A Proof-Producing Translator for Verilog Development in HOL

Andreas Lööw
Chalmers University of Technology
Gothenburg, Sweden

Magnus O. Myreen
Chalmers University of Technology
Gothenburg, Sweden

***Abstract*—We present an automatic proof-producing translator targeting the hardware description language Verilog. The tool takes a circuit represented as a HOL function as input, translates the input function to a Verilog program and automatically proves a correspondence theorem between the input function and the output Verilog program ensuring that the translation is correct. As illustrated in the paper, the generated correspondence theorems furthermore enable transporting circuit reasoning from the HOL level to the Verilog level. We also present a formal semantics for the subset of Verilog targeted by the translator, which we have developed in parallel with the translator. The semantics is based on the official Verilog standard and is, unlike previous formalization efforts, designed to be usable for automated and interactive reasoning without sacrificing a clear correspondence to the standard. To illustrate the translator's applicability, we describe case studies of a simple verified processor and verified regexp matchers and synthesize them for two FPGA boards. The development has been carried out in the HOL4 theorem prover.**

I. Introduction

When building fully-verified systems, so called *verified stacks* [1]–[3], software and hardware cannot be treated as separate, disconnected worlds. For a system to be fully correct, it is not enough that its software and the hardware constituents are correct considered independently of each other: the software and hardware components must also be integrated correctly. To be able to state and prove all-encompassing system correctness claims, the software, hardware, and all integrations between them must be made available inside the same formal system.

In a recent paper [4], we present a methodology to construct verified stacks inside the HOL4 interactive theorem prover [5]. For the software side of the stacks, we rely on (and extend) the CakeML project [6]. For the hardware side, we introduce a new hardware development methodology. In this paper, we provide the technical details of this hardware development methodology.

Formal verification already has an established position in (industrial) hardware development, but only in a limited sense. The dominating formal verification approach consists of relying on so-called automatic verification tools, offering e.g. model checking and equivalence checking. Such tools, in any form resembling the current state-of-the-art,

- cannot handle the intricacies of *reasoning inside an explicitly stated formal semantics for an industry-relevant hardware description language* (HDL), such as Verilog or VHDL, and consequently do not formally relate their guarantees to such semantics,
- nor can they handle *complex full-system specifications*; instead, system-externally justified simplifications, translations, and decompositions must be introduced to make the complexity manageable for the tools.

The alternative approach of interactive theorem proving (ITP) has the potential, we claim, to overcome the limitations listed above. This would be because, inside ITP systems, such as HOL4, automatic and manual proofs can be combined in logically safe ways. The ITP approach has been taken many times before (see Sec. VIII), and there has been previous work on formalizing the semantics of mainstream low-level HDLs, and previous work on formally verifying the correctness of concrete, or parameterized, circuits, inside ITP systems. However, to the best of our knowledge, *there has not been prior work combining these two efforts*, i.e. verification of non-trivial circuits with respect to a detailed explicitly stated formal semantics for one such description language.

Our hardware development methodology is designed to make hardware verification easy in HOL, and at the same time, support a solid connection to a formal semantics of a subset of Verilog. The heart of the approach is *an automatic proof-producing tool* that, given a functional version of a Verilog program in HOL, produces Verilog code and proves a *correspondence theorem* stating that the Verilog code and the functional code have the same behavior according to a formal semantics of Verilog we have developed. Furthermore, the correspondence theorem enables transporting system correctness results for the functional code to the Verilog code, as illustrated in Sec. II.

This paper makes the following contributions:

- We elaborate on our previously published *hardware development methodology* that is centered around functional versions of Verilog programs in HOL. (Sec. II, III, IX)
- We present the details of our *automatic tool* that makes the methodology have a solid connection to our explicitly defined operational semantics for Verilog. (Sec. VI)
- The description of our *formal semantics for a subset of Verilog* is also a contribution. The semantics is carefully carved out to be faithful to the Verilog standard, manageable in complexity for HOL proofs, and sufficiently large to express interesting synthesizable hardware. (Sec. V)

All source code for this work can be found at `https://github.com/CakeML/hardware`.

II. Example

This section illustrates the ideas behind our hardware development methodology through an example. Subsequent sections provide technical details.

The first step in our methodology is to embed (express) the circuit-to-be-verified inside HOL. To do this, the user must first define a new state record type containing the variables the circuit is to operate over. If the circuit is to interact with the external world, a second state record representing world states must be defined also. The user must then define the circuit in terms of next-state functions operating over these two state records. For the purposes of our example, consider the following HOL function AB as a model of a circuit:

```
A fext s =
  if fext.pulse then s with count := s.count + 1w else s
B s = if 10w <+ s.count then s with done := T else s
AB fext init 0 = init
AB fext init (Suc n) = let s' = AB fext init n in
  ⟪count := (A (fext n) s').count; done := (B s').done⟫
```

The HOL syntax s *with* $x := y$ means setting field x to y in record s, $\langle\!| \ldots |\!\rangle$ constructs a new record instance, and `<num>`w is notation for constant words. The AB circuit is a combination of two circuits, A and B. The A circuit checks for an external pulse signal: A adds one to count whenever pulse is detected. The parallel circuit B assigns true to done whenever count is larger than 10. The combined AB function takes three arguments: a function *fext* from time (a natural number) to the user-defined external-world state record, modeling the circuit-external world; *init*, an instance of the user-defined circuit-internal state record; and a third argument specifying the number of clock cycles to evaluate the circuit for.

The second step in our methodology is to run the circuit through our proof-producing translation tool. For AB, the tool generates an in-logic Verilog AST ABv representing a Verilog module consisting of two processes corresponding to A and B:

```
always_ff @ (posedge clk) // A
 if (pulse) count <= count + 8'd1;

always_ff @ (posedge clk) // B
 if (8'd10 < count) done = 1;
```

The tool is proof-producing, meaning that each run of the tool automatically proves a *correspondence theorem* stating that the generated Verilog code ABv behaves the same as the input function AB – thereby guaranteeing the correctness of the translation. Crucially, these correspondence theorems moreover allow us to transport properties proved about AB (by any means inside HOL) to the generated Verilog code ABv *without any manual reasoning involving Verilog semantics*.

The third step in our methodology is proving and transporting circuit properties. For our running example of the AB circuit, we prove a simple property as follows. If we assume that the input pulse is true infinitely often,

$$\mathsf{pulse\_spec}\ \mathit{fext} \iff \forall n.\ \exists m.\ (\mathit{fext}\ (n+m)).\mathsf{pulse},$$

then we can easily prove that done will eventually be set to true:

$$\vdash \mathsf{pulse\_spec}\ \mathit{fext} \Rightarrow \exists n.\ (\mathsf{AB}\ \mathit{fext}\ \mathit{init}\ n).\mathsf{done}.$$

Properties proved of HOL functions such as AB can easily be transported to properties of its generated Verilog code thanks to the automatically proved correspondence theorems. For our running example, we can prove the following, (to repeat:) *without manual reasoning about the Verilog semantics*:

```
⊢ pulse_spec_verilog fext ∧ vars_has_type Γ ABtypes ⇒
  ∃ n Γ'.
    mrun fext ABv Γ n = Inr Γ' ∧
    mget_var Γ' "done" = Inr (VBool T).
```

Here mrun is the top-level Verilog semantics. The theorem states that there exists some number of clock cycles n such that the Verilog semantics successfully produces a state for the nth clock cycle, and in that state mget_var tells us that done is set to true, i.e. Verilog value VBool T.

