Lutsig: A Verified Verilog Compiler for Verified Circuit Development

Andreas Lööw
Chalmers University of Technology
Gothenburg, Sweden

Abstract

We report on a new verified Verilog compiler called Lutsig. Lutsig currently targets (a class of) FPGAs and is capable of producing technology mapped netlists for FPGAs. We have connected Lutsig to existing Verilog development tools, and in this paper we show how Lutsig, as a consequence of this connection, fits into a hardware development methodology for verified circuits in the HOL4 theorem prover. One important step in the methodology is transporting properties proved at the behavioral Verilog level down to technology mapped netlists, and Lutsig is the component in the methodology that enables such transportation.

CCS Concepts: • **Hardware → Hardware description languages and compilation**; **Logic synthesis**; **Theorem proving and SAT solving**; *Methodologies for EDA*.

Keywords: hardware synthesis, compiler verification, hardware verification

ACM Reference Format:
Andreas Lööw. 2021. Lutsig: A Verified Verilog Compiler for Verified Circuit Development. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21), January 18–19, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3437992.3439916

1 Introduction

We envision a future where hardware development can be carried out entirely inside an interactive theorem prover (ITP). As a step towards this future, we present a methodology for the development of correct hardware artifacts and provide the tools needed for the methodology.

First, to motivate the methodology, we take a short detour into the software world. In today's formal methods ecosystem for software development, we find tools for the following development methodology: (i) prove a correctness theorem about your program at the source level, (ii) use a verified compiler to transform your program to machine code, and, lastly, (iii) transport the source-level program correctness theorem down to the generated machine code by composing the source-level program correctness theorem with the compiler correctness theorem. When carried out inside an ITP, the development methodology is capable of producing artifacts with remarkably small trusted computing bases (TCBs) [20]. For example, the verified CakeML compiler [35] and its accompanying formal methods tools hosts such a development methodology inside the ITP HOL4 [32]. To put trust in the correctness of software produced inside an ITP according to the methodology, users need only to trust the correctness specification used in the program correctness theorem, that the formalization of the target machine's instruction set architecture (ISA) used to model the behavior of the machine code accurately captures the actual behavior of the target machine, and the ITP itself.

Returning back to the hardware world, we believe the above development methodology is equally useful when applied to hardware as when applied to software. When applied to the hardware, the methodology enables the production of hardware artifacts with the same TCBs as the software TCBs outlined above except that we will have to trust a formalization of a model of hardware instead of a formalization of a target machine's ISA (i.e., a model of a target machine). In the hardware world, however, no toolchain for carrying out hardware development entirely inside an ITP exist today. Instead, hardware development must be carried out by connecting together multiple (unverified) tools, resulting in a much larger TCB: Also individual tools and intermediate formalisms and languages need to be trusted.

In this paper, we describe a new compiler, called Lutsig, for the hardware description language (HDL) Verilog that we have verified using the ITP HOL4 [32]. The compiler targets technology mapped netlists. As a result, for the first time, the above development methodology can be carried out in the hardware world down to technology mapped netlists inside an ITP. This improves the state-of-the-art (Sec. 8), but does not (yet) allow us to carry out all of hardware development inside an ITP as compilation steps following technology mapping still have to be carried out outside HOL4 (Sec. 3). Specifically, we make the following contributions:

CPP '21, January 18–19, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8299-1/21/01…$15.00
https://doi.org/10.1145/3437992.3439916

- we develop and describe a verified compiler from a subset of Verilog, by far the most widely used HDL today [11], down to netlists, and a hybrid verified translation-validation-based technology mapper for (a class of) FPGAs, together forming our compiler Lutsig;
- we connect the compiler to existing development tools for proving Verilog circuits correct [23]; and
- we show that the compiler and the connected Verilog development tools can host the above small TCB development methodology – in particular, we show how to map correctness theorems proved at the Verilog level down to the technology mapped netlist level.

All source code and proofs are available at https://github.com/CakeML/hardware.

2 Why Existing Approaches to Hardware Development Are Insufficient

This section further motivates moving hardware development inside an ITP by further outlining problems with today's non-ITP methodologies. We again focus on transporting theorems from the source level down to some target level.

One problem for non-ITP development methodologies we highlight is the problem of individual correctness of the compilation tools used. But the main problem for non-ITP development methodologies is that they do not provide a way to avoid intermediate compilation steps ending up in the TCB.

Individual tool correctness. Of course, it is important that each tool involved in the compilation is correct. Today's unverified tools, however, contain bugs [16]. This problem can to some extent be addressed in non-ITP methodologies by relying on translation validation [30] (or as it is known in the hardware world, logical equivalence checking or formal equivalence checking). A translation validation tool for a compilation tool takes the input given to the compilation tool and the output produced by the compilation tool and finds an equivalence proof between the input and the output. This means that we no longer need to trust the compilation tool to be correct. Instead, we only need to trust the translation validation tool (or its associated proof checker, if such a checker is available). However, it is not enough that each individual tool functions correctly when we want to transport a source-level correctness theorem down to our target level; the tools and the correctness theorem must also "fit together", and this problem is not addressed by translation validation. We consider this problem next.

Intermediate compilation steps in the TCB. When transporting correctness theorems from the source level down to our target level, we will pass by many different representations and tools on our way. For the transportation to succeed, these tools and our correctness theorem must fit together – they must be composable. In an ITP setting, if the transportation succeeds, then, when all steps have been composed together, intermediate steps will have been removed out of the TCB. This is not the case in a non-ITP setting.

One composition problem we will face is that the prover we used to prove the source-level correctness theorem we want to transport and the compilation tools in use might (subtly) differ in how they interpret the HDL we have implemented our circuit in. If we do not check our compositions mechanically, which is not done in today's methodologies, bugs stemming from composition problems might go unnoticed. This problem is particularly important in hardware development, because today's two most used HDLs, Verilog and VHDL, are infamous for their gotchas and idiosyncrasies (for Verilog, see e.g. Sutherland and Mills [34]). To address this, instead of checking compositions mechanically, numerous attempts at designing new HDLs have resulted in a small ecosystem of HDLs meant to replace or supplement Verilog and VHDL, such as e.g. Lava [6], Bluespec [27], and Chisel [5] (see also Gammie [12]). Using a well-designed language instead of Verilog and VHDL shrinks the TCB, but the replacement language still contributes to the TCB. In other words, replacing Verilog and VHDL improves the situation but does not resolve the situation entirely. Moving hardware development inside an ITP, on the other hand, completely eliminates the language used to express the source-level circuit from the TCB. As a result, from a TCB perspective, the choice of language does not matter.

A similar composition problem occurs when composing tools together into a compilation chain down to the abstraction level we are targeting. The tools must communicate with each other, and if the languages used for communication are interpreted differently by the tools there is a risk of bugs being introduced in the compilation process. One can (also here) introduce new, supposedly well-designed, languages for communication between tools to address this problem. New such languages include LLHD [31] and FIRRTL [19]. Indeed, for communication between tools, compared to Verilog and VHDL, (e.g.) "LLHD's simplicity offers a much smaller 'surface for implementation errors'" [31]. In contrast, moving hardware development inside an ITP renders communication language choice unimportant from a TCB perspective – the move eliminates the "surface for implementation errors" completely. Again, improving the languages involved can shrink the TCB, but still leaves the languages in the TCB rather than removing them from the TCB.

3 Compiler Overview

This section gives an overview of Lutsig's compilation passes and shows how Lutsig fits into the ITP development methodology described in the introduction of this paper.

Fig. 1 shows a compilation chain from HOL circuits down to FPGA bitstreams we have made Lutsig part of. In this chain, the compilation from HOL circuits (A) to Verilog circuits (B) is handled by the proof-producing Verilog translator

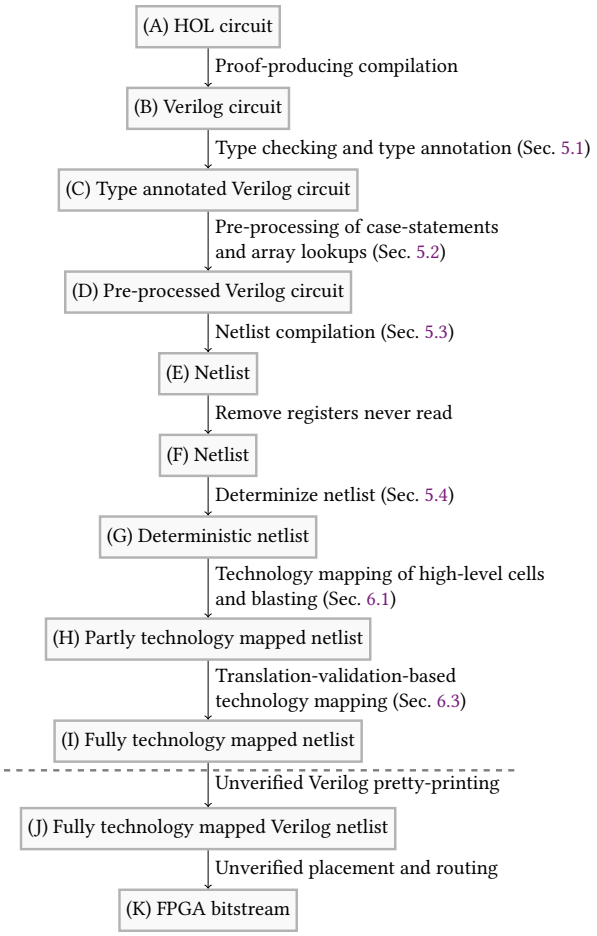

Figure 1. An overview of Lutsig's compilation passes and illustration of how Lutsig fits into circuit development.

described in Lööw and Myreen [23]. We chose to use this translator because it was simple to connect to Lutsig because Lutsig's Verilog semantics is based on the Verilog semantics used by the translator. However, we should keep in mind that this connection just illustrates one particular use of Lutsig. Lutsig is not limited to using this particular translator as its front-end; combining Lutsig with any front-end tool capable of somehow providing HOL4 correctness proofs for circuits implemented in terms of Lutsig's Verilog semantics gives us a toolchain for hosting the ITP development methodology we are interested in here.[1]

Continuing down the compilation chain, regardless of front-end used, Lutsig handles the compilation of Verilog circuits (B) down to technology mapped netlists (I). All steps between (B) and (I) are verified except the last step (i.e., (H) to (I)), which is instead based on translation validation. The compilation steps below the dotted line in Fig. 1 are carried out outside the formal development. That is, we currently rely on (unverified) external Verilog-based tools to handle the last stages of compilation; in particular placement and routing, but also (among others) clocking and details such as encoding the compilation result as an FPGA bitstream. Moving those stages of compilation into HOL4 is left as future work since we expect an approach similar to the translation validation approach taken in the second part of Lutsig's technology mapper (i.e., the step from (H) to (I)) to be applicable for those compilation steps as well – we have, however, not investigated this in detail yet. In terms of TCB, this means that we are not yet fully independent of Verilog.

