Reconciling Verified-Circuit Development and Verilog Development

Andreas Lööw
Imperial College London
London, UK

***Abstract*—In software development, verified compilers like the CompCert compiler and the CakeML compiler enable a methodology for software development and verification that allows software developers to establish program-correctness properties on the verified compiler's target level. Inspired by verified compilers for software development, the verified Verilog synthesis tool Lutsig enables the same methodology for Verilog hardware development. In this paper, we address how Verilog features that must be understood *as hardware constructs*, rather than *as software constructs*, fit into hardware development methodologies, such as Lutsig's, inspired the development methodology enabled by software compilers. We explore this issue by extending the subset of Verilog supported by Lutsig with one such feature: `always_comb` blocks. In extending Lutsig's Verilog support with this, seemingly minor, feature, we are, perhaps surprisingly, required to revisit Lutsig's methodology for circuit development and verification; this revisit, it turns out, requires reconciling traditional Verilog development and the traditional program-verification methodology offered by verified software compilers. All development for this paper has been carried out in the HOL4 theorem prover.**

***Index Terms*—hardware development, hardware synthesis, Verilog**

I. Introduction

In software development, verified compilers enable the following interactive-theorem-proving-based verified-program development (VPD) methodology:

1) *develop and compile* your program in the same way as when using an unverified compiler;
2) *prove* a source-level correctness theorem about your program (by whatever means you have available – the methodology is independent of how the correctness theorem is established); and, lastly,
3) *transport* the source-level program-correctness theorem down to your verified compiler's target level by simple composition of the source-level program-correctness theorem and the compiler's (program-independent) correctness theorem.

VPD has been successfully deployed in many different software contexts, such as e.g. imperative programming [1], functional programming [2], concurrent programming [3], just-in-time compilation [4], [5], compiler-implementation correctness (by compiler bootstrapping) [2], [6], usability such as compositional/separate compilation [7], security such as constant-time preservation [8], and performance such as time/space reasoning [9]–[11].

In this paper, however, our interest lies in hardware development rather than software development. Previous work on verified hardware-synthesis tools [12]–[15] – also known as hardware compilers – show that VPD is equally applicable to hardware contexts, thereby providing a methodology for circuit development and verification. In this paper, we augment existing work on VPD in hardware contexts by considering source-level language Verilog features that must be understood *as hardware constructs* rather than *as software constructs*.

To handle such hardware constructs, we propose a hardware development methodology combining VPD and traditional Verilog development (TVD). While radical methodological redesign is certainty a worthwhile enterprise [16]–[26], we here dedicate our energy towards an enterprise in which we want to maintain as much as possible of the look-and-feel of both VPD and TVD. Specifically, as we further elaborate in the next section (Sec. II), we want to maintain both (1) VPD's ability to transport source-level correctness theorems down to the compiler's target level and (2) TVD's synthesis-modeling-idiom-based approach to synthesis.

We validate the proposed methodology combining VPD and TVD by adapting and extending Lutsig [14], a verified synthesis tool for synchronous Verilog designs, for the methodology. Specifically, we extend Lutsig's Verilog support with one of Verilog's features that must be understood as a hardware construct: `always_comb` blocks, which allows hardware designers to declare that certain parts of their behavioral Verilog code are to be synthesized to combinational logic. Combinational logic is stateless logic and stands in contrast to sequential logic (modeled as e.g. `always_ff` blocks), which is stateful logic.

All in all, we make the following two contributions:

- We propose a development methodology combining VPD, i.e. the traditional development methodology based on verified compilers, and TVD, i.e. traditional Verilog development, in a way that inherits the strengths of both and simultaneously avoids their main weaknesses.
- We validate the methodology by showing that it allows us to add support for `always_comb` blocks to Lutsig, the Verilog semantics used in Lutsig, and a proof-producing Verilog code generator connected to Lutsig.

All the work for this paper has been carried out in the HOL4 theorem prover [27]. All source code and proofs are available at https://github.com/CakeML/hardware.

https://doi.org/10.34727/2021/isbn.978-3-85448-053-2_15

II. Background: VPD and TVD

This section serves two purposes: firstly, it introduces VPD and TVD in more detail, and, secondly, it establishes notation and terminology used in the rest of the paper.

A. Verified-program development (VPD)

We now give a more detailed description of VPD, following the exposition of Leroy [1]. In VPD, we start off with a source program P_S implemented in a source language S and a compiled program P_T implemented in a target language T produced by a compiler: $Comp\ P_S = \texttt{OK}\ P_T$. If the compiler is unable, for whatever reason, to compile P_S, then a compile-time error is reported: $Comp\ P_S = \texttt{Error}$. The source language S has a semantics L_S, and the target language T has a semantics L_T. The two semantics L_S and L_T associate sets of observable behaviors B to source and target programs. We write $P \Downarrow_L B$ to denote that a program P executes with observable behavior B under semantics L.

We say that a compiler $Comp$ is verified when we have proved $\forall P_S\, P_T$, $Comp\ P_S = \texttt{OK}\ P_T \implies P_S \approx P_T$ for some notion of semantic preservation $\approx$. The only notion of semantic preservation we use in this paper is backward simulation: $P_S \approx P_T \iff \forall B,\ P_T \Downarrow_{L_T} B \implies P_S \Downarrow_{L_S} B$; that is, any behavior of the target program must be a behavior allowed by the source semantics.

Compiler users, however, are not ultimately interested in the correctness of the compiler $Comp$ they are using; rather, when compiling a source program P_S with a compiler, users are ultimately interested in the correctness of the target program P_T produced by the compiler. This is, of course, also part of VPD. Since it is easier to prove the correctness of P_S and *transport* the result to P_T than it is to prove the correctness of P_T directly, VPD is as follows: Following Leroy's exposition, users are asked to formalize what they mean by their program being correct by providing a predicate $Spec$ over observable behaviors. We write $P \models_L Spec$ for $\forall B,\ P \Downarrow_L B \implies Spec\ B$. Now, for a successful compiler run $Comp\ P_S = \texttt{OK}\ P_T$, if the user's compiler $Comp$ has been verified (with backward simulation as the notion of semantic preservation), then the user can derive $P_T \models_{L_T} Spec$ (i.e., what the user is ultimately interested in) from $P_S \models_{L_S} Spec$ by simple composition.

