CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Thesis for the Degree of Doctor of Philosophy

Building Verified Hardware and Verified Stacks in HOL

Andreas Lööw

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Gothenburg, Sweden
2021

Building Verified Hardware and Verified Stacks in HOL
Andreas Lööw
ISBN 978-91-7905-518-9

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr. 4985
ISSN 0346-718X

Department of Computer Science and Engineering
Chalmers University of Technology and
University of Gothenburg
SE-412 96 Gothenburg, Sweden
Telephone +46 (0)31-772 1000

Printed at Reproservice, Chalmers University of Technology
Gothenburg, Sweden, 2021

Abstract

This thesis explores building provably correct software and hardware inside the HOL4 interactive theorem prover. Interactive theorem provers such as HOL4 are proof environments where manual (human) and automated (machine) proofs can be composed in logically safe ways and all proof steps (be it manual or automated) are mechanically checked.

In this thesis, we are in particular interested in systems consisting of both software and hardware, such as so-called verified stacks. A verified stack is a computer system accompanied by a correctness theorem ensuring the correctness of its running software down to the computer system's hardware implementation.

One contribution of this thesis is that we provide new tools to build verified stacks. Specifically, we provide a new proof-producing Verilog code generator capable of translating hardware circuits proved correct inside HOL4 to the hardware description language Verilog. We also provide a verified Verilog synthesis tool, called Lutsig, for (a class of) FPGAs. Lutsig translates Verilog designs, such as those generated by our proof-producing Verilog code generator, to technology-mapped netlists. With the combined help of the Verilog code generator and Lutsig, it is possible for hardware designers to design and prove circuits correct inside HOL4 and then translate their circuits down to the netlist level while simultaneously carrying along proved properties.

Another contribution is that we apply the new tools in concrete case studies. In particular, one of our case studies contributes to the tradition of building verified stacks as follows. In the case study, we use our Verilog code generator in the construction of a verified proof-of-concept processor that we synthesize for an FPGA board. Building upon this work, we use the processor as the hardware basis for verified stacks based on CakeML programs, including a stack for compiling CakeML programs and a stack for checking proofs. To be able to construct such stacks, we adapt and extend the verified CakeML compiler and its development methodology to support targeting the new processor we have constructed. The CakeML compiler previously only supported compilation to x86, ARM and other architectures without verified implementations.

Contents

Chapter 1

Introduction

This introduction provides some background material relevant for this thesis and describes how the papers included in this thesis fit together.

1.1 The grand vision

This thesis concerns proving computer components correct using mathematics. When we mathematically prove a component correct we say that we verify it.

With this terminology in place, the grand vision underlying the papers included in this thesis is simple to state: verify everything, everywhere, always.

Or, the vision put in more precise form: This thesis concerns building trustworthy computer systems. Here, by "computer system", I (really) mean the whole system: software, hardware, connections between different components – you name it.

Previous projects with ambitious goals similar to the above are easy to enumerate exhaustively since there are essentially only three:

- the CLI stack project [9, 96] (late 1980s and early 1990s),
- the Verisoft project [3] (2003–2010), and
- the DeepSpec project [5] (2016–2021).

Of course, the exact number depends on how we count. For example, the Rigorous Engineering of Mainstream Systems (REMS) project [94], the Provably Correct Systems (ProCoS) project [38, 42], and Jeffrey Joyce's work on "totally verified systems" [54] have overlapping aims with the above-mentioned projects. A similar vision to ours (but not the execution of the vision) is even present in Hoare's "An axiomatic basis for computer programming" from 1969 [44]:

> The most important property of a program is whether it accomplishes the intentions of its user. [...] When the correctness of a program, its compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible

> to place great reliance on the results of the program, and predict their properties with a confidence limited only by the reliability of the electronics.

Trustworthy computer systems are known by multiple names; I use the name "verified stack" here. The term "stack" is a descriptive term since computer systems can be usefully thought of in terms of layers stacked on top of each other. Both when building and using a computer system, such a view of computer systems is useful. Specifically, when working in one layer of a stack (i.e., a computer system), we should only have to be concerned with the layer immediately above us and the layer immediately below us. For example, when writing a computer program in a mainstream language like Python, C++, or Haskell, we should not have to think about the hardware implementation details of the computer we will use to run our program. Similarly, when verifying a software program we should not have to think about hardware implementation details.

The kind of modularity described in the above examples can be achieved by separating different stack layers by specifications. That is, we see the layers as modules consisting of both implementations and specifications, where the specifications of the modules of the stack can be used to glue together the modules into one coherent computer system independently of how the modules are implemented (i.e., the module implementations are hidden behind the module specifications). Such an approach results in computer systems with layers that can be modularly developed and verified independently of each other.