In Sec. 7, to show how the suggested compilation chain of Fig. 1 works in practice, we present a case study following the setup. But first, we describe the different components of Lutsig in the coming sections.

4 Source Language and Target Language

As a first step towards describing the compiler, we describe the source language and target language of the compiler.

The source language of Lutsig is a subset of Verilog. The Verilog semantics used is based on earlier work on Verilog semantics by Lööw and Myreen [23]. In the earlier work, it was important that the Verilog semantics soundly captured the Verilog standard [4]. For this paper, sound capture of the standard is not important for circuit correctness results, as the Verilog semantics is not part of the TCB of circuits developed according to the development methodology we follow. However, faithfulness to the standard is important in order to be able to call the compiler a Verilog compiler.

The target language of Lutsig is a simple custom language for netlists consisting of lookup tables (LUTs), cells for arithmetic hardware found in the class of FPGAs we target, and registers. The same netlist language is used for representing intermediate circuits during compilation; intermediate circuits are expressed in terms of high-level cells that are later mapped to the final target cell set.

4.1 Source Language: Verilog

Lutsig supports the subset of Verilog described in Fig. 2. The syntax is animated by a functional big-step operational semantics [28] designed with the aim of being a sound simplification of the simulation semantics provided by the Verilog standard [4]. The semantics is based on previous work [23], which should be consulted for details on the semantics.[2] The syntax and semantics is, since this paper, accompanied by a no-frills type system.

[1] Of course, Lutsig can also be used in combination with a Verilog front-end tool not capable of providing HOL4 proofs – but the combination of such a tool with Lutsig does not give us a solution to the problem of removing intermediate compilation steps out of the TCB as outlined in Sec. 2. Nevertheless, if we simply want to use Lutsig as a trustworthy Verilog compiler without transporting proofs, then a simple non-verified Verilog parser would suffice as a front-end.

[2] The same scope limitations as described in Lööw and Myreen [23] still hold, in particular we do not consider Verilog's implicit resizing of arrays (which would be simple but uninteresting to support).

c	::=	$b \mid [b]$	for Boolean b
t	::=	`logic` | `logic[`n`]`	for $n \in \mathbb{N}$
op	::=	`&&` | `||` | `==` | `+`	
e	::=	c	literal constant
	|	v	variable
	|	v`[`e`]`	array indexing
	|	$\neg e$	unary not
	|	e op e	binary operator
s	::=	s `;` s	sequencing
	|	`if` e `then` s `else` s	if-statement
	|	`case` e [e `:` s] s? `endcase`	case-statement
	|	e = e	blocking assignment
	|	e `<=` e	non-blocking assignment
	|	e `= X`	blocking `X` assignment
	|	e `<= X`	non-blocking `X` assignment
d	::=	t v = c	
	|	t v `= X`	
p	::=	`always_ff @(posedge clk)` s	
m	::=	`module` $[(v, t)]$ $[d]$ $[p]$	

Figure 2. Verilog values c, Verilog types t, expressions e, statements s, variable declarations d, processes p, and modules m. The notation $[x]$ denotes a list of xs, and x? denotes an optional x.

In Lutsig's Verilog semantics, the top-level construct is a module `module` $[(v, t)]$ $[d]$ $[p]$ consisting of type declarations for inputs $[(v, t)]$, variable declarations $[d]$, and processes $[p]$. The semantics of a module is given by a function run $fext$ $fbits$ (Module $exttys$ $decls$ ps) n expressed in a sum monad where errors are represented by Inl and success by Inr. On success, run returns an environment with the variables in $decls$. In the semantics, non-determinism is modeled using the two functions $fext$ and $fbits$ and quantification in theorems where the semantics is used, see Lutsig's correctness theorem in Sec. 6.2 for an example. The function $fext : \mathbb{N} \to \text{string} \to \text{error} + \text{value}$ represents the world outside the circuit and maps clock cycles to states of the external world. The variables read from snapshots of the external world must be typed according to $exttys$. The function $fbits : \mathbb{N} \to \text{bool}$ represents an infinite stream of non-deterministic bits and is used to give semantics to non-deterministic constructs in the language (described below). Executing the semantics consists of initializing all variables according to $decls$ and then running the processes ps for n clock cycles. The Verilog standard allows for processes to be interleaved non-deterministically. In the compiler, we did not find a use for the additional optimization freedom such non-deterministic interleavings offer, and consequently, a clock cycle in the semantics consist of executing processes sequentially in declaration order.

Verilog processes consist of statements s and expressions e, and they in turn mostly consist of the usual imperative-language constructs. We highlight two constructs that stand out among the crowd of otherwise usual constructs. The first construct we highlight is `X` assignments. In Lutsig's Verilog semantics, an assignment `v = X` overwrites the variable `v` with non-deterministic bits from $fbits$. This semantics deviates from the standard, and the deviation is motivated in Sec. 4.1.2. The second construct we highlight is non-blocking assignments, written `<=`, which are used for communication between processes. Blocking assignments, written `=`, have the usual imperative-language semantics. To be able to express the semantics of non-blocking assignment, Lutsig's Verilog semantics has two separate environments Γ and Δ that are used to keep track of variables' state during execution. Variable reads and blocking assignments only interact with Γ. Non-blocking assignments, on the other hand, do not update Γ directly, but instead update Δ, which is merged into Γ at the end of each clock cycle, such that the updates in Δ become available in Γ in the next clock cycle. Informally, non-blocking writes do not interfere with the execution of the current clock cycle and will instead only be made available to all processes from the next clock cycle.

4.1.1 Simulation and Synthesis Semantics?

One of Verilog's (many) idiosyncrasies that must be taken into consideration when developing a compiler is that Verilog, in practice, is understood as having two semantics: one simulation semantics and one synthesis semantics. The most recent (System)Verilog standard [4] provides a simulation semantics for Verilog (called scheduling semantics in the standard). However, the standard, unfortunately, does not provide any synthesis semantics: That is, it does not define which language constructs are synthesizable (a "synthesizable subset" of the language) and how these synthesizable constructs should be synthesized. Effectively, this leaves it up to each Verilog compiler to provide its own synthesis semantics.

Before Verilog was merged into SystemVerilog, the (now superseded) Verilog standard [1][3] had an accompanying synthesis standard [2]. This synthesis standard could be used as a starting point for a formal synthesis semantics. However, recall the context set up in the introduction of this paper: We are interested in building a compiler that allows us to transport theorems from the Verilog level down to the netlist level. In this context, the problem is not finding a starting point for the formalization of a synthesis semantics: Rather, we want a single semantics used everywhere, because having theorems expressed in a simulation semantics and a compiler proved semantics preserving with respect to a (separate) synthesis semantics opens up problems with composing said theorems with the compiler correctness theorem. Similar composition problems occur in informal settings. Indeed, Mills and Cummings [26] outline some "RTL coding styles" (anti-patterns) that yield simulation and synthesis mismatches.

[3] Verilog 2005 [3] was published between Verilog 2001 and the merge of Verilog into SystemVerilog, but Verilog 2005 is a minor update of Verilog 2001.

To avoid mismatch problems and ensure simple composability of circuit correctness theorems with the compiler correctness theorem, Lutsig takes Verilog's simulation semantics as its synthesis semantics, except for X values, as described below. Consequently, the top-level correctness theorem for Lutsig (Sec. 6.2) is stated in terms of Lutsig's formalization of Verilog's simulation semantics.

4.1.2 X Values. We make one important deviation from Verilog's simulation semantics in Lutsig's Verilog semantics. The deviation concerns Verilog's (in)famous X values [25, 33, 36]. The simulation semantics for X values provided by the standard is not a good fit for synthesis purposes; this section motivates our deviation from the standard and provides our alternative X semantics. Lutsig's determinization pass, presented in Sec. 5.4, illustrates one example of an optimization enabled by having an X value semantics fit for synthesis.

For background: In Verilog, a bit can take on four different values: `0`, `1`, `X` and `Z`. The value `Z` is only relevant for constructs not supported by Lutsig (such as nets with multiple drivers), so we do not consider it here. The values `0` and `1` are the two standard bit values. Remains to be explained, then, is `X`. In the Verilog standard [4, p. 83] the value is said to "represents an unknown logic value." We now enumerate some aspects of the standard's `X` value semantics and then conclude that (some of) these aspects stand in the way for the "don't care" usage of `X` values commonly seen in synthesis.

