Cummings 2000], widely recognized among hardware designers, that aim to ensure deterministic and synthesizable hardware designs. By targeting this specific subset of Verilog, we can develop a formal semantics that enables more efficient deductive verification. Rather than interpreting the execution of a Verilog module as the concurrent execution of processes with nondeterminism, we can simplify it as a conventional state-transition system that deals with the module's state as a whole.

Additionally, we observe another opportunity for further optimization of the semantics once we eliminate nondeterminism. Even when treating a Verilog module as a state-transition system, we may still face a significant bottleneck when handling state transitions defined as *relations* within an *operational* semantics. For instance, when a user intends to verify a Verilog module by stating and proving properties such as invariants, operational semantics imposes proof obligations, requiring the user to *deduce* the state-transition-relation propositions provided as hypotheses. As the size of the target module increases, this task becomes particularly burdensome, resulting in longer deduction times for the theorem prover.

Summarizing the aforementioned points, our ultimate goal is to establish a *denotational* semantics of Verilog. By employing the denotational semantics, we obtain a state-transition *function* that simplifies the process of proving properties through function applications rather than deducing transition relations. It is important to reiterate that our goal is *not* to develop the most precise and comprehensive formal semantics of Verilog, but rather to design one that can be conveniently employed for verification purposes.

In order to demonstrate the usability of our proposed formal semantics, we have prototyped a verification framework, embedded in the Coq proof assistant, that contains verification techniques associated with our semantics. Furthermore, on top of the framework, as a case study, we formally proved the correctness of a pipelined processor described in Verilog over a single-cycle processor as the specification.

To sum up, the contribution of this paper is listed as follows[1]:

- We design formal semantics of Verilog tailored for deductive verification (Section 3). Our proposed semantics is denotational across all the syntax levels – from expressions to modules – thus the semantics for a module is simply a state-transition function.
- We prototype a verification framework (Section 4), embedded in Coq, that takes advantage of the denotational semantics we designed, along with verification techniques to enable proofs more efficient.

[1]Our formal semantics of Verilog, framework, and case study are all described in and machine-checked by Coq, submitted as the supplementary material.

```
1  module example #(parameter integer g = 0)
2    (input logic clk,
3     input logic rst_n,
4     output logic t);
5    reg r;
6    wire s, u;
7
8    assign s = !u;
9
10   always_comb begin
11     t = r;
12     t = t || s;
13   end
14
15   always @(posedge clk) begin
16     if (!rst_n) r <= g;
17     else if (u) r <= t;
18   end
19
20   another_module ins(.clk, .rst_n, .u);
21 endmodule
```

$$\{\ \mathcal{S}[u] = 1;\ \}$$
$$\downarrow$$
$$\{\ \mathcal{S}[u] = 1;\ \mathcal{S}[s] = 0;\ \}$$
$$\downarrow$$
$$\{\ \mathcal{S}[u] = 1;\ \mathcal{S}[s] = 0;\ \mathcal{S}[t] = \mathcal{R}[r];\ \}$$
$$\downarrow$$
$$\left\{\begin{array}{l} \mathcal{S}[u] = 1;\ \mathcal{S}[s] = 0;\ \mathcal{S}[t] = \mathcal{R}[r]; \\ \mathcal{R}_{\text{next}}[r] = \mathcal{R}[r]; \end{array}\right\}$$

Fig. 1. An example Verilog module and its denotation steps

- On top of the framework, we formally prove the correctness of a pipelined processor as a case study (Section 5), demonstrating that the proofs are machine-checked faster (Section 6).

2 OVERVIEW

2.1 The Verilog Language

Verilog is a hardware-description language (HDL) used to describe hardware designs at the register-transfer level (RTL). Users can design Verilog modules, and if the design is synthesizable, it can be transformed into actual hardware circuits. Not all Verilog designs are synthesizable, since Verilog is also extensively used for simulation. Verilog includes syntactic components that define simulation processes and their timing controls, which are typically not synthesizable. The entire Verilog syntax and semantics have been described formally in the IEEE Standards [Sys 2018]. In order to encompass the behaviors of both synthesizable and simulation modules, the standard has adopted what is known as "scheduling semantics."

The scheduling semantics treats modules as concurrent reactive systems. Under this semantics, each logic block is seen as a *process* that can be triggered by an *event*. There are various ways to generate an event; e.g., an "update" event is generated when a logic value is updated (assigned). Assignments can occur while a block is being executed, resulting in triggering other blocks to be executed concurrently. In this case, the scheduling semantics makes a *nondeterministic choice* regarding which block to execute; it can either resume execution after the assignment or initiate the execution of a newly triggered block[2].

Hardware designers strive to minimize nondeterminism in their designs. If a design exhibits any nondeterministic behaviors, the resulting synthesized circuit may deviate from the intended behavior as the synthesizer makes its own choices among possible behaviors. Due to this concern, when employing Verilog to describe synthesizable hardware blocks, designers have adhered to *guidelines* [Cummings 2000] as a general principle to eliminate any instances of nondeterminism or race conditions. This paper specifically focuses on this particular set of designs.

Figure 1 presents an example Verilog module, deliberately created to elucidate our motivation. This module follows the guidelines and thus is synthesizable in a deterministic manner. A module

[2]The standard [Sys 2018] explicitly describes this nondeterminism in dedicated sections: 4.7 Nondeterminism and 4.8 Race conditions.

is a basic building block that implements a certain functionality. Using its definition, a module can be instantiated in the definition of another module, forming a module hierarchy. In this way, two modules communicate to each other. A module definition can be parameterized by taking `parameters`, and each instance can have different parameter values. The module shown in the figure has a integer parameter called `g` that has the default value 1.

A module definition contains input and output ports. The example module takes two inputs `clk` and `rst_n` and a single output `t`. The clock (`clk`) and the reset (`rst_n`) signals are typical inputs to a module: `rst_n` is 0 before the module starts to run, and keeps asserted (to 1) during the execution. The clock regularly changes its value between 0 and 1.

Verilog has two groups of data objects: nets and variables. *Nets* are used just to connect signals and logics, making them stateless. **`wire`** is a typical data type belonging to nets. On the other hands, *variables* can hold data. **`reg`** is a typical data type belonging to variables. One common misconception is that **`reg`** exclusively represents Verilog registers. In practice, **`reg`** is indeed used to declare registers, but in fact it can also represent combinational logics (in **`always`** blocks introduced soon). A signal can also have a type **`logic`**, and in this case it can be either a net or a variable, determined by the context.

Throughout this paper, we will adopt a consistent approach to eliminate any potential confusion regarding nets and variables. First of all, **`reg`** will be exclusively used for declaring registers. We will also use the word "wire" to encompass not only the net type **`wire`** but also any signals that are driven combinationally (inside **`always`** blocks). Likewise, we will use the word "register" literally to denote stateful Verilog registers.

2.2 Challenges in Interpreting Assignments

Every signal in Verilog is declared first, and assigned later. In the example shown in Figure 1, `t` is declared as an output **`logic`**, and a register (**`reg`** `r`) and two wires (**`wire`** `s, u`) are declared at the top of the module. All the signals in the example module contain a single bit.

Right after the declarations, we see an *assignment* statement for `s` (**`assign`** `s = !u` at line 8, called a continuous assignment in the standard). A typical challenge in designing formal semantics arises here; at the moment of defining the semantics of this assignment, we have not evaluated the value of `u` yet, which is used in the right-hand side. In Verilog, it is totally legal to use (read) a signal prior to its assignment, and still the value should be derived from the assignment. (In this sense, Verilog is not entirely a procedural language like C.) The scheduling semantics used in the standard addresses this situation by ensuring that the assignment is executed whenever any operand on the right-hand side is updated. In this example, the assignment will be executed whenever `u` is updated.

An "always" block (at line 10) comes after the assignment, and in this example **`always_comb`** indicates that the statement intends to define a combinational-logic circuit. We first see a *blocking assignment* (`t = r`), which blocks all the statements after this assignment within a statement block, until the assignment is executed completely. The effect of the assignment is then propagated throughout the remaining statements. In this example, `t = r` is executed first, and the next statement is executed with the value of `t` being equal to `r`. As `r` is previously declared as a register, `t` takes the current register value of `r`. Upon executing the next statement (`t = t || s`), the value of `t` on the right-hand side should be equal to `r`. Here we encounter a similar problem again, this time with `s` not yet evaluated. Despite an assignment for `s` prior to the "always" block, we can obtain the value once the execution is successfully carried out.

The standard handles the execution of the "always" block in a similar manner to continuous assignments. In this example, the **`always_comb`** block is executed whenever there is an update to `r` or `s`. (The standard calls `r` and `s` a *sensitivity list*, and **`always_comb`** automatically generates the sensitivity list by looking at its body.) An inherent inefficiency/subtlety arises within the scheduling

semantics in this scenario. When `t = r` is executed, based on the semantics, an update event for `t` is generated and may trigger the execution of other blocks that are "sensitive" to `t`, *even if* `t` has not been fully evaluated yet. This trigger becomes meaningless in such cases[3]. As expected, if the design adheres faithfully to the guidelines, the combinational logic can be fully evaluated *without* the need for generating intermediate update events.

Another "always" block (at line 15) is described next, now with a *timing control* `@(posedge clk)`, indicating that the statement is executed whenever the clock value (`clk`) transitions from 0 to 1 (thus making a positive edge). In this block, we see *nonblocking assignments* `r <= g` and `r <= t`. A nonblocking assignment does not block any subsequent statements, i.e., the assignment and the subsequent statements run concurrently. If a design adheres to the guidelines, nonblocking assignments are always used for register updates. While assigning the next register value for `r`, we once again encounter the same problem of not knowing the value of `t` (in `r <= t`).

Lastly, the module includes a module instance (at line 20). `ins` is an instance of a module called `another_module`, which just takes the clock (`clk`) and reset (`rst_n`) signals as inputs and returns a single bit `u`. Note that in describing input/output connections a single name can be used (e.g., `.clk`) if the declaration and the binding names are identical. Henceforth, we will assume that `another_module` always outputs 1 for the signal `u`.

2.3 Challenges in Relational Semantics

Working with a design that adheres to the guidelines, the situation improves as there is no longer a need to generate intermediate update events. However, we can still observe the ongoing issue of handling assignments with unevaluated signals. In such cases, it is common and natural to define formal semantics in a *relational* way. In this approach, each wire `p` has a relation with a value v if the assignment to `p` evaluates to v. Defining this relation as $\mathcal{E}(p, v)$, we can derive the semantics for the assignments for `s` (by `assign`) and `t` (by `always`) in the example as follows:

$$\mathcal{E}(u, v_u) \vdash \mathcal{E}(s, !v_u) \qquad \mathcal{E}(s, v_s) \vdash \mathcal{E}(t, \mathcal{R}[r] \,||\, v_s),$$

where $\mathcal{R}$ is a register state defined as a finite map containing the current register values.

Although seemingly intuitive for addressing the evaluation problem, relational semantics imposes an additional burden on users to prove properties of modules such as invariants, particularly in the context of deductive verification. In this example, we can claim an invariant that the value of `r` is always 0, with the value of `u` always being 1 as assumed above. In order to prove this invariant, it is necessary to prove the following property as an induction step:

$$\mathcal{E}(u, v_u) \wedge \mathcal{E}(t, v_t) \quad \vdash \quad \mathcal{R}[r] = 0 \longrightarrow (\text{if } (v_u) \text{ then } v_t \text{ else } \mathcal{R}[r]) = 0.$$

This induction step claims that whenever `r` is updated, its assigned value should be 0 (`if (u) r <= t;`).

The burden comes from deduction of the values from the relation $\mathcal{E}$, and it becomes significantly more intricate when the relations are "chained," e.g., evaluating `t` requires the evaluation of `s`, which in turn requires `u`. Deducing a large number of such relations extends the time it takes for a theorem prover to check the proofs; users may have to wait longer, even while they are in the midst of writing the proof. We call this problem a decrease of *proof-engineering performance*.

We aim to solve this problem by designing the formal semantics in a *denotational* way. For a given Verilog module, instead of working with a transition relation, we define a *state-transition function* as the denotation of the module. This denotation specifically takes input signals and the current register state as arguments and returns the next register state and output signals. Returning

[3]Furthermore, the standard allows the behavior where the update event of the last assignment is ignored, which violates the semantics of combinational logic. Section 6 will provide details of this semantic flaw.

to the example, the transition function generates the next register state, denoted as $\mathcal{R}_{\text{next}}$, that contains the value for `r`, derived through the following sequence:

$$\begin{aligned}
\mathcal{R}_{\text{next}}[r] &= \text{if } (\mathcal{S}[u]) \text{ then } \mathcal{S}[t] \text{ else } \mathcal{R}[r] \\
&= \text{if } (\mathcal{S}[u]) \text{ then } (\mathcal{R}[r] \,||\, \mathcal{S}[s]) \text{ else } \mathcal{R}[r] \\
&= \text{if } (\mathcal{S}[u]) \text{ then } (\mathcal{R}[r] \,||\, !\mathcal{S}[u]) \text{ else } \mathcal{R}[r] \\
&= \text{if } (1) \text{ then } (\mathcal{R}[r] \,||\, !1) \text{ else } \mathcal{R}[r] \\
&= \mathcal{R}[r] \,||\, 0 = \mathcal{R}[r],
\end{aligned}$$

where $\mathcal{S}$ contains the current wire values. Since $\mathcal{R}_{\text{next}}[r] = \mathcal{R}[r]$, it is obvious to prove that the value of `r` is always 0. (The reset value is 0 by `r <= g` with `g = 0`.)