For synthesis, we ship a Verilog pretty-printer that can be used to print the generated Verilog code ABv to file to be used (after adding the necessary module boilerplate code) as input for off-the-shelf synthesis toolchains that target (e.g.) FPGAs.

A. Larger examples

The example above was intentionally kept simple for ease of presentation. However, the methodology has been shown to work on larger examples. Sec. VII describes how we have applied it in our paper on verified stacks [4] to produce a verified implementation of a processor for execution of CakeML programs. In the same section we also describe how to build circuits that perform matching against regular expressions.

III. Hardware development methodology summary

Fig. 1 summarizes the flow the translator enables, from high-level specifications down to runnable FPGA application, as exemplified in the previous section.

We want to stress that just because we are working in HOL does not mean that our approach is an instance of high-level synthesis (HLS). As can be seen in the example from the previous section, the input and output languages are at the same (relatively low) abstraction level. Even when writing HOL circuits, we are still thinking in terms of hardware: we are thinking in terms of clock cycles and logical gates. From one perspective, a HOL circuit is just another HOL function (in other words, a functional program), expressed in a restricted subset. This perspective is what enables using HOL circuits for reasoning. But from another perspective, we have a circuit in *almost Verilog*, that can be turned to *actual Verilog* by our proof-producing translator.

IV. Overview

The following sections provide more technical detail: We will first describe the subset of Verilog we target and its semantics (Sec. V), followed by the internals of the translator (Sec. VI). We then explain how we have applied our methodology in two case studies (Sec. VII). Lastly, we discuss related work (Sec. VIII) and consider trusted base issues (Sec. IX).

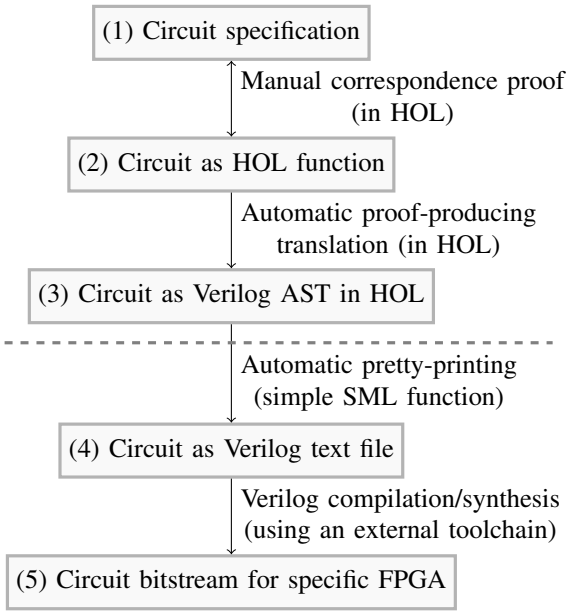

Fig. 1. An overview of our hardware development methodology. For non-trivial circuits, human-guided proofs are needed for the connection between layers 1 and 2, whereas the other steps are always automatic. The correspondences above the dotted line are proved functionally correct, and the correspondences below the dotted line are not covered by our formal development. The main focus of this paper is the translator connecting layers 2 and 3.

V. Verilog

In this section we describe the syntax and formal semantics of the subset of Verilog targeted by the translator. Verilog is a large language, the current standard [7] is more than 1300 pages long. But many parts of the standard are irrelevant for describing hardware: these, non-synthesizable, parts of the standard include, e.g., concepts used to express test benches and other testing infrastructure. We have not included all synthesizable constructs in our formalization; rather, the subset we target consists of synthesizable constructs needed in synthesizable behavioral process-based Verilog programs. Our aim is that this subset should be large enough to describe simple synchronous hardware. The subset has already been sufficient for the case studies we present in Sec. VII.

Our Verilog formalization is based on the standard's event-driven simulation semantics, which the standard defines in informal English prose. Two criteria have guided our formalization work. First and foremost, we wanted a *sound semantics*, in the sense that the functional correctness of a circuit in our semantics implies functional correctness with respect to the standard. Furthermore, it has been important to us that our semantics is *usable for reasoning inside an ITP*. We have addressed both of these concerns by keeping the semantics intentionally small, and have tried to only including well-understood constructs.

A. Abstraction level (Verilog as an output language)

Verilog is the exit representation used in our hardware methodology, meaning that Verilog is the communication medium used for interaction with external synthesis pipelines.

v	::=	`bool` | $[v]$	
op	::=	`+` | `*` | `<` | `&` | `>>` | ...	
e	::=	v	literal constant
	|	x	variable reference
	|	e`[`e`]`	indexing
	|	e`[`n`:`m`]`	slicing, for $n, m \in \mathbb{N}$
	|	$\neg e$	unary not
	|	e op e	binary operator
	|	e `?` e `:` e	ternary if
s	::=	s `;` s	sequential sequencing
	|	`if` e `then` s `else` s	if-statement
	|	`case` e [e `:` s] `endcase`	case-statement
	|	e = e	blocking assignment
	|	e <= e	non-blocking assignment
p	::=	[`always_ff @ (posedge clk)` s]	

Fig. 2. Verilog values v, expressions e, statements s, and programs p. Variable declarations are not included in the figure.

We target behavioral process-based Verilog, with concepts such as integer addition and multiplication available as primitive notions. We chose this level because it is important to not work at a too low level of abstraction when using Verilog as the input language for synthesis tools.

It is all too easy to fall into the trap of thinking that "real" hardware consists of "the standard gates", such as ANDs and XORs, and flip flops, and consequently that this should be our target representation. This is incorrect, as what real hardware consists of is a technology-dependent question. In the case of FPGAs, real hardware consists of LUTs, DSP blocks, BRAMs, etc., rather than some homogeneous collection of gates. For example, if we compile away the notion of addition by compiling to ANDs, XORs, etc. before handing of our circuits to a synthesis pipeline, the pipeline might fail to exploit special-purpose hardware constructs available for efficient addition computations (as noted by e.g. Beyer et al. [8]).

B. Subset of Verilog included

We will now discuss the Verilog constructs included in our semantics, and how they relate to the standard. Fig. 2 gives an overview of the constructs included.

Programs. In Verilog, programs consist of hierarchically connected modules. Each module has a set of input and output ports, which are used to connect the different modules together. Common top-level constructs inside a module beyond other instantiated modules include data declarations, procedural blocks and continuous assignments.