One concern related to `X` values is how the standard logical operators should be extended to handle `X` inputs (in other words, how to handle "`X` propagation"). Some operators are extended in an intuitive way by the standard: For example, for logical and `&&` [4, pp. 265–266] we have that both `1'b0 && 1'bx` and `1'bx && 1'b0` evaluate to `1'b0`, and e.g. `1'b1 && 1'bx` and `1'bx && 1'bx` both evaluate to `1'bx`. Bitwise and `&` [4, p. 266] is extended similarly, and we have that e.g. `3'b00x & 3'b100` evaluate to `3'b000`. Also the conditional operator is given an intuitive extension, e.g. `1'bx ? 4'b01xx : 4'b00x1` evaluates to `4'b0xxx`.

Some other operators, however, are extended in less intuitive ways. For example, addition is one example of an operator that can be considered too "`X`-pessimistic" (a term used in discussions on `X` value semantics): "[I]f any operand bit value is the unknown value x [...], then the entire result value shall be x" [4, p. 261]. So, e.g., `3'b000 + 3'b00x` evaluates to `3'bxxx`. Similarly, for a variable `a`, e.g. `a * 0` does not necessarily evaluate to `0`, nor does `a - a`.

At the same time, other constructs in the language are seemingly too "`X`-optimistic". One example of such a construct is if-statements. For example, after executing the following code fragment the (1-bit) variable `a` will always be `1'b0`:

```
if (1'bx)
 a = 1'b1;
else
 a = 1'b0;
```

This is because the first branch is taken if and only if the condition expression evaluates to "a nonzero known value" [4, p. 299]. Another too `X`-optimistic construct is array assignments: An assignment to an array `a` such as e.g. `a[3'bxxx] = 1'b0` "shall perform no operation" according to the standard, because an index containing `X`s is considered invalid [4, pp. 148–149].

Another peculiarity with Verilog's `X` semantics is illustrated by the equality operators provided in the language. The operators do not keep track of when an `X` value is compared with itself: None of Verilog's equality operators provides the intuitive semantics that comparing `1'bx` with `1'bx` is `1'bx`, but comparing, say, a 1-bit variable, `a` with itself is always `1'b1`. Indeed, let `a = 1'bx` and consider the following table [4, pp. 264–265]:

op	`1'bx` op `1'bx`	`a` op `a`	`1'b1` op `1'bx`
`==`	`1'bx`	`1'bx`	`1'bx`
`===`	`1'b1`	`1'b1`	`1'b0`
`==?`	`1'b1`	`1'b1`	`1'b1`

The above semantics is not fit for synthesis purposes, since one important usage of `X` values in synthesis is to signal "don't care". For example, if we assign `X` to a variable, we signal to the compiler that we do not care about the value of the variable in the situation it was assigned and the compiler is free to assign any value to the variable. This opens up optimization opportunities for the compiler. However, clearly, this way of using `X` values is not compatible with the simulation semantics outlined above. For example, recall the if-statement above with the condition `1'bx`. If we replace the condition `1'bx` with `1'b1`, then `a` will always be equal to `1'b1` after the if-statement. That is, replacing `X` values with concrete values can add behavior to programs!

To solve this problem, and to get an easy-to-understand semantics, Lutsig's Verilog semantics deviates from the standard. In Lutsig's semantics, bits can only take on the two standard values `0` and `1`. This means that no special attention needs to be given to how `X` values propagate through the operators supported by Lutsig. `X` assignments are given meaning by interpreting them as sources of non-determinism: Formally, bits from *fbits* are used to overwrite the old left-hand side. Other sources of `X` values are handled by aborting the execution: For example array out-of-bounds accesses abort the execution instead of returning `X` [4, pp. 148–149, p. 279].[4]

4.2 Target Language: Netlists

The syntax of Lutsig's target netlist language is described in Fig. 3. Cells are connected together with members of the cell_input type, called *i* in Fig. 3. A "variable" in the context of

[4] We leave it up to future case studies to decide if this is a good design decision or not. In retrospect, we could have followed the standard more closely here while at the same time avoiding the problems outlined in this section by returning non-deterministic bits from out-of-bounds accesses.

c	::=	$b \mid [b]$	for Boolean b
t	::=	`logic` | `logic[`n`]`	for $n \in \mathbb{N}$
i	::=	c	literal constant
	|	v	variable
	|	v`[`c`]`	array indexing
cell	::=	v = `ndet(`t`)`, v = `not(`i`)`, v = `and(`i_0, i_1`)`,	
		v = `or(`i_0, i_1`)`, v = `equal(`i_0, i_1`)`, v = `add(`i_0, i_1`)`,	
		v = `mux(`i_0, i_1, i_2`)`, v = `array_write(`i_0, n, i_1`)`,	
		v = k`-LUT([`i`])`, (v_0, v_1) = `carry4(`i_0`,[`i`],[`i`])`	
d	::=	$t\ v = (c, i?)$	
	|	$t\ v = (\mathsf{X}, i?)$	
cir	::=	`circuit` $[(v, t)]$ $[d]$ $[cell]$	

Figure 3. Netlist values c, types t, inputs i, cells *cell*, register declarations d, and circuits *cir*.

cell inputs refers to the output of another cell, a register, or a circuit input. The syntax is animated by a functional big-step operational semantics that runs a provided circuit for n clock cycles: circuit_run *fext fbits* (Circuit *exttys regs nl*) n. Execution starts by initializing the registers *regs*, and for each clock cycle all cells nl are executed in order and registers with inputs are updated after all cells have been executed. The formal semantics is a straightforward implementation of this evaluation scheme. For example, executing all cells in order is done by folding (in the sum monad) with the cell semantics function cell_run. For cell_run in turn, e.g. the semantics of and cells with Booleans inputs is particularly simple:

```
cell_run fext s (Cell2 CAnd out in1 in2) ≝ do
  in1 ← cell_input_run fext s in1;
  in2 ← cell_input_run fext s in2;
  in1 ← get_cbool in1;
  in2 ← get_cbool in2;
  Inr (cset_var s (NetVar out) (CBool (in1 ∧ in2)))
od
```

The exact set of cells available for use in technology mapped netlists depends on which hardware technology is targeted. For this paper, we target Xilinx 7 series FPGAs [39]. Lutsig's technology mapped output netlists for this class of FPGAs contain only k-LUT (with $k \leq 6$) and carry4 cells (and registers) – the other cells in the netlist language are only used for intermediate compilation steps (and are consequently not part of the TCB).

We now describe the cells included in the netlist language. k-LUT cells are k-bit input 1-bit output lookup tables that can be configured to implement any Boolean function with the same number of input and output bits. The (maximum) value of k depends on specific FPGA targeted. carry4 cells represents the carry chain logic available in the kind of FPGAs we target [40][5]. Lutsig utilizes carry4 cells to implement addition and wide equality checks (see Sec. 6). The semantics of `ndet(`t`)` is that the cell non-deterministically (using *fbits*) generates a value of type t. The semantics of `mux(`i_0, i_1, i_2`)` is that the cell outputs i_1 when i_0 is true, otherwise i_2. The semantics of `array_write(`i_0, n, i_1`)` is that the cell outputs the array input i_0 with element n replaced by the value of input i_1. The remaining cells have the obvious semantics.

[5]In our formalization, we have merged the two inputs CYINIT and CI into one input.

5 Verilog to Netlist Compilation

In this section we describe, in order, the different compilation passes an input Verilog program goes through when Lutsig transforms it into a not-yet-technology-mapped netlist. Technology mapping is explained in the next section.

We have composed the correctness theorems for the different passes together with the verified part of the technology mapper into a top-level correctness theorem for Lutsig, and the theorem is presented in the next section (in Sec. 6.2).

5.1 Type Checking and Type Annotating

Compilation starts with a type checking and type annotation pass. The type annotations are needed in subsequent pre-processing passes (Sec. 5.2), and the main Verilog to netlist pass (Sec. 5.3) needs to know that the input Verilog program is well-typed. The type checker typecheck takes an input Verilog program, type checks and type annotates the program, and returns the annotated program on success, and otherwise signals a type error.

The type checker is sound in the following sense:

```
⊢ typecheck (Module exttys decls ps) =
  Inr (Module exttys' decls' ps') ⇒
  exttys' = exttys ∧ decls' = decls ∧
  vertype_prog (K None) (Module exttys' decls' ps') ∧
  run fext fbits (Module exttys' decls' ps') n =
  run fext fbits (Module exttys decls ps) n
```

That is, if the type checker terminates successfully, then *exttys* and *decls* are unchanged, the annotated program is well-typed and correctly annotated (vertype_prog) in an empty typing environment (K None), and furthermore the annotated program behaves in the same way as the input program.

5.2 Pre-processing

Before the main Verilog to netlist pass (Sec. 5.3), non-constant index array lookups and case-statements are compiled away by a series of Verilog to Verilog pre-processing passes. We now outline these pre-processing passes.

Array lookups. The pre-processing passes for handling array reads and array writes with non-constant index lookups both compile away such lookups by replacing them with case-statements built out of constant lookups only. Given how similar the two passes are, we only cover the array reads pre-processing pass in this paper.