B. Traditional Verilog development (TVD)

We now turn to TVD. As Weste and Harris [28, p. 699] put it, hardware description languages (HDLs) like Verilog are "better understood as shorthand for describing digital hardware" than programming languages. Continuing, Weste and Harris describe TVD as follows:

1) "[...] begin your design process by planning, on paper or in your mind, the hardware you want."
2) "Then, write the HDL code that implies that hardware to a synthesis tool."

In TVD, an important concept is *modeling idioms*, which enable the hardware designer to express not only the behavior of their design but *what kind of hardware they want*. Modeling idioms are what allow the hardware designer to write Verilog code that "implies" the hardware design the hardware designer has formed "on paper or in [their] mind."

Examples of modeling idioms include e.g. `always_ff` and `always_comb` blocks, allowing hardware designers to specify if sequential or combinational logic should be inferred by the synthesis tool. In general, what modeling idioms are available depends on what technology is targeted. E.g., the synthesis manual for Xilinx's (unverified) synthesis suite Vivado [29, p. 111] contains modeling idioms and guidelines for modeling block RAMs (BRAMs), a type of memory available in Xilinx FPGAs. The modeling idioms related to BRAMs are presented as Verilog design fragments, instructing the hardware designer how to write their Verilog code such that the synthesis tool will infer features such as write enable inputs, byte-write-enable inputs, optional output registers, etc.

III. Reconciling VPD and TVD

Having introduced both VPD and TVD, we are now in a position to combine the best of two worlds: we want the methodology for circuit development and verification offered by Lutsig to provide the strengths of both VPD, i.e., theorem transportation, and TVD, i.e., synthesis-tool control by modeling idioms.

As a first step, as we want to apply the VPD methodology to Verilog hardware development, we must specialize $Comp$, S, L_S, T, and L_T to appropriate hardware instances. Since we, in this paper, are working with Lutsig, we set: $Comp$ = Lutsig, S = Verilog (abbreviated "ver"), and T = technology-mapped netlists for (a class of) FPGAs (abbreviated "nl"). For L_T, Lutsig uses a simple netlist language. What remains to specify is L_S – and this is where our problems begin.

The problems surrounding L_S arise from the fact that, traditionally conceived, Verilog has two semantics: one *simulation semantics* and one *synthesis semantics*. The reason for having two semantics, we will see, is TVD. This, however, does not fit cleanly into VPD since in VPD the source language S is supposed to have *one and only one* semantics L_S; since otherwise theorem transportation cannot be carried out by simple composition.

We now discuss the two semantics in the context of synthesis tool design and how they relate and fit into VPD and TVD. We first introduce the two semantics, we then survey the state of the art, and then conclude by stating how our development methodology – combining VPD and TVD – as implemented in Lutsig contributes to the state of the art.

Simulation semantics. The simulation semantics is given by the (System)Verilog standard [30]. The semantics is large, complicated, and full of gotchas [31], but at the end of the day, is an informally specified event-based operational semantics.

Synthesis semantics. The situation for the synthesis semantics is less straightforward.

Firstly, one minor hurdle to overcome is that the authoritative source for the semantics is unclear. Since the Verilog standard does not provide a synthesis semantics and the Verilog synthesis standard [32] has been withdrawn, it is up to each synthesis tool to provide their own synthesis semantics. Current tool-specific

synthesis manuals, such as e.g. the synthesis manuals for Vivado [29] and Quartus [33], however, largely contain similar material as the withdrawn synthesis standard (similar modeling idioms, design and coding-style recommendations, etc.), except specified in a more detailed fashion since such manuals are both tool- and target-technology-specific. We therefore use the withdrawn Verilog synthesis standard as the basis for our discussion here.

Secondly – the major hurdle – the synthesis semantics, both as specified in the synthesis standard and the tool-specific synthesis manuals, is not a full semantics like the simulation semantics; rather, it is just a collection of modeling idioms and design recommendations built on top of the simulation semantics. This ends up causing problems since some of the modeling idioms prescribe semantics incompatible with the simulation semantics: specifically, some of the modeling idioms have not only *nonfunctional consequences* but also *functional consequences*; in other words, some modeling idioms have consequences for the (functional) behavior of synthesized circuits! In TVD, the problems this causes are known as *simulation-and-synthesis mismatches*. Some mismatches are highlighted in (the informative) App. B in the synthesis standard. E.g., we are warned that the following module[1] will cause a simulation-and-synthesis mismatch since the assignments to `y` and `tmp` are "mis-ordered" (since the block is supposed to describe combinational logic – that is, stateless logic – and `tmp` is read before being assigned):

```
module andor1b(output reg y, input a, b, c);
reg tmp;

always @* begin
  y = tmp | c;
  tmp = a & b;
end
endmodule
```

State-of-the-art VPD. To some extent, VPD and TVD were reconciled already in the first version of Lutsig. However, except for X assignments, which, according to the synthesis standard, "tells the simulator to treat the signal as having an unknown value and tells the synthesis tool to treat the signal as a don't care" [32, p. 106], not much attention was directed towards simulation-and-synthesis mismatches. This was because the supported subset of Verilog was sufficiently small and software-like that the parts of Verilog that risk causing simulation-and-synthesis mismatches were, in effect, avoided.[2]

Now, on the other hand, when adding support for `always_comb` to Lutsig, i.e., a feature that must be understood *as a hardware construct* rather than *as a software construct*, i.e., a feature that must be understood in terms of modeling idioms, further reconciliation between VPD and TVD is needed. At the same time, we should acknowledge that problems similar to our present problems can be found in software development as well. E.g., one aspect of what has happened is that we have ended up with *nonfunctional expectations* on our synthesis tool – and VPD, in its minimal incarnation, only covers *functional expectations*, specifically semantics preservation. Nonfunctional expectations are, of course, sometimes put on software compilers [35], since functional software-compiler guarantees say (most commonly) nothing about code size, memory usage, cache performance, overall performance, security, etc. Indeed, some of the software VPD work mentioned in the introduction provide examples of VPD work addressing nonfunctional properties, such as security [8] and space reasoning [9].

Another point of comparison is how so-called undefined behavior (UB) is handled in languages such as C [36], [37]. UB leaves some parts of the language in question left with unspecified semantics (to allow for compiler optimizations). UB forms a subset of the language to avoid. Simulation-and-synthesis mismatches are similar to UB in the sense that sources of such mismatches can be seen as parts of Verilog to avoid. However, the two are not equivalent since the concept that induces simulation-and-synthesis mismatches, modeling idioms, has no analog in UB-based approaches to language semantics.