1.2 Contents

This thesis consists of four papers. The papers contribute, in different ways, to the grand vision of building trustworthy computer systems. In the first paper, we build verified stacks that can (e.g.) compile functional programs and check machine-readable proofs. In the remaining three papers, we describe tools that can be used for constructing verified stacks and, in particular, tools for constructing trustworthy hardware.

Concretely, this thesis contributes the following artifacts:

- A new formal semantics for the hardware description language Verilog, which enables formal reasoning about Verilog circuits
- A new verified Verilog synthesis tool called Lutsig, which generates technology-mapped FPGA netlists
- A new proof-producing Verilog code generator, which allows hardware developers to build provably correct Verilog circuits

- A new verified processor called Silver, which is implemented in Verilog
- A new extension to the verified CakeML compiler (a compiler from the SML-like language CakeML to machine code), which makes it target Silver

The contributed artifacts are meaningful both as independent artifacts and as artifacts for verified stack constructions. The artifacts are connected to each other:

- All artifacts that relate to Verilog make use of our new formal semantics for Verilog.
- Lutsig can synthesize the output of the Verilog code generator (when the output is within the subset of Verilog Lutsig supports).
- The verified processor Silver is implemented in Verilog, developed with the help of the Verilog code generator, and functions as the meeting point between software and hardware in our stack constructions.
- Our CakeML extensions connects the CakeML compiler and the Silver processor.
- Etc.

There are many ways to summarize a collection of papers. Throughout this chapter, a big-picture-oriented summary is given, where the artifacts introduced above and related concepts are described as-needed as they fit into the larger picture. As part of this summary, in Sec. 1.5, a paper-oriented summary is given, and when concluding this chapter, in Sec. 1.8.4, an artifact-oriented summary is given.

1.3 The interactive theorem prover HOL4

Since this thesis applies mathematics to software and hardware development, this thesis belongs to the subfield of computer science known as formal methods. Other approaches to ensure the reliability of software and hardware include empirical methods such as testing, which we do not cover here.

This thesis, like the three previous verified-stacks projects mentioned above, follows the approach to mathematics known as interactive theorem proving. In interactive theorem proving, a human and a machine cooperate in constructing proofs. See Geuvers [27], Harrison et al. [37], and Ringer et al. [90] for general background information on interactive theorem proving.

Specifically, all proofs in the papers included in this thesis have been constructed in cooperation with, and have subsequently been checked by, the HOL4 interactive theorem prover [95]. The main way to conduct proof in HOL4 is that a human and HOL4 cooperatively construct a tactic-based proof. HOL4 is also easily scriptable, so HOL4 users can easily add new custom tactics and automated proof procedures as needed.

It is important to use a trustworthy interactive theorem prover since the prover is part of the trusted computing base (TCB) of the theorems proved with help of the prover. Using a trustworthy prover is in particular important in verified-stack projects, since, in such projects, we want to keep the TCB as small as possible. Specifically, the TCB of stacks verified with the help of an interactive theorem prover consists of

- the top-level specification of the stack (e.g., a specification for a user-facing application),
- the bottom-level specification of the stack (e.g., the semantics of the hardware language the bottom level of the stack is implemented in),
- the theorem prover used in verifying the stack (e.g., as here, HOL4).

As long as we are using a trustworthy prover, the above is a remarkably small TCB. Note specifically how no intermediate layers are part of the TCB. This means that correctness arguments for the individual intermediate layers and their connections to surrounding layers can be arbitrarily complex without contributing to the TCB. The number of intermediate layers is also irrelevant for the TCB.

The HOL4 prover is a trustworthy prover since to trust the prover we need only to trust a small kernel of the prover. Specifically, HOL4 is an LCF-style prover, and the architecture of its kernel and the rest of the system is based on the LCF tradition [32, 33, 86]. MacKenzie [69], Klein [56, Sec. 2], Pollack [89], and Wiedijk [111] provide more in-depth discussions about trust concerns in the context of computer mathematics.

1.4 Research questions

We now introduce two research questions that have motivated the work carried out for this thesis. We return to the questions in the end of this chapter.

1.4.1 First research question

The first research question addressed by this thesis is:

> Can we build fully verified stacks based on substantial programs?

Previous verified-stack projects have only, to the best of my knowledge, run small programs on top of their verified hardware stacks. A reasonable next step is thus to investigate if more substantial fully verified programs can be run on top of fully verified hardware.

The CakeML compiler – a verified compiler for the SML-like language CakeML – served as a good starting point to answer the above research question for three reasons:

- The compiler is capable of compiling substantial programs, including compiling itself.
- There exists a small collection of substantial programs written for the compiler.
- The compiler's correctness theorem is specified on the level of machine code, i.e. down to a level appropriate to relate to hardware.