For an array a, e.g. `a[1]` is considered a lookup with a constant index, whereas `a[i + 1]` where `i` is a variable is considered a lookup with a non-constant index. All read lookups with non-constant indices are pre-processed away by the pass. As an example, say an array `a` has (Verilog) type `logic[`n_a`]`, another array `i` has type `logic[`n_i`]`, and two variables `b` and `c` have type `logic`, then the pre-processing pass would transform the following code fragment

```
b = a[i] & c;
```

into the following code fragment

```
case (i)
0: tmpvar34 = a[0];
1: tmpvar34 = a[1];
2: tmpvar34 = a[2];
3: tmpvar34 = a[3];
...
n: tmpvar34 = a[n];
endcase

b = tmpvar34 & c;
```

where `tmpvar34` is a fresh variable and n = $\min(n_a, 2^{n_i})$. The index length n_i is retrieved from the annotation in the Verilog AST added by the type checker. When the index is, like in the example, simply a variable, n_i can easily be recomputed by looking up the variable in the typing environment, but for more involved expressions, like e.g. `i + 1`, (re-doing part of) type inference would be needed if the type checker did not add annotations. Lastly, note that all remaining array read lookups are in-bounds after the pre-processing pass (which simplifies compilation to netlists further down in the compilation chain).

Case-statements. Case-statements, both those introduced by the array lookup pre-processing passes and those from the input program, are transformed into series of if-statements by a case-statements pre-processing pass. We illustrate the compilation scheme involved by the following example

```
case (i + 1)
 8'b0000_0000: j = 0;
 8'b0000_0001: j = 5;
 default: j = 12;
endcase
```

which would be transformed into

```
tmpvar66 = i + 1;

if (tmpvar66 == 8'b0000_0000)
 j = 0;
else if (tmpvar66 == 8'b0000_0001)
 j = 5;
else
 j = 12;
```

where `tmpvar66` is a fresh variable.

The pass simplifies subsequent passes, because they do not have to take case-statements into consideration. But, clearly, the compilation scheme is highly inefficient: We do not need two 8-bit wide equality checks to differentiate `8'b0000_0000` from `8'b0000_0001`.

5.3 Verilog to Netlist Compilation

The Verilog to netlist pass in Lutsig is based on the unverified Verilog compiler CSYN's compilation algorithm [14]. We now outline some interesting aspects of the compilation algorithm as implemented in Lutsig.

Compiling expressions. The two most important compilation functions are compile_stmt and compile_exp, which compiles Verilog statements and Verilog expressions. We explain compile_exp first. Grossly simplified, (part of) the correctness theorem for compile_exp is:

$$
\begin{aligned}
&\vdash \text{compile\_exp } s\ e = \text{Inr } (s',nl,inp) \wedge \\
&\quad \text{erun } vfext\ venv\ e = \text{Inr } vv \wedge \ldots \Rightarrow \\
&\quad \exists\, nenv'\ nv. \\
&\qquad \text{sum\_foldM (cell\_run } nfext) \ nenv\ nl = \text{Inr } nenv' \wedge \\
&\qquad \text{cell\_input\_run } nfext\ nenv'\ inp = \text{Inr } nv \wedge \\
&\qquad \text{same\_value } vv\ nv
\end{aligned}
$$

That is, the theorem states that after executing the generated netlist nl, the cell input inp has the same value as the input expression e evaluates to.

The pass targets a set of high-level cells, and consequently most compilation schemes for expressions are straightforward. For example, compiling an addition expression is simply a matter of recursively invoking the expression compiler twice and merging the results from the two resulting netlists with a new addition cell CAdd:

$$
\begin{aligned}
&\text{compile\_exp } s \text{ (Arith } e_1 \text{ Plus } e_2) \stackrel{\text{def}}{=} \text{do} \\
&\quad (s,nl_1,inp_1) \leftarrow \text{compile\_exp } s\ e_1; \\
&\quad (s,nl_2,inp_2) \leftarrow \text{compile\_exp } s\ e_2; \\
&\quad (s,tmpvar) = \text{fresh\_tmpvar } s; \\
&\quad newcell = \text{Cell2 CAdd } tmpvar\ inp_1\ inp_2; \\
&\quad newvar = \text{NetVar } tmpvar; \\
&\quad \text{Inr } (s,nl_1 +\!\!+ nl_2 +\!\!+ [newcell],\text{VarInp } newvar \text{ None}) \\
&\text{od}
\end{aligned}
$$

Mapping the CAdd cell to cells actually available on FPGAs is the responsibility of technology mapping (Sec. 6).

Compiling non-X assignments. We now explain some interesting parts of compile_stmt, and we start with the compilation of non-X assignments. We first remark that each variable in the input Verilog program is mapped to a register. A separate post-processing pass not detailed in this paper removes registers never read, to avoid unnecessary registers. For each register, the compilation algorithm needs to generate a net for its next-state function induced by the input Verilog program. This is done by two stacks of maps σ_{b} and σ_{nb}, which the compilation algorithm carries around as state. The stack structure mirrors the block structure of the input program and is explained later in this paper when the compilation of if-statements is explained. The maps in each stack

map variable names (string) to cell inputs (cell_input option). A stack forms a map by delegating lookups to the maps in the stack and letting maps on top shadow maps below. If a variable is mapped by no map, the stack maps the variable to the register allocated for it. During compilation, the stacks of maps are updated as described below.

Blocking assignments update σ_{b} and non-blocking assignments update σ_{nb}. For a blocking assignment without indexing, the following compilation scheme, where the cset_net call updates $\sigma_{\mathrm{b}}(var)$ to inp, compiles the assignment:

compile_stmt s (BAssn (NoIndexing var) (Some e)) $\stackrel{\text{def}}{=}$ do
(s,nl,inp) $\leftarrow$ compile_exp s e;
Inr (s *with* bsi := cset_net s.bsi var inp,nl)
od

For blocking assignments with indexing, the compilation scheme is similar except that we also need to generate an `array_write` cell. Generating the `array_write` cell is simple because after the pre-processing passes (Sec. 5.2) have been run all array indices are in-bound constants.

The compilation schemes for non-blocking assignments are identical, except that σ_{nb} is updated instead of σ_{b}.

Compiling variable reads boils down to a lookup in σ_{b}, i.e., the contents of σ_{nb} does not matter for reads:

compile_exp s (Var var) $\stackrel{\text{def}}{=}$ Inr (s,[],cget_net s.bsi var)

The contents of σ_{nb} become relevant at the end of compilation, similarly to how the contents of Δ becomes relevant at the end of each clock cycle in Lutsig's Verilog semantics. Informally, σ_{b} tracks Γ and σ_{nb} tracks Δ, and as the contents of Δ overrides the contents of Γ at the end of each clock cycle, an initially reasonable-looking compilation approach is to let σ_{nb} override σ_{b} in the cell input generation for each variable's register in the sense that σ_{nb} decides the cell input if it contains an entry for the variable, otherwise σ_{b} is used as a fallback. This idea works for simple programs such as

```
module // ...
always_ff @(posedge clk) begin
 a <= 0; a = 1;
 b = 1; b <= 0;
end
endmodule
```

where according to Verilog's simulation semantics both `a` and `b` should be `0` at the end of each clock cycle (as non-blocking assignments always shadow blocking ones). Before we can present an example program that is miscompiled by the suggested compilation scheme, we must explain how if-statements are compiled.

Compiling if-statements. If-statements `if (c) then st else sf` are compiled into muxes in the following way: The expression `c` is compiled by compile_exp such that we get a cell input inp_{c} and a netlist nl_{c} for the expression, and `st` and `sf` are compiled by recursively calling compile_stmt in accordance with the following pseudo-code (ignoring other state components beyond σ_{b} and σ_{nb}, and denoting maps in the two stacks by sigmas as well):

compile_stmt $(\sigma_\epsilon :: \sigma_{\mathrm{b}}, \sigma_\epsilon :: \sigma_{\mathrm{nb}})$ `st` =
$(\sigma_{\mathrm{b}}^{\mathtt{st}} :: \sigma_{\mathrm{b}}, \sigma_{\mathrm{nb}}^{\mathtt{st}} :: \sigma_{\mathrm{nb}}, nl_{\mathtt{st}})$,
compile_stmt $(\sigma_\epsilon :: \sigma_{\mathrm{b}}, \sigma_\epsilon :: \sigma_{\mathrm{nb}})$ `sf` =
$(\sigma_{\mathrm{b}}^{\mathtt{sf}} :: \sigma_{\mathrm{b}}, \sigma_{\mathrm{nb}}^{\mathtt{sf}} :: \sigma_{\mathrm{nb}}, nl_{\mathtt{sf}})$

where σ_ϵ is an empty map. To update σ_{b}, for each variable var in either $\sigma_{\mathrm{b}}^{\mathtt{st}}$ or $\sigma_{\mathrm{b}}^{\mathtt{sf}}$, create a new cell

$$\mathtt{mux}(inp_{\mathrm{c}}, (\sigma_{\mathrm{b}}^{\mathtt{st}} :: \sigma_{\mathrm{b}})(var), (\sigma_{\mathrm{b}}^{\mathtt{sf}} :: \sigma_{\mathrm{b}})(var))$$

and add a mapping to σ_{b} from var to the new mux. Lastly, generate muxes for and update σ_{nb} in the same way except use $\sigma_{\mathrm{nb}}^{\mathtt{st}}$ and $\sigma_{\mathrm{nb}}^{\mathtt{sf}}$ instead of $\sigma_{\mathrm{b}}^{\mathtt{st}}$ and $\sigma_{\mathrm{b}}^{\mathtt{sf}}$.