Recall that we aim to keep the look-and-feel of TVD in Lutsig's combination of VPD and TVD. We therefore must include modeling idioms in Lutsig's synthesis methodology rather than try to formulate a synthesis story under a – potentially more familiar for software developers – UB framework.

State-of-the-art TVD. Today's commercial (unverified) synthesis tools leave much to be desired; within the same tool, simulation-and-synthesis mismatches are handled along the whole spectrum of: silently miscompiling Verilog designs, issuing warnings, and aborting the compilation process entirely. In consequence, the result of a successful synthesis run is unclear for hardware developers: since an error-free synthesis run does not guarantee an actually successful synthesis run, some form of postsynthesis inspection, e.g. testing or manual visual inspection, is needed to ensure that the functional and nonfunctional properties we are interested in survived or were established during synthesis.

Lutsig's methodology. The conclusion we draw from the above discussion is that, to handle both TVD and VPD, Lutsig must implement both Verilog's semantics: the simulation semantics for VPD-style theorem transportation, and the synthesis semantics, in the form of synthesis idioms, for synthesis-idiom-based TVD.

In Lutsig, TVD is handled on an informal best-effort basis, since strict compliance prohibits too many optimizations, and VPD is handled, as it must, formally.

An interesting question is how much of TVD can be handled formally. For this paper, to illustrate that part of TVD can be treated formally, the feature of focus of this paper, `always_comb` blocks, diverges in Lutsig from the above general pattern of treating TVD informally: we prove that if the two semantics assign different behaviors to an `always_comb` block (e.g., because of "mis-ordered" writes) in a given input design, then Lutsig will abort – since Lutsig cannot abide

[1] Here presented verbatim, using an `always @*` block rather than an `always_comb` block since the synthesis standard was published before the first SystemVerilog standard – the synthesis standard based on the Verilog 2001 standard [34].

[2] Clearly, a discussion concluding "Lutsig takes Verilog's simulation semantics as its synthesis semantics" [14, p. 50] is insufficient for handling `always_comb` blocks.

by both semantics if they point in different directions. It is Lutsig's two top-level theorems (Sec. VIII and IX) that together formally show that Lutsig successfully handles both semantics for `always_comb` blocks. We leave the consideration of other synthesis idioms as future work.

Lutsig's contribution to establishing functional properties. Like for the first version of Lutsig, we have proved that Lutsig is semantics preserving (Sec. VIII). Specifically, after our discussion, it should now be clear that Lutsig must be semantics preserving with respect to Verilog's simulation semantics. We call Lutsig's formalization of the simulation semantics L_{ver}; i.e., in terms of VPD, we have $L_S = L_{\text{ver}}$. The semantics is the same Verilog semantics used as in the first version of Lutsig, with the exception that we now have added support for `always_comb` blocks (as described in Sec. V).

Since Lutsig allows for VPD development, after the hardware designer has transported a source-level correctness theorem down to the netlist level, the designer can rest assured that the synthesis process has not introduced any functional bugs. For functional correctness, VPD effectively *forces Lutsig to adopt (in stark contrast to other Verilog synthesis tools) a uniform error handling mechanism*: if Lutsig cannot guarantee semantics preservation, it must abort. Like the first version of Lutsig, and other verified compilers and synthesis tools, silent miscompilation is guaranteed to never occur.

Lutsig's contribution to establishing nonfunctional properties. We improve the state of the art in establishing nonfunctional hardware property by proving that Lutsig's synthesis algorithm correctly implements the modeling idiom that `always_comb` must generate combinational logic (Sec. IX), i.e., enables proven-correct TVD for `always_comb` blocks. For other modeling idioms, Lutsig does not improve the state of the art with respect to establishing nonfunctional properties.

Other approaches to circuit correctness. The first Lutsig paper [14] compares VPD-style hardware development, as followed here, to other approaches to circuit correctness, such as translation validation (known as formal equivalence checking in the hardware world), so we do not repeat that discussion here.

IV. Using Lutsig in practice

The rest of the paper consists of putting the discussion up till now into practice by adding support for `always_comb` to Lutsig and surrounding components. But before heading into technical details, we show how all pieces of the development fit together by demonstrating how hardware designers can use Lutsig in combination with a proof-producing Verilog code generator, developed in conjunction with Lutsig, to transport correctness properties down to the netlist level.[3]

[3]We emphasize that what is demonstrated here is one of multiple potential use cases of Lutsig. Like any Verilog synthesis tool, Lutsig can be made part of different hardware-development flows. In particular, one can imagine many different front-ends capable of generating Lutsig Verilog ASTs and, in various ways, producing proofs of correctness for those ASTs. In this paper, the proof-producing code generator we use fits our purposes here. Someone wanting to verify and synthesize existing Verilog code will have other needs. For developers not interested in verification at all, there is a (unverified) Verilog-text-file front-end for Lutsig available such that Lutsig can be used like a conventional Verilog synthesis tool.

```
module avg(input logic clk,
           input logic[7:0] signal,
           output logic[7:0] avg);

logic[7:0] h0 = 0, h1 = 0, h2 = 0, h3 = 0;

always_ff @(posedge clk) begin
 h0 <= signal; h1 <= h0; h2 <= h1; h3 <= h2;
end

always_comb begin
 avg = h0 + h1 + h2 + h3;

 // Div by 4 by shifting
 avg[0] = avg[2]; avg[1] = avg[3]; avg[2] = avg[4];
 avg[3] = avg[5]; avg[4] = avg[6]; avg[5] = avg[7];
 avg[6] = 0; avg[7] = 0;
end

endmodule
```

Fig. 1. Example Verilog module

Example module. The Verilog module in Fig. 1, implementing a moving-average filter, serves as a running example in this section. The module utilizes Lutsig's new support for `always_comb` blocks. Sec. V provides more details on Lutsig's Verilog support.

Proving Verilog designs correct. Lutsig is accompanied by a proof-producing Verilog code generator. The code generator is explained in more detail in Sec. VI. In short, the code generator constructs a Verilog module P_{ver} given a HOL embedding P_{HOL} of a Verilog circuit. As the code generator is proof-producing, the code generator enables hardware designers to transport properties proved about the input HOL circuit P_{HOL}, e.g. $P_{\text{HOL}} \models_{L_{\text{HOL}}} \mathit{Spec}$, to the generated Verilog module P_{ver}, i.e. $P_{\text{ver}} \models_{L_{\text{ver}}} \mathit{Spec}$, by simple composition.