Our denotational semantics captures the aforementioned evaluation sequence through the derivation of the *least fixed point* (LFP) of the current wire state, by traversing the module body multiple times. In each iteration, we *add* wire values that can be evaluated with those that have been *evaluated thus far*. The right-hand side of Figure 1 shows the iterations for the example. The first iteration begins with no wire values, resulting in only the value of `u` obtained from the module instance `ins` (at line 20). In the second iteration, using the value of `u`, we additionally obtain the value of `s` (`assign s = !u` at line 8). After four iterations, we can eventually derive all the wire values and the register updates.

Details of our denotational semantics, including this iteration, will be discussed in Section 3.2. Note that our contribution does not lie in the algorithm for deriving a transition function. Instead, we argue that this approach enables fast and practical derivations, effectively reducing the burden associated with using relational semantics. We will demonstrate the practicality of our denotational semantics by verifying nontrivial hardware designs in our case study (Section 5).

3 FORMAL SYNTAX AND SEMANTICS OF VERILOG

In this section, we provide our formal syntax and semantics of Verilog. The syntax is defined as an abstract syntax tree (AST), directly representing Verilog modules. We then propose our formal semantics in Section 3.2, by denoting through each syntax level.

Notations. Before we provide the formal syntax and semantics, we define notations used generally throughout this paper. An overline (e.g., $\overline{l}$) denotes a list. $[]$, $(e :: \overline{l})$, and $(\overline{l_1} + \overline{l_2})$ denote nil, cons, and append, respectively. We use $\langle\cdot\rangle$ to denote a tuple. A tuple may have a label (e.g., $\langle\cdot\rangle_{\text{label}}$) for distinction. Round parantheses are used to denote a tuple as well (e.g., (t_1, t_2)), but in this case no labels are attached. Lastly, we use just a number to access the n-th element of a tuple (e.g., t.1, t.2).

3.1 Syntax

The syntax and semantics of Verilog have been already documented as the IEEE Standards [Sys 2018], which we will refer to as the standard in this paper. As mentioned in Section 2, our focus is specifically on designs that follow the guidelines [Cummings 2000] to ensure deterministic behaviors in their concurrent executions. In this paper, we will discuss a part of syntax we support, but still enough to discuss nontrivial features and our proposed formal semantics. The complete formal syntax and semantics are defined in our Coq artifact.

Expressions, L-values, and events. Figure 2 describes the formal syntax of Verilog. An *expression* (represented as "e") is the smallest unit to form a value in Verilog. An expression is inductively defined, either a literal (c), an ID (id), a hierarchy selection ($e.e$), an index selection ($e[e]$), a range ($e[e:e]$), a concatenation ($\{\overline{e}\}$), a duplicate ($\{e\{\overline{e}\}\}$), a casting ($e'(e)$), a unary operation ($\text{op}(e)$), a binary operation ($\text{op}(e, e)$), a conditional expression ($e \ ? \ e : e$), or an "inside" expression ($e \in \overline{e}$).

Modules			
m	::=	module $\langle \mathsf{id}, \overline{\langle \mathsf{id}, e\rangle}, \overline{\mathsf{id}}, \overline{\mathsf{id}}, \overline{gb}\rangle$	
Generate blocks			
gb	::=	bl	(Non-generate blocks)
	|	if (e) then id : gb else id : gb	(Conditional generate blocks)
Blocks			
bl	::=	logic id	(Declarations)
	|	assign $\langle lv, e\rangle$	(Assignments)
	|	always $\overline{st}$	(Always blocks)
	|	$\langle \mathsf{id}, \mathsf{id}, \overline{\langle \mathsf{id}, e\rangle}, \overline{\langle \mathsf{id}, e\rangle}, \overline{\langle \mathsf{id}, lv\rangle}\rangle_{\mathsf{module}}$	(Module instances)
Statements			
st	::=	lv = e	(Blocking assignments)
	|	lv <= e	(Nonblocking assignments)
	|	if (e) then $\overline{st}$ else $\overline{st}$	(Conditional statements)
	|	case(e) $\overline{\langle e, \overline{st}\rangle}$ $\overline{st}$	(Cases)
	|	@(ee) $\overline{st}$	(Event-control statements)
Expressions			
e	::=	c | id	(Literals / IDs)
	|	$e.e$ | $e[e]$	(Hierarchy / Index selections)
	|	$e[e : e]$	(Ranges)
	|	$\{\overline{e}\}$ | $\{e\{\overline{e}\}\}$	(Concatenations / Duplications)
	|	e'(e)	(Castings)
	|	op(e) | op(e, e)	(Unary / Binary operations)
	|	e ? e : e | $e \in \overline{e}$	(Conditional / Inside expressions)
lv	::=	e	(L-values)
ee	::=	posedge e | negedge e	(Event expressions)

Fig. 2. Formal syntax of Verilog (excerpts)

The Verilog standard syntax defines L-values separately as a subset of expressions. For simplicity, we define the L-value (represented as "lv") same as the expression. Our formal semantics will not accept any expressions that cannot be L-values.

An *event expression* (represented as "ee") is either a positive-edge event (posedge e) or a negative-edge event (negedge e)[4]. Similar to our L-value, the event expression takes only a part of expressions (in posedge or negedge), but we define it same as the expression for convenience.

Statements. Statements contain elements that are used to build a circuit block. A *statement* (represented as "st") is either a blocking assignment (lv = e), a nonblocking assignment (lv <= e), a conditional statement (if (e) then $\overline{st}$ else $\overline{st}$), a case statement (case(e) $\overline{\langle e, \overline{st}\rangle}$ $\overline{st}$), or an event-control statement (@(ee) $\overline{st}$). Blocking assignments are used to assign values to wires, forming a combinational logic circuit in the end. Nonblocking assignments are used to assign values to registers, forming a sequential logic. An event-control statement runs only when the associated event happens.

(Generate) Blocks. Blocks are circuits that run concurrently for each cycle. A *block* is either a declaration (logic id), a continuous assignment (assign $\langle lv, e\rangle$), an "always" block (always $\overline{st}$), or a module instance ($\langle \mathsf{id}, \mathsf{id}, \overline{\langle \mathsf{id}, e\rangle}, \overline{\langle \mathsf{id}, e\rangle}, \overline{\langle \mathsf{id}, \mathsf{id}\rangle}\rangle_{\mathsf{module}}$). A module instance takes a module ID (id),

[4]We do not support manually providing a sensitivity list for logics; it is common practice to allow a synthesizer to infer the list for a given block by using either "`always @(*)`" or "`always_comb`".

```
Definition m := #[
module example #(parameter integer g = 1)
  (input logic clk,
   input logic rst_n,
   output logic t);
  reg r;
  wire s, u;
  assign s = !u;
  always_comb begin
    t = r;
    t = t || s;
  end
  ...
endmodule].
```

```
m := VModuleDeclAnsi example
  (VParamDeclData
    (VDataTypeOrImplicitDat ..)
    (VParamAssignOne g ..))
  (VAnsiPortDeclsCons
    (VAnsiPortDeclVar ..)
    (VAnsiPortDeclsCons ..))
  (VModuleItemsCons (VVarDeclOne ..)
    (VModuleItemsCons
    (VNetDeclOne VNetTypeWire VPackedDimsNil ..)
      (VModuleItemsCons
        (VContAssignNet ..)
        (VModuleItemsCons ..))))
  : VModuleDecl
```

Fig. 3. Parsing a Verilog module in Coq

i	$\in$	$\mathbb{N}$			(Indices)
b	$\in$	$\mathbb{Z} * \mathbb{N} * \mathbb{B}$	::=	$\langle val, size, sign \rangle_{\text{b}}$	(Bits)
h	$\in$	$\mathcal{H}$	::=	$[] \mid b \mid \overline{\langle i, h \rangle_{\text{arr}}} \mid \overline{\langle \text{id}, h \rangle_{\text{str}}}$	(Hierarchical maps (HMap))
p	$\in$	$\mathcal{P}$	::=	$i \mid \text{id} \mid p + p$	(Hierarchical paths)

Fig. 4. Representation of bits and hierarchical maps

an instance ID (id), parameter values ($\overline{\langle \text{id}, e \rangle}$), input values ($\overline{\langle \text{id}, e \rangle}$), and output binds ($\overline{\langle \text{id}, lv \rangle}$) in order. The Verilog syntax has a notion of *generate blocks*; as its name says, blocks are "generated" under certain conditions. A generate block is either just a non-generate block (bl) or a conditional generate block (if (e) then id : gb else id : gb). The standard syntax requires that any condition in a generate block must be a "constant expression," which are not defined explicitly in our formal syntax. Once again, as constant expressions are a subset of expressions, we consider the condition as an expression, evaluate it in the formal semantics, and decide whether to generate the associated block or not.

Modules. A verilog *module* (module $\langle \text{id}, \overline{\langle \text{id}, e \rangle}, \overline{\text{id}}, \overline{\text{id}}, \overline{gb} \rangle$) takes a module ID (id), parameter values ($\overline{\langle \text{id}, e \rangle}$), input declarations ($\overline{\text{id}}$), output declarations ($\overline{\text{id}}$), and generate blocks as a body ($\overline{gb}$). Note that parameter values (also the ones in a module instance) must be constant expressions, according to the standard syntax, but again we take and evaluate them as expressions.

3.1.1 Parsing Verilog modules in Coq. We have described our formal syntax in Coq by defining inductive abstract syntax tree (AST) types. In order to accommodate Verilog modules as they are, we established custom notations[5] for each syntactic data structure, allowing Coq to parse and transform a Verilog module into its corresponding AST.

Figure 3 shows a Verilog module written down in Coq and the corresponding AST (parsed by Coq). `m` is a Coq value defined by using custom notations; the actual value of `m` is an AST, where the highest-level constructor is `VModuleDeclAnsi`, representing a Verilog module.

3.2 Our Proposed Formal Semantics

We now present our formal semantics of Verilog. We begin by introducing the data structures we have defined to represent semantic values. Subsequently, we proceed to present the semantics for each level of syntax, ranging from expressions to modules.

3.2.1 Data structures. Figure 4 presents data structures used in our formal semantics. To represent bits, we employ a tuple containing the value and size as natural numbers ($\mathbb{N}$, including zero), and

[5]We heavily utilized Coq's "custom entries" to implement Verilog notations.

```
1 module example (..);
2   reg r;
3   generate if (..) begin: gen
4     reg r;
5     ...
6   end endgenerate
7   another_module ins(.r);
8   ...
9 endmodule].
```

$$s := \begin{bmatrix} \langle \mathsf{r}, b_0 \rangle_{\mathrm{str}}; \\ \langle \mathsf{gen}, [\langle \mathsf{r}, b_1 \rangle_{\mathrm{str}}; \cdots] \rangle_{\mathrm{str}}; \\ \langle \mathsf{ins}, \cdots \rangle_{\mathrm{str}} \\ \cdots \end{bmatrix}$$

Fig. 5. A semantic state defined as a hierarchical map

the sign as a Boolean ($\mathbb{B}$). This choice of representation is primarily motivated by utilizing a wide range of lemmas and proof-automation tactics for bits-as-numbers in Coq. A drawback of this representation is that it is not canonical. For instance, the equivalence of two bits ($\equiv$) should be defined as follows:

$$b_1 \equiv b_2 \triangleq (b_1.val - b_2.val) \,\%\, 2^{b_1.size} = 0 \ \wedge \ b_1.size = b_2.size \ \wedge \ b_1.sign = b_2.sign.$$

Next, we introduce a new data structure called "hierarchical maps" (HMap, henceforce) to represent general Verilog values and states. As shown in the figure, an HMap h is defined inductively, encompassing an empty map ([]), a "bits" element (b), an array ($\overline{\langle i, h \rangle_{\mathrm{arr}}}$), or a struct ($\overline{\langle \mathsf{id}, h \rangle_{\mathrm{str}}}$). Structs in an HMap not only represent ordinary struct values in Verilog, but also cover generate blocks and module instances. Consequently, a Verilog semantic state can be defined by an HMap as well. Figure 5 provides an example that an HMap is used as a module state. Note that structs are inductively defined for both a generate block (declared as gen in the figure) and a module instance (declared as ins).

3.2.2 The key concepts. Before delving into our proposed formal semantics, we would like to emphasize the two key high-level concepts we have employed in designing formal semantics to enhance deductive verification.

- Denotational: we design denotational semantics across all levels of syntax. As a result, the denotation of a Verilog module becomes just a state-transition function rather than a relation. In order to realize it, we employed simple fail monads to determine whether the evaluation of a syntactic component (e.g., an expression, a statement, etc.) was successful or not.
- Per-logic: we aim to avoid carrying the entire transition function for a module, especially when only specific state changes are necessary. In particular, when an invariant is stated for specific registers only, we would like to apply the transition functions solely to those registers. This approach is primarily motivated by the performance aspect of proof engineering; checking proofs tends to be slower when larger structures are involved.