In our formalization, to restrict the scope of the initial version of our project, we consider a flattened module hierarchy (i.e., programs consist of a single module). This can be understood as saying that we do not (formally) consider code used to "glue" modules together. (Meredith et al. [9] take the same approach in their Verilog semantics in the K framework, see the discussion in Sec. VIII.) We did not find an immediate use for continuous assignments in our case studies as they did not include three-state buses and as all combinational logic could be placed inside procedural blocks, so we did

not include such assignments in the formalization. As we are only interested in single-clock domain synchronous hardware, all procedural blocks in our formalization are `always_ff` blocks waiting for a positive edge (`posedge`) from a program-common clock. Consequently, a Verilog program p (Fig. 2) in our formalization consists of a data declarations section and a list of `always_ff` blocks. We use the terms process and procedural block synonymously throughout.

Statements and expressions. In our formalization, statements s and expressions e (both in Fig. 2), the syntactical elements inside procedural blocks, consist, beyond non-blocking assignments, mostly of standard imperative-language constructs. The state update of a non-blocking assignment is not visible until the next clock cycle, and such assignments are used for communication between processes.

We only allow pure expressions, because the standard does not enforce an evaluation order, and we want our generated Verilog programs to be portable between Verilog toolchains. This means, e.g., that our expressions cannot contain assignments.

Variables and nets. In Verilog, there are two kinds of data objects, namely: variables and nets. Variables are included in our formalization as they are used for holding temporary values (in other words, wiring subcircuits), and describing registers capable of holding state between cycles. (Whether a variable corresponds to a register is up to the synthesizer in use to decide, i.e., nothing we have to concern ourselves with on our abstraction level [9].) Nets, however, are mainly useful for handling cases where there are multiple (continuous) drivers, which we are not interested in.

Values. Values v (Fig. 2) in our formalization consist of Booleans and nestable (balanced) arrays.

Boolean values. Verilog Booleans can take on four values: `0`, `1`, `X`, and `Z`. However, we can still use standard Booleans in our formalization. We do not include `Z` as a possible Boolean value because it is only used for nets (with multiple drivers), and nets are not covered in our formalization. The value `X` represents an unknown value. Such value might be useful in simulation-based testing, but for proving we do not need an explicit representation of "unknown", because inside ITPs we already have access to such concepts directly in the logic. For example, if we want to prove that a circuit is correct regardless of the initial value of some particular Boolean variable (as we did in the example in Sec. II), we simply quantify our theorem statement over all possible Boolean values for that variable.

Array values. We support packed arrays, both 1-dimensional and nested variants, in our formalization. Arrays are indexed as if all levels were declared as `logic[msb:lsb] arr` with $\text{msb} \geq 0$ and $\text{lsb} = 0$. The formalization includes modulo-arithmetic operations over 1-dimensional arrays. Arrays are represented as nested HOL lists in our formalization.

Array resizings. Verilog is well known for its many idiosyncrasies. We have tried to the extent possible to avoid including any obscure or complex constructs in our formalization, to minimize the risk of formalization bugs caused by us misunderstanding the Verilog standard. But for Verilog's idiosyncratic handling of array resizings and signed numbers, one is not left with much choice, as these concepts are implicitly part of every Verilog expression.

In particular, some Verilog expressions are context-determined, both with respect to size and signedness. The size and signedness of a context-determined expression are not decided just by the subexpressions of the expression, but also by the context the expression is part of. For example, for three Verilog arrays `a`, `b`, and `c`, where `a` and `b` are of length 16 and `c` of length 32, the addition in the expression `c = a + b` will be a 32-bit addition as `c` is considered part of the context of the addition. To limit the amount of context taken into consideration by context-determined expressions, one can nest expressions inside the concatenation operator, whose operands are self-determined: e.g., `c = { a + b }` expresses a 16-bit addition. As for signedness, i.e. (here) if `a` and `b` are zero-extended or sign-extended, `c` is not considered part of the context even in the concatenation-less expression. That is, if `a` and `b` are signed, then they will be sign-extended, regardless of the signedness of `c`. But if one of `a` and `b` is unsigned and the other signed, then both will be zero-extended.

Not only are context-determined expressions an obstacle to overcome when formalizing the language, they also make up an obstacle when translating to Verilog from strongly typed languages with explicit resizing annotations, such as the subset of HOL we are using to describe circuits. Such resizing annotations cannot blindly be removed in translation, as Verilog's implicit resizings semantics do not necessarily result in the same kinds of resizings. Fortunately, explicit resizing annotations are also available in Verilog. In our formalization all resizing operations are explicit, so we can translate explicit resize operations to explicit resize operations in our translator.

This should be sound for Verilog programs produced by the translator, but is not entirely satisfactory. A better approach would be to also formulate the implicit resizing rules, so that the translator could have proved that no such implicit resizings occur in its translated expressions (i.e., even in the presence of the implicit resizing rules, the translation is still correct). (Furthermore, the translator could have also removed explicit resize operations where the implicit resize rules already imply the resizing, making the output code a little cleaner.)

Signed and unsigned operations. Signedness matters not only for resizings, i.e. to decide whether to do zero-extension or sign-extension, but also for operations such as arithmetical shifts, less-than comparisons, and similar comparison operators. For example, whether `a < b` is a signed or unsigned less-than operation in Verilog depends on whether `a` and `b` are both signed or not (and more generally, the signedness of various elements in the context the operation occurs in). We do not formalize these signedness rules directly, but instead keep all variables and expressions unsigned (so that we can ignore all signedness rules), and only convert to signed values (and then directly back to unsigned values) temporarily when needed using explicit sign casting. For example, to make sure to get a signed less-than operation one can write `{ $signed(a) < $signed(b) }` where the single-element concatenation operation again limits the context considered. Concretely, this

means that in the formalization there are two different (e.g.) less-than operands, one for signed less-than and one for unsigned less-than, and in pretty-printing sign casts are introduced for the signed variant.

Types. We have not formalized Verilog's static type system. Instead, type errors are checked at runtime in our formalization.

C. Formal semantics

Our formal semantics is a clocked functional operational semantics in three layers. The two first layers consist of an evaluation function erun for expressions and an evaluation function prun for stepping a process one clock cycle. Stepping processes one clock cycle always terminate in finite time, so there is no need for clocks in these layers. The third layer consist of a clocked evaluation function mrun (for "module run") that steps a program forward a specified number of clock cycles, by stepping every process in the program once per cycle by calling the prun function. Runtime errors are handled by returning a sum value, indicating either failure, Inl, or success, Inr. If an error occurs in a process, then the entire program execution is aborted (by returning failure).