One interesting aspect of this compilation scheme is that it causes the generated netlists for both branches to always be executed, including netlists for dead branches. This is different from when targeting a language with jumps, such as e.g. an assembly language. As an example, for an if-statement `if (c) then st else sf`, consider some condition `c` that is always true and a statement `sf` that always crashes with a runtime error. That `sf` always crashes does not affect the if-statement's behavior, because the code will never be executed. However, as the generated netlist for the if-statement will consist of nl_{c} followed by $nl_{\mathtt{st}}$ followed by $nl_{\mathtt{sf}}$ followed by the muxes used to merge the results from the two branches, the netlist for `sf` executes every clock cycle. Consequently, it must be ensured that no Verilog code, not even dead code, can generate a netlist that crashes with a runtime error. To ensure that no bad netlists are generated, the pass assumes its input to be well-typed. From well-typedness and array lookup pre-processing, it follows that no output netlist will crash with a runtime error.

Compiling non-`X` assignments, continued. Now understanding the compilation scheme for if-statements, we understand why the following program would be miscompiled by the earlier suggested compilation scheme for non-`X` assignments saying that σ_{nb} should simply shadow σ_{b} in register input generation:

```
module // ...
always_ff @(posedge clk) begin
 if (a)
  b <= 0;
 b = 1;
end
endmodule
```

If we would let σ_{nb} shadow σ_{b}, then starting from $\mathtt{a} = \mathtt{b} = 0$ we would get $\mathtt{b} = 1$ from the Verilog program, but $\mathtt{b} = 0$ from the generated netlist, because the cell $\mathtt{mux}(\mathtt{a}, 0, \mathtt{b})$ (where `a` and `b` refer to the registers for `a` and `b`) generated from the if-statement would overshadow the blocking assignment (since $\sigma_{\mathrm{nb}}(\mathtt{b})$ will map to the mentioned mux and $\sigma_{\mathrm{b}}(\mathtt{b})$ will map to 1 after the `always_ff` block has been processed).

Instead of designing a more complex compilation scheme, we have restricted Lutsig to only accept modules not containing blocking and non-blocking assignments to the same variable.[6] As a result, σ_{nb} never shadows anything in σ_{b}, and, it turns out, the earlier suggested compilation scheme for register inputs now works without problems. The same restriction can be found in other compilers: For example Vivado Design Suite [41, pp. 233–234] introduces a "usage restriction" saying not to mix blocking and non-blocking assignments (although, Vivado can synthesize programs with mixed assignments "without error"), and (as of this writing) the latest Yosys compiler [38] silently miscompiles (some) programs with mixed assignments.

Compiling `X` assignments. When compiling a blocking `X` assignment, a new `ndet` cell is generated and σ_{b} for the left-hand side variable is updated to map to the cell output of the new `ndet` cell. Again, the compilation scheme for non-blocking assignments is the same except that σ_{nb} is updated instead.

Because the netlists for both branches of (all) if-statements in the input Verilog program are executed each clock cycle, we cannot "reuse" the *fbits* stream of non-deterministic bits from the Verilog level at the netlist level in Lutsig's correctness theorem. Consider the following code fragment:

```
if (c)
  a = 1'bx;
```

For this code fragment, one bit of *fbits* will be consumed if `c` evaluates to true, otherwise no bits will be consumed. At the netlist level, the generated netlist will always consume one bit of *fbits* regardless of what `c` evaluates to. This means that we cannot state a correctness theorem guaranteeing that the output netlist has the exact same behavior as the input Verilog code if we simply reuse *fbits*. Instead, as seen in the top-level correctness theorem in Sec. 6.2, we say that for every *nfbits* on the netlist level, there is a *vfbits* on the Verilog level such that the behavior of the Verilog code coincides with the behavior of the netlist (Inr case) or Verilog evaluation aborts with an error (Inl case). The resulting theorem may initially seem too weak to be useful, but the circuit correctness theorems we are interested in transporting from the Verilog level to the netlist level include the claim that no runtime errors occur in the Verilog code, and consequently we will never reach the Inl case in the theorem.

5.4 Netlist Determinization

A determinization pass removes all non-determinism from the circuit as the target cell set does not include any non-deterministic cells. In the pass, for the registers, all `X` initializations are replaced with zero initializations. For the cells, it would be possible to similarly replace all `ndet` cells with zeros. But, in some cases, by carefully picking another value to replace an `ndet` cells with, we can optimize away other cells as well in the determinization process. To illustrate this, consider the following Verilog code:

```
if (c) begin
  // ...
  a = 1'b1;
  // ...
end else begin
  // ...
  a = 1'bx;
  // ...
end
```

The Verilog to netlist pass will (as long as `a` is not assigned in any of the branches after the assignments highlighted in the above example) generate a mux cell to merge the two writes `a = 1'b1` and `a = 1'bx`. Note that if we replace the `1'bx` value with `1'b1`, rather than naively replacing all `X` values with zeroes, we can optimize away the mux because it will now always output the same value.

The mux optimization idea illustrated in the example is the core idea of the determinization pass, except that the pass operates on the netlist level rather than on the Verilog level. To be able to select appropriate replacement values, the pass traverses the netlist twice: (1) During the first traversal appropriate replacement values are identified, and (2) during the second traversal `ndet` cells and cells that become redundant after replacing the `ndet` cells with the values from the first traversal are removed.

(1) Finding replacement values. The first traversal incrementally builds up a map σ : cell_input $\rightarrow$ dfill option where dfill = TBD ctype | HBD cvalue. We call cell inputs with a TBD ("to be determined") entry in σ TBD inputs and cell inputs with a HBD ("has been determined") entry HBD inputs.

Before the traversal the map is empty. For all `ndet` cells visited, a TBD entry is added to the map to keep track of which inputs can be replaced with new values. A HBD entry is added when a cell can be optimized away by setting a TBD input to a specific value. In our simple implementation, only mux cells add HBD entries. Specifically, if a mux has one TBD input and the other input is constant (the condition input does not matter), the TBD entry for the TBD input is replaced with a HBD entry with the constant from the constant input. One can easily imagine other ways to add HBD entries: For example, addition cells with one TBD input could add HBD entries filled with zeroes as the addition cell could then be optimized away (regardless of what the other input to the cell is).

(2) Replacing `ndet` cells. The σ built up during the first traversal is used during the second traversal. During the second traversal, TBD cell inputs are replaced by constant zero inputs and HBD cell inputs are replaced by the value contained in the input's HBD entry. All `ndet` cells and mux

[6] Because of the same restriction, we were able to take a small shortcut in Lutsig's Verilog semantics, storing complete arrays in Δ rather than parts-to-be-updated, because we know Δ will never shadow anything in Γ.

cells with inputs that are constant and equal (after processing based on σ) are removed.

Correctness. Consider again Lutsig's top-level theorem in Sec. 6.2. After the determinization pass, the circuit is independent of *nfbits* because it no longer contains any nondeterministic constructs – and clearly we can always find a *vfbits* because the determinization pass never adds new behaviors to the circuit.

6 Technology Mapping

Technology mapping in Lutsig is divided into two passes: The first pass is verified (Sec. 6.1), and the second pass is based on translation validation (Sec. 6.3). The first pass maps high-level cells and furthermore functions as a pre-processing pass for the second pass by splitting all low-level array cells into Boolean cells. The second pass maps all low-level (now-Boolean) cells not mapped by the first pass to LUTs. The second pass is based on translation validation to allow future developments as we expect running a realistic optimizing technology mapper in-logic would be too computationally expensive.

6.1 Verified Technology Mapping

The first pass maps high-level cells such as addition cells to cells natively available on the target FPGAs and splits – or "blasts" – low-level cells that operate over arrays to cells that operate over Booleans. As a result, when the second pass finalizes the mapping process, it only has to map Booleans cells. The second pass is based on graph covering, and after the first pass' pre-processing we know that we will always be able to find a covering since a k-input Boolean cell can always (if needed) be covered by a k-LUT. Because the FPGAs we target only offer Boolean registers, the first pass also blasts all array registers into Boolean registers.

We have proved that the first pass is correct in the sense that the following relation between a non-blasted circuit state s and a blasted state bs is invariant under running a non-blasted circuit and its blasted version:

```
blast_reg_rel s bs ≝
 ∀ reg.
   case cget_var s (RegVar reg 0) of
    Inl e ⇒ T
   | Inr (CBool v) ⇒
    cget_var bs (RegVar reg 0) = Inr (CBool v)
   | Inr (CArray v) ⇒
    ∀ i. i < length v ⇒
        cget_var bs (RegVar reg i) = Inr (CBool (el i v))
```

Informally, Boolean registers remain untouched and array registers are blasted into a series of Boolean registers each containing one bit from the array register they were blasted out of.

The pass functions as follows. When the pass traverses a netlist, a blast map σ : cell_input $\rightarrow$ cell_input list[7] is maintained to keep track of which cells have been blasted where. For low-level cells, blasting is straightforward; for example, blasting a mux cell $v = \mathsf{mux}(i_0, i_1, i_2)$ with two array inputs i_1, i_2 of length n results in n mux cells

$$\begin{aligned} &v_0 = \mathsf{mux}(\sigma(i_0), \sigma(i_1)[0], \sigma(i_2)[0]),\\ &v_1 = \mathsf{mux}(\sigma(i_0), \sigma(i_1)[1], \sigma(i_2)[1]),\\ &\ldots,\\ &v_{n-1} = \mathsf{mux}(\sigma(i_0), \sigma(i_1)[n-1], \sigma(i_2)[n-1]) \end{aligned}$$

where $v_0, \ldots, v_{n-1}$ are fresh variables. Note how cell inputs are updated using σ. In case no mapping for a cell input exists, the cell input is left untouched. After the mux cell has been blasted, σ is updated such that $\sigma(v) = [v_0, \ldots, v_{n-1}]$.