3.2.3 Semantics for expressions, L-values, and events. Figure 6 describes formal semantics for expressions, L-values, and events. Denotation of an expression ($[\![\cdot]\!]_e$) is a function that takes the current evaluation position ($p \in \mathcal{P}$) and the current state evaluated up to the current position ($s \in \mathcal{S}$) and returns the evaluated value ($\in \mathcal{V}$) or fails to evaluate. $\mathcal{S}$ and $\mathcal{V}$ are both HMaps, i.e., $\mathcal{S} = \mathcal{V} = \mathcal{H}$. At the moment evaluating an expression, s should contain all the values of 1) the parameters and inputs of the module, 2) the current registers, and 3) the wires evaluated so far. Throughout the section, we will simply call s the current state. While the most cases are straightforward, there are nontrivial cases worth analyzing in detail.

- $[\![\mathsf{id}]\!]_e$: search for a value of id, considering the current position p. For instance, in Figure 5, there are two registers with the ID r, one at the top level and the other in the generate block gen. If the current position is $p = [\mathsf{gen}]$, the value should be the one mapped by [r; gen] in

$$
\begin{array}{rcl}
[\![\cdot]\!]_e & : & \mathcal{P} * \mathcal{S} \to \mathsf{Fail}\ \mathcal{V} \\
[\![c]\!]_e(p,s) & = & c \\
[\![\mathsf{id}]\!]_e(p,s) & = & s\lfloor \mathsf{id} :: p \rfloor_{\mathsf{v}} \\
[\![e_p.\mathsf{id}]\!]_e(p,s) & = & [\![e_p]\!]_e(p,s)[\mathsf{id}] \\
[\![e_1[e_2]]\!]_e(p,s) & = & [\![e_1]\!]_e(p,s)[[\![e_2]\!]_e(p,s)] \\
[\![e_1[e_2:e_3]]\!]_e(p,s) & = & [\![e_1]\!]_e(p,s)[[\![e_2]\!]_e(p,s) : [\![e_3]\!]_e(p,s)] \\
[\![\{\overline{e_i}\}]\!]_e(p,s) & = & \{\overline{[\![e_i]\!]_e(p,s)}\} \\
[\![\{e_c\{\overline{e_i}\}\}]\!]_e(p,s) & = & \{[\![e_c]\!]_e(p,s)\{\overline{[\![e_i]\!]_e(p,s)}\}\} \\
[\![e_1\text{'}(e_2)]\!]_e(p,s) & = & ([\![e_1]\!]_e(p,s))\text{'}([\![e_2]\!]_e(p,s)) \\
[\![\mathsf{op}(e)]\!]_e(p,s) & = & \mathsf{op}_{\mathsf{h}}([\![e]\!]_e(p,s)) \\
[\![\mathsf{op}(e_1,e_2)]\!]_e(p,s) & = & \mathsf{op}_{\mathsf{h}}([\![e_1]\!]_e(p,s),[\![e_2]\!]_e(p,s)) \\
[\![e_c\ ?\ e_t : e_f]\!]_e(p,s) & = & \mathsf{if}\ ([\![e_c]\!]_e(p,s))\ \mathsf{then}\ [\![e_t]\!]_e(p,s)\ \mathsf{else}\ [\![e_f]\!]_e(p,s) \\
[\![e \in \overline{e_i}]\!]_e(p,s) & = & \mathsf{if}\ ([\![e]\!]_e(p,s) \in \overline{[\![e_i]\!]_e(p,s)})\ \mathsf{then}\ 1\ \mathsf{else}\ 0 \\
\\
[\![\cdot]\!]_{lv} & : & \mathcal{P} * \mathcal{D} * \mathcal{S} \to \mathsf{Fail}\ \mathcal{P} \\
[\![\mathsf{id}]\!]_{lv}(p,d,s) & = & d\lfloor \mathsf{id} :: p \rfloor_{\mathsf{p}} \\
[\![e.\mathsf{id}]\!]_{lv}(p,d,s) & = & \mathsf{id} :: [\![e]\!]_{lv}(p,d,s) \\
[\![e[e_i]]\!]_{lv}(p,d,s) & = & [\![e_i]\!]_e(p,s) :: [\![e]\!]_{lv}(p,d,s) \\
\\
[\![\cdot]\!]_{ee} & : & \mathcal{P} * \mathcal{S} \to \mathsf{Fail}\ \mathcal{V} \\
[\![\mathsf{posedge}\ e]\!]_{ee}(p,s) & = & [\![e]\!]_e(p,s) \\
[\![\mathsf{negedge}\ e]\!]_{ee}(p,s) & = & [\![e]\!]_e(p,s)
\end{array}
$$

Fig. 6. Formal semantics for expressions, L-values, and events

the state. We defined a search function, denoted as $s\lfloor \mathsf{id} :: p \rfloor_{\mathsf{v}}$, and used it exactly in denoting ID expressions.

- $[\![e_p.\mathsf{id}]\!]_e$: evaluate e_p and search for a value mapped by id in the evaluated value. $h[\mathsf{id}]$ is an ordinary struct field accessor, which fails to evaluate if h is not a struct.
- $[\![e_1\text{'}(e_2)]\!]_e$: evaluate e_1 and e_2 and apply a casting function defined in HMap. Currently, the casting function only supports bits-to-bits casts, though we believe the other cases are not really practical.
- $[\![\mathsf{op}(e_1,e_2)]\!]_e$: evaluate e_1 and e_2 and apply the binary operation defined in HMap (op_{h}). The operations faithfully follow the evaluation rules described in the standard, e.g., adding two bits with different sizes and signs.

To enhance readability, we have chosen not to explicitly show the monadic "bind" operations in Figure 6 to obtain the values of subexpressions. If any subexpression fails to evaluate, the overall evaluation of the expression also fails. We will adopt this convention to other denotations as well.

Denotation of an L-value ($[\![\cdot]\!]_{lv}$) is a function that takes the current position ($p \in \mathcal{P}$), declarations evaluated so far ($d \in \mathcal{D}$), and the current state ($s \in \mathcal{S}$) and returns the evaluated path ($\in \mathcal{P}$) or fails to evaluate. $\mathcal{D}$ is also an HMap (i.e., $\mathcal{D} = \mathcal{H}$), where each leaf value has no semantic meaning. Like the current state s, the declarations d contains all the register/wire declarations up to the current position p. Among the cases, it appears that $[\![\mathsf{id}]\!]_{lv}$ is the only nontrivial one. The denotation of id (as an L-value) corresponds to a path that can be obtained by searching for the position of id from d, following a similar approach as used for $[\![\mathsf{id}]\!]_e$. We have defined a path-finding function, denoted as $d\lfloor \mathsf{id} :: p \rfloor_{\mathsf{p}}$, that employs the same search algorithm as $s\lfloor \mathsf{id} :: p \rfloor_{\mathsf{v}}$.

Denotation of an event ($[\![\cdot]\!]_{ee}$) is a function that has the same type as $[\![\cdot]\!]_e$. Currently, our semantics only supports a *single clock domain* and does not allow using posedge and negedge at the same

$$
\begin{array}{rcl}
[\![\cdot]\!]_{st} & : & \mathcal{P} * \mathcal{D} * \mathcal{S} \to \mathsf{Fail}\ (\mathcal{S}_{\mathrm{u}} * \mathcal{R}_{\mathrm{u}}) \\
[\![lv = e]\!]_{st}(p,d,s) & = & \langle [[\![lv]\!]_{lv}(p,d,s) := [\![e]\!]_e(p,s)], [] \rangle \\
[\![lv <= e]\!]_{st}(p,d,s) & = & \langle [], [[\![lv]\!]_{lv}(p,d,s) := [\![e]\!]_e(p,s)] \rangle \\
[\![\mathsf{if}\ (e)\ \mathsf{then}\ \overline{st_t}\ \mathsf{else}\ \overline{st_f}]\!]_{st}(p,d,s) & = & \left\langle \begin{array}{l} ([\![\overline{st_t}]\!]_{\overline{st}}(p,d,s)).1 \uplus \{[\![e]\!]_e(p,s)\}\ ([\![\overline{st_f}]\!]_{\overline{st}}(p,d,s)).1, \\ ([\![\overline{st_t}]\!]_{\overline{st}}(p,d,s)).2 \uplus \{[\![e]\!]_e(p,s)\}\ ([\![\overline{st_f}]\!]_{\overline{st}}(p,d,s)).2 \end{array} \right\rangle \\
[\![\mathsf{case}(e_c)\ []\ \overline{st_d}]\!]_{st}(p,d,s) & = & [\![\overline{st_d}]\!]_{\overline{st}}(p,d,s) \\
[\![\mathsf{case}(e_c)\ (\langle e_0, \overline{st_0}\rangle :: \overline{\langle e_i, \overline{st_i}\rangle})\ \overline{st_d}]\!]_{st}(p,d,s) & = & \left\{ \begin{array}{l} sr_0 \leftarrow [\![\overline{st_0}]\!]_{\overline{st}}(p,d,s); \\ sr_i \leftarrow [\![\mathsf{case}(e_c)\ \overline{\langle e_i, \overline{st_i}\rangle}\ \overline{st_d}]\!]_{st}(p,d,s); \\ \left\langle \begin{array}{l} sr_0.1 \uplus \{[\![e_c]\!]_e(p,s) \equiv [\![e_0]\!]_e(p,s)\}\ sr_i.1, \\ sr_0.2 \uplus \{[\![e_c]\!]_e(p,s) \equiv [\![e_0]\!]_e(p,s)\}\ sr_i.2 \end{array} \right\rangle \end{array} \right\} \\
[\![@(ee)\ \overline{st}]\!]_{st}(p,d,s) & = & \left\langle \begin{array}{l} ([\![\overline{st}]\!]_{\overline{st}}(p,d,s)).1 \uplus \{[\![ee]\!]_{ee}(p,s)\}\ [], \\ ([\![\overline{st}]\!]_{\overline{st}}(p,d,s)).2 \uplus \{[\![ee]\!]_{ee}(p,s)\}\ [] \end{array} \right\rangle \\
\\
[\![\cdot]\!]_{\overline{st}} & : & \mathcal{P} * \mathcal{D} * \mathcal{S} \to \mathsf{Fail}\ (\mathcal{S}_{\mathrm{u}} * \mathcal{R}_{\mathrm{u}}) \\
[\![[]]\!]_{\overline{st}}(p,d,s) & = & \langle [], [] \rangle \\
[\![st_0 :: \overline{st_i}]\!]_{\overline{st}}(p,d,s) & = & \left\{ \begin{array}{l} sr_0 \leftarrow [\![st_0]\!]_{st}(p,d,s); \\ sr_i \leftarrow [\![\overline{\langle st_i \rangle}]\!]_{st}(p,d,s \uplus sr_0.1); \\ \langle sr_i.1, sr_0.2 \uplus sr_i.2 \rangle \end{array} \right\}
\end{array}
$$

Fig. 7. Formal semantics for statements

time – it is highly impractical to use both edges, to say the least. In this context, we employ a workaround by assuming that the clock signal is set to 1 and does not toggle. For a given event, we only evaluate the expression part (i.e., e in posedge e or negedge e), which now serves as a clock-gating condition to determine whether the corresponding statement should be executed or not. As part of our future work, we aim to develop a more rigorous approach to reasoning about various clock uses.

3.2.4 Semantics for statements. Figure 7 describes formal semantics for statements. Denotation of a statement ($[\![\cdot]\!]_{st}$) is a function that takes the current position ($p \in \mathcal{P}$), declarations evaluated so far ($d \in \mathcal{D}$), and the current state ($s \in \mathcal{S}$), returning the updates for wires ($\in \mathcal{S}_{\mathrm{u}}$) and registers ($\in \mathcal{R}_{\mathrm{u}}$) or fails to evaluate. It is worth detailing each case in this denotation.

For $[\![lv = e]\!]_{st}$ and $[\![lv <= e]\!]_{st}$: a blocking assignment updates a wire value, while a nonblocking assignment updates a register. $[p := v]$ constructs an HMap singleton, where v is attached at the last of a path p. For example, $[[\mathsf{id}_a;\ \mathsf{id}_b] := v]$ constructs an HMap $[\langle \mathsf{id}_b, [\langle \mathsf{id}_a, v\rangle_{\mathrm{str}}]\rangle_{\mathrm{str}}]$.