Moreover, I and the developers of the CakeML compiler thought of the compiler as a "realistic" compiler, in the sense that no "unrealistic" (unsatisfiable) assumptions were made in the compiler's correctness theorem. Connecting the compiler to verified hardware allowed us to put this perception about realism to test. In particular, previously, when targeting unverified hardware, assumptions about the runtime environment had to be baked in into the compiler's correctness theorem. For example, assumptions about file input and output.

1.4.2 Second research question

The second research question this thesis addresses is:

> Can the traditional development methodology for verified software based on verified software compilers be replicated for hardware development for a mainstream hardware description language?

Before this thesis, there existed previous work on the semantics of mainstream hardware description languages (HDLs) – both for Verilog (e.g., Gordon [30] and Meredith et al. [73]) and for VHDL (e.g., the collection of approaches in the book *Formal Semantics for VHDL* [57]). There also existed previous work on verified hardware synthesis (e.g., Braibant and Chlipala [15]). But no previous work combined work on mainstream HDLs with work on verified synthesis: the work on mainstream HDLs did not address the verification of synthesis tools, and the work on verified synthesis tools was based

on nonmainstream languages such as Bluespec with a much smaller userbase than the mainstream hardware languages.

The work on the verified Verilog synthesis tool Lutsig is what ended up addressing the research gap formed by the lack of verification projects for verified synthesis tools for a mainstream HDL.

As stated in the research question above, one particular aspect that motivated the work on Lutsig was whether the "traditional development methodology for verified software" could be applied to a mainstream HDL for hardware development. Here, by the "traditional development methodology" we refer to the development approach where software-program-correctness reasoning is carried out on the source level and then transported to the compiled program – the target level – by composing the source-level reasoning with the compiler's correctness theorem. To apply this development approach to Verilog development, as done in the work on Lutsig, both Verilog's hardware-oriented features – such as X values – and Verilog's quirks – such as simulation-and-synthesis mismatches – have to be taken into consideration.

The development methodology described above is important since it can be understood as addressing the practicalities of small-TCB development inside an interactive theorem prover. More precisely, the problem is as follows. When developing software and hardware inside an interactive theorem prover, we need a way to represent the artifacts we build. Two aspects of such representations are important:

- How easy is it to reason about the representation?
- What is the connection between the representation and what is represented? I.e., how large is the formalization gap? I.e., how does the representation strategy contribute to the TCB?

To be able to develop reliable software and hardware inside an interactive theorem prover, we need convincing answers to both questions since a representation that is easy to reason about is not useful if the connection between the representation and what is represented is not clear, and a clear connection between the representation and what is represented is not useful if we cannot reason about the representation. However, finding one single representation that provides satisfying answers to both questions is often difficult since a representation that is easy to reason about usually means a high-level representation, and a representation with a clear connection to what it represents usually means a low-level representation.

Verified compilers provide an answer to the above problem since verified compilers enable us to work with multiple representations and translate between them automatically without increasing the TCB. Specifically, a verified

compiler allows us to establish target-level correctness by

1. first carrying out source-level reasoning, i.e., carry out reasoning based on a high-level representation where reasoning is simple,
2. followed by using the compiler to automatically build a low-level representation, with a clear connection to what is represented, and finally,
3. easily transporting the reasoning about the high-level representation down to the low-level representation by simple composition with the compiler's correctness theorem.

1.5 The papers

This section provides short summaries of the contents of the papers included in this thesis. The following papers are included:

Paper 1 Verified Compilation on a Verified Processor, by Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. Presented at Conference on Programming Language Design and Implementation (PLDI) in 2019.

Paper 2 A Proof-Producing Translator for Verilog Development in HOL, by Andreas Lööw and Magnus Myreen. Presented at Conference on Formal Methods in Software Engineering (FormaliSE) in 2019.

Paper 3 Lutsig: A Verified Verilog Compiler for Verified Circuit Development, by Andreas Lööw. Presented at Conference on Certified Programs and Proofs (CPP) in 2021.

Paper 4 Lutsig 2.0: Verilog, Synthesis-Tool Verification, and Circuit-Verification Methodology, by Andreas Lööw. Paper draft; not yet published.

The papers are included in full as separate chapters following this introductory chapter. The summaries here focus on what is concretely done and achieved in each paper and highlight the relationships between the included papers. For the scientific contribution of each paper, in terms of novelty claims, and how the paper relates to previous work, see the papers themselves. The full abstracts are included in the beginning of each chapter for each included paper.

Paper 1–3 have been published. The three published papers are included as published (except minor clarifications and minor language fixes), with the exception that the appendix of Paper 3 has been inlined into the main body of the paper. Paper 4 is a paper draft and is not yet published.

1.5.1 Paper 1: Verified Compilation on a Verified Processor

In Paper 1 we build verified stacks and we introduce our proof-producing Verilog code generator.