Blasting high-level cells requires more attention. For example, addition can be implemented purely in terms of LUTs, but the class of FPGAs we target has special hardware support for addition which we want to exploit. In our implementation, we tried to mirror how existing compilers for the class of FPGAs we target map addition operations: A network of `carry4` cells in combination with xor cells implemented as LUTs, such that the fast carry chains available on the FPGAs we target are exploited. Our implementation likewise maps `equal` cells to networks of LUTs and `carry4` cells.

Another special case is `array_write` cells. Blasting $v =$ `array_write`(i_0, n, i_1) does not generate any new cells: It is sufficient to update $\sigma(v)$ such that $\sigma(v)[i]$ equals $\sigma(i_1)$ if $i = n$ and $\sigma(i_0)[i]$ otherwise.

6.2 Lutsig's Top-Level Correctness Theorem

Composing the correctness theorems for the different passes of the verified compiler (Sec. 5) and the verified technology mapper (Sec. 6.1) results in the following top-level theorem for the verified part of Lutsig:

```
⊢ let m = Module exttys decls ps in
    compile keep m = Inr circuit ∧ writes_ok ps ∧
    vertype_fext exttys vfext ∧ same_fext vfext nfext ⇒
    ∃ cenv vfbits.
      circuit_run nfext nfbits circuit n = Inr cenv ∧
      case run vfext vfbits m n of
        Inl e ⇒ T
      | Inr venv ⇒ verilog_netlist_rel keep venv cenv
```

A few details are worth mentioning: The *keep* argument to compile is a list of registers that must not be optimized away. One example usage is keeping registers that are never read internally but will later be exposed as circuit outputs. The predicate writes_ok prohibits blocking and non-blocking assignments to the same variable (see Sec. 5.3). The predicate verilog_netlist_rel is similar to the predicate blast_reg_rel from

[7] The actual type is a little more involved, but for this paper this level of detail is sufficient.

Sec. 6.1, but only guarantees correspondence for registers in *keep* (as other registers might have been optimized away).

6.3 Translation-Validation-Based Technology Mapping

The second pass maps cells not mapped by the first pass to LUTs. The output netlist from the pass is fully mapped, consisting only of cells natively available on the FPGAs we target. The pass first (1) finds a mapping, and then, as a separate step, (2) proves the mapping correct.

(1) Finding a mapping. The first step does not involve any kind of proof: The responsibility of the first step is to find, by any means, a mapping to later be validated by the second step. For this purpose, we have implemented a simple unverified placeholder technology mapper in SML. The technology mapper is based on conventional graph covering techniques [18]. The technology mapper does not carry out any optimization when finding a covering: The mapper constructs coverings and selects a covering to use using a simple greedy algorithm. For the purpose of this paper, finding any covering is sufficient: As no proofs are required, if a "good" covering (for some definition of good) is needed, the problem of finding a covering can be outsourced to any existing mapper instead of relying on our placeholder mapper. For this paper, we opted for implementing a placeholder mapper because it was simpler than integrating an existing mapper.

(2) Proving the mapping correct. The responsibility of the second step is to generate a theorem on the form

$$\begin{aligned}&\vdash \ldots \Rightarrow\\ &\quad \mathsf{circuit\_run}\ \mathit{fext}\ \mathit{fbits}\ (\mathsf{Circuit}\ \mathit{exttys}\ \mathit{regs}\ \mathit{nl}_1)\ n =\\ &\quad \mathsf{circuit\_run}\ \mathit{fext}\ \mathit{fbits}\ (\mathsf{Circuit}\ \mathit{exttys}\ \mathit{regs}\ \mathit{nl}_2)\ n\end{aligned}$$

where nl_1 is the fully mapped netlist produced by the first step and nl_2 is the partially mapped netlist that was given as input to the first step. Note that the registers *regs* are left untouched. Thus, if we can prove each register's cell inputs in the two circuits equivalent, the equivalence of the two circuits easily follows. Cell output names are preserved by our technology mapper, making matching outputs between the two netlists simple. As a result, the second step functions as follows: For each cell output in nl_1, prove the cell output equal to the cell output with the same name in nl_2. With the help of some in-logic computations needed to sanity-check nl_1 as the netlist was generated outside the logic, the equivalence of the two circuits easily follows from the equivalence of the cell outputs.

For already mapped cells, given that their inputs are equal in the two netlists, the equality of their outputs follows directly. For cells mapped to LUTs, the equivalence is shown as follows for each LUT:

(2a) generate a Boolean expression for the LUT and a Boolean expression for the cells covered by the LUT using HOL4 automation,

(2b) prove the two expressions equivalent using a SAT solver,

(2c) conclude using further HOL4 automation that the equivalence of the LUT and the cells follows.

Step (2a) works in a fashion similar to the proof-producing HOL-to-Verilog translation tool from the Verilog development tools [23] connected to Lutsig. The following predicate

$$\begin{aligned}&\mathsf{Eval}\ \mathit{fext}\ \mathit{st}\ \mathit{nl}\ \mathit{inp}\ b \stackrel{\text{def}}{=}\\ &\quad \forall\, \mathit{st}'.\ \mathsf{is\_initial\_state}\ \mathit{st} \wedge\\ &\qquad \mathsf{sum\_foldM}\ (\mathsf{cell\_run}\ \mathit{fext})\ \mathit{st}\ \mathit{nl} = \mathsf{Inr}\ \mathit{st}' \Rightarrow\\ &\qquad \mathsf{cell\_input\_run}\ \mathit{fext}\ \mathit{st}'\ \mathit{inp} = \mathsf{Inr}\ (\mathsf{CBool}\ b)\end{aligned}$$

allows the automation to express that after running the netlist nl starting from state st, reading the cell input inp will return the Boolean b. When we generate a Boolean expression for a cell input, we say that we are Boolifying the input.

We have proved theorems similar to the following theorem for all cells that can occur at this stage of the compilation:

$$\begin{aligned}&\vdash \mathsf{all\_distinct}\ (\mathsf{flat}\ (\mathsf{map}\ \mathsf{cell\_outputs}\ \mathit{nl})) \wedge\\ &\quad \mathsf{mem}\ (\mathsf{Cell2}\ \mathsf{CAnd}\ \mathit{out}\ \mathit{in}_1\ \mathit{in}_2)\ \mathit{nl} \wedge\\ &\quad \mathsf{Eval}\ \mathit{fext}\ \mathit{st}\ \mathit{nl}\ \mathit{in}_1\ \mathit{in1b} \wedge \mathsf{Eval}\ \mathit{fext}\ \mathit{st}\ \mathit{nl}\ \mathit{in}_2\ \mathit{in2b} \Rightarrow\\ &\quad \mathsf{Eval}\ \mathit{fext}\ \mathit{st}\ \mathit{nl}\ (\mathsf{VarInp}\ (\mathsf{NetVar}\ \mathit{out})\ \mathsf{None})\ (\mathit{in1b} \wedge \mathit{in2b})\end{aligned}$$

Informally, the theorem says that if no cell shadows any other cell (all_distinct ...), there is an and cell with inputs in_1 and in_2 in the netlist nl, and the two inputs have been Boolified to $in1b$ and $in2b$, respectively, then the Boolification of the and cell's output out is $in1b \wedge in2b$. Using this set of theorems we have proved, Boolifying a set of cells is simply a matter of visiting each cell in netlist order and for each cell specializing the theorem for its cell type.

Step (2b) utilizes the existing SAT infrastructure available in HOL4 [37]. HOL4's SAT infrastructure relies on the presence of an external (unverified) SAT solver, like e.g. MiniSat [10], from which the infrastructure can reconstruct a HOL proof based on the output from the SAT solver. That is, the SAT solver itself remains outside the TCB. The proof obligations delegated to the SAT solver consist of proving the Boolifications from step (2a) of the LUT output and the corresponding cell output in the partially mapped netlist equivalent. Because each LUT is processed separately, the Boolean expressions sent to the SAT solver are kept small.

Step (2c) is straightforward given the theorem proved by the SAT infrastructure in HOL4 in the previous step.

7 Case Study and Evaluation

We now show an example of how to use Lutsig in verified circuit development and then compare Lutsig to a mature unverified commercial compiler.

Example usage. As a case study, we follow Fig. 1 and show how to prove an implementation of a moving average filter correct with the help of Lutsig. Both the correctness

criteria and the HOL implementation of the circuit (A)[8] are defined in terms of the HOL word library. Using the helper function

```
presignal fext n shift ≝
  if n < shift then 0w else (fext (n − shift)).signal,
```

we say that the output signal from our moving average filter is correct if it is the following signal:

```
avg_spec fext n ≝
  if n = 0 then 0w
  else if (fext (n − 1)).enabled then
    (presignal fext n 1 + presignal fext n 2 +
     presignal fext n 3 + presignal fext n 4) // 4w
  else (fext (n − 1)).signal
```

A straightforward but space-inefficient implementation is given by the HOL circuit avg (A):

```
avg_step fext s ≝ let
  s = s with h3 := s.h2;
  s = s with h2 := s.h1;
  s = s with h1 := s.h0;
  s = s with h0 := fext.signal;
  s = s with sum := s.h0 + s.h1 + s.h2 + s.h3;
  s = div_by_4 s
in
  if fext.enabled then s with avg := s.sum
  else s with avg := fext.signal
avg fext s 0 ≝ s
avg fext s (Suc n) ≝ avg_step (fext n) (avg fext s n)
```

As Lutsig does not support division, we implement division by bit-shifting, and as Lutsig does not support bit-shifting, we implement bit-shifting using array operations (div_by_4 in the circuit definition, and for completeness, the definition of div_by_4 is included in App. A). Proving the implementation correct is trivial, and gives us the following theorem:

```
⊢ (avg fext avg_init n).avg = avg_spec fext n        (1)
```

We now derive a Verilog implementation vavg (B) from avg. The core component of the Verilog development tools [23] we have connected to Lutsig is a proof-producing translator from shallowly embedded Verilog-like HOL circuits to deeply embedded Verilog circuits. The Verilog code derived from avg_step is included in App. B. For each run of the translator, the translator produces a theorem stating that the input HOL circuit and the output Verilog circuit have the same behavior. Composing the theorem produced by the translator and the HOL circuit correctness theorem Thm. 1, we can easily produce a Verilog circuit correctness theorem:

```
⊢ lift_vfext vfext fext ⇒
  ∃ s. run vfext vfbits vavg n = lnr s ∧                (2)
       get_reg s "avg" = lnr (w2ver (avg_spec fext n))
```

[8]The parenthesized letters refer to the stages introduced in Fig. 1.