The Verilog module in Fig. 1 was in fact generated by the code generator from a HOL circuit. With the help of the code generator, we have proved that, if we by $s[n]$ mean the value of signal s at clock cycle n, the generated Verilog module satisfies the specification (in 8-bit modular arithmetic) $\texttt{avg}[n] = \dfrac{\sum_{i=1}^{4} \texttt{signal}[n-i]}{4}$, i.e., the module is correct.

Going to the netlist level. Now having both a Verilog module (Fig. 1) and a correctness result for the module, we can synthesize a netlist implementation of the module, by invoking Lutsig, and transport the correctness result to the netlist implementation, by composing the Verilog-level correctness result with Lutsig's correctness theorem (i.e., in general notation, derive $P_{\text{nl}} \models_{L_{\text{nl}}} \mathit{Spec}$ from $P_{\text{ver}} \models_{L_{\text{ver}}} \mathit{Spec}$). We discuss Lutsig in more detail in Sec. VII and the functional correctness of Lutsig in Sec. VIII. Since the behavior of the variable `avg` is specified using an `always_comb` block, no register should be generated for the variable; this is further discussed in Sec. IX in the context of the nonfunctional correctness property we have proved about Lutsig.

FPGAs. At this point, our formal development ends. To run the netlist implementation produced by Lutsig on an FPGA, the netlist needs to be placed and routed onto an FPGA chip and then encoded into a bitstream for the chip. In our experiments,

we used the unverified synthesis suite Vivado 2020.2 for these last steps. According to our manual testing, the netlist Lutsig synthesizes for the Verilog module in Fig. 1 runs correctly on top of the FPGA board we used for testing.

V. Formal semantics

In this section we first describe the updated source language of Lutsig (Sec. V-A); that is, we describe the subset of Verilog that Lutsig supports and Lutsig's Verilog semantics L_{ver} for this subset. We then describe the updated target language of Lutsig (Sec. V-B), that is, Lutsig's netlist language.

A. Lutsig's Verilog semantics

In Lutsig, circuits are represented as Verilog modules. A Verilog module, in turn, in Lutsig, consists of:

- a set of input signals (including a clock signal `clk`),
- a set of variables, some marked externally visible,
- a set of **`always_comb`** blocks, and
- a set of **`always_ff`** `@(`**`posedge`** `clk)` blocks.

Lutsig's Verilog semantics is a functional operational semantics that takes the following four inputs:

- a Verilog module m to execute,
- the number of clock cycles n to execute the module,
- a function $\mathit{fext} : \mathbb{N} \to \mathsf{string} \to \mathsf{value}$ modeling snapshots of the nondeterministic world outside the module, and
- a function $\mathit{fbits} : \mathbb{N} \to \mathsf{bool}$ modeling a stream of nondeterministic bits[4].

Since Lutsig's Verilog must be convenient to use in formal reasoning, Lutsig's Verilog is not, in contrast to full Verilog, based on nondeterministic event processing. Since Lutsig targets synchronous designs, the complexities of an event-driven semantics can be fully avoided. Of particular interest is the process-level semantics of Lutsig's Verilog semantics, since the expression-level and statement-level semantics have not been updated for this new version of Lutsig. In short, Lutsig's Verilog semantics for executing one clock cycle is:

- For clock cycle zero, i.e. before the first clock tick, initialize all variables (for a variable without a specified initial value, assign a nondeterministic value) and then run all **`always_comb`** blocks in dependency order.
- For all other clock cycles, run all **`always_ff`** blocks in declaration order followed by all **`always_comb`** blocks in dependency order.

A module's **`always_ff`** blocks are, in Lutsig's Verilog, executed in declaration order since the order of execution does not affect the final result of execution as long as not more than one process writes to the same variable and all writes to variables that are read by processes other than the process making the writes are nonblocking (a type of assignment used for communication between processes in Verilog).

A module's **`always_comb`** blocks are, in Lutsig's Verilog, executed in dependency order since the order of execution *does* matter since blocking writes are used even for variables shared between processes. All **`always_comb`** blocks are sorted before execution by their variable dependencies in the sense that no process writes to a variable that has been read by an earlier process. If the processes cannot be sorted in this way, the semantics aborts with an error. Sorting the processes complicates the semantics, since a sorting algorithm is embedded into the semantics. (We have, however, proved that the algorithm sorts correctly.) The sorting algorithm picks one particular permutation, but users of the semantics should think of it as an arbitrary permutation of the input **`always_comb`** blocks that satisfy the mentioned dependency-order criteria.[5]

Our intention is that Lutsig's non-event-driven Verilog semantics should coincide with the event-driven simulation semantics of full Verilog, as defined by the Verilog standard, as long as good coding style is followed; e.g., as mentioned above, not writing blockingly in an **`always_ff`** block to a variable shared between processes. As part of future work, we plan to formally prove a correspondence between the two semantics to make the relationship between them more precise. Such future semantics work is important for Lutsig when arguing that Lutsig is a *Verilog* synthesis tool, but such work is simultaneously independent of Lutsig in the sense that it would not require Lutsig's implementation and proofs to be updated, as long as the work does not unveil problems in the non-event-driven semantics (and hence requiring us to revisit the semantics).

B. Lutsig's netlist semantics

For this version of Lutsig, to support the compilation of **`always_comb`** blocks, we split netlist registers into two groups: pseudoregisters and real registers. Pseudoregisters are only needed to represent intermediate compilation results – i.e., pseudoregisters are always compiled away before the compilation process is finished. We explain how pseudoregisters are used in the compilation process in Sec. VII. After adding pseudoregisters, a netlist in Lutsig consists of two lists of cells and two lists of registers: one list of cells for the real registers and one list of cells for the pseudoregisters.

There is a formal semantics in functional-operational style associated with Lutsig's netlists. The semantics takes the same kind of arguments as Lutsig's Verilog semantics except a netlist is given rather than a Verilog module. Netlist execution is similar to Lutsig's Verilog execution. First, we define a netlist step to be running all pseudoregister cells, updating all pseudoregisters, and then running all remaining cells. Now, with this terminology in mind, we can describe the full semantics:

- For clock cycle zero, initialize all registers and then do a netlist step.
- For all other clock cycles, update all real registers and then do a netlist step.

[4]See Lööw [14] for a discussion on how X values are treated in Lutsig. We do not repeat the discussion on X values here since such concerns are orthogonal to our current concerns.

[5]Picking one particular permutation rather than an arbitrary permutation simplifies some proofs in the development. But since picking an arbitrary permutation would simplify the user-facing presentation of the semantics, it might be worth revisiting this choice.