Most concepts on the expression and statement level, such as array indexing, if-statements, and case-statements are formalized in a straightforward manner. For example, in relational style, because it is easier to show part of the semantics in this way, if-statements follow the obvious rules:

$$\frac{\mathsf{erun}\ \mathit{fext}\ s\ c = \mathsf{Inl}\ \mathit{err}}{\mathsf{prun}\ \mathit{fext}\ s\ (\mathsf{IfElse}\ c\ \mathit{pt}\ \mathit{pf}) = \mathsf{Inl}\ \mathit{err}},$$

$$\frac{\mathsf{erun}\ \mathit{fext}\ s\ c = \mathsf{Inr}\ (\mathsf{VArray}\ a)}{\mathsf{prun}\ \mathit{fext}\ s\ (\mathsf{IfElse}\ c\ \mathit{pt}\ \mathit{pf}) = \mathsf{Inl}\ \mathsf{TypeError}},$$

$$\frac{\mathsf{erun}\ \mathit{fext}\ s\ c = \mathsf{Inr}\ (\mathsf{VBool\ T}) \qquad \mathsf{prun}\ \mathit{fext}\ s\ \mathit{pt} = s'}{\mathsf{prun}\ \mathit{fext}\ s\ (\mathsf{IfElse}\ c\ \mathit{pt}\ \mathit{pf}) = s'},$$

$$\frac{\mathsf{erun}\ \mathit{fext}\ s\ c = \mathsf{Inr}\ (\mathsf{VBool\ F}) \qquad \mathsf{prun}\ \mathit{fext}\ s\ \mathit{pf} = s'}{\mathsf{prun}\ \mathit{fext}\ s\ (\mathsf{IfElse}\ c\ \mathit{pt}\ \mathit{pf}) = s'}.$$

Here, the parameter s represents the current circuit state, and we recognize *fext* from the example from Sec. II, allowing modeling of circuit-external behavior, such as e.g. memory modules or non-deterministic input sources. In the expression-level and statement-level Verilog semantics, *fext* is a function from variable names (strings) to Verilog values, as time is handled on the module level, as will be illustrated. The semantic rule for reading external inputs is straightforward as well:

$$\frac{\mathit{fext}\ \mathit{var} = \mathit{res}}{\mathsf{erun}\ \mathit{fext}\ s\ (\mathsf{InputVar}\ \mathit{var}) = \mathit{res}}.$$

We will now focus on the most interesting part of the formalization, namely, how concurrency is handled, how the Verilog event queue is modeled, and how blocking and non-blocking assignments interact with the event queue.

For the module level, a Verilog program is a list of processes, with an association list Γ assigning values to variables. During execution of a cycle, all non-blocking assignments are stored in another association list Δ (of the same type as Γ) used as a queue. The semantics is defined in monadic style in the actual development, but here we present functions (that we have proved equal to the original functions) with the monadic combinators unrolled for clarity. A function

$$\begin{aligned}&\mathsf{mstep}\ \mathit{fext}\ []\ s = \mathsf{Inr}\ s\\ &\mathsf{mstep}\ \mathit{fext}\ (p{::}ps)\ s = \mathsf{case\ prun}\ \mathit{fext}\ s\ p\ \mathsf{of}\\ &\quad \mathsf{Inl}\ e\ \Rightarrow\ \mathsf{Inl}\ e\\ &\ \mid \mathsf{Inr}\ s'\ \Rightarrow\ \mathsf{mstep}\ \mathit{fext}\ ps\ s'\end{aligned}$$

steps all processes in a given list one clock cycle starting in state s. Another function

$$\begin{aligned}&\mathsf{mstep\_commit}\ \mathit{fext}\ ps\ \Gamma = \mathsf{case\ mstep}\ \mathit{fext}\ ps\ (\Gamma,[])\ \mathsf{of}\\ &\quad \mathsf{Inl}\ e\ \Rightarrow\ \mathsf{Inl}\ e\\ &\ \mid \mathsf{Inr}\ (\Gamma',\Delta')\ \Rightarrow\ \mathsf{Inr}\ (\Delta' +\!\!+ \Gamma')\end{aligned}$$

constructs a new initial state for a new cycle (with an empty non-blocking writes queue), executes all given processes, and, lastly, "commits" all of the queued non-blocking writes by appending them to the program variables. The top-level function

$$\begin{aligned}&\mathsf{mrun}\ \mathit{fext}\ ps\ \Gamma\ 0 = \mathsf{Inr}\ \Gamma\\ &\mathsf{mrun}\ \mathit{fext}\ ps\ \Gamma\ (\mathsf{Suc}\ n) = \mathsf{case\ mrun}\ \mathit{fext}\ ps\ \Gamma\ n\ \mathsf{of}\\ &\quad \mathsf{Inl}\ e\ \Rightarrow\ \mathsf{Inl}\ e\\ &\ \mid \mathsf{Inr}\ \Gamma'\ \Rightarrow\ \mathsf{mstep\_commit}\ (\mathit{fext}\ n)\ ps\ \Gamma'\end{aligned}$$

allows for stepping a collection of processes, that is, a program, a specified number of cycles.

Note that mrun runs processes in the order they occur in its input list ps. In Verilog, processes are executed concurrently in an interleaved and non-deterministic manner. But we are only interested in processes that do not "interfere" with each other, so program execution can be modeled faithfully without considering non-deterministic interleavings. If we let vwrites denote the variables written to blockingly by a process, vnwrites denote the variables written to non-blockingly by a process, vreads denote the variables read by a process, and disjoint denote that two sets are disjoint (i.e., $\mathsf{disjoint}\ s\ t \iff s \cap t = \emptyset$), then we can formalize non-interference as follows:

$$\begin{aligned}&\mathsf{valid\_program}\ ps \iff\\ &\forall\, i\ j.\\ &\quad 0 \le i \wedge i < \mathsf{length}\ ps \wedge 0 \le j \wedge j < \mathsf{length}\ ps \wedge i \neq j \Rightarrow\\ &\quad \mathsf{let}\ p\ =\ \mathsf{el}\ i\ ps;\ q\ =\ \mathsf{el}\ j\ ps\ \mathsf{in}\\ &\quad\ \mathsf{disjoint}\ (\mathsf{vreads}\ p)\ (\mathsf{vwrites}\ q) \wedge\\ &\quad\ \mathsf{disjoint}\ (\mathsf{vnwrites}\ p \cup \mathsf{vwrites}\ p)\ (\mathsf{vwrites}\ q \cup \mathsf{vnwrites}\ q).\end{aligned}$$

The definition makes sure that processes only communicate through non-blocking assignments. As non-blocking assignments do not propagate during cycle execution, the order of execution among processes does not matter – and Verilog's event-driven semantics collapses into what we have above – which simplifies matters significantly, as the semantics can stay deterministic. As valid_program is purely syntactical, satisfaction can be checked by evaluation inside HOL.