The code generator enables hardware designers to develop circuits in higher-order logic (HOL) and translate them to Verilog circuits. The code generator is based on our Verilog semantics that we also introduce in this paper. The code generator is proof-producing in the sense that it for every run produces (using the HOL4 API) a HOL4 theorem stating that the input HOL circuit and the output Verilog circuit have the same behavior. This allows hardware designers to prove theorems about HOL circuits (specifically, shallowly embedded Verilog designs) and easily transport such theorems to generated Verilog circuits (specifically, generated deeply embedded Verilog designs) generated by the code generator.

Moreover, in the paper we use the code generator to build verified stacks. For this, the paper introduces our verified Silver processor. We build Silver and verify it down to its Verilog implementation with the help of the code generator. In the paper, we extend the verified CakeML compiler such that we can run the compiler on top of Silver. To run the compiler and other programs on top of the processor, we synthesize the Verilog implementation of the processor, using standard (unverified) Verilog synthesis tools, for one of our FPGA boards. All the programs we ran on top of the processor, including both the CakeML compiler itself and a proof checker, have associated correctness theorems stating that the programs satisfy their specifications when run on top of the Verilog implementation of the processor.

Note: In Paper 2 and 3, the code generator is called a "translator". This is because the proof-producing tool in the CakeML project that inspired the Verilog code generator is called "translator" (the CakeML translator translates shallowly embedded CakeML programs to deeply embedded CakeML programs). In the introduction of this thesis, we use the term "code generator" throughout.

1.5.2 Paper 2: A Proof-Producing Translator for Verilog Development in HOL

Paper 2 provides further information on the proof-producing code generator introduced and used in Paper 1. The paper also provides a more in-depth discussion on the Verilog semantics introduced in Paper 1.

1.5.3 Paper 3: Lutsig: A Verified Verilog Compiler for Verified Circuit Development

Paper 3 introduces the verified Verilog synthesis tool Lutsig. Lutsig targets technology-mapped netlists for (a class of) FPGAs[1] and is based on the Verilog semantics developed in Paper 1 and 2. As a result, as of this paper it is now possible to reliably synthesize Verilog designs produced by the proof-producing code generator from Paper 1 and 2 (previously, to do synthesis one had to involve unverified synthesis tools) – as long as the designs are implemented in the (currently small) subset Lutsig supports. For example, the Silver processor from Paper 1 falls outside the supported Verilog subset, but in a case study in the paper we show that a moving-average filter can be successfully compiled down to a netlist. Since Lutsig is verified, properties proved at the Verilog level can be transported down to the netlist level with the help of Lutsig. The moving-average filter case study in the paper shows how to do this kind of property transportation in practice.

1.5.4 Paper 4: Lutsig 2.0: Verilog, Synthesis-Tool Verification, and Circuit-Verification Methodology

Paper 4 continues the work started in Paper 3. To be able to synthesize more Verilog designs, Lutsig must support a larger subset of Verilog. For this reason, the paper introduces a new version of Lutsig that supports a larger subset, specifically, Verilog's `always_comb` construct. Moreover, in extending Lutsig, it becomes apparent that we must think deeper about differences between verifying software compilers and hardware-synthesis tools. In particular, where does concepts from the Verilog world such as synthesis idioms fit into the larger verification picture?

1.5.5 Statement of contribution

This section provides short summaries of my contributions to the four papers included in this thesis.

- For Paper 1, I designed, developed and verified the Silver processor. I was also responsible for the lab hardware setup, including FPGA development and surrounding tooling. I was involved, from the hardware side, in the

[1] Additional background for readers unfamiliar with FPGAs: Lutsig covers a large part of the synthesis process, but not all of the process. For example, before a technology-mapped netlist can be loaded onto an FPGA, it must first be placed and routed. Moreover, the placed-and-routed netlist must be encoded into an FPGA bitstream before it can be loaded onto an FPGA. These subsequent compilation steps are not included in Lutsig and other tools must be used for these steps.

integration of the CakeML compiler and the processor (e.g., adapting the processor and its surrounding components for the integration), but the software work was carried out by the other authors of the paper.

- For Paper 2, I alone have developed everything described by the paper. My co-author gave advice and adjusted the presentation in the paper.
- I am the sole author and developer of both Paper 3 and Paper 4.

1.6 Changes since the papers were published

In this section I look back at the papers included in this thesis. In particular, I discuss what has changed since the papers were published and discuss some limitations.