It is important that the netlist semantics is simple since the semantics is part of the trusted base of circuits produced with the help of Lutsig. In fact, for netlists without pseudoregisters, such as the final output netlists generated by Lutsig, it is easy to prove that the above semantics collapses into the following clean semantics $L_{\text{nl}'}$:

- For clock cycle zero, initialize all registers and then run all cells.
- For all other clock cycles, update all registers and then run all cells.

VI. The proof-producing Verilog code generator

For this paper, we have extended the proof-producing Verilog code generator bundled with Lutsig with support for translating **`always_comb`** blocks, such that we can prove circuits containing such blocks correct.[6]

The code generator can generate a deeply embedded Verilog circuit given a shallowly embedded Verilog circuit. To shallowly embed a Verilog circuit means to express it as a HOL function (i.e., a functional program). Shallowly embedded circuits are convenient to work with since HOL4 has well-developed infrastructure for reasoning about functional programs. The code generator is an SML function which is proof-producing in the sense that it, for every run, proves a HOL theorem (using the HOL4 API) ensuring that the input circuit and output circuit have the same behavior.

Since the input language to the code generator is Verilog, although shallowly embedded, there is no need to provide a new set of hardware-modeling idioms (i.e., a new synthesis semantics) for the input language. In other words, the input circuits should be seen as Verilog circuits, and, when shallowly embedding Verilog circuits, according to the style the code generator expects, the hardware developer should think of themselves as doing Verilog development.

The code generator assumes that circuits are embedded in the style we now describe. Verilog processes must be embedded as next-state functions over (module-specific) state records. For each process, the generated Verilog code closely mirrors the given input HOL function. E.g., recall that the **`always_ff`** block in the Verilog module in Fig. 1 is simply "`h0 <= signal; h1 <= h0; h2 <= h1; h3 <= h2;`"; the next-state function the block is generated from is:

$$\begin{array}{l}\mathsf{avg\_ff}\ \mathit{fext}\ s\ s' \stackrel{\text{def}}{=} \mathsf{let}\\ \quad s' = s'\ \mathsf{with\ h0} := \mathit{fext}.\mathsf{signal};\\ \quad s' = s'\ \mathsf{with\ h1} := s.\mathsf{h0};\\ \quad s' = s'\ \mathsf{with\ h2} := s.\mathsf{h1}\ \mathsf{in}\\ \quad s'\ \mathsf{with\ h3} := s.\mathsf{h2}\end{array}$$

Note how field updates are translated to assignments in Verilog in a straightforward manner (the syntax r with f $:=$ v means that field f of record r is updated to value v). Also note how two state records s and s' are passed around; these two state records are the basis of the nonblocking-assignments embedding style used. The record s contains the values of all variables at the start of the current clock cycle, and the record s' contains the current values of all variables. To see why both records are needed, consider e.g. the assignments to `h0` and `h1` in the generated **`always_ff`** block: since the assignment to `h0` is nonblocking, the updated value of `h0` is not available until the next clock cycle, and the HOL embedding of the `h1` assignment must therefore read the value of `h0` from the s record (not the s' record) to model Verilog's semantics correctly.

The rest of the HOL circuit embedding style closely mirrors Lutsig's Verilog semantics. First, there is a function

$$\begin{array}{l}\mathsf{procs}\ []\ \mathit{fext}\ s\ s' \stackrel{\text{def}}{=} s'\\ \mathsf{procs}\ (p{::}ps)\ \mathit{fext}\ s\ s' \stackrel{\text{def}}{=} \mathsf{procs}\ ps\ \mathit{fext}\ s\ (p\ \mathit{fext}\ s\ s')\end{array}$$

for combining a list of next-state functions into one single next-state function. The function allows for building one next-state function for all **`always_ff`** blocks in the module and one next-state function for all **`always_comb`** blocks. One important caveat is that the **`always_comb`** blocks must be provided in dependency order, otherwise the HOL circuit will not correctly mirror Lutsig's Verilog semantics since Lutsig's Verilog semantics sorts all **`always_comb`** blocks by dependency before execution. The resulting two next-state functions formed by composing all **`always_ff`** blocks and **`always_comb`** blocks, respectively, using procs, can then be given to the following function, also mirroring Lutsig's Verilog semantics, to build a full circuit:

$$\begin{array}{l}\mathsf{mk\_circuit}\ \mathit{sstep}\ \mathit{cstep}\ s\ \mathit{fext}\ 0 \stackrel{\text{def}}{=} \mathit{cstep}\ (\mathit{fext}\ 0)\ s\ s\\ \mathsf{mk\_circuit}\ \mathit{sstep}\ \mathit{cstep}\ s\ \mathit{fext}\ (\mathsf{Suc}\ n) \stackrel{\text{def}}{=} \mathsf{let}\\ \quad s = \mathsf{mk\_circuit}\ \mathit{sstep}\ \mathit{cstep}\ s\ \mathit{fext}\ n;\\ \quad s = \mathit{sstep}\ (\mathit{fext}\ n)\ s\ s\ \mathsf{in}\\ \quad \mathit{cstep}\ (\mathit{fext}\ (\mathsf{Suc}\ n))\ s\ s\end{array}$$

E.g., the HOL representation of the Verilog module in Fig. 1 is mk_circuit (procs [avg_ff]) (procs [avg_comb]).

Lastly, one more level of encoding is needed to handle variable initialization, which is simple and we do not detail here.

VII. Lutsig

We now discuss Lutsig's new support for **`always_comb`** blocks. To simultaneously honor both Verilog's simulation semantics and Verilog's synthesis semantics – in this paper, specifically, for the latter, the modeling idiom that **`always_comb`** blocks must always be mapped to combinational logic – Lutsig must take on the responsibility to abort if the two semantics differ in what semantics they assign to some **`always_comb`** block in a given design. In this section, we discuss how Lutsig implements this responsibility. In Sec. VIII, we show that Lutsig successfully achieves its responsibility towards Verilog's simulation semantics, by presenting a theorem stating that Lutsig is semantics preserving with respect to Lutsig's formalization of Verilog's simulation semantics. In Sec. IX, we show that Lutsig successfully achieves its responsibility towards Verilog's synthesis semantics (for **`always_comb`** blocks), by presenting a theorem stating that **`always_comb`** blocks are never be mapped to registers (or other stateful constructs).

[6]Unrelatedly, we have also changed how nonblocking assignments are shallowly embedded, such that a larger set of Verilog designs can be embedded.