As for the expression-level and statement-level semantics, the only construct that interacts with the queue of non-blocking

writes is in fact non-blocking assignments; meaning that reads are always based on Γ. The semantics of (successful) blocking (=) and non-blocking assignments (<=) for, e.g., Boolean variables are given by the following rules:

$$\frac{\mathsf{erun}\ \mathit{fext}\ (\Gamma,\ \Delta)\ e = \mathsf{Inr}\ v \qquad (x, v') \in \Gamma \qquad \texttt{same\_shape}\ v\ v'}{\mathsf{prun}\ \mathit{fext}\ (\Gamma,\ \Delta)\ (x = e) = \mathsf{Inr}\ ((x, v)::\Gamma,\ \Delta)},$$

$$\frac{\mathsf{erun}\ \mathit{fext}\ (\Gamma,\ \Delta)\ e = \mathsf{Inr}\ v \qquad (x, v') \in \Gamma \qquad \texttt{same\_shape}\ v\ v'}{\mathsf{prun}\ \mathit{fext}\ (\Gamma,\ \Delta)\ (x <= e) = \mathsf{Inr}\ (\Gamma,\ (x, v)::\Delta)}.$$

Two points are worth making here. Firstly, we make sure that the assigned variable's type does not change by ensuring that the value shape is the same before and after (using `same_shape`), rather than utilizing a separate static type system. Secondly, assignment rules for arrays (not shown here) are similar, but more complex. For arrays, one has to support writes to part of an array (e.g., `a[5] <= a[3]`), but such generalizations are straightforward. Conceptually, such writes only update part of the array written to, but, for simplicity, in our semantics we store the entire updated array in the queue of non-blocking writes.

D. Validation

To validate our reading of the Verilog standard we have compared the result of running 30 small handwritten expression-level examples (available in the source code repository) in our semantics to simulating them using Icarus Verilog (10.2), Xilinx Vivado Design Suite (2018.2), and Verilator (3.926). The examples exercise a subset of the operators supported by the semantics, and include array resizings and computations involving signed numbers. We focused on the expression level, rather than the statement level, as we in particular wanted to validate that our current handling of resizings and signed numbers work at least for small expressions. The arrays the examples operate over are all of short length (3–5 elements), meaning that testing all possible inputs was feasible. We did not find any discrepancies between our semantics and the three simulators. (When experimenting, we did, however, find bugs in Icarus Verilog related to resizing and sign handling. The bugs were resolved immediately by the maintainers.)

Another source of validation, which exercises also the statement-level semantics, is that our case studies (Sec. VII) worked as expected.

VI. The translator

For any formal hardware development methodology, it is important to consider how circuits are modeled in the prover:

1) Are the circuits *shallowly embedded*? In other words: are circuits just normal logic functions that ought to be understood as circuits?
2) Or are the circuits *deeply embedded*? In other words: is the syntax of circuits explicitly modeled and a separate evaluation function/relation gives them their meaning?

Reasoning about shallow embeddings is significantly simpler than reasoning about deep embeddings. However, deep embeddings offer a far more clear correspondence between the embeddings and the entity being modeled than shallow embeddings do.

Our proof-producing translator allows users to do reasoning in a shallow embedding and yet have the benefit of the deep embedding (i.e., a clear correspondence to the target representation) since properties proved of the shallow embedding can effortlessly be transported to the deep embedding.

A. Input language

The input language of the translator should be thought of as a hardware description language in the same sense as Verilog is a hardware description language. More precisely, the input language should be thought of as a language describing Verilog programs, which in turn describe hardware. When a programmer writes their HOL circuits (that is, HOL functions), they should have in mind what the translator's Verilog output will look like, and what in turn those Verilog constructs mean in terms of hardware. In this sense, we are doing Verilog development.

We decided to use standard HOL words (bit vectors) and Booleans for the input language rather than some custom data types modeling the Verilog data types in a more direct fashion because using standard data types allows us to re-use theories and proof tools from the HOL standard library when proving circuits correct. For example, there are pre-built tools for bit-blasting HOL words for using SAT solvers to find HOL proofs.

B. Implementation overview

As the translator is proof-producing, to trust the output of the translator we only need to trust the correctness of our Verilog formalization (that is, that it correctly captures the standard document) rather than the translator implementation itself; an implementation bug in the translator can at most result in the translator failing to produce a translation correctness proof.

As shown in the example in Sec. II, the translator takes a circuit represented as a next-state function consisting of smaller next-state functions as input. The translator translates the top-level circuit function into a Verilog program, with one process per inner next-state function. In cases where processes communicate, the translator introduces non-blocking assignments.

The translator implementation is split into two passes. A first, proof-producing pass that operates on one function at a time turns each function into a Verilog process, where all assignments are blocking. A second, verified pass replaces blocking assignments with non-blocking assignments where needed, and combines the processes produced by the first pass into a single complete Verilog program.

C. Pass one: process translation

The first pass is a proof-producing SML function, operating through the HOL4 API. To exemplify, the first pass turns the A function from the example from Sec. II into:

```
always_ff @ (posedge clk)
 if (pulse) count = count + 8'd1;
```

Note that the assignment is not yet non-blocking, as introducing such assignments is the responsibility of the second pass.

The first pass operates on one function at a time. Each input function is turned into a process (except the top function, which is AB in the example of Sec. II). There is no need to support auxiliary helper functions since all helper functions can be inlined by rewriting rules in HOL before translation.

The translator constructs its proofs using relations between various HOL (shallow) and Verilog (deep) entities. The translator defines relS to relate the input circuit's state record with Verilog states. Similarly, relS_fextv relates the external-state representations. These relations are used to define EvalS, which is central to the proof automation. We define EvalS $fext\ s\ \Gamma\ s'\ vp$ to say that, if states s and Γ are related, then execution of Verilog program vp results in some Verilog state Γ' that is related to new shallow state s':

$$\begin{array}{l}\mathsf{EvalS}\ fext\ s\ \Gamma\ s'\ vp \iff \\ \forall fextv\ \Delta. \\ \quad \mathsf{relS}\ s\ \Gamma \wedge \mathsf{relS\_fextv}\ fextv\ fext \Rightarrow \\ \quad \exists \Gamma'\ \Delta'. \\ \qquad \mathsf{prun}\ fextv\ (\Gamma,\Delta)\ vp = \mathsf{Inr}\ (\Gamma',\Delta') \wedge \\ \qquad \mathsf{relS}\ s'\ \Gamma'.\end{array}$$

We write EvalS $fext\ s\ \Gamma\ (f\ s)\ vp$ to state that Verilog program vp is related to HOL function f.

Internally, the first pass, following the Verilog process semantics, is separated into two layers: one layer for expressions and one layer for statements. For the expression level, there is an EvalS-like Eval relation, used to state translation correctness on the expression level:

$$\begin{array}{l}\mathsf{Eval}\ fext\ s\ \Gamma\ P\ e \iff \\ \forall fextv\ \Delta. \\ \quad \mathsf{relS}\ s\ \Gamma \wedge \mathsf{relS\_fextv}\ fextv\ fext \Rightarrow \\ \quad \exists v.\ \mathsf{erun}\ fextv\ (\Gamma,\Delta)\ e = \mathsf{Inr}\ v \wedge P\ v.\end{array}$$