1.6.1 The Silver processor and the CakeML compiler

I do not have anything new in particular to say about the Silver processor or the connection between the CakeML compiler and the processor. As I see it, the main limitation of the stacks we created was clear already at the time we created them; the main limitation of our stacks is the performance of the stacks, specifically, the performance of the Silver processor. For example, compiling CakeML programs on top of the Silver processor, when synthesized for one of our FPGA chips, takes multiple hours whereas compiling the same program on my laptop finishes almost immediately. Another important limitation is that we only verify the processor down to its Verilog implementation. To load the Verilog implementation onto an FPGA, the implementation needs to be synthesized for the FPGA and encoded into a bitstream. These steps are not part of our formal story, and the tools we used in the project to perform those tasks are therefore part of the TCB of the stacks we built. Hopefully, given enough development investment, Lutsig should eventually be able to synthesize Silver – thereby removing a large part of the unverified tools we relied on in the project out of the TCB.

Erbsen et al. [22], in a paper published after Paper 1, point out some other limitations: "[...] all of [the previous verified stack projects] have simplified the [stack construction and verification] task by restricting interactivity of the application, inventing new simplified instruction sets, and using unrealistic input and output mechanisms." I do not consider these limitations as important as the two mentioned above (i.e., performance and the unverified tools we relied on); I explain why below.

It is true that we in our stack constructions restrict user interactivity and to some extent it is also true that we use "unrealistic input and output mechanisms",

since we preload all application input into memory before starting applications. We took this approach since we wanted to make the input mechanism as simple as possible, to minimize the TCB. But, in retrospect, in the end, it is not clear that preloading input results in a smaller TCB, since it simply moves the complexity of the input mechanisms from the environment model of the stack to whatever is used to preload the input into memory (in our case, an unverified Python script). With that said, I do not see any particular obstacles in replacing the input/output mechanism we utilize in our stacks with an input/output mechanism that supports interactivity, so I do not see this as a severe limitation of the stacks we can create. Instead, I see the particular input/output mechanism we ended up using as a partly accidental/inessential consequence of us only using batch program in our case studies (a compiler, a proof checker, etc.) rather than a necessary choice and inherit limitation of our approach.

As for "inventing new simplified instruction sets" – this is also true to some extent. We, indeed, use a custom ISA rather than an established ISA like x86 or ARM. However, I see the choice to introduce a new ISA rather than using an already established but simple ISA like RISC-V to be largely an artifact of how the processor was developed rather than a consequence of some limitation of our approach. The Silver processor implementation is so simple that the specific encoding used for instructions is unimportant, and I consequently do not see any obstacles in using e.g. the RISC-V ISA instead of the Silver ISA. But at the same time, I agree that using an established ISA rather than a custom ISA would be preferable since by using an established ISA it would be more clear to readers that there are no simplifying assumptions built into the instruction-encoding scheme used. With that said, I am not aware of any simplifications made in the instruction-encoding scheme nor should there by anything else novel about the Silver ISA; the ISA is more an accidental artifact of the development process than anything else. For example, in terms of realism, the Silver ISA is expressive enough to serve as a compilation target for the CakeML compiler. But at the same time, when it comes to verified stacks, including their ISAs and other details, stacks should be evaluated based on what was done, not what could have been done.

1.6.2 The proof-producing Verilog code generator

The proof-producing Verilog code generator was introduced and described in Paper 1 and 2. We call the code generator as it existed when Paper 1 and 2 were published version 1 of the code generator. For Paper 3, I had to revisit the code generator since the Verilog semantics it is based on was updated for that paper, but no major changes were done. For Paper 4, I reimplemented a large part of the code generator. We call the reimplemented version of the

code generator version 2 of the code generator. I reimplemented a large part of the code generator since it was partly needed to support the new features introduced but also to address some internal architectural problems that made using the code generator difficult (I explain this in more detail further down).

All in all, for all versions, the proof-producing Verilog code generator consists of three parts:

- A way to shallowly embed Verilog designs in HOL
- A formal semantics for Verilog, i.e., a way to deeply embed Verilog designs
- A proof-producing algorithm to translate shallowly embedded Verilog designs to their corresponding deeply embedded representation

In the following, I comment on some ways in which the code generator has evolved during the work on the four papers included in this thesis, and I provide some more details that did not fit into any of the papers.

Shallowly embedding Verilog designs in HOL

Both version 1 and 2 of the code generator expect Verilog designs to be shallowly embedded as collections of next-state functions, i.e., functions from state records to updated state records. In both versions, every next-state function is translated to a Verilog process. In version 1, all functions were translated to `always_ff @(posedge clk)` blocks (where `clk` is a circuit-global clock). In version 2, both `always_ff` blocks and `always_comb` blocks are supported.

Inside Verilog processes, we have statements and expressions. We now describe their embeddings.

Expressions are embedded with the help of the HOL word library and standard HOL Booleans. Verilog arrays are embedded as HOL words, and the various Verilog array operators supported by the code generator are simply embedded as their corresponding HOL-word-library analogues. Verilog Booleans are embedded as HOL Booleans in a similarly straightforward manner.