Concretely, the above responsibility boils down to ensuring that there is no sequential logic inside any `always_comb` block. This is where pseudoregisters come in: all variables written to by an `always_comb` block are mapped to pseudoregisters, and all other variables are mapped to real registers. All pseudoregisters must then be compiled away before the synthesis process is over, otherwise Lutsig aborts with an error.

A. Variable-level and element-level analysis

To keep the implementation of Lutsig simple, the decision whether to map a variable to a pseudoregister or a real register is done on the level of variables. E.g., all elements of an array variable are either all mapped to pseudoregisters or to real registers. In full Verilog, the analysis is instead based on longest static prefixes [30, p. 282]. Such more fine-grained analysis allows for different parts of an array to be mapped to different kinds of logic, which could possibly be practically useful, but would clutter the solution presented here without providing additional insight.

Note, however, that some amount of element-level analysis is still needed. E.g., consider a module containing only one variable `a` with type `logic[1:0]` and the following block:

```
always_comb begin
  a[0] = inp0;
  a[1] = inp1;
end
```

The block represents combinational logic since all elements of the array are assigned. But if one of the assignments would have been left out, then the block would not represent combinational logic. Hence, an analysis on the element level cannot be fully avoided.

B. Lutsig's synthesis passes

In Lutsig, pseudoregisters are removed at a late stage in the synthesis pipeline. The following pipeline passes in Lutsig are important for our discussion here:

SYNT	Synthesize the given Verilog design to a netlist
REM	Remove unused registers (variable-level analysis)
DET	Remove all nondeterminism from the netlist
MAP	Compile and technology-map away array cells
REM	Remove unused registers (element-level analysis)

Pseudoregisters are introduced in SYNT and not removed until MAP. Since MAP is done on the element level (rather than the variable level as the passes before it), it was natural to place the removal of pseudoregisters there. The downside of this approach is that we had to update *all* intermediate passes of Lutsig, such as REM and DET, to handle the more complex netlist semantics with pseudoregisters. (Note that REM is run twice, which we motivate in the next section.)

C. Problems in compiling combinational logic

We now highlight how Lutsig handles some of the problems related to compiling combinational logic. Our presentation is example driven and many of the examples relate to detecting simulation-and-synthesis mismatches. It is important to consider not only designs that are rejected by Lutsig but also designs that are accepted, since compiler-correctness theorems like Lutsig's (of the form *Comp* $P_S = \texttt{OK}\ P_T \implies P_S \approx P_T$) do not protect against compiler bugs that cause compilers to fail on valid input code (i.e., bugs causing the compiler to return `Error` when it should have returned `OK`). To exemplify, consider the extreme case of a compiler that always returns `Error`: such a compiler is vacuously correct, but, of course, not particularly useful.

1) Combinational logic in `always_ff` blocks: Code inside `always_comb` blocks must always represent combinational logic only, but code inside `always_ff` blocks can represent both combinational and sequential logic. E.g., consider a module consisting of three variables `a`, `b`, and `c` with type `logic[1:0]` with one single block:

```
always_ff @(posedge clk) begin
  a = inp0;
  b[0] = inp1;
  b[1] = inp2;
  c <= a + b;
end
```

Such code should not generate registers for `a` and `b` since those registers would never be read. REM makes sure the registers for `a` and `b` generated by SYNT are optimized away before the synthesis process is over. REM is run twice since we want to catch easy cases (such as `a` in the example) early but at the same time also make sure to catch cases requiring element-level analysis (such as `b` in the example).

2) Sequential logic in `always_comb` blocks: Lutsig must check that all `always_comb` blocks actually model combinational logic. E.g., Lutsig must reject the following block:

```
always_comb a = a + 1;
```

For this paper, we have extended MAP to handle this.

MAP handles the compilation of netlist-level array constructs such as array cells and array registers, by mapping them to array constructs natively available or to Boolean subcircuits. MAP is centered around a map σ from cell inputs to lists of "marked" cell inputs. MAP visits all netlist cells in order and the map σ is updated as the netlist is visited to keep track of mapped cells. For real registers, all inputs are marked legal from the start of compilation. For pseudoregisters, all inputs are initially marked as illegal inputs. If an illegal input is referenced during compilation (i.e. the (relevant part of the) σ entry for the cell input is marked illegal), the compilation is aborted.

We now consider two examples. First, note that the reference to `a` on the right-hand side in the above `always_comb` block will cause the compilation to abort. Now, instead consider the following Verilog code exemplifying code Lutsig accepts (although note that the illustration is done on the Verilog level rather than on the netlist level that MAP is actually run at):

```
always_comb begin
  // since b is a pseudoregister, we have:
  // sigma(b) = [illegal, illegal]

  b[0] = inp0; // sigma(b) = [illegal, inp0]
  b[1] = inp1; // sigma(b) = [inp1, inp0]

  // we can read the full b here since all
  // elements of b are legal
```

```
  b = b + 1;
end
```

Note that since nonsynthesizable code is rejected by Lutsig, it is not important what semantics Lutsig's Verilog semantics assigns to nonsynthesizable code. For some nonsynthesizable code, Lutsig's semantics diverges from Verilog's simulation semantics. E.g., recall that all blocks are unconditionally executed each clock cycle in Lutsig's semantics. In contrast, in Verilog's simulation semantics, `always_comb` blocks are only executed when something they depend on is updated. But since combinational logic is idempotent – that is, we can execute it multiple times without affecting the result – executing the same `always_comb` multiple times is harmless. However, if the `always_comb` block does not actually model combinational logic, this reasoning does not hold, and the two semantics might diverge.

3) Intrablock order problems: Recall the `andor1b` module with "mis-ordered" assignments discussed in Sec. III. The σ-based MAP pass also handles such code correctly. E.g., Lutsig rejects the following code with the same problem:

```
always_comb begin
  b = a + 1; // sigma(a) says a illegal here!
  a = inp;
end
```

4) Interblock order problems: Recall that Lutsig's non-event-based Verilog semantics sorts `always_comb` blocks before execution (see Sec. V). E.g., to assign sensible semantics to the following code, the order of the blocks needs to be reversed before execution:

```
always_comb b = a + 1;
always_comb a = inp;
```

The same order problem occurs in compilation: To compile the above code correctly, Lutsig must first sort the `always_comb` blocks by their dependencies. To sort, Lutsig uses the same sorting algorithm as used in Lutsig's Verilog semantics.