In our semantics, evaluating an expression never changes the program state; evaluation simply results in to some Verilog value. Because of this, the Eval predicate is parameterized by a post-condition predicate P that can be instantiated to various predicates stating what an expression should evaluate to. The translator uses the predicates BOOL, WORD, and WORD_ARRAY for expressing translation correspondences between Booleans, words, and functions from words to words (representing arrays), respectively. For example, the definition of BOOL is simply that the corresponding HOL Boolean should be wrapped in the Verilog semantics' Boolean constructor:

$$\mathsf{BOOL}\ b\ v \iff v = \mathsf{VBool}\ b.$$

To make things more concrete, consider the expression translator (which produces Eval theorems) and consider the simple HOL expression $1w \oplus 2w$. For this input, the translator responds with

$$\begin{array}{l}\vdash \mathsf{Eval}\ fext\ s\ \Gamma\ (\mathsf{WORD}\ (1w \oplus 2w)) \\ \quad (\mathsf{ABOp}\ (\mathsf{Const}\ (\mathsf{w2ver}\ 1w))\ \mathsf{BitwiseXor} \\ \qquad (\mathsf{Const}\ (\mathsf{w2ver}\ 2w))),\end{array}$$

where (ABOp ...) is the resulting Verilog code as represented in the internal AST. To produce the above theorem, the translator utilizes the pre-proved theorem

$$\begin{array}{l}\vdash \mathsf{Eval}\ fext\ s\ \Gamma\ (\mathsf{WORD}\ w_1)\ v_1 \wedge \\ \quad \mathsf{Eval}\ fext\ s\ \Gamma\ (\mathsf{WORD}\ w_2)\ v_2 \Rightarrow \\ \quad \mathsf{Eval}\ fext\ s\ \Gamma\ (\mathsf{WORD}\ (w_1 \oplus w_2)) \\ \qquad (\mathsf{ABOp}\ v_1\ \mathsf{BitwiseXor}\ v_2).\end{array}$$

To discharge the antecedent of the theorem, the translator recursively calls itself with the operands of the input expression. These recursive calls can be resolved directly as translating literals is a base-case in the translator's recursive algorithm; so, the calls will return the needed Eval theorems directly.

This kind of syntax-directed divide and conquer approach is the main mechanism behind the entire translation process. The algorithm has access to a repertoire of similar pre-proved theorems, and can use them to translate other operations, such as arithmetic operations and array indexing. The same kind of decomposition is possible on the statement level, for, e.g., if-statements, where the following theorem is used for translations:

$$\begin{array}{l}\vdash \mathsf{Eval}\ fext\ s\ \Gamma\ (\mathsf{BOOL}\ C)\ Ce \wedge \mathsf{EvalS}\ fext\ s\ \Gamma\ L\ Lv \wedge \\ \quad \mathsf{EvalS}\ fext\ s\ \Gamma\ R\ Rv \Rightarrow \\ \quad \mathsf{EvalS}\ fext\ s\ \Gamma\ (\mathsf{if}\ C\ \mathsf{then}\ L\ \mathsf{else}\ R)\ (\mathsf{IfElse}\ Ce\ Lv\ Rv).\end{array}$$

For if-statements, the statement-level algorithm calls the expression-level algorithm to discharge the Eval part of the antecedent, and recursively call itself to discharge the two EvalS parts in the antecedent.

Not all constructs can be translated by specializing pre-proved theorems. Some constructs, such as case-expressions, need special, but ultimately straightforward and uninteresting, machinery for their translation. We leave out most of such details here, but make a few remarks in what follows.

One construct that needs special translation treatment is variables. Because our input is shallowly embedded circuits, some hardware concepts must be modeled indirectly. Temporary local immutable variables are modeled natively as let-expressions, but imperative concepts, such as state between clock cycles and local mutable variables, are modeled indirectly through HOL state record fields. Both types of variables (let-expressions variables and state record fields) are translated to standard (mutable) variables in Verilog. For example, the HOL function

$$\begin{array}{l}\mathsf{R}\ s = \mathsf{let}\ s' = s\ with\ ⟪\mathsf{a} := 1w;\ \mathsf{b} := 2w⟫; \\ \qquad\qquad s'' = s'\ with\ \mathsf{a} := s'.\mathsf{a} + 1w; \\ \qquad\qquad tmp = 1w \\ \quad \mathsf{in\ case}\ s''.\mathsf{c}\ \mathsf{of} \\ \qquad 0w \Rightarrow s''\ with\ \mathsf{c} := tmp \\ \qquad \mid v \Rightarrow s''\ with\ \mathsf{c} := 0w\end{array}$$

produces the following output when used as an input circuit:

```
always_ff @ (posedge clk)
 b = 8'd2; a = 8'd1;
 a = a + 8'd1; tmp = 8'd1;
 case c
  8'd0 : c = tmp;
  default : c = 8'd0;
 endcase
```

In the example we see that let-expressions are used both for binding intermediate states and local (immutable) variables. Both types of let-expressions are, unsurprisingly, translated by divide and conquer. For let-expressions used for binding intermediate state, input functions are only allowed to refer to the most recent "state variable" (in R first s, then s', and then s''), which makes translating such let-expressions straightforward. Let-expressions used to introduce local variables require more work. When an unknown variable is reached during translation, the output EvalS and Eval theorems will be weakened by preconditions on their environment Γ constraining it to include the encountered variable. The constructed Verilog code and the generated precondition are coupled using a free HOL variable, and when the recursion, on its way up, reaches the relevant let-binding site, the precondition can be weakened to only require that the variable in question has the correct shape. The shape precondition is required for the blocking assignment the matching let-binding site introduces to not fail with a (runtime) type error. The shape precondition is then propagated to the top because we need to keep track of which variables to declare at the top level in the generated Verilog program.

Furthermore, in the same example we see that the translator supports nested record updates. Some care must be taken when translating such expressions, consider e.g. $\langle|\mathsf{a} := 0w;\ \mathsf{b} := s.\mathsf{a}|\rangle$ and $\langle|\mathsf{b} := s.\mathsf{a};\ \mathsf{a} := 0w|\rangle$, where s is the current state record: in HOL the expressions are equivalent, but if refined naively into Verilog as sequential mutable variable updates they are no longer equivalent. This is an important point, as for an expression to be translatable, its syntax must allow a dual reading in which the HOL and Verilog semantics coincide.

Lastly, we have yet to discuss multidimensional arrays. Such arrays are represented as functions from words to words in the input language. The current machinery for multidimensional arrays is simple and quite limited, and only supports arrays up to three dimensions.

Returning to the input language discussion, the function R above is fairly representative of what kind of functions are accepted as input by the translator. That is, functions consisting of nested let-expressions, in turn consisting of operations that fairly directly, in a syntactical sense but not necessarily semantical sense, map to our subset of Verilog. Of course, larger functions than R, with more nesting, can be translated.

The translation approach taken here is inspired by Myreen and Owens' HOL to CakeML proof-producing code generator [10]. Translating from HOL to Verilog is both easier and more difficult. It is easier, at least in our case, because we accept a smaller and more specialized subset of HOL as input, and it is more difficult because the distance between HOL and Verilog is larger than the distance between HOL and CakeML.

D. Pass two: full program translation

The second pass takes EvalS theorems produced by the first pass and composes them into a theorem for a whole Verilog program. In terms of the example from Sec. II, the first pass produces two EvalS theorems, one for A and one for B, and the second pass takes them as input and produces a correspondence theorem for the whole circuit AB.