For $[\![\mathsf{if}\ (e)\ \mathsf{then}\ \overline{st_t}\ \mathsf{else}\ \overline{st_f}]\!]_{st}$: the most intuitive approach to denoting this if-then-else statement would be first to evaluate e and to evaluate either $\overline{st_t}$ or $\overline{st_f}$ based on the value of e, formally:

$$[\![\mathsf{if}\ (e)\ \mathsf{then}\ \overline{st_t}\ \mathsf{else}\ \overline{st_f}]\!]_{st}(p,d,s) = \mathsf{if}\ ([\![e]\!]_e(p,s))\ \mathsf{then}\ [\![\overline{st_t}]\!]_{\overline{st}}(p,d,s)\ \mathsf{else}\ [\![\overline{st_f}]\!]_{\overline{st}}(p,d,s).$$

When constructing a per-logic transition function, this denotation becomes problematic, particularly when e involves nondeterministic values (e.g., module inputs), since the evaluation does not proceed further if the value of e is not evaluated. To overcome this problem, we have adopted the notion called "predicated updates," where each update carries a predicate so that the update happens only if the predicate holds. $(h_1 \uplus\{p\}\ h_2)$ denotes a predicated update; for example:

$$[\langle \mathsf{id}_a, v_a\rangle_{\mathrm{str}}] \uplus\{p\}\ [\langle \mathsf{id}_b, v_b\rangle_{\mathrm{str}}] = [\langle \mathsf{id}_a, \mathsf{if}\ (p)\ \mathsf{then}\ v_a\ \mathsf{else}\ []\rangle_{\mathrm{str}};\ \langle \mathsf{id}_b, \mathsf{if}\ (p)\ \mathsf{then}\ []\ \mathsf{else}\ v_b\rangle_{\mathrm{str}}].$$

Here, we see that the predicate p is applied to each ID, and thus the resulting update map is defined per-logic.

$$
\begin{array}{rcl}
\overline{M} \vdash [\![\cdot]\!]_{bl} & : & \mathcal{P} * \mathcal{D} * \mathcal{S} \to \mathsf{Fail}\ (\mathcal{D}_\mathrm{u} * \mathcal{S}_\mathrm{u} * \mathcal{R}_\mathrm{u}) \\
\overline{M} \vdash [\![\mathsf{logic\ id}]\!]_{bl}(p,d,s) & = & \langle [p := \langle \mathsf{id}, [] \rangle], [], [] \rangle \\
\overline{M} \vdash [\![\mathsf{assign}\ \langle lv, e \rangle]\!]_{bl}(p,d,s) & = & \langle [], [[\![lv]\!]_{lv}(p,d,s) := [\![e]\!]_e(p,s)], [] \rangle \\
\overline{M} \vdash [\![\mathsf{always}\ \overline{st}]\!]_{bl}(p,d,s) & = & sr \leftarrow [\![\overline{st}]\!]_{\overline{st}}(p,d,s);\ \langle [], sr.1, sr.2 \rangle; \\
\multicolumn{3}{r}{\overline{M} \vdash [\![\langle \mathsf{id}_m, \mathsf{id}_i, \overline{\langle \mathsf{id}_p, e_p \rangle}, \overline{\langle \mathsf{id}_i, e_i \rangle}, \overline{\langle \mathsf{id}_o, lv_o \rangle} \rangle_{\mathsf{module}}]\!]_{bl}(p,d,s) =} \\
\multicolumn{3}{r}{\left\{ \begin{array}{l}
s_p \leftarrow \uplus_p [\mathsf{id}_p := [\![e_p]\!]_e(p,s)]; \\
s_i \leftarrow \uplus_i [\mathsf{id}_i := [\![e_i]\!]_e(p,s)]; \\
rs \leftarrow [\![\overline{M}[\mathsf{id}_m]]\!]_M(s_p \uplus s_i, s\lfloor \mathsf{id}_i :: p \rfloor_\mathrm{v}); \\
s_o \leftarrow \uplus_o [[\![lv_o]\!]_{lv}(p,d,s) := rs.2\lfloor \mathsf{id}_o :: [] \rfloor_\mathrm{v}]; \\
\langle [], s_o, [(\mathsf{id}_i :: p) := rs.1] \rangle
\end{array} \right\}}
\end{array}
$$

Fig. 8. Formal semantics for blocks

For $[\![\mathsf{case}(e_c)\ \overline{\langle e_i, \overline{st_i} \rangle}\ \overline{st_d}]\!]_{st}$: a case statement can be regarded as multiple if-then-else statements. Therefore, predicated updates are also used here to evaluate the statement, where each predicate compares a target expression (e_c) with each case item (e_i).

For $[\![@(ee)\ \overline{st}]\!]_{st}$: an event-control statement can be also regarded as an if-then-else statement without the "else" part. A predicated update is used here as well, based on the evaluation of the event ee.

The denotation of a statement is easily extended to the one for a sequence of statements, presented also in Figure 7. Denotation of a statement sequence ($[\![\cdot]\!]_{\overline{st}}$) is a function that has the same type as $[\![\cdot]\!]_{st}$. For the empty sequence, there are no updates for wires or registers. For a nonempty sequence ($st_0 :: \overline{st_i}$), we first evaluate the updates by st_0 (resulting in sr_0 in the figure). When evaluating the updates by $\overline{st_i}$, the denotation takes ($s \uplus sr_0.1$) as the current state. The update-merge operator ($\uplus$) merges two HMaps as updates as if the left-hand-side update happens first; an appropriate merge algorithm will be applied, if the two HMaps contain the update values to the same path.

3.2.5 Semantics for blocks. Figure 8 describes formal semantics for blocks. Denotation of a block ($[\![\cdot]\!]_{bl}$) is a function that takes the same arguments (p, d, s) as the one for statements and returns the updates for declarations ($\in \mathcal{D}_\mathrm{u}$), wires ($\in \mathcal{S}_\mathrm{u}$), and registers ($\in \mathcal{R}_\mathrm{u}$) or fails to evaluate. The denotation particularly assumes a global argument $\overline{M}$ (i.e., $\overline{M} \vdash [\![\cdot]\!]_{bl}$), a finite map from a module ID to its body, used when denoting a module instance. While the other denotations are quite straightforward, the most complex case is indeed the denotation of a module instance:

(1) We begin by collecting all the parameter and input values to the instance (resulting in s_p and s_i in the figure, respectively).
(2) We then obtain the module body ($\overline{M}[\mathsf{id}_m]$) and denote the module using $[\![\cdot]\!]_M$ that will be defined soon. The denotation of a module takes two HMap states: one for the parameter/input values and the other for the current register values. In this case, $s_p \uplus s_i$ forms the former, and the latter is obtained from the current state ($s\lfloor \mathsf{id}_i :: p \rfloor_\mathrm{v}$). The return value is the pair of register updates and output values, resulting in rs in the denotation.
(3) The next step is to connect the output values to corresponding wires as updates, resulting in s_o in the denotation.
(4) Finally, we construct the denotation for the module instance: there are no new declarations ([]), the only wire updates are from the outputs (s_o), and the only register updates are the ones in the module instance ($[(\mathsf{id}_i :: p) := rs.1]$).

3.2.6 Semantics for generate blocks. Figure 9 describes formal semantics for generate blocks. Denotation of a generate block ($[\![\cdot]\!]_{gb}$) is a function that takes the same arguments (p, d, s) as the one for blocks and returns the updates for declarations ($\in \mathcal{D}_\mathrm{u}$), wires ($\in \mathcal{S}_\mathrm{u}$), and registers ($\in \mathcal{R}_\mathrm{u}$).

$$
\begin{array}{rcl}
[\![\cdot]\!]_{gb} & : & \mathcal{P} * \mathcal{D} * \mathcal{S} \to \mathcal{D}_{\mathrm{u}} * \mathcal{S}_{\mathrm{u}} * \mathcal{R}_{\mathrm{u}} \\
[\![bl]\!]_{gb}(p,d,s) & = & [\![bl]\!]_{bl}(p,d,s) \text{ or } \langle [],[],[]\rangle \text{ if fail} \\
[\![\mathsf{if}\ (e_c)\ \mathsf{then}\ \mathsf{id}_t : gb_t\ \mathsf{else}\ \mathsf{id}_f : gb_f]\!]_{gb}(p,d,s) & = & \left\{ \begin{array}{l} \mathsf{if}\ ([\![e_c]\!]_e(p,s)) \\ \mathsf{then}\ [\![gb_t]\!]_{gb}(\mathsf{id}_t :: p,d,s) \\ \mathsf{else}\ [\![gb_f]\!]_{gb}(\mathsf{id}_f :: p,d,s) \end{array} \right\} \\
\\
[\![\cdot]\!]_{\overline{gb}} & : & \mathcal{P} * \mathcal{D} * \mathcal{S} \to \mathcal{D} * \mathcal{S} * \mathcal{R}_{\mathrm{u}} \\
[\![[]]\!]_{\overline{gb}}(p,d,s) & = & \langle d,s,[]\rangle \\
[\![gb_0 :: \overline{gb_i}]\!]_{\overline{gb}}(p,d,s) & = & \left\{ \begin{array}{l} dsr_0 := [\![gb_0]\!]_{gb}(p,d,s); \\ dsr_i := [\![\overline{gb_i}]\!]_{\overline{gb}}(p, d \Subset dsr_0.1, s \Subset dsr_0.2); \\ \langle dsr_i.1, dsr_i.2, dsr_0.3 \Subset dsr_i.3\rangle \end{array} \right\}
\end{array}
$$

Fig. 9. Formal semantics for generate blocks

It is worth noting that generate block is the first syntax level where the denotation returns a non-monadic value, i.e., it is just a regular function.

For a block *bl* (as a generate block), the denotation is simply uncovering the monad; if the block denotation fails, it returns "no updates" ($\langle [], [], [] \rangle$). For a conditional generate block (if (e_c) then $\mathsf{id}_t : gb_t$ else $\mathsf{id}_f : gb_f$), we first evaluate the condition expression e_c and denote either gb_t or gb_f based on the value of e_c. Also, the current position p should be extended to either $(\mathsf{id}_t :: p)$ or $(\mathsf{id}_f :: p)$ as the position indeed moves into the sub-generate block. It is important to note that the predicated update is not used in this case, since the condition e_c in a generate block must be a constant, thus deterministic.

Next, we extend the denotation of a generate block to the one for a sequence of generate blocks, described also in Figure 9. Denotation of a generate-block sequence ($[\![\cdot]\!]_{\overline{gb}}$) is a function that takes the same arguments (p, d, s) as $[\![\cdot]\!]_{gb}$. On the other hand, $[\![\cdot]\!]_{\overline{gb}}$ returns *the next states* for declarations and wires (not just the updates), and the update for registers. In this sense, $[\![[]]\!]_{\overline{gb}}(p,d,s)$ just returns d and s from the arguments and no updates for registers ($[]$). It is quite straightforward to denote $(gb_0 :: \overline{gb_i})$ by denoting gb_0 and $\overline{gb_i}$ and merging the states appropriately, as shown in the figure. The left-merge operator ($\Subset$) is used in this case, where any value in the left-hand side is preserved if the right-hand side also has a value to the same path. In other words, if a module has two blocks updating the same state values, the latter one is ignored. If the module is well-formed, we would not have such a case. We have an additional purpose for employing this operation within our verification framework, which will be elaborated in Section 4.3.

3.2.7 Semantics for modules. Finally, we provide formal semantics for modules, presented in Figure 10. Denotation of a module *through a single iteration* ($[\![\cdot]\!]_m$) is a function that takes the declarations and states evaluated so far ($d \in \mathcal{D}$ and $s \in \mathcal{S}$), and returns the next declarations/states ($\in \mathcal{D} * \mathcal{S}$) and the register updates ($\in \mathcal{R}_{\mathrm{u}}$). For a given module (module $\langle \mathsf{id}_m, \overline{\langle \mathsf{id}_p, e_p \rangle}, \overline{\mathsf{id}_i}, \overline{\mathsf{id}_o}, \overline{gb} \rangle$), we first generate declarations for parameters/inputs (d_p and d_i, respectively) and states for parameters (s_p). We then proceed to evaluate $[\![\overline{gb}]\!]_{\overline{gb}}$, starting from the top position (i.e., $[]$), using the merged declarations between the one given as an argument (d) and the other one for parameters/inputs ($d_p \uplus d_i$), and the merged states between s and s_p. In essence, this denotation aims to derive additional declarations/states from the given ones. From this perspective, we can consider a fixpoint denotation ($[\![\cdot]\!]_m^{\mathrm{lfp}}$), which iteratively derives as many declarations/states as possible.