The embedding of statements is less straightforward. In particular some extra thought needs to be allocated to how to embed blocking and nonblocking assignments. In Verilog, nonblocking assignments are used for (some) shared variables and the effect of a nonblocking assignment is not visible until the next clock cycle. Therefore, when embedding a nonblocking assignment in a next-state function, we cannot naively directly update the state record the next-state function operates over since the effect of the assignment would be visible immediately.

In version 1, the above is solved by saying that a process that writes to a shared variable is not allowed to read the variable after it has written to the

variable. Concretely, this is implemented in two steps in the code-generation algorithm. In the first step, all processes are translated from shallowly embedded Verilog to deeply embedded Verilog process-by-process. In this first step, all assignments, including assignments to shared variables, are translated to blocking assignments. After the first step, a second step that operates over deeply embedded Verilog designs starts. This second step translates all blocking assignments to shared variables to nonblocking assignments. Since processes are not allowed to read shared variables after they have written to them, the translation of blocking assignments to nonblocking assignments does not affect the execution of the processes when considered in isolation (for a more precise statement of this, see Paper 2). After this second step, the processes can be merged into one single module since they no longer interfere with each other, since all assignments to shared variables are nonblocking.

In version 2, I changed how nonblocking assignments are embedded. The main reason for this change was to allow to embed more designs (i.e., to make the embedding style more expressive). For example, code like the following can be found in Verilog projects:

```
always_ff @(posedge clk) begin
 a <= inp;
 b <= a;
 c <= b;
end
```

This block is not embeddable in the embedding style used in version 1 of the code generator since both `a` and `b` are read after being written to. It is often possible to manually rewrite code into embeddable form, but if we want to be able to embed Verilog code as written by Verilog programmers, a more powerful embedding style is needed.

To allow embedding blocks like the above, version 2 of the code generator encodes blocks using next-state functions operating over two state records. More precisely, if the state record type is s, then the type of the next-state functions of is $s \to s \to s$ (modulo some details not of relevance here). The first input record contains the values of all variables at the beginning of the current clock cycle, and the second input record contains the most recent values of all variables. Recall that version 2 of the code generator supports `always_comb` blocks. Such blocks are not important for our discussion here since they should never contain nonblocking assignments. For `always_ff` blocks, if we for all next-state functions

- never update the first input record,
- commit all updates to the second record,

- read the values of shared variables from the first record, and
- read the values of nonshared variables from the second record,

then there is a direct correspondence between the next-state function and the same function as implemented as a Verilog process. (Paper 4 includes a concrete example of how this embedding style looks in practice.)

With the new embedding style utilized in version 2, we no longer need a two-step translation process. In version 2 of the code generator, assignments to shared variables are directly translated to nonblocking assignments, and assignments to nonshared variables are, as before, translated to blocking assignments.

The embedding style utilized in version 2 is more complex than the style used in version 1, but since the former allows for embedding a larger number of Verilog designs I consider it a better embedding style. (Moreover, the translation implementation of version 2 turned out much cleaner than the implementation of version 1.)

The code-generation algorithm

This section concerns a minor implementation detail I thought was worth mentioning that changed between version 1 and 2 of the code generator.

Version 1 and 2 of the code generator are based on the same underlying algorithm. However, in version 1, one short-sighted shortcut that was taken in the implementation of the code generator is that the two predicates Eval and EvalS used to express correspondences between shallowly embedded and deeply embedded Verilog code both depended on a predicate relS. The predicate relS is a per-circuit generated predicate used to express that a circuit-specific state record and a Verilog state (for the Verilog semantics) represent the same state. Since Eval and EvalS are used in the implementation of the code generator, this meant that the whole code generator must be recompiled every time a new circuit is to be translated (since the constants used in the code generator depends on a circuit-specific predicate). Without changing the underlying algorithm, this was addressed in version 2 by parameterizing Eval and EvalS over what state relation to use. Most of the code-generator infrastructure thereby becomes independent of circuit-specific constants and therefore does not need to be recompiled when applied to new circuits.

For Paper 4 (i.e. version 2 of the code generator), the above change in combination with adding support for `always_comb` blocks and changing the handing of nonblocking assignments resulted in most of the implementation of the code generator being redone. Therefore, most implementation details mentioned in Paper 2 (i.e., version 1 of the code generator) are now out-of-date

– but the underlying ideas are still the same in the updated version of the code generator.

1.6.3 Verilog synthesis

In Paper 1 and 2 unverified tools are used for Verilog synthesis. After Lutsig was introduced in Paper 3, synthesis from Verilog down to technology-mapped netlists can be done by Lutsig instead. However, introducing Lutsig does not necessitate us to revisit Paper 1 and 2. Lutsig shrinks the TCB of circuits developed using our tools since our formalization of Verilog's semantics no longer needs to be part of the TCB, but the overall development flow is the same as before Lutsig was introduced: (1) use the proof-producing code generator to generate a Verilog design, (2) use a Verilog synthesis tool to map the generated Verilog design to hardware. Lutsig only affects the second step (i.e., (2)), so the first step (i.e., (1)) can remain the same (since both the code generator and Lutsig are based on the same Verilog semantics).