The second pass is verified instead of proof-producing, and consist of a HOL function intro_cvars and associated proof infrastructure. The function operates over Verilog syntax and takes a user-provided list of "communication variables" (in the Sec. II example, just `count`) and replaces all assignments to these variables with non-blocking assignments. The proof infrastructure helps to build a whole-program correspondence theorem out of the process theorems produced by the first pass.

The second pass requires that processes do not read from communication variables they have written to earlier in the same cycle. This style requirement should be seen as a strategy to shallowly embed non-blocking assignments, as, process-locally, if a variable is not read after being written to, it does not matter if the writes to it are blocking or non-blocking. More precisely, the style requirement ensures that intro_cvars is semantic preserving in the sense that for any Verilog process p without non-blocking assignments, $\mathsf{prun}\ \mathit{fext}\ (\Gamma, \Delta)\ p = \mathsf{Inr}\ (\Gamma', \Delta')$ implies that there exist Γ'_{cs} and Δ'_{cs} such that $\mathsf{prun}\ \mathit{fext}\ (\Gamma, \Delta)\ (\mathsf{intro\_cvars}\ cs\ p) = \mathsf{Inr}\ (\Gamma'_{cs}, \Delta'_{cs})$, and (Γ', Δ') and $(\Gamma'_{cs}, \Delta'_{cs})$ only differs in that writes to communication variables cs have been moved from Γ' to Δ'_{cs}.

The correspondence theorems for whole programs, the target output of second pass, are on the same form as the process-level EvalS theorems. Namely, state equivalence between a HOL state and a Verilog state is invariant under stepping. The proof infrastructure available for intro_cvars in combination with a small amount of circuit-specific boilerplate setup code can be used to combine the collection of processes generated by the first pass into a single complete Verilog program and generate a whole-program correspondence theorem if the processes satisfy the above style requirement and the valid_program predicate.

VII. Case studies

We present two case studies: a verified processor, built to be usable in verified stack constructions, and a method to construct verified regexp (regular expression) matchers.

A. Processor case study

As part of our paper on verified stacks [4] referred to in the introduction, we showed how we have designed, verified, and synthesized a simple processor using the hardware development methodology that is elaborated in this paper. The processor is capable of hosting programs compiled by the (verified) CakeML compiler [6], including the compiler itself. One of the main results in the paper is a theorem stating that running the compiler correctly compiles programs even when running on top of the processor (down to the Verilog level). The processor and CakeML compiler work that was required for this result is described in more detail in our verified stacks paper, but we briefly recapitulate some hardware-relevant points here as the case study illustrates that the translator scales beyond the small examples presented in the paper so far.

The processor implements a small custom RISC instruction set architecture. We designed the processor to be as simple

as possible (it is, e.g., not pipelined) but still capable enough to host compiled CakeML programs. For this case study we targeted a PYNQ-Z1 board, using the board's DRAM for data and instruction memory. The processor has support for hardware accelerators and consists of two processes: one for the processor itself, and one for the hardware accelerator currently in use. (We have yet to develop any hardware accelerator beyond a placeholder integer addition accelerator.) The processor's external environment, such the DRAM module and an interrupt interface used for communicating with the external world, is modeled by a *fext* function. An earlier unverified but translatable version of the processor consisted of three processes, where the third process modeled a BRAM module used for data and instruction memory; illustrating that BRAMs can also be modeled in the translator's input language (modulo a minor problem with packed vs. unpacked arrays).

Running the Verilog translator on the processor takes just a few seconds (compiling everything from scratch, including the "pre-proved theorems" needed by the translator, takes a few minutes). The output Verilog code is around 300 lines of code.

Using the Vivado toolchain, the synthesized processor can be clocked at 40 MHz. The processor loaded onto the FPGA board is capable of running programs compiled by the CakeML compiler. In particular, as the compiler can compile itself, we have been able to run the compiler itself on top of the processor.

B. Regexp matcher case study

Our second case study is a method to construct verified regexp matchers implemented in Verilog. The method is based on an existing HOL tool[1] that can compile regexps to DFAs represented as a simple driver function and a (potentially large) next-state table. To enable this new hardware-producing flow, we first translated the driver function to a circuit in HOL. The circuit is a one-process program, and has a serial interface that can receive one character per clock cycle, and one output bit indicating whether the string formed by the characters seen so far is accepted or not. There is also a reset input bit to start a new match. We then proved that the circuit accepts the same language as the original DFA when they both follow the same next-state table.

To show that the flow works, we constructed a matcher implementation for the simple regexp `fo+bar`. The regexp-to-DFA compiler tool is proof-producing, and as our Verilog translator is proof-producing as well, running the flow, we were able to compose the above manual circuit correctness theorem and the two generated correspondence theorems into a theorem stating that the Verilog circuit accepts the same language as the initial regexp. Being able to compose theorems from different developments in this way illustrates one of the advantages of embedding circuits inside ITPs over relying on traditional verification methods such as model checking. We also synthesized the Verilog circuit with some glue code for a Basys 3 FPGA board connected to a keyboard, and the circuit worked as expected. As only the table differs between different regexp matchers, generating matchers for other regexps is straightforward. (Our driver is register based; for larger regexps, a BRAM-based driver would be more appropriate. Such a driver would require another round of manual verification.)

[1]The tool is located in `examples/formal-languages/regular` in the HOL4 distribution

VIII. Related work

Producing correct hardware with the help of an ITP has been addressed in many ways. For example, two earlier verified stack projects [2], [3] report synthesizing for FPGAs: The pre-Verisoft PVS VAMP processor was specified in a custom-built gate-level language which relied on an unverified `pvs2hdl` tool [8] for translation to Verilog for synthesis, and a tool with similar functionality was available for the Isabelle/HOL version of VAMP [11]. In contrast, the DeepSpec Kami project [12] targets the high-level HDL Bluespec, and relies on the usual unverified Bluespec toolchain for synthesis.

More generally, it seems that one common synthesis strategy is to do verified or proof-producing compilation down to a "simple" or "low-level" language (or simply start from such a language) that can "easily", *but without proof*, be translated to some mainstream low-level HDL, such as Verilog or VHDL, and then feed this output to some external synthesis toolchain. For example Iyoda [13], Pizani Flor et al. [14], and Braibant and Chlipala [15] follow this approach. But also the opposite direction is possible; Hunt et al.'s [16] tool instead loads final-product Verilog programs into their ACL2 setup. In the context of hardware security, tools capable for loading (subsets of) VHDL and Verilog into Coq are available [17], [18]. The vital difference between our project and earlier projects, is that earlier projects have not provided proofs all the way down to an explicitly stated Verilog/VHDL semantics.

As for Verilog semantics work, the most complete Verilog formalization we are aware of is Meredith et al.'s rewriting logic-based K framework formalization [9], later ported to Isabelle/HOL by Khan et al. [19]. Meredith et al. model a larger subset of Verilog than we do, including e.g. continuous assignments and non-synthesizable concepts such as delayed statements. This requires a more complete, and complicated, event queue model and event execution model. For other previous Verilog semantics work, see the related work section of Zhu et al. [20]. But whereas we have applied our Verilog semantics for verification by utilizing it in our translator, by in turn having applied the translator by extracting our case studies, missing from the mentioned and earlier Verilog formalization initiatives are convincing applications, where the authors apply their semantics in non-trivial circuit verification work. For example, Khan et al. prove that multiplication by two and left-shifting by one are equivalent in their semantics, and then note that "proving more general theorems about complex designs would be extremely difficult".