Now having procured the Verilog implementation vavg (B), the verified part of Lutsig (Sec. 6.2) can now compile down vavg to a partly technology mapped netlist navg' (H). Now, translation-validation-based technology mapping (Sec. 6.3) provides us with a fully technology mapped netlist navg (I).

At this point, we have all theorems needed to derive the final correctness theorem for the netlist navg. Composing the Verilog circuit correctness theorem Thm. 2, Lutsig's correctness theorem (Sec. 6.2), and the theorem produced by translation-validation-based technology mapping (Sec. 6.3), we have derived the following correctness theorem for the netlist implementation of the filter:

```
⊢ lift_nfext nfext fext ⇒
  ∃ s. circuit_run nfext nfbits navg n = lnr s ∧
       get_reg_blasted s "avg" (w2net (avg_spec fext n))
```

This is as far down the abstraction hierarchy our current development takes us inside HOL. To produce an FPGA bitstream (K) out of navg that can be loaded onto an FPGA, we need to consult external tools. For communication with external tools, we have developed a (unverified) pretty-printer that can print technology mapped netlists to Verilog netlists (J). The pretty-printed netlists can be simulated and synthesized to FPGA bitstreams by tools such as Vivado Design Suite, which is the tool we used in this case study. According to the manual testing we have carried out, the circuit works according to its specification both during simulation and when loaded onto an FPGA board.

Evaluation. We now provide a short evaluation of the compiler. The compiler performs reasonably on the moving average filter case study. Vivado 2018.2 (with default settings) compiles the Verilog program derived from avg to 29 LUTs[9], 2 `carry4` cells, and 32 registers. Lutsig compiles it to 32 LUTs, 6 `carry4` cells (two cells for each (8-bit) addition in the program), and 32 registers.

However, Lutsig stands no chance against mature tools like Vivado on larger examples. The formally verified high-level compiler Vericert [15] that compiles from CompCert C to Verilog bases its Verilog semantics on the same Verilog semantics Lutsig bases its Verilog semantics on and is therefore a good fit for generating test input for Lutsig – since they consequently deal with similar subsets of Verilog. However, as Vericert is verified in Coq rather than HOL4, Vericert cannot provide us with HOL4 correctness proofs for the Verilog code it produces. But since we for the moment are only interested in evaluating the performance of Lutsig, we do not need correctness proofs from the front-end used. Concretely, we use the following C program to evaluate Lutsig:

[9]The exact number of LUTs depends on what we mean by a LUT. For example, for the class of FPGAs we target LUTs have two outputs and sometimes two small one-bit-output LUTs can be merged into one such LUT. Taking this into consideration when counting gives us 24 two-bit-output LUTs rather than 29 one-bit-output LUTs.

```
int main() {
 int max = 5, acc = 0;

 for (int i = 0; i != max; i++)
  acc += i;

 return acc + 2;
}
```

Vericert compiles this program to two Verilog processes each containing an 11-cases case-statement. The full Verilog program generated by Vericert is included in App. C. Vivado compiles the Verilog program to 59 LUTs, 27 `carry4` cells, and 138 registers. Lutsig compiles the program to 1087 LUTs, 92 `carry4` cells, and 225 registers. Case-heavy programs are a particularly bad fit for Lutsig: The large number of cells is a result of the inefficiency of Lutsig's pre-processing-based compilation of case-statements (and the lack of optimization passes in Lutsig). For this example program, the verified part of Lutsig takes around 1 second to execute in logic, the in-logic validation of the netlist generated by Lutsig's unverified technology mapper takes around 8 seconds, and Lutsig's SAT-based translation validation pass takes around 50 seconds. If we replace the 32-bit registers generated by the `int` variables in the C program with 8-bit registers, then Vivado compiles the program to 49 LUTs, 2 `carry4` cells, and 36 registers and Lutsig compiles the program to 326 LUTs, 30 `carry4` cells, and 57 registers. For this smaller program, all of compilation takes around 10 seconds.

8 Related Work

In the software world, realistic verified compilers such as the CakeML compiler [35] and the CompCert compiler [22] exist. In the hardware world, equally mature verified compilers are nowhere to be found. Previous work on verified hardware compilers is limited but exists.

The verified hardware compiler implemented in Coq by Braibant and Chlipala [8] share many similarities with our work, but their source and target languages are different from ours. Their source language is a Bluespec-inspired language called Fe-Si. Fe-Si programs share many similarities with the subset of Verilog Lutsig supports, but Fe-Si programs are more high-level as they do not specify cycle-per-cycle behavior. Unlike our source language, Fe-Si does not include `X` values. The Fe-Si compiler, like Lutsig, target netlists. Neither the Fe-Si compiler nor Lutsig succeed in moving all of hardware development entirely inside an ITP, but Lutsig comes one step further towards this goal as the Fe-Si compiler does not include a technology mapper.

Bourgeat et al. [7] provide another verified hardware compiler implemented in Coq for another Bluespec-inspired language called Kôika. With Kôika, Bourgeat et al. try to address one of the drawbacks of working on a higher abstraction level than traditional HDLs like Verilog allow. Specifically, Kôika allows the hardware designer to specify a schedule that can be used to verify that the Kôika compiler builds the kind of hardware the designer had in mind when implementing their design in Kôika. Mismatch problems between designer intent and what is constructed by the hardware compiler can happen in Verilog as well, one (infamous) example being the unintended inferred latches problem. Whether one should design and verify hardware on a high or low abstraction level depends on the context the work is carried out in, and there are pros and cons to both alternatives. A good high-level language brings advantages, e.g. by enabling rapid prototyping, but also, as illustrated by Kôika, disadvantages in terms of compiler understandability – the more abstract the language, the more magical the black boxes known as the compilers associated with the language become for language users.

Another previous project is the partly verified BEDROC high-level synthesis system [21]. Designed with verification in mind, the aim of the project was full verification, but the work was never finished. A small part of the system was verified inside an ITP (Nuprl), the rest of the verification was carried out by non-ITP means.

Beyond the above mentioned projects, efforts for applying ITPs to hardware development seem to have been focused on topics outside compiler verification: In particular inventing and embedding hardware DSLs and verifying circuits have received more attention [9, 13, 17, 24, 29].

9 Conclusion

We have presented a new verified compiler called Lutsig that compiles Verilog programs down to technology mapped netlists for FPGAs. We have also illustrated the utility of the compiler as a tool in small TCB hardware development by transporting properties proved at the compiler's source level down to the compiler's target level.

Our case studies tell us that further work is needed on improving compiler output quality (e.g. in terms of number of LUTs). Moreover, the subset of Verilog Lutsig currently supports is arguably too closely tied to the kind of Verilog code produced by the proof-producing Verilog translator used in our main case study. For Lutsig to be more widely applicable, a larger subset of Verilog must be supported. Adding support for constructs commonly seen in production Verilog code not currently supported by Lustig, such as `always_comb` blocks and continuous assignments, is therefore part of future work. Another important missing feature is support for Verilog designs consisting of multiple modules. Our plan is to add support for important missing features incrementally now that all initial components of Lutsig are in place.

Acknowledgments

This research was supported with funding from the Swedish Foundation for Strategic Research. We thank Magnus Myreen for comments on drafts of this paper.