1.7 An amazing story; or: to my surprise, formal methods actually work

To illustrate the value of formally proving computer systems correct, two verification success stories are presented in this section.

Today, it is not a difficult task to find interactive-theorem-proving success stories; one can find everything from theoretical results in abstract mathematics to concrete results in computer science such as verified compilers (see e.g. Ringer et al. [90]). Stacks verified using interactive theorem proving provide one particularly good source of formal-methods success stories. If you ask me, the most exciting part about building verified stacks is seeing the physical realization of the stack actually work in practice when running a proven-correct concrete program. Both the stories presented here are such stories.

The first success story comes from Albin et al. [2] and their work on the CLI stack. Albin et al. report on the testing of the FM9001 microprocessor, which is a processor integrated into the CLI stack. To be more precise, the FM9001 was fabricated by LSI Logic, Inc., as an ASIC, and Albin et al. report on the post-fabrication testing of the physical device. The story in the paper is a story about success, since the processor works as expected for all tests it was subjected to. In Albin et al.'s words: "The testing included both executing FM9001 machine code and also low-level testing with a Tektronix LV500 chip tester. To date, all tests have confirmed that the FM9001 behaves as formally specified." In one part of their report, Albin et al. enumerate a series of successful software tests they

have performed, and they express their satisfaction seeing their proven-correct stack actually do what they have proven it should do as follows:

> In all these cases, the FM9001 microprocessor worked as expected. Perhaps the most satisfying of these software tests is the Nim game-playing program. It was most rewarding to see this rather subtle Piton program, which has been formally proved, with Nqthm, to win if possible, actually win, when possible, while executing on a physical FM9001—and winning within a proven real-time performance envelope.

The second success story is similar but instead comes from work done for Paper 1. In fact, since I am part of the story, I know it is not only a success story – it is an amazing story. With that said, let us jump into the story.

After having finished proving the correctness of a Silver-and-CakeML stack capable of compiling functional programs, it was time to test the stack in practice. The only program we had run on top of Silver before this was a small hello world program. Running the hello world program led us to identify some small integration problems, such as e.g. some small problems in the unverified Python glue scripts used to load machine code into memory, handle input/output (e.g., to print text to a terminal), etc. But with those small problems addressed, it was time to run the full compiler.

In development not based on formal methods, going from running a simple hello world program to running a full compiler is an unthinkable step. I nevertheless loaded the proven-correct machine code of the CakeML compiler into the memory of the FPGA board I had synthesized Silver for and told the compiler to compile a small example program. At first, nothing happened. This is to be expected, since the compiler does not give any output until the compilation process is complete. After an hour of compilation, still nothing. When running the CakeML compiler on my laptop, compiling the same small example program takes almost no time at all. But since the processor inside my laptop is much more complex than Silver, it is of course to be expected that Silver is not as fast. It therefore made sense to continue waiting. At this point, however, the day was over and I went home and let the compilation process run overnight.

When coming back the next day, I was expecting to find the processor in some unexplainable error state. To my surprise, this was not the case. Instead, the compilation process had finished successfully, and the compiler had printed the compilation result, i.e., the machine code of the small example program. Comparing the outputted machine code with reference machine code moreover showed that the compiler had not only outputted machine code, it had outputted exactly the machine code it should output. That is, the compiler executed and

finished successfully the very first time we ran it on top of Silver. An amazing end of an amazing story – to my surprise, formal methods actually work!

1.8 Conclusion

Concluding this chapter, we now ask what remains of the two research questions introduced earlier in this chapter, given the work that has been done in the included papers.

1.8.1 The first research question

The first research question – on running substantial verified programs on top of verified hardware – is answered positively by Paper 1 and 2. Specifically, Paper 1 shows how we have built verified stacks capable of, among others, compiling functional programs and checking proofs. At the same time, it should be acknowledged that further work is possible, as described further in Sec. 1.8.3.

1.8.2 The second research question

The second research question – on enabling the development of verified hardware in a mainstream HDL similarly to how verified software is developed using verified software compilers – is answered positively by Paper 3 and 4. Specifically, Paper 3 introduces the verified Verilog synthesis tool Lutsig and illustrates how to apply Lutsig to hardware development. In Paper 4, Lutsig is developed further, and the supported subset of Verilog is extended. As with the first research question, the limitations of our attempt at answering the second research question should be acknowledged; some further work is described in Sec. 1.8.3.

1.8.3 Future work and questions

Of course, as with any work, it is possible to improve the work carried out in the papers included in this thesis.