IX. Discussion

We will now discuss the trusted computing base (TCB) of our hardware development methodology. To trust our methodology it is necessary to trust, beyond the usual suspects,

an external synthesis toolchain (to be used in the final step of the methodology), and the soundness of our Verilog formalization.

Ideally, analogously to the situation in software development [21], our circuits would exit our ITP at the lowest possible level of abstraction. In hardware development, what the lowest level is depends on what technology we are targeting. For FPGAs, which we are interested in, the lowest level is at the level of FPGA bitstreams. Targeting this level from our input level would require access to a combination of proof-producing and verified tools for technology mapping, placement and routing, etc. At the time of writing, no such toolchain exists. There are multiple reasons for this; one of them being that the FPGA bitstream formats rarely have publicly available documentation. Instead, for transporting our Verilog circuits to FPGA bitstreams, our hardware development methodology relies on existing unverified tools.

Synthesis toolchain. Here, to trust a synthesis toolchain means trusting it to compile our generated Verilog code to an FPGA bitstream in a semantics preserving manner with respect to Verilog's simulation semantics. When relying on external tools, some confidence in the correctness of the final bitstream can be built by employing standard industrial techniques such as testing and formal equivalence checking. Unfortunately, the evidence produced via testing and equivalence checking is not connected to any formal semantics and can thus not be properly connected to our proofs.

Verilog formalization. Also the communication channel between our ITP developments and the external synthesis tools must be trusted. As Verilog is our communication medium between these two parts, trusting our methodology means trusting our reading and formalization of the Verilog standard.

An inherent risk with targeting a large standard-defined language like Verilog without an official formal semantics is that it is impossible to prove readings of such language standards correct. Consequently, when arguing for our formalization's correctness, we have to fall back on empirical methods, such as testing our semantics against Verilog simulators, and carrying out case studies based on our methodology. An alternative means of validation, which we have not pursued, would be to prove a correspondence between our semantics and a semantics at the level of detail of Meredith et al.'s semantics [9]. In this approach, one would, e.g., show that the order of process execution order does, indeed, not matter for Verilog programs satisfying valid_program. This would validate our semantics further, but would not address the question why the more detailed (and therefore more complicated) semantics is correct.

X. Conclusion

We have constructed a proof-producing translation tool from HOL circuits to Verilog circuits, and, as a prerequisite for this, developed a formal semantics for a subset of Verilog. The semantics is carefully written to faithfully model the Verilog standard, while still being simple enough to use for reasoning. The latter is important for our proof-producing translator, which must carry out automatic reasoning to construct proofs guaranteeing that its translations are correct.

The translator enables a hardware development flow where users develop theorems and theories based on shallowly embedded HOL circuits that can easily be transported to corresponding deeply embedded Verilog circuits.

Acknowledgments. We thank Carl-Johan Seger and Koen Claessen for helpful feedback. This work was partly supported by the Swedish Foundation for Strategic Research.

References

[1] J Strother Moore, "A grand challenge proposal for formal methods: A verified stack," in *10th Anniversary Colloquium of UNU/IIST*, 2003.

[2] E. Alkassar, M. A. Hillebrand, D. Leinenbach, N. W. Schirmer, and A. Starostin, "The Verisoft approach to systems verification," in *Verified Software: Theories, Tools, Experiments (VSTTE)*, 2008.

[3] A. W. Appel, L. Beringer, A. Chlipala, B. C. Pierce, Z. Shao, S. Weirich, and S. Zdancewic, "Position paper: The science of deep specification," *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 375, no. 2104, 2017.

[4] A. Lööw, R. Kumar, Y. K. Tan, M. O. Myreen, M. Norrish, O. Abrahamsson, and A. Fox, "Verified compilation on a verified processor," in *Programming Language Design and Implementation (PLDI)*, 2019, to appear.

[5] K. Slind and M. Norrish, "A brief overview of HOL4," in *Theorem Proving in Higher Order Logics (TPHOLs)*, 2008.

[6] A. Fox, M. O. Myreen, Y. K. Tan, and R. Kumar, "Verified compilation of CakeML to multiple machine-code targets," in *Certified Programs and Proofs (CPP)*, 2017.

[7] "IEEE standard for SystemVerilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2017*, 2018.

[8] S. Beyer, C. Jacobi, D. Kroening, and D. Leinenbach, "Correct hardware by synthesis from PVS," Tech. Rep., 2002. [Online]. Available: http://www-wjp.cs.uni-sb.de/publikationen/BJKL02.pdf

[9] P. Meredith, M. Katelman, J. Meseguer, and G. Roşu, "A formal executable semantics of Verilog," in *Formal Methods and Models for Codesign (MEMOCODE)*, 2010.

[10] M. O. Myreen and S. Owens, "Proof-producing translation of higher-order logic into pure and stateful ML," *Journal of Functional Programming (JFP)*, vol. 24, no. 2–3, 2014.

[11] S. Tverdyshev, "Formal verification of gate-level computer systems," Ph.D. dissertation, Saarland University, 2009.

[12] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: A platform for high-level parametric hardware specification and its modular verification," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 1, no. ICFP, 2017.

[13] J. Iyoda, "Translating HOL functions to hardware," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-682, 2007.

[14] J. P. Pizani Flor, W. Swierstra, and Y. Sijsling, "Pi-Ware: Hardware description and verification in Agda," in *Types for Proofs and Programs (TYPES 2015)*, 2018.

[15] T. Braibant and A. Chlipala, "Formal verification of hardware synthesis," in *Computer Aided Verification (CAV)*, 2013, vol. 8044.

[16] W. A. Hunt, M. Kaufmann, J Strother Moore, and A. Slobodova, "Industrial hardware and software verification with ACL2," *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 375, no. 2104, 2017.

[17] X. Guo, R. G. Dutta, P. Mishra, and Y. Jin, "Automatic code converter enhanced PCH framework for SoC trust verification," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 12, 2017.

[18] M. Bidmeshki and Y. Makris, "VeriCoq: A Verilog-to-Coq converter for proof-carrying hardware automation," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2015.

[19] W. Khan, A. Tiu, and D. Sanán, "VeriFormal: An executable formal model of a hardware description language," in *Singapore Cyber-Security RandD Conference (SG-CRC)*, 2017.

[20] H. Zhu, J. He, and J. P. Bowen, "From algebraic semantics to denotational semantics for Verilog," *Innovations in Systems and Software Engineering*, vol. 4, no. 4, 2008.

[21] R. Kumar, E. Mullen, Z. Tatlock, and M. O. Myreen, "Software verification with ITPs should use binary code extraction to reduce the TCB," in *Interactive Theorem Proving (ITP)*, 2018.