Not all processes can be ordered by their dependencies. Since combinational logic must not include combinational loops, the sorting algorithm used in Lutsig rejects code containing circular dependencies like the following:

```
always_comb a = b + 1;
always_comb b = a + 1;
```

5) If statements: Lutsig handles if statements correctly. E.g. the following code is rejected:

```
always_comb
  if (c)
    a = inp;
//else
// a = 'x;
```

If instead the else branch is uncommented, then Lutsig synthesizes the code successfully. The original block without an else branch gets stuck in the synthesis process since SYNT generates a mux with `inp` and the pseudoregister generated for `a` as inputs and MAP eventually detects that a pseudoregister is referenced and aborts the synthesis process.

6) Case statements and nested if statements: Compiling case statements is similar to compiling if statements: if a variable is assigned in one branch, then it must be assigned in all other branches as well. Let the variable `c` have type `logic[1:0]` and consider the following code:

```
always_comb
  case (c)
    2'b00: a = 1;
    2'b01: a = 4;
    2'b10: a = 1;
    2'b11: a = 2;
//default: a = 'x;
  endcase
```

A sufficiently smart synthesis tool would realize that `a` is assigned for all possible values of `c`. However, Lutsig's synthesis algorithm is not smart and requires the commented-out `default` branch above to realize that all cases are covered. The same holds for the analogous situation with nested if statements. In fact, Lutsig handles case statements by expanding them to nested if statements, so Lutsig's limited case statement handling is a consequence of Lutsig's limited if statement handling.

VIII. Functional Correctness of Lutsig

We now state Lutsig's functional-correctness theorem, thereby showing that Lutsig successfully abides by (its formalization of) Verilog's simulation semantics. The theorem statement is the same as in the previous version of Lutsig; the HOL4 proof of the theorem, however, has been updated to take into account the new functionality added in this paper. If we let $P \Downarrow_L^{n,fbits} S$ denote that design P's externally visible state is S under the semantics L after n clock cycles with nondeterminism source $fbits$, then Lutsig's correctness theorem is as follows:

$$\begin{aligned}&\texttt{Lutsig}\ P_{\text{ver}} = \texttt{OK}\ P_{\text{nl}} \implies \\ &\exists S_{\text{nl}}, P_{\text{nl}} \Downarrow_{L_{\text{nl}'}}^{n,fbits} S_{\text{nl}} \wedge \\ &\exists fbits', P_{\text{ver}} \Downarrow_{L_{\text{ver}}}^{n,fbits'} S_{\text{ver}} \implies S_{\text{nl}} = S_{\text{ver}}\end{aligned}$$

Per the usual convention, all free variables in the theorem are implicitly universality quantified. Note that since the netlist P_{nl} in the theorem statement never contains pseudoregisters, we can use the simplified netlist semantics $L_{\text{nl}'}$ which does not handle pseudoregisters.

Although the theorem statement is more complex than straightforward backward simulation as presented in Sec. II-A, the theorem still allows for theorem transportation from the Verilog level down to the netlist level by simple composition (i.e., VPD): Given a circuit-correctness theorem stating that a Verilog module P_{ver} never crashes (regardless of what $fbits$ is supplied), say $\exists S_{\text{ver}}, P_{\text{ver}} \Downarrow_{L_{\text{ver}}}^{n,fbits} S_{\text{ver}} \wedge Spec\ S_{\text{ver}}$ for some specification $Spec$, if Lutsig successfully synthesize P_{ver} to a netlist P_{nl}, then we can easily derive $\exists S_{\text{nl}}, P_{\text{nl}} \Downarrow_{L_{\text{nl}'}}^{n,fbits} S_{\text{nl}} \wedge Spec\ S_{\text{nl}}$.

IX. Nonfunctional Correctness of Lutsig

We now turn to the nonfunctional correctness of Lutsig. Recall that Verilog's synthesis semantics enables hardware designers to express hardware design ideas to their synthesis tool through modeling idioms. The theorem presented in this section, which we have proved in HOL4, shows that Lutsig correctly handles `always_comb` blocks in the sense that the theorem captures the modeling idiom that `always_comb` blocks must be mapped to combinational logic [30, p. 207].

We formalize this modeling idiom as follows: for any run $\texttt{Lutsig}\ P_{\text{ver}} = \texttt{OK}\ P_{\text{nl}}$, if a variable is written to in an `always_comb` block in P_{ver}, then no register with the same name as the variable will be included in P_{nl}. Formally, the theorem is as follows:

$$\texttt{Lutsig}\ P_{\text{ver}} = \texttt{OK}\ P_{\text{nl}} \implies$$
$$\forall var, var \in \texttt{comb\_vars}\ P_{\text{ver}} \implies var \notin \texttt{regs}\ P_{\text{nl}}$$

Note that the theorem relates concepts in the input design P_{ver} (writes) to concepts in the final netlist P_{nl} (registers) – this means that we must, in our proofs, carry information from the very first compilation phase down to the very last.[7]

X. Conclusion

We now conclude. In our discussion on the relationships between Verilog's simulation semantics, Verilog's synthesis semantics, VPD, and TVD, we identify Verilog's modeling idioms as the core cause of tensions between VPD and TVD. To put our discussion to test, we have added support for `always_comb` blocks to the verified synthesis tool Lutsig.

Our discussion on VPD and TVD paves the way for further Lutsig extensions that add support for Verilog constructs associated with simulation-and-synthesis mismatches, such as support for BRAM inference.

Another interesting direction for future work to explore is how a more detailed hardware semantics would affect the `always_comb` discussion. In this paper our Verilog semantics is at the level of cycle-by-cycle behavior – what are the alternatives for a more detailed hardware semantics that, while at the same time as keeping source-level reasoning feasible, allow us to turn the nonfunctional property we have proved in this paper into a part of the compiler's functional correctness theorem?