The denotation of a module in its ultimate form ($[\![\cdot]\!]_M$) takes the input values (s_i) as the starting state and the current register values (s_r), and returns register updates ($\in \mathcal{R}_{\mathrm{u}}$) and output values

$$
\begin{array}{rcl}
[\![\cdot]\!]_m & : & \mathcal{D} * \mathcal{S} \to \mathcal{D} * \mathcal{S} * \mathcal{R}_{\mathrm{u}} \\
[\![\mathsf{module}\ \langle \mathsf{id}_m, \overline{\langle \mathsf{id}_p, e_p \rangle}, \overline{\mathsf{id}_i}, \overline{\mathsf{id}_o}, \overline{gb} \rangle]\!]_m(d, s) & = & \left\{ \begin{array}{l} d_p \leftarrow \uplus_p[\mathsf{id}_p := []]; \\ d_i \leftarrow \uplus_i[\mathsf{id}_i := []]; \\ s_p \leftarrow \uplus_p[\mathsf{id}_p := [\![e_p]\!]_e([], [])]; \\ \overline{[\![gb]\!]_{gb}}([], d \Subset (d_p \uplus d_i), s \Subset s_p) \end{array} \right\} \\
[\![\cdot]\!]_m^{\mathsf{lfp}} & : & \mathcal{D} * \mathcal{S} \to \mathcal{D} * \mathcal{S} * \mathcal{R}_{\mathrm{u}} \\
[\![m]\!]_m^{\mathsf{lfp}}(d, s) & = & \mathsf{lfp}_{(d,s)}\ [\![m]\!]_m(d, s) \\
[\![\cdot]\!]_M & : & \mathcal{S} * \mathcal{R} \to \mathcal{R}_{\mathrm{u}} * \mathcal{S} \\
[\![m]\!]_M(s_i, s_r) & = & \left\{ \begin{array}{l} dsr \leftarrow [\![m]\!]_m^{\mathsf{lfp}}([], s_i \uplus s_r); \\ \langle dsr.3, (dsr.2|_{m.\overline{\mathsf{id}_o}}) \rangle \end{array} \right\}
\end{array}
$$

Fig. 10. Formal semantics for modules

($\Subset \mathcal{S}$). The fixpoint denotation $[\![\cdot]\!]_m^{\mathsf{lfp}}$ is used to derive the next states and the register updates (dsr), and the return values are constructed accordingly. Note that in order to obtain output values, we filter the next states using the output IDs, represented as $(dsr.2|_{m.\overline{\mathsf{id}_o}})$.

4 A VERIFICATION FRAMEWORK EMBEDDED IN COQ

In this section, we introduce our verification framework designed for formally verifying Verilog modules. Our framework is prototyped in the Coq proof assistant and is closely associated with the formal semantics outlined in Section 3. To begin, we establish state transition systems for Verilog modules. We then proceed to introduce our verification methodology, which involves utilizing cuts (Section 4.3) and performing pre-evaluation of transition functions (Section 4.4).

4.1 State Transition Systems

To begin, we introduce the fundamental concept of state transition systems.

Definition 4.1. *For a given state type S, a* **state transition system** *(STS) of S, denoted as $\mathsf{STS}[S]$, consists of 1) a set of initial states $S_0 \subseteq S$ and 2) a state-transition relation $(\rightarrow_S) : S \to S \to \mathbb{P}$. ($\mathbb{P}$ denotes propositions.) We will just use $(\rightarrow)$ if S is clear from context. For states s_1 and s_2, we will call $(s_1 \rightarrow s_2)$ a (transition) step from s_1 to s_2.*

Based on the definition of STS, we define a number of conventional terms. For states s_1 and s_n we call $(s_1 \rightarrow^* s_n)$ *steps* iff $s_1 = s_n$ or there exist intermediate states $s_2, \cdots, s_{n-1}$ such that $(s_i \rightarrow s_{i+1})$ for $i = 1, \cdots, (n-1)$. For a state s, we say s is reachable iff there is a state $s_0 \in S_0$ such that $(s_0 \rightarrow^* s)$. $S_{\mathsf{rch}} \subseteq S$ denotes the set of reachable states.

A set $S_I \subseteq S$ is called an *invariant* iff $S_{\mathsf{rch}} \subseteq S_I$, denoted as $\mathsf{STS}[S] \sqsubseteq S_I$. By definition, in order to prove that a set S_I is an invariant, it suffices to prove 1) $S_0 \subseteq S_I$ and 2) $\forall s_i, s_{i+1}.\ s_i \in S_I \rightarrow (s_i \rightarrow_S s_{i+1}) \rightarrow s_{i+1} \in S_I$.

Subsequently, we extend STS to define a labeled transition system.

Definition 4.2. *For a given state type S and label type L, a* **labeled transition system** *(LTS) of S and L, denoted as $\mathsf{LTS}[S, L]$, consists of 1) a set of initial states $S_0 \subseteq S$ and 2) a labeled transition relation $(\xrightarrow[S]{L}) : S \to L \to S \to \mathbb{P}$. For states s_1 and s_2 and a label l, $(s_1 \xrightarrow{l} s_2)$ denotes the relation.*

An LTS is by definition an STS, where the transition step can be defined as $(s_1 \rightarrow_S s_2) \triangleq \exists l \in L.\ s_1 \xrightarrow{l} s_2$. Therefore, it is fair to apply all the notions established for STS to LTS, e.g., reachability, invariants, etc.

We proceed to define our simulation relation [Brookes and Rounds 1983], the correctness notion between an implementation and a specification.

Definition 4.3 (Simulation). *For given state types S^i and S^s (for an implementation and a specification, respectively), a label type L, and a label-validity function $\mathcal{L} : L \to \mathbb{B}$, we call $(\sim) : S^i \to S^s \to \mathbb{P}$ a* **simulation** *relation between the implementation and the spec iff the following conditions hold:*

(1) $\forall s_0^i \in S_0^i, s_0^s \in S_0^s.\ s_0^i \sim s_0^s$.

(2) $\forall s_1^i, s_2^i \in S^i, l \in L.\ s_1^i \xrightarrow{l} s_2^i \to$
$\forall s_1^s.\ s_1^i \sim s_1^s \to$
$(\mathcal{L}(l) = \mathit{false} \wedge s_2^i \sim s_1^s) \vee (\exists s_2^s.\ s_1^s \xrightarrow{l} s_2^s \wedge s_2^i \sim s_2^s \wedge \mathcal{L}(l) = \mathit{true})$.

We override the relation symbol $(\sim)$ for the simulation among STSes, e.g., $STS[S^i] \sim STS[S^s]$.

In order to encompass various hardware behaviors, we incorporated the *label-validity function* $\mathcal{L}$ into the simulation relation. This function explicitly indicates the validity of labels at each state-transition step. When designing Verilog modules, it is common to generate validity signals for output signals, as the outputs may not always be valid. The label-validity function captures these signals and determines the validity of output labels, allowing the specification to simulate only the valid ones. Consequently, it is essential to thoroughly examine both the label-validity function and the simulation relation to ensure a sound definition of correctness. The application of our simulation will be demonstrated in the case-study section (Section 5).

4.2 Transition Systems for Verilog

In order to set up a transition system for a given Verilog module within our framework, users are required to provide certain configurations, outlined as follows:

- The clock ID: as explained in Section 3.2, our formal semantics supports only a single clock domain. The framework needs to know the unique clock ID (given as an input signal) used across all modules.
- The reset ID: the framework also supports only a single reset signal, used in all modules.
- Negative or positive reset: it is necessary to know when the reset is asserted.

For simplicity and convenience (but without loss of generality), henceforth, we will assume that the clock ID is designated as clk (thus clk is always asserted to 1, as explained in Section 3.2) and the reset ID is rst_n as a negative reset, i.e., the module state should be reset whenever rst_n = 0. Also, note that the framework does not consider the case where the module is reset in the middle of its run.

4.2.1 The reset state. Based on the module semantics defined in Section 3.2, the denotation of a module inherently contains the reset logic. In practice, Verilog modules take rst_n as an input and define the reset value for each register within its corresponding "always" block. For instance, if we evaluate the denotation ($[\![\cdot]\!]_M$) for the module shown in Figure 1, we will see the register updates similar to the following form:

$$[\langle \mathsf{r}, \mathsf{if}\ (\mathsf{not}\ ([\![\mathsf{rst\_n}]\!]_e([], s_i)))\ \mathsf{then}\ 0\ \mathsf{else}\ \cdots\rangle_{\mathrm{str}};\ \cdots],$$

where s_i contains the input values including the one for rst_n. Therefore, if we put rst_n = 0 to the module denotation, we obtain the reset state. This process is formally defined as follows:

Definition 4.4 (The pre-reset state). *The* **pre-reset state**, *denoted as $s_{(-1)}$, is an HMap just containing the input value 0 for rst_n (1 for the positive reset rst), i.e., $s_{(-1)} \triangleq [\langle rst\_n, 0\rangle_{str}]$.*

Definition 4.5 (The reset state). *For a given module m, the* **reset state** s_0^m *is an HMap containing all the register-reset values defined in the module, derived from the pre-reset state and the module denotation:* $s_0^m \triangleq (\llbracket m \rrbracket_M(s_{(-1)}, []))$.1.

Note that it is not necessary to provide any input values except rst_n, since the reset values should be constants that are not dependent to the inputs. We provide [] for the current register state, as there are no register values before the reset.

In contrast to the reset state, in order to have a regular transition function, we need to ensure that rst_n is 1. We implement it by providing $(s_i \uplus [\langle \text{rst\_n}, 1 \rangle_{\text{str}}])$ as the input values to the module, instead of directly using the current state s_i as it is.

Definition 4.6 (The transition function for a Verilog module). *For a given module m,* $\mathcal{T}_m : \mathcal{S} * \mathcal{R} \to \mathcal{R} * \mathcal{S}$ *is called* **the transition function** *defined as:*

$$\mathcal{T}_m(s_i, s_r) \triangleq rs := \llbracket m \rrbracket_M(s_i \uplus [\langle rst\_n, 1 \rangle_{str}], s_r);\ \langle s_r \uplus rs.1, rs.2 \rangle.$$

Note that we merge the current register state (s_r) with the updates ($rs.1$) to obtain the next state.

4.2.2 Transition systems for a Verilog module. We are now prepared to build transition systems for a Verilog module, beginning with the STS.

Definition 4.7 (STS for a Verilog module). *A state transition system for a Verilog module m (denoted as STS*$[m]$*) is composed of the followings:*

- *The initial state* $\mathcal{R}_0 \triangleq \{s_0^m\}$
- *The state-transition relation* $(s_1 \to_{\mathcal{R}} s_2) \triangleq \exists s_i, s_o.\ \mathcal{T}_m(s_i, s_1) = \langle s_2, s_o \rangle$

Note that in this definition the state-transition relation is nondeterministic regarding input/output values that are not explicitly disclosed within the relation. This nondeterminism indicates that any invariants should be proven considering arbitrary input values.

The LTS of a Verilog module, on the other hand, involves input/output values that are disclosed as labels.

Definition 4.8 (LTS for a Verilog module). *A labeled transition system for a Verilog module m (denoted as LTS*$[m]$*) with a label type* $(\mathcal{S} * \mathcal{S})$ *is composed of the followings:*

- *The initial state* $\mathcal{R}_0 \triangleq \{s_0^m\}$
- *The state-transition relation* $(s_1 \xrightarrow{l} s_2) \triangleq \mathcal{T}_m(l.1, s_1) = \langle s_2, l.2 \rangle$

4.3 Verification with Abstraction

When proving an invariant or simulation, sometimes it is more preferable to forget assignments to specific wires (e.g., if those wires are not relevant to the invariant at all) or to abstract the assignments by imposing constraints on the wire values to satisfy desired properties. This abstraction becomes essential when the wires are assigned with large combinational-logic circuits, since our denotational semantics inlines each assignment to other interconnected wires/registers. We have previously demonstrated this inlining of wires in the example shown in Figure 1. Assuming that the value of `u` involves a complex logic that cannot be reduced to a constant, the register `r` will also depend on such intricate logic. In this case, it would be beneficial to abstract this complex logic to enhance proof-engineering performance.

In this section, we provide *wire abstraction* as our solution to this problem. The high-level intuition is to treat the target wire like a module input with a free variable, i.e., an arbitrary value. Our verification framework formally provides this abstraction mechanism; we begin by providing the formal definition of *cuts*:

$$
\begin{array}{l}
\{\ \mathcal{S}[u] = v_u;\ \} \\
\rightarrow \{\ \mathcal{S}[u] = v_u;\ \mathcal{S}[s] = !v_u;\ \} \\
\rightarrow \{\ \mathcal{S}[u] = v_u;\ \mathcal{S}[s] = !v_u;\ \mathcal{S}[t] = \mathcal{R}[r]\ ||\ !v_u;\ \} \\
\rightarrow \left\{ \begin{array}{l} \mathcal{S}[u] = v_u;\ \mathcal{S}[s] = !v_u;\ \mathcal{S}[t] = \mathcal{R}[r]\ ||\ !v_u; \\ \mathcal{R}_{\mathsf{next}}[r] = \mathsf{if}\ (v_u)\ \mathsf{then}\ (\mathcal{R}[r]\ ||\ !v_u)\ \mathsf{else}\ \mathcal{R}[r]; \end{array} \right\}
\end{array}
$$

Fig. 11. Denotation steps with a cut to u in the example from Figure 1

Definition 4.9 (Cuts)**.** *For a given Verilog module m, we call a state $s_c^\circ \in \mathcal{S}$ consisting of (ID, value) pairs (i.e., $s_c^\circ = \overline{\langle id_c, v_c \rangle_{str}}$)* **cuts***, where each ID originally represents an internal wire of m. $m|^{s_c^\circ}$ denotes the module that the cuts s_c° are applied to m. The transition function for $m|^{s_c^\circ}$ is defined by applying the cuts to $\mathcal{T}_m$ as inputs, i.e., $\mathcal{T}_{m|^{s_c^\circ}}(s_i, s_r) \triangleq \mathcal{T}_m(s_c^\circ \uplus s_i, s_r)$.*