A Definition of div_by_4

$$
\begin{aligned}
&\mathsf{div\_by\_4}\ s \stackrel{\text{def}}{=} \mathsf{let}\\
&\quad s = s\ \mathit{with}\ \mathsf{sum} := (0 :+ \mathsf{word\_bit}\ 2\ s.\mathsf{sum})\ s.\mathsf{sum};\\
&\quad s = s\ \mathit{with}\ \mathsf{sum} := (1 :+ \mathsf{word\_bit}\ 3\ s.\mathsf{sum})\ s.\mathsf{sum};\\
&\quad s = s\ \mathit{with}\ \mathsf{sum} := (2 :+ \mathsf{word\_bit}\ 4\ s.\mathsf{sum})\ s.\mathsf{sum};\\
&\quad s = s\ \mathit{with}\ \mathsf{sum} := (3 :+ \mathsf{word\_bit}\ 5\ s.\mathsf{sum})\ s.\mathsf{sum};\\
&\quad s = s\ \mathit{with}\ \mathsf{sum} := (4 :+ \mathsf{word\_bit}\ 6\ s.\mathsf{sum})\ s.\mathsf{sum};\\
&\quad s = s\ \mathit{with}\ \mathsf{sum} := (5 :+ \mathsf{word\_bit}\ 7\ s.\mathsf{sum})\ s.\mathsf{sum};\\
&\quad s = s\ \mathit{with}\ \mathsf{sum} := (6 :+ \mathsf{F})\ s.\mathsf{sum};\\
&\quad s = s\ \mathit{with}\ \mathsf{sum} := (7 :+ \mathsf{F})\ s.\mathsf{sum}\\
&\mathsf{in}\ s
\end{aligned}
$$

That div_by_4 is a correct implementation of division can be proved with almost no effort:

$$\vdash \mathsf{div\_by\_4}\ s = s\ \mathit{with}\ \mathsf{sum} := s.\mathsf{sum}\ //\ 4w$$

B Verilog Process Derived from avg

```
always_ff @(posedge clk) begin
 h3 = h2;
 h2 = h1;
 h1 = h0;
 h0 = signal;
 sum = h0 + h1 + h2 + h3;
 sum[0] = sum[2];
 sum[1] = sum[3];
 sum[2] = sum[4];
 sum[3] = sum[5];
 sum[4] = sum[6];
 sum[5] = sum[7];
 sum[6] = 0;
 sum[7] = 0;

 if (enabled)
  avg = sum;
 else
  avg = signal;
end
```

C Vericert Output

The small C example program in Sec. 7 and the output in this section is taken from Yann Herklotz's talk at the PLDI'20 student research competition. We had to slightly modify the C program, as we had to replace the loop exit condition `i < max` with `i != max` because Lutsig does not support less-than comparisons. Below follows the Verilog code Vericert gives as output when given the C example program as input. We had to slightly modify the output such that it would fit the subset of Verilog Lutsig supports.

```
module main(reg_7, reg_8, clk, finish, ret);
 input [0:0] reg_7;
 input [0:0] reg_8;
 input [0:0] clk;
 output reg [0:0] finish;
 output reg [31:0] ret;
 reg [31:0] state;
 reg [31:0] reg_1;
 reg [31:0] reg_2;
 reg [31:0] reg_3;
 reg [31:0] reg_4;

 always @(posedge clk)
  if (reg_8 == 1'd1)
   state <= 4'd11;
  else
   case (state)
    4'd8: state <= 3'd7;
    3'd4: state <= 3'd7;
    2'd2: state <= 1'd1;
    4'd10: state <= 4'd9;
    3'd6: state <= 3'd5;
    1'd1: state <= 1'd1;
    4'd9: state <= 4'd8;
    3'd5: state <= 3'd4;
    2'd3: state <= 1'd1;
    4'd11: state <= 4'd10;
    3'd7:
     if (reg_1 == reg_3)
      state <= 2'd3;
     else
      state <= 3'd6;
    default: ;
   endcase

 always @(posedge clk)
  case (state)
   4'd8: ;
   3'd4: ;
   2'd2: reg_4 <= 32'd0;
   4'd10: reg_2 <= 32'd0;
   3'd6: reg_2 <= reg_2 + (reg_1 + 32'd0);
   1'd1: begin
    finish <= 1'd1;
    ret <= reg_4;
   end
   4'd9: reg_1 <= 32'd0;
   3'd5: reg_1 <= reg_1 + 32'd1;
   2'd3: reg_4 <= reg_2 + 32'd2;
   4'd11: reg_3 <= 32'd5;
   3'd7: ;
   default: ;
  endcase
endmodule
```

References

[1] 2001. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2001* (2001). https://doi.org/10.1109/IEEESTD.2001.93352

[2] 2005. Verilog Register Transfer Level Synthesis. *IEEE Std 62142-2005* (2005). https://doi.org/10.1109/IEEESTD.2005.339572

[3] 2006. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005* (2006). https://doi.org/10.1109/IEEESTD.2006.99495

[4] 2018. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017* (2018). https://doi.org/10.1109/IEEESTD.2018.8299595

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Annual Design Automation Conference (DAC)*. https://doi.org/10.1145/2228360.2228584

[6] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/289423.289440

[7] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The Essence of Bluespec: A Core Language for Rule-Based Hardware Design. In *Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3385412.3385965

[8] Thomas Braibant and Adam Chlipala. 2013. Formal Verification of Hardware Synthesis. In *Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-642-39799-8_14

[9] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017). https://doi.org/10.1145/3110268

[10] Niklas Eén and Niklas Sörensson. 2004. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*. https://doi.org/10.1007/978-3-540-24605-3_37

[11] Peter Flake, Phil Moorby, Steve Golson, Arturo Salz, and Simon Davidmann. 2020. Verilog HDL and Its Ancestors and Descendants. *Proceedings of the ACM on Programming Languages* 4, HOPL (2020). https://doi.org/10.1145/3386337

[12] Peter Gammie. 2013. Synchronous Digital Circuits as Functional Programs. *ACM Computing Surveys* 46, 2 (2013). https://doi.org/10.1145/2543581.2543588

[13] Michael J. C. Gordon. 1986. Why higher-order logic is a good formalism for specifying and verifying hardware. In *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*.

[14] David J. Greaves. 1995. The CSYN Verilog Compiler and Other Tools. In *International Workshop on Field-Programmable Logic and Applications (FPL)*. https://doi.org/10.1007/3-540-60294-1_113

[15] Yann Herklotz, James Pollard, Nadesh Ramanathan, and John Wickerson. 2020. Formal Verification of High-Level Synthesis. https://yannherklotz.com/docs/drafts/formal_hls.pdf Under review.

[16] Yann Herklotz and John Wickerson. 2020. Finding and Understanding Bugs in FPGA Synthesis Tools. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*. https://doi.org/10.1145/3373087.3375310

[17] Warren A. Hunt, Matt Kaufmann, J Strother Moore, and Anna Slobodova. 2017. Industrial hardware and software verification with ACL2. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 375, 2104 (2017). https://doi.org/10.1098/rsta.2015.0399

[18] Mike Hutton, Vaughn Betz, and Jason Anderson. 2016. FPGA Synthesis and Physical Design. In *Electronic Design Automation for IC Implementation, Circuit Design, and Process Technology*, Luciano Lavagno, Igor L. Markov, Grant Martin, and Louis K. Scheffer (Eds.). CRC Press, Chapter 16.

[19] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *International Conference on Computer-Aided Design (ICCAD)*. https://doi.org/10.1109/ICCAD.2017.8203780

[20] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O. Myreen. 2018. Software Verification with ITPs Should Use Binary Code Extraction to Reduce the TCB. In *Interactive Theorem Proving (ITP)*. https://doi.org/10.1007/978-3-319-94821-8_21

[21] Miriam Leeser, Richard Chapman, Mark Aagaard, Mark Linderman, and Stephan Meier. 1993. High Level Synthesis and Generating FPGAs with the BEDROC System. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology* 6, 2 (1993). https://doi.org/10.1007/bf01607881

[22] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Communications of the ACM (CACM)* 52, 7 (2009). https://doi.org/10.1145/1538788.1538814

[23] Andreas Lööw and Magnus O. Myreen. 2019. A Proof-Producing Translator for Verilog Development in HOL. In *International Workshop on Formal Methods in Software Engineering (FormaliSE)*. https://doi.org/10.1109/FormaliSE.2019.00020

[24] Thomas F. Melham. 1993. *Higher Order Logic and Hardware Verification*. Cambridge University Press.

[25] Don Mills. 2004. Being Assertive with Your X. In *Synopsys Users Group Conference (SNUG)*.

[26] Don Mills and Clifford E. Cummings. 1999. RTL Coding Styles That Yield Simulation and Synthesis Mismatches. In *Synopsys Users Group Conference (SNUG)*.

[27] Rishiyur Nikhil. 2004. Bluespec SystemVerilog: Efficient, Correct RTL from High-Level Specifications. In *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. https://doi.org/10.1109/MEMCOD.2004.1459818

[28] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-step Semantics. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-662-49498-1_23

[29] João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. 2018. ΠWare: Hardware Description and Verification in Agda. In *International Conference on Types for Proofs and Programs (TYPES 2015)*. https://doi.org/10.4230/LIPIcs.TYPES.2015.9

[30] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation validation. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. https://doi.org/10.1007/BFb0054170

[31] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020. LLHD: A Multi-level Intermediate Representation for Hardware Description Languages. In *Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3385412.3386024

[32] Konrad Slind and Michael Norrish. 2008. A Brief Overview of HOL4. In *Theorem Proving in Higher Order Logics (TPHOLs)*. https://doi.org/10.1007/978-3-540-71067-7_6

[33] Stuart Sutherland. 2013. I'm Still In Love With My X!. In *Design and Verification Conference (DVCon)*.

[34] Stuart Sutherland and Don Mills. 2007. *Verilog and SystemVerilog Gotchas: 101 Common Coding Errors and How to Avoid Them*. Springer. https://doi.org/10.1007/978-0-387-71715-9

[35] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *Journal of Functional Programming (JFP)* 29 (2019). https://doi.org/10.1017/S0956796818000229

[36] Mike Turpin. 2003. The Dangers of Living with an X. In *Synopsys Users Group Conference (SNUG)*.

[37] Tjark Weber and Hasan Amjad. 2009. Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic* 7, 1 (2009). https://doi.org/10.1016/j.jal.2007.07.003

[38] Clifford Wolf. [n.d.]. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys.

[39] Xilinx 2018. *7 Series FPGAs Data Sheet: Overview (DS180, v2.6)*. Xilinx.

[40] Xilinx 2019. *Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide (UG953, v2019.2)*. Xilinx.

[41] Xilinx 2020. *Vivado Design Suite User Guide: Synthesis (UG901, v2019.2)*. Xilinx.