Lastly, no approach to hardware development, regardless of hardware language used, completely shields the hardware designer from the synthesis aspects we have discussed in this paper. It would therefore be interesting to consider how much of our discussion on VPD and TVD translates into hardware development and synthesis-tool verification for other hardware languages. The questions we raise in this paper will reappear in similar form regardless of the hardware language used. After all, not even so-called high-level synthesis (HLS), i.e., generating hardware from software languages like C, can completely hide the synthesis process from hardware developers. E.g., the manual [38, p. 17] for Vitis, an unverified HLS tool for C, C++, and OpenCL, states that "arbitrary, off-the-shelf software cannot be efficiently converted into hardware" and that, moreover, "even if [a] software program can be automatically converted (or synthesized) into hardware, achieving acceptable quality of results, will require additional work such as rewriting the software to help the HLS tool achieve the desired performance goals." The pessimism of the manual [38, p. 28] continues: "Software written for CPUs and software written for FPGAs is fundamentally different. You cannot write code that is portable between CPU and FPGA platforms without sacrificing performance." To prepare its readers for hardware development using Vitis, the manual informs its readers what they need to know about the Vitis synthesis process to design efficient hardware; in other words, the HLS hardware designer, much like the Verilog hardware designer, must be aware of how to control their synthesis tool and how to communicate to their synthesis tool what kind of hardware they want. In total, the Vitis manual is 660 pages, reflecting the fact that not even HLS manages to abstract away the complexities of synthesis.

[7] Before we started working on the proof, Lutsig did not actually satisfy our formalization of the `always_comb` modeling idiom. This was because the SYNT pass (see Sec. VII) used the presence of writes *in the design that was given to that pass* to decide which variables to map to real registers and which to pseudoregisters rather than the presence of writes *in the design as given by the user* (i.e., P_{ver} in the above theorem) – the former does not reliably track the latter since writes may be optimized away in the compilation process!

Acknowledgment

We thank Magnus Myreen, Adam Chlipala, David Greaves, Tom Melham, Warren Hunt, Koen Claessen, Wolfgang Ahrendt, Philippa Gardner, and Kashish Raimalani for comments on drafts of this paper. This work was supported by the Swedish Foundation for Strategic Research.

References

[1] X. Leroy, "A formally verified compiler back-end," *Journal of Automated Reasoning*, vol. 43, no. 4, 2009.

[2] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: A verified implementation of ML," in *Principles of Programming Languages (POPL)*, 2014.

[3] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell, "CompCertTSO: A verified compiler for relaxed-memory concurrency," *Journal of the ACM*, vol. 60, no. 3, 2013.

[4] A. Barrière, S. Blazy, O. Flückiger, D. Pichardie, and J. Vitek, "Formally verified speculation and deoptimization in a JIT compiler," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, 2021.

[5] M. O. Myreen, "Verified just-in-time compiler on x86," in *Symposium on Principles of Programming Languages (POPL)*, 2010.

[6] ——, "A minimalistic verified bootstrapped compiler (proof pearl)," in *Conference on Certified Programs and Proofs (CPP)*, 2021.

[7] D. Patterson and A. Ahmed, "The next 700 compiler correctness theorems (functional pearl)," *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, 2019.

[8] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu, "Formal verification of a constant-time preserving C compiler," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, 2019.

[9] A. Gómez-Londoño, J. Åman Pohjola, H. T. Syeda, M. O. Myreen, and Y. K. Tan, "Do you have space for dessert? A verified space cost semantics for CakeML programs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, 2020.

[10] R. M. Amadio, N. Ayache, F. Bobot, J. P. Boender, B. Campbell, I. Garnier, A. Madet, J. McKinna, D. P. Mulligan, M. Piccolo, R. Pollack, Y. Régis-Gianas, C. Sacerdoti Coen, I. Stark, and P. Tranquilli, "Certified complexity (CerCo)," in *Foundational and Practical Aspects of Resource Analysis (FOPARA)*, 2014.

[11] Z. Paraskevopoulou and A. W. Appel, "Closure conversion is safe for space," *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, 2019.

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

[13] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind, "The essence of Bluespec: A core language for rule-based hardware design," in *Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[14] A. Lööw, "Lutsig: A verified Verilog compiler for verified circuit development," in *Conference on Certified Programs and Proofs (CPP)*, 2021.

[15] Y. Herklotz, J. D. Pollard, N. Ramanathan, and J. Wickerson, "Formal verification of high-level synthesis," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, 2021.

[16] F. Schuiki, A. Kurth, T. Grosser, and L. Benini, "LLHD: A multi-level intermediate representation for hardware description languages," in *Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[17] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations," in *International Conference on Computer-Aided Design (ICCAD)*, 2017.

[18] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *Annual Design Automation Conference (DAC)*, 2012.

[19] R. Nikhil, "Bluespec SystemVerilog: Efficient, correct RTL from high-level specifications," in *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, 2004.

[20] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in Haskell," in *International Conference on Functional Programming (ICFP)*, 1998.

[21] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "Cλash: Structural descriptions of synchronous hardware using Haskell," in *Euromicro Conference on Digital System Design*, 2010.

[22] L. Vega, J. McMahan, A. Sampson, D. Grossman, and L. Ceze, "Reticle: A virtual machine for programming modern FPGAs," in *Conference on Programming Language Design and Implementation (PLDI)*, 2021.

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

[24] W. L. Harrison, I. Graves, A. Procter, M. Becchi, and G. Allwein, "A programming model for reconfigurable computing based in functional concurrency," in *International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2016.

[25] R. Nigam, S. Atapattu, S. Thomas, Z. Li, T. Bauer, Y. Ye, A. Koti, A. Sampson, and Z. Zhang, "Predictable accelerator design with time-sensitive affine types," in *Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[26] M. Christensen, T. Sherwood, J. Balkind, and B. Hardekopf, "Wire sorts: A language abstraction for safe hardware composition," in *Conference on Programming Language Design and Implementation (PLDI)*, 2021.

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

[28] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed. Pearson, 2011.

[29] *Vivado Design Suite User Guide: Synthesis (UG901, v2020.2)*, Xilinx, 2021.

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

[31] S. Sutherland and D. Mills, *Verilog and SystemVerilog Gotchas: 101 Common Coding Errors and How to Avoid Them*. Springer, 2007.

[32] "Verilog register transfer level synthesis," *IEEE Std 62142-2005*, 2005.

[33] *Intel Quartus Prime Pro Edition User Guide: Design Recommendations (UG-20131, v21.1)*, Intel, 2021.

[34] "IEEE standard for Verilog hardware description language," *IEEE Std 1364-2001*, 2001.

[35] L. Simon, D. Chisnall, and R. Anderson, "What you get is what you C: Controlling side effects in mainstream C compilers," in *European Symposium on Security and Privacy (EuroS&P)*, 2018.

[36] C. Hathhorn, C. Ellison, and G. Roşu, "Defining the undefinedness of C," in *Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[37] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell, "Into the depths of C: Elaborating the de facto standards," in *Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[38] *Vitis High-Level Synthesis User Guide (UG1399, v2021.1)*, Xilinx, 2021.