Figure 11 shows the denotation steps of the example introduced in Figure 1 with a cut $s_c^\circ = [\langle \mathsf{u}, v_u \rangle_{\mathsf{str}}]$ being applied to the transition function. Instead of the original value of u (which was 1 in Figure 1), a free variable v_u is now assigned to u. Note that u will never be updated by its original assignment, since once the wire value is set, no denotation of an assignment can overwrite it (by the left-merge operator $\Subset$ introduced in Figure 9). After u obtains the value v_u, it is propagated to all its uses. When v_u is replaced with the original value of u, we retrieve the original transition function, formally stated in the following definition and lemma:

Definition 4.10 (Cut assignments)**.** *For a given module m and cuts $s_c^\circ = \overline{\langle id_c, v_c \rangle_{str}}$, we call the state $s_c^\bullet(s_i, s_r) \in \mathcal{S}$* **cut assignments***, derived by replacing each v_c in s_c° with the original assigned value of id_c. The original value can be obtained from $s_s \triangleq (\llbracket m \rrbracket_m^{lfp}([], s_i \uplus s_r)).2$, which contains all the current wire and register values, hence by $s_s[id_c]$. We will just use $s_c^\bullet$ if s_i and s_r are clear from context.*

Lemma 4.1. *For a given module m and any cut assignments $s_c^\bullet$, the following equation holds:*

$$\forall s_i, s_r.\ \mathcal{T}_m(s_c^\bullet(s_i, s_r) \uplus s_i, s_r) = \mathcal{T}_m(s_i, s_r).$$

Proof. Straightforward by performing inductions through all syntax levels, namely from expressions to generate blocks. ∎

The above lemma provides a justification for employing cuts in invariant proofs. Considering the STS for $m|^{s_c^\circ}$, denoted as $\mathsf{STS}[m|^{s_c^\circ}]$, it has the reset state same as $\mathsf{STS}[m]$ but the transition function becomes $\mathcal{T}_{m|^{s_c^\circ}}(s_i, s_r) = \mathcal{T}_m(s_c^\circ \uplus s_i, s_r)$. Exploiting this fact, we can prove a theorem that any invariant in $\mathsf{STS}[m|^{s_c^\circ}]$ also holds in $\mathsf{STS}[m]$:

Theorem 4.2 (Invariant proof with cuts)**.** *For a given module m, let s_c° be cuts constrained by $\mathcal{P}_c : \mathcal{S} \rightarrow \mathbb{P}$. If an invariant $S_I \subseteq \mathcal{S}$ holds in $STS[m|^{s_c^\circ}]$ and the cut assignments $s_c^\bullet$ satisfies the constraint (for any reachable state s_r), then the invariant also holds in $STS[m]$, formally:*

$$(\mathcal{P}_c(s_c^\circ) \rightarrow STS[m|^{s_c^\circ}] \sqsubseteq S_I) \rightarrow (s_r \in \mathcal{R}_{rch} \rightarrow \mathcal{P}_c(s_c^\bullet(s_i, s_r))) \rightarrow STS[m] \sqsubseteq S_I.$$

Proof. Since the invariant S_I holds in $\mathsf{STS}[m|^{s_c^\circ}]$, S_I still holds with $\mathsf{STS}[m|^{s_c^\bullet}]$ since $s_c^\bullet$ meets the constraint for any reachable states (i.e., $\mathcal{P}_c(s_c^\bullet)$). By Theorem 4.1, the transition function in $\mathsf{STS}[m|^{s_c^\bullet}]$ equals to the one in $\mathsf{STS}[m]$. Therefore, S_I also holds in $\mathsf{STS}[m]$. ∎

Cuts can be used in simulation proofs as well, following a similar approach justified by Theorem 4.2 for invariants:

Theorem 4.3 (Simulation proof with cuts)**.** *For an implementation module m^i and a specification module m^s, let $s_c^{\circ i}$ and $s_c^{\circ s}$ be cuts of m^i and m^s, respectively, constrained by $\mathcal{P}_c : \mathcal{S} \rightarrow \mathcal{S} \rightarrow \mathbb{P}$. If a*

simulation ($\sim$) *holds between* $STS[m^i|^{s_c^{\circ i}}]$ *and* $STS[m^s|^{s_c^{\circ s}}]$ *and the two cut assignments* $s_c^{\bullet i}$ *and* $s_c^{\bullet s}$ *satisfy the constraint, then the simulation also holds between* $STS[m^i]$ *and* $STS[m^s]$*, formally:*

$$(\mathcal{P}_c(s_c^{\circ i}, s_c^{\circ s}) \rightarrow STS[m^i|^{s_c^{\circ i}}] \sim STS[m^s|^{s_c^{\circ s}}]) \rightarrow (s_r^i \sim s_r^s \rightarrow \mathcal{P}_c(s_c^{\bullet i}, s_c^{\bullet s})) \rightarrow STS[m^i] \sim STS[m^s].$$

In our case study (Section 5), we will demonstrate the extensive application of cuts in both invariant and simulation proofs.

4.4 (Pre-)Evaluation of transition functions

The transition function we established from the denotation of a module is not fully usable yet. When we initially developed the transition function, our goal was to ensure its evaluation is as fast as possible. However, if the function still contains the entire module body, it takes longer to evaluate the function each time we pass arguments to it.

$$\mathcal{F}(s) = [\![\text{a} \parallel \text{b}]\!]_e([], s) \quad \rightsquigarrow \quad s[a] \;||_{\text{h}}\; s[b] \quad \text{(after pre-evaluation)}$$

The example above illustrates this problem. Rather than directly using the original transition function $\mathcal{F}$ that still contains an expression (a || b), we would like to pre-evaluate the function as much as possible, without applying the argument s. Consequently, the resulting function will not contain any syntactic elements but will solely involve the manipulation of values (represented as $||_{\text{h}}$ below, denoting the logical OR operation for HMaps).

The need for this process arises from the fact that a function is *not evaluated until* its arguments are applied. At least in Coq, the only way to evaluate such a function is by applying what are known as *reduction tactics* to simplify the function. In our framework, we particularly used a faster reduction tactic called `vm_compute` [Grégoire and Leroy 2002], enabling us to obtain evaluated functions much more quickly. We will show the experimental results of this pre-evaluation approach in Section 6.

5 A CASE STUDY: PROVING A PIPELINED PROCESSOR

In this section, we present the correctness proof of a pipelined processor as a case study, conducted using our verification framework presented in Section 4. Both the processor implementation and specification are both described in Verilog. We formally proved the correctness by showing a simulation between the processor and its spec. To expedite the simulation proof, we effectively employed several cuts introduced in Section 4.3.

We conducted this case study to demonstrate the effectiveness of our proposed formal semantics in verifying hardware designs that incorporate nontrivial optimizations. Finding an existing processor for this purpose was challenging, as establishing a precise specification for the processor itself is nontrivial even before providing the correctness proof. For instance, if a processor allows self-modification or relaxed memory models, a simple single-cycle processor would not suffice as the specification. While there has been active research in establishing proper specifications, we acknowledge that it falls beyond the scope of this paper. Moreover, to substantiate the efficiency of our semantics, we aimed to use a Verilog processor whose design closely resembles the one already proven in Coq. Considering these circumstances, we made the decision to develop our own processor.

5.1 The Processor Implementation

Our processor implements a subset of the RV32I ISA [Waterman and Asanović 2019], a base integer instruction set of the RISC-V ISA. While the processor faithfully implements all the integer/memory operations in RV32I, it does not support the following instructions: `FENCE` (memory model) and `ECALL/EBREAK` (environment call and breakpoints).

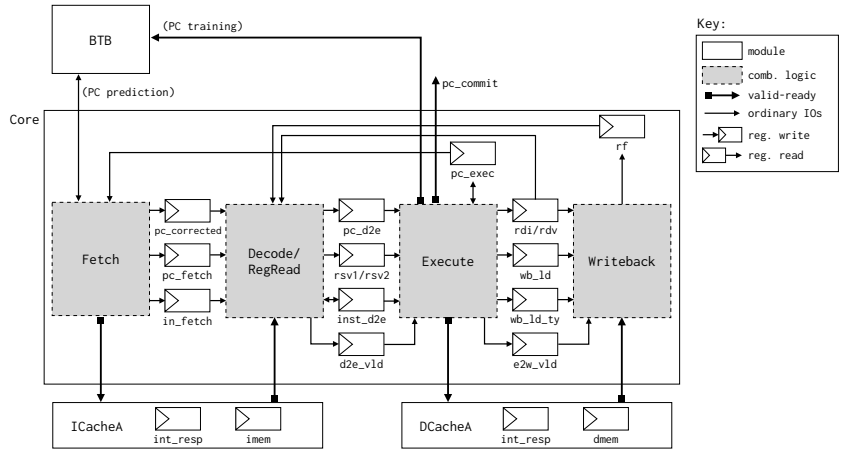

Fig. 12. The processor implementation

The processor is outlined in Figure 12. It is composed of four Verilog modules, namely `Core`, `BTB`, `ICacheA`, and `DCacheA`, which consist of ~500 lines of code. `Core` is the main pipeline, interacting with the atomic instruction cache `ICacheA` and data cache `DCacheA`, each of which is defined as a Verilog module. The pipeline consists of four stages – Fetch, Decode, Execute, and Writeback – containing combinational-logic blocks (represented as gray boxes in the figure). Between any two stages, there are pipeline registers that transfer various data from the previous stage to the next stage. Below, we introduce important design aspects for each module and the pipeline.

5.1.1 The atomic instruction/data caches. `Core` is connected with the instruction cache and data cache to fetch instructions and to load/store data, respectively. The caches do not respond to any request instantaneously; these caches are considered *atomic*, since they do not allow any additional requests while processing a specific request until the corresponding response comes out. In our case study, we implemented these two caches (`ICacheA` and `DCacheA`) in a conventional way. In addition to the data array (`imem` or `dmem`), each cache has a register called `int_resp` that temporarily stores the response data. The requestor receives the data whenever it sends a "ready" indication. This implementation is commonly used, as many FPGA synthesis tools convert this design into an atomic memory called a Block RAM (BRAM) in FPGA. Regarding the IO interface for each cache, request and response interfaces implement the part of the AXI protocol [Axi 2013], and thus each interface follows the conventional valid-ready protocol.

5.1.2 The Fetch stage. The Fetch stage "fetches" instructions by reading the PC value (`pc_fetch`) and sends a read request to `ICacheA` (outside of `Core`) using the PC value as the address. Once `ICacheA` responds with the corresponding instruction, the next stage (`Decode`) will process it accordingly. When `Fetch` makes a request, it sets the register `in_fetch` to 1 to indicate that the read request is in progress. `Fetch` does not make any additional requests if `in_fetch` is 1 and `Decode` is not ready to perform with the instruction response. `in_fetch` will be set to 0 as soon as the Decode stage receives the instruction.

In the Fetch stage, determining the exact next PC is not possible, since the instruction is before its decode/execution. Therefore, a *branch predictor* predicts the next PC by collecting information from later stages. One well-known branch predictor is a branch target buffer (BTB) [Perleberg and Smith 1993], which maintains a small buffer to store current and next PC pairs. We have faithfully implemented the BTB module and attached it to the Fetch stage. If the PC prediction is found to

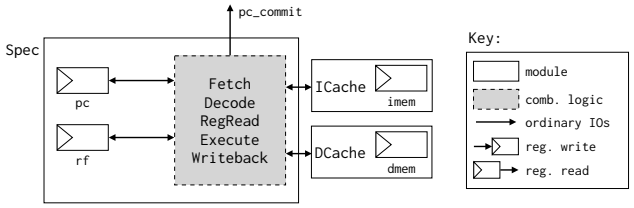

Fig. 13. The processor specification

be incorrect, which is determined in the Execute stage, `Fetch` is indicated from `Execute` that the prediction is wrong along with the correct PC value.

5.1.3 The Decode stage. The Decode stage is responsible for "decoding" the instruction fetched from the preceding Fetch stage. `Decode` extracts information from the instruction, such as the instruction type (`opcode`), the indices of the two source and destination registers (`rs1`, `rs2`, and `rd`). The decoded instruction is then stored in the register `inst_d2e`, allowing the next stage `Execute` to read and execute it. `Decode` also forwards the PC of the decoded instruction (`pc_d2e`) from `Fetch`.

In addition to decoding instructions, `Decode` also reads the values of the source registers (`rs1` and `rs2`) from the register file (`rf`). However, there is a possibility that the older instructions – in the Execute or Writeback stage – have not yet performed their potential writes to the register file. This situation is known as a *data hazard*, which is a well-known challenge in designing a pipelined processor, and there are several solutions. In our processor implementation, we either 1) bypass the register value from `rdi/rdv` (before the Writeback stage) to resolve read-after-write (RAW) hazards or 2) stall if a load is requested to `DCacheA` or the destination-register index in `inst_d2e` conflicts with the reads.

5.1.4 The Execute stage. The Execute stage "executes" the decoded instruction along with the corresponding source-register values. To begin with, `Execute` checks if the data from `Decode` is correct in terms of PC. It maintains a register `pc_exec`, which always holds the correct PC for execution, and thus compares `pc_exec` with `pc_d2e` to determine whether to proceed with the execution or not. If the two PCs are different, `Execute` indicates `Fetch` to correct the PC and discards all the data received from `Decode`.

During the execution phase, for regular bit-arithmetic instructions, `Execute` calculates the destination-register value and forwards the result to the Writeback stage (`rdi/rdv`). For control-transfer instructions (jumps or branches), it calculates the next PC value and updates `pc_exec`. For memory-request instructions, it sends the appropriate request to `DCacheA`. For load requests, it sets `wb_ld` to 1 to let the Writeback stage receive the data and write it to the register file. It also sets `wb_ld_ty` properly to distinguish between different load types (`LW`, `LH`, or `LB`).

Upon successful execution, `Execute` generates a module-level output called `pc_commit`, which contains the PC value of the instruction that was just executed. The processor specification also has the same output, and thus these outputs become part of the simulation proof. Section 5.3 will discuss the proof in detail.

5.1.5 The Writeback stage. The final stage, `Writeback`, is responsible for writing data back to the register file `rf`. It handles two cases: one involving the executed value from `Execute` and the other involving the load value from `DCacheA`. The logic `wb_ld` determines which case the write operation corresponds to.

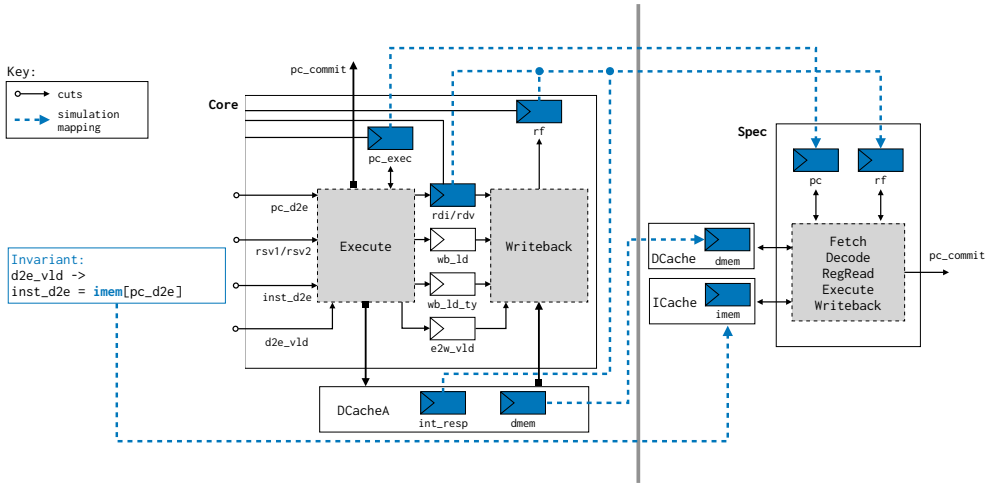

Fig. 14. The simulation relation between the processor implementation and specification

5.2 The Processor Specification

Figure 13 shows the specification of our processor implementation. In contrast, the spec is a single Verilog module `Spec` that does not involve any pipelines, comprising ~300 lines of code. Each instruction is executed *per-cycle*, and both the register file and the data cache are updated immediately at the same cycle. This communication occurs with the *instantaneous* caches, denoted as `ICache` and `DCache` in the figure. Unlike the atomic caches, these caches do not have any registers to store responses. `Spec` also generates an output `pc_commit` every time it executes an instruction.

5.3 The Simulation Proof

In this section, we present the formal proof of simulation between the implementation (Section 5.1) and the specification (Section 5.2). We begin by constructing two LTSs for the implementation and the spec based on the definition introduced in Section 4.2. The simulation relation is then defined between these two LTSs, which is presented in Section 4.1. The simulation proof involves a number of invariants and incorporates several cuts introduced in Section 4.3. We will first explain our intuition of the proof, followed by an introduction to the invariants and cuts.

5.3.1 The high-level intuition. The most crucial aspect of a simulation proof is obviously the simulation relation itself. For a given implementation and a specification, there can exist various relations that correctly satisfy the simulation, indicating that the correctness proof is not necessarily unique. That said, the challenges involved in the proofs can vary depending on the chosen relation.

In hardware design, the state space of an implementation is typically larger than the one of its spec. In other words, the implementation contains more registers than the spec, with the extra registers being used for optimizations. For instance, in our case study, the spec has a total of four registers: PC (`pc`), register file (`rf`), instruction cache (`imem`), and data cache (`dmem`). Conversely, the implementation has more registers. When looking into the pipeline, there are several intermediate registers between any consecutive stages to facilitate data transfer, e.g., `inst_d2e`, `wb_ld`, etc. In this sense, the initial task in proving simulation is to establish a mapping between the implementation and the spec. Typically, the mapping assigns several registers of the implementation to a single register in the spec.

In determining the simulation for our processor implementation and spec, a crucial factor lies in defining which PC/instruction we designate as the "current" ones, so they can be mapped to

their counterparts in the spec. The implementation is pipelined with four stages, leading to three potential PCs and instructions located after each stage. Figure 14 illustrates our simulation relation. As seen in the figure, we chose the ones around the Execute stage – `pc_exec` for the PC and `inst_d2e` for the instruction. The intuition behind this decision is that the Execute stage should execute an instruction *only if* `pc_d2e` matches `pc_exec` (i.e., PC prediction is correct), indicating that `inst_d2e` is the instruction fetched from the instruction cache with `pc_exec` as the memory address.

5.3.2 Simulation proof with invariants and cuts. Now we present the detailed simulation relation and its proof. As explained in the previous section, among several candidates, our simulation relates the PC and instruction at the Execute stage to their counterparts in the spec. It is worth noting that `Execute` makes an output `pc_commit` with `pc_commit_vld = 1` only if the PC conveyed from the Decode stage (`pc_d2e`) matches the one maintained in `Execute` (`pc_exec`), indicating that it is correct to execute with the instruction and register-read values from `Decode`. Hence, we define our label mapping in a way that the spec takes a transition step only if `pc_commit_vld = 1`.

Formally, the simulation relation $(\sim) : \mathcal{S} \to \mathcal{S} \to \mathbb{P}$ and the label mapping $\mathcal{L} : \mathcal{S} * \mathcal{S} \to$ option $(\mathcal{S} * \mathcal{S})$ (from Definition 4.8) are defined as follows:

$$
\begin{array}{lcl}
s^{\mathsf{i}} \sim s^{\mathsf{s}} & \triangleq & s^{\mathsf{s}}[\mathsf{pc}] = s^{\mathsf{i}}[\mathsf{pc\_exec}] \wedge \\
& & s^{\mathsf{s}}[\mathsf{rf}] = \ \text{if } (s^{\mathsf{i}}[\mathsf{e2w\_vld}]) \\
& & \quad \text{then } s^{\mathsf{i}}[\mathsf{rf}][s^{\mathsf{i}}[\mathsf{rdi}] := \ \text{if } (s^{\mathsf{i}}[\mathsf{wb\_ld}]) \\
& & \qquad \text{then } \mathit{get\_ld\_val}\,(s^{\mathsf{i}}[\mathsf{wb\_ld\_ty}],\ s^{\mathsf{i}}[\mathsf{DCacheA}][\mathsf{int\_resp}]) \\
& & \qquad \text{else } s^{\mathsf{i}}[\mathsf{rdv}]] \\
& & \quad \text{else } s^{\mathsf{i}}[\mathsf{rf}] \wedge \\
& & s^{\mathsf{s}}[\mathsf{ICache}][\mathsf{imem}] = s^{\mathsf{i}}[\mathsf{ICacheA}][\mathsf{imem}] \wedge \\
& & s^{\mathsf{s}}[\mathsf{DCache}][\mathsf{dmem}] = s^{\mathsf{s}}[\mathsf{DCacheA}][\mathsf{dmem}], \\
\mathcal{L}(s_i, s_o) & \triangleq & \text{if } (s_o^{\mathsf{i}}[\mathsf{pc\_commit\_vld}]) \\
& & \text{then Some } (s_i, [\mathsf{pc\_commit} := s_o^{\mathsf{i}}[\mathsf{pc\_commit}]]) \\
& & \text{else None.}
\end{array}
$$

We examine each state mapping one-by-one, considering the label mapping as well:

- PC: if `pc_commit_vld` is 0, `pc_exec` remains unchanged since no execution happens; the spec does not make a step as well, matching the label mapping to None. If `pc_commit_vld = 1`, it indicates that the execution has occurred. In this case, `pc_exec` gets updated to the next PC value, based on the decoded instruction `inst_d2e` and the register-read values `rsv1` and `rsv2`. The spec also takes a step and updates `pc`. *Assuming* that the spec updates `pc` based on the same decoded instruction and register-read values used in the implementation, we can easily prove the simulation for PC.
- Register file: this is the most complicated part of the simulation, since in the implementation, the register file is updated *not* in the Execute stage *but in Writeback*. Therefore, we should *speculate* the next register-file value from the other registers. As defined, the speculation is basically to reflect what Writeback would do for the update, using the registers conveyed from Execute, e.g., `rdi/rdv`, `wb_ld`, etc. A function *get_ld_val* is employed to calculate the load value based on the load type.
- Instruction cache: the two instruction memories are equal from their resets and never update.
- Data cache: the reasoning is similar to that of the PC. The data cache is only updated when executing a store instruction. Since the data memory `dmem` in `DCacheA` is updated immediately, the subsequent data-memory values should be still equal.

As mentioned above, during the simulation proof, we made the assumption that the Execute stage uses the same decoded instruction and register-read values as the spec when executing an instruction. This assumption provides a strong motivation to define cuts (with constraints) and to

Proof-check time (sec)	Preprocessing	Invariants	Simulation	Total
The Kami processor	181.29	896.03	841.16	1918.48
Our processor	15.98	76.27	65.41	157.66

Fig. 15. Evaluation of proof checks between the Kami processor and ours (in seconds)

utilize them in the simulation proof, as described in Theorem 4.3. Specifically, as shown in Figure 14, we declare cuts for `inst_d2e`, `rsv1`, and `rsv2` in the implementation with constraints asserting that these values match the corresponding ones in the spec, formally:

$$
\begin{aligned}
s^{\mathsf{i}} \sim_c s^{\mathsf{s}} \quad \triangleq \quad & s^{\mathsf{i}}[\mathsf{d2e\_vld}] = 0 \lor \\
& s^{\mathsf{i}}[\mathsf{pc\_d2e}] \neq s^{\mathsf{i}}[\mathsf{pc\_exec}] \lor \\
& \left(\begin{array}{l} s^{\mathsf{i}}[\mathsf{inst\_d2e}] = s^{\mathsf{s}}[\mathsf{inst\_cur}] \land \\ s^{\mathsf{i}}[\mathsf{rsv1}] = s^{\mathsf{s}}[\mathsf{rsv1}] \ \land s^{\mathsf{i}}[\mathsf{rsv2}] = s^{\mathsf{s}}[\mathsf{rsv2}] \end{array} \right)
\end{aligned}
$$

where s^{i} and s^{s} contain the registers and cut wires of the implementation and the spec, respectively. Note that the equalities of instructions and register-read values apply only when `d2e_vld = 1` and `pc_d2e = pc_exec`. We also declared cuts for `inst_cur`, `rsv1`, and `rsv2` in the spec to have access for these internal wires. Using these cuts and constraints, the simulation can be proven without knowing any logics in the previous stages (Fetch and Decode) in the implementation.

Subsequently, we should prove that the above constraint holds with cut assignments, as required in Theorem 4.3. The constraint proof is outlined in Figure 14 as well. To begin, we need to prove an invariant that `inst_d2e` is the instruction of `pc_d2e`, which can be proven easily:

$$\forall s^{\mathsf{i}}.s^{\mathsf{i}} \in S_{\mathsf{rch}} \longrightarrow s^{\mathsf{i}}[\mathsf{d2e\_vld}] = 1 \longrightarrow s^{\mathsf{i}}[\mathsf{inst\_d2e}] = s^{\mathsf{i}}[\mathsf{ICacheA}][\mathsf{imem}][\mathsf{pc\_d2e}].$$

By using this invariant and the original simulation definition, we can prove the constraints. For instance, the equality of the instructions can be proven as follows:

$$
\begin{aligned}
s^{\mathsf{i}}[\mathsf{inst\_d2e}] &= s^{\mathsf{i}}[\mathsf{ICacheA}][\mathsf{imem}][\mathsf{pc\_d2e}] && \text{(by the invariant)} \\
&= s^{\mathsf{i}}[\mathsf{ICacheA}][\mathsf{imem}][\mathsf{pc\_exec}] && \text{(by the constraint precondition)} \\
&= s^{\mathsf{s}}[\mathsf{ICache}][\mathsf{imem}][\mathsf{pc}] && \text{(by the simulation relation)} \\
&= s^{\mathsf{s}}[\mathsf{inst\_cur}]. && \text{(by the wire definition in the spec)}
\end{aligned}
$$

The equalities for register-read values (`rsv1`/`rsv2`) are proven in a similar manner.

6 EVALUATION AND DISCUSSION

Proof-engineering performance evaluation. In order to demonstrate the effectiveness of our semantics and verification framework in terms of proof-engineering performance, we conducted a comparison between the performance of our case study and the Kami processor [Choi et al. 2017; Erbsen et al. 2021]. The Kami processor is described in the Kami language, a rule-based high-level HDL. The formal semantics of Kami is defined in an operational way, meaning that state transitions are represented *relationally*. The Kami processor closely resembles our case-study processor, as both are pipelined with four stages and include a branch target buffer (BTB) connected to the Fetch stage. In terms of data hazard, the Kami processor is less optimized than ours; it only supports stall, lacking value bypassing. Lastly, the Kami processor also assumes complete separation between the instruction and data caches, with the instruction cache remaining unchanged.

Although we acknowledge that comparing our processor with Kami may not be entirely fair due to the differences in language and verification processes, we believe it is still valuable to make this comparison for two reasons. Firstly, the Kami processor design is still very similar to ours. Secondly, the relational formal semantics of Kami, which we claimed as a burden in deductive verification, makes it worthwhile to include in the comparison.

Figure 15 presents our evaluation results. We performed the experiment with Coq 8.16.1, 2.6 GHz Intel Core i7 CPU, and 16 GB RAM. The results clearly show that it takes much shorter time (~12.2× faster) to machine-check the corectness proof of our processor than Kami. We include the column "Preprocessing" in the experiment, which counts time for inlining [Choi et al. 2017] in Kami and pre-evaluation of transition functions for ours.

Relation between the standard and our semantics. As observed in the preceding sections, it appears that our proposed semantics significantly deviates from the standard Verilog semantics [Sys 2018]. It can be argued that we should justify our semantics by demonstrating its equivalence to the standard. That said, we claim that proving this equivalence is not worth the effort, as the current standard itself exhibits a flaw in its semantics [Lööw 2022]. Specifically, conflicting definitions exist within the standard, allowing a scenario where a process is triggered by an update event generated by an intermediate update value in a combinational-logic block, but ignores the last update event, thus *failing to read the final value*. We believe that this violation contradicts the desired behavior of combinational logics. This issue has been acknowledged over a decade, and various approaches [Lööw 2022; Meredith et al. 2010] have been proposed during that time. However, the latest version of the standard apparently has not yet addressed this problem. Considering these circumstances, we currently do not see the value in establishing a formal connection between our semantics and the standard.

7 RELATED WORK

Formal semantics of Verilog. Most of the previous approaches to establishing formal semantics of Verilog have focused on formalizing the scheduling semantics described in the standard [Sys 2018]. In this tradition, a couple of approaches [Zhu et al. 2006, 2011] have developed both operational and denotational semantics from what is called algebraic semantics as a basis, though they only targeted a very limited subset of Verilog, e.g., without considering wire assignments.

One notable achievement involved the K framework to define executable operational semantics based on rewriting logic [Meredith et al. 2010], evidently the most comprehensive formal semantics so far. However, it is worth noting that this work lacks any verification cases, and to the best of our knowledge, no subsequent verification efforts have been made. Therefore, it is difficult to assess whether their semantics is practically applicable for formal verification purposes.

As described in Section 6, there have been identified semantic issues related to nondeterminism in the scheduling semantics. Recent successes have primarily concentrated on addressing these problems [Lööw 2021; Lööw 2022; Lööw and Myreen 2019], achieving a rigorous operational semantics. Their semantics has been defined formally in the HOL4 interactive theorem prover, and also used to develop a verified compiler from Verilog to FPGA logic. It is worth emphasizing again that the problems tackled in the aforementioned approaches are separate from our own achievements in terms of designing efficient formal semantics.

Deductive verification with low-level HDLs. There have not been a lot of approaches to the deductive verification of hardware described in Verilog. A converter called VeriCoq [Bidmeshki and Makris 2015] from Verilog to Coq has been developed to implement proof-carrying hardware IPs (PCHIPs). VeriCoq supports module-level conversion, where the resulting Coq code is allegedly the equivalent representation of the source Verilog module. That said, since the converter is not formally verified, one cannot ensure the Verilog module and the Coq representation are indeed equivalent. Furthermore, their conversion supports a very limited subset of Verilog, e.g., "generate" blocks and functions/tasks are not supported, which are employed obviously a lot in practice.

The most recent, notable achievement is the verified stacks between assembly programs and processor execution [Lööw et al. 2019]. They have developed the Silver ISA, a general-purpose

RISC ISA connected to the backend of CakeML [Myreen and Owens 2012], and the Silver processor described in Verilog, executing Silver assembly programs. They also formally proved the correctness of the programs described in Silver ISA and the one for the Silver processor in the HOL theorem prover, and went so far as to connect the two correctness proofs to have the verified stack. Since the Silver processor is described in Verilog, the formal semantics of was required in order for verification; they employed the one already defined in their previous work [Lööw and Myreen 2019]. That said, the processor is quite simple in that it is not pipelined, executes instructions in-order, and thus is similar to the specification of the ISA.

Several approaches have developed their own low-level HDLs (not Verilog) and verification platforms, embedded in high-level languages. Lava [Bjesse et al. 1998] is a tool to specify, design, and verify hardware in Haskell. Esterel [Schneider 2001] is a programming language for reactive systems also used to specify and design hardware, later embedded into HOL. Π-Ware [Flor and Swierstra 2015] is a HDL and its formal semantics embedded in Agda. The three languages introduced above have not been equipped with enough abstraction to make the design and verification scalable; as far as we see, no case studies have been performed for these languages where multiple modules are involved.

We would like to emphasize that our work is indeed distinctive in terms of taking a hardware design with nontrivial optimizations (a pipelined processor) written in Verilog as-is, and formally proving the correctness effectively through the use of our formal semantics and verification tools optimized for deductive verification.

Deductive verification with high-level HDLs. There have been continuous, successful approaches to design and verify hardware with rule-based high-level HDLs [Bourgeat et al. 2020; Braibant and Chlipala 2013; Choi et al. 2017; Pit-Claudel et al. 2021]. These rule-based languages are based on Bluespec SystemVerilog [Nikhil 2004]; rules are executed atomically (by guarded atomic actions), and thus the execution satisfies what is known as one-rule-at-a-time semantics. The languages and their formal semantics are embedded in Coq; on top of this level of abstraction, the correctness of multiprocessor and cache-coherent memory subsystem has been formally proven [Choi et al. 2017], and a compiler from the rule-based HDL to RTL circuitry has been verified as well [Pit-Claudel et al. 2021].

One notable recent achievement in this line of work is PDL [Zagieboylo et al. 2022], a high-level HDL designed specifically for constructing and verifying pipelined processors. This language supports various ingredients to design pipelined processors, while the compiler (from PDL to BSV) guarantees the one-instruction-at-a-time semantics (higher abstraction than "one rule at a time") for the target designs. For each compilation, the correctness of the pipeline is verified by the Z3 SMT solver. They also conducted various case studies, including a series of RISC-V pipelined processors.

8 CONCLUSION AND FUTURE WORK

We have designed a formal semantics of Verilog that can be effectively used in deductive verification. By designing a denotational semantics, we establish a state-transition function for the module, which reduces the burden for deducing transition relations in verification. In order to demonstrate the benefits of the transition function, we have prototyped a verification framework embedded in the Coq proof assistant and introduced a useful verification methodology using wire cuts. Utilizing this framework, we formally verified a RISC-V pipelined processor over the single-cycle specification as a case study, demonstrating the practicality of our formal semantics and framework.

Our current focus for future work lies in expanding our semantics to handle fine-grained clock event controls and diverse clock domains, including asynchronous clocks. As described in Section 3, our syntax and semantics are limited to a single clock domain with basic clock gating functionality.

REFERENCES

2013. *AMBA AXI3 and AXI4 Protocol Specification.* Technical Report. ARM. https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification

2018. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018), 1–1315. https://doi.org/10.1109/IEEESTD.2018.8299595

Mohammad-Mahdi Bidmeshki and Yiorgos Makris. 2015. VeriCoq: A Verilog-to-Coq converter for proof-carrying hardware automation. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. 29–32. https://doi.org/10.1109/ISCAS.2015.7168562

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '98)*. ACM, New York, NY, USA, 174–184. https://doi.org/10.1145/289423.289440

Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. 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* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 243–257. https://doi.org/10.1145/3385412.3385965

Thomas Braibant and Adam Chlipala. 2013. Formal Verification of Hardware Synthesis. In *CAV 2013, 25th International Conference on Computer Aided Verification (Lecture Notes in Computer Science, Vol. 8044)*. Springer, 213–228. http://gallium.inria.fr/~braibant/fe-si/

Stephen D. Brookes and William C. Rounds. 1983. Behavioural equivalence relations induced by programming logics. In *Automata, Languages and Programming*, Josep Diaz (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 97–108.

Joonwon Choi, Adam Chlipala, and Arvind. 2022. Hemiola: A DSL and Verification Tools to Guide Design and Proof of Hierarchical Cache-Coherence Protocols. In *Computer Aided Verification*, Sharon Shoham and Yakir Vizel (Eds.). Springer International Publishing, Cham, 317–339.

Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 24 (aug 2017), 30 pages. https://doi.org/10.1145/3110268

Edmund M. Clarke and E. Allen Emerson. 1981. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*. Springer-Verlag, Berlin, Heidelberg, 52–71.

E. M. Clarke, E. A. Emerson, and A. P. Sistla. 1986. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.* 8, 2 (April 1986), 244–263. https://doi.org/10.1145/5397.5399

Clifford Cummings. 2000. Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill! *SNUG (Synopsys Users Group) 2000 User Papers* (2000).

Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. 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* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 604–619. https://doi.org/10.1145/3453483.3454065

Joao Paulo Pizani Flor and Wouter Swierstra. 2015. Π-Ware: An Embedded Hardware Description Language using Dependent Types. *TYPES 2015* (2015), 67.

Benjamin Grégoire and Xavier Leroy. 2002. A Compiled Implementation of Strong Reduction. *SIGPLAN Not.* 37, 9 (sep 2002), 235–246. https://doi.org/10.1145/583852.581501

Andreas Lööw. 2021. Lutsig: A Verified Verilog Compiler for Verified Circuit Development. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Virtual, Denmark) *(CPP 2021)*. Association for Computing Machinery, New York, NY, USA, 46–60. https://doi.org/10.1145/3437992.3439916

Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. 2019. Verified Compilation on a Verified Processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1041–1053. https://doi.org/10.1145/3314221.3314622

Andreas Lööw. 2022. A small, but important, concurrency problem in Verilog's semantics? (Work in progress). In *2022 20th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. 1–6. https://doi.org/10.1109/MEMOCODE57689.2022.9954591

Andreas Lööw and Magnus O. Myreen. 2019. A Proof-Producing Translator for Verilog Development in HOL. In *2019 IEEE/ACM 7th International Conference on Formal Methods in Software Engineering (FormaliSE)*. 99–108. https://doi.org/10.1109/FormaliSE.2019.00020

Kenneth L. McMillan. 1993. *Symbolic Model Checking.* Kluwer Academic Publishers, USA.

Patrick Meredith, Michael Katelman, José Meseguer, and Grigore Roşu. 2010. A formal executable semantics of Verilog. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*. 179–188. https://doi.org/10.1109/MEMCOD.2010.5558634

Magnus O. Myreen and Scott Owens. 2012. Proof-Producing Synthesis of ML from Higher-Order Logic. In *International Conference on Functional Programming (ICFP)*. ACM Press, 115–126. https://doi.org/10.1145/2364527.2364545

R. Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* 69–70. https://doi.org/10.1109/MEMCOD.2004.1459818

C.H. Perleberg and A.J. Smith. 1993. Branch target buffer design and optimization. *IEEE Trans. Comput.* 42, 4 (1993), 396–412. https://doi.org/10.1109/12.214687

Clément Pit-Claudel, Thomas Bourgeat, Stella Lau, Arvind, and Adam Chlipala. 2021. Effective Simulation and Debugging for a High-Level Hardware Language Using Software Compilers. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 789–803. https://doi.org/10.1145/3445814.3446720

Klaus Schneider. 2001. *A Verified Hardware Synthesis of Esterel Programs.* Springer US, Boston, MA, 205–214. https://doi.org/10.1007/978-0-387-35409-5_20

Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. 2015. Modular Deductive Verification of Multiprocessor Hardware Designs. In *Proc. CAV*.

Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213.* Technical Report. RISC-V Foundation.

Drew Zagieboylo, Charles Sherk, Gookwon Edward Suh, and Andrew C. Myers. 2022. PDL: A High-Level Hardware Design Language for Pipelined Processors. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 719–732. https://doi.org/10.1145/3519939.3523455

Huibiao Zhu, Jifeng He, and J. Bowen. 2006. From algebraic semantics to denotational semantics for Verilog. In *11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'06)*. 13 pp.–. https://doi.org/10.1109/ICECCS.2006.1690363

Han Zhu, Huibiao Zhu, Si Liu, and Jian Guo. 2011. Towards Denotational Semantics for Verilog in PVS. In *2011 Fifth International Conference on Secure Software Integration and Reliability Improvement - Companion.* 1–2. https://doi.org/10.1109/SSIRI-C.2011.10