One aspect of how the Silver processor was received that surprised me was the skepticism the processor's custom ISA received. To me, the ISA is an artifact of the processor's development process. But, of course, that it is truly an artifact and not a workaround or something similar is something that needs to be supported by an argument. For Lutsig, we have the opposite situation. Lutsig is a very understandable artifact, since Verilog is a well-known language. What is great about this is that even developers with no understanding of the

internals of hardware-synthesis tools can form an image of how useful Lutsig is.

Understandability ties into how I see Lutsig develop over time. The goal of the first version of Lutsig was (of course) not to build a full synthesis tool comparable to unverified commercial tools. Rather, the idea was to build a simple but functioning tool capable of transporting circuit-correctness properties from the Verilog level down to the netlist level. This, if you ask me, is a good starting point for developing a more serious synthesis tool. Since Lutsig is an understandable product, Lutsig is something that can receive critique. Critique of particular interest is what can be considered to be missing from Lutsig. For example, support for level-sensitive latches and support for multiple clock domains are things that have already been mentioned to me. Forming a picture of what is missing based on both external critique and self-driven case studies will help form an idea about in what direction Lutsig should be developed.

Something similar can be said about the verified stacks we have built. Even if there are understandability aspects of the ISA that could be improved, the final artifacts themselves (i.e., the stacks) are understandable products. For example, a verified stack capable of compiling functional programs is understandable as long as you are familiar with compilation. Moreover, the stacks fit into a similar incremental development story as the one imagined for Lutsig. The stacks developed in this thesis are not the first verified stacks to be developed; rather, the stacks address limitations of previous stacks (specifically, running substantial programs). At the same time, the stacks introduced here have limitations themselves. This opens up for further work – however, the bar for future work is, as a consequence of our stacks, now raised. In other words, progress has been made.

Aiming at being more concrete about the future, here follows a short wish list of concrete things that I would have done given more time:

- As mentioned in previous sections, the performance of the stacks we build in Paper 1 is in need of improvement, in particular the Silver processor. To come closer to more realistic stacks, a next step in the development could be to build a stack sufficiently performant to compile programs in reasonable time (rather than, as now, multiple hours).

- Extending the Verilog support of Lutsig such that one could compile the Silver processor using Lutsig would allow us to shrink the TCB of the stacks we have built in Paper 1 (since a large part of unverified hardware synthesis could then be removed from the TCB). Of course, a version of Lutsig with extended Verilog support would also be useful for compiling other larger circuits as well. Moreover, language features like module open up questions about modular verification.

- An even more substantial extension, and a further step towards further realism, would be to start to think about what would be needed for a multicore-based (or even multiprocessor-based) verified stack. Concurrency would of course be a highly nontrivial extension to our work and would require revisiting almost every single layer of our stacks; we would at least need a way to formally reason about concurrent programs, a verified compiler for concurrent programs, and a verified multicore processor.

- A project orthogonal to the above items could be to look into the construction of a more formal connection between the Verilog standard's Verilog semantics and Lutsig's Verilog semantics, since such a connection is valuable for compiling unverified circuits (and circuits verified with respect to other non-Lutsig formalizations of the Verilog standard).

Beyond the two research questions addressed in the beginning of this conclusion, there are many big-picture questions one could ask about the work in this thesis that are not addressed in this thesis: How should arguments for the value formal methods for hardware functional correctness supposedly provide be adapted to the fact that the most recent highly publicized processor problems have been security vulnerabilities (like Heartbleed, Meltdown, and Spectre [39, 72]) rather than functional bugs like the famous FDIV bug [112] from 1994? How should one go about making hardware development based on interactive theorem proving attractive for hardware developers without any background in either the relevant kinds of formal methods or functional programming? Moreover, since we in the work for this thesis have developed a synthesis tool for (currently) FPGAs, hardware-oriented questions beyond the immediate context of verified stacks are relevant as well. For example, what are the appropriate application domains for FPGAs and how are FPGAs best programmed in those different application domains [58, 80, 91, 102, 114]? Specifically, how fit for those application domains is the synthesis tool we have developed (or how fit can it be made)? And how, more generally, should hardware development for non-FPGA target technologies be carried out?

1.8.4 Concluding remarks

Concluding this section, we give one last summary of the contents of the papers. This time we orient the summary around the artifacts contributed by this thesis.

Two artifacts are of relevance for all four included papers: Our Verilog semantics and our proof-producing Verilog code generator. To build correct Verilog artifacts, a formal Verilog semantics is needed. Since all papers include

some kind of Verilog artifact, all papers relate to our Verilog semantics. Moreover, since our Verilog code generator allows for the construction of provably correct Verilog artifacts and all our papers relate to at least one provably correct Verilog artifact, the code generator is of relevance for all papers as well. Specifically, the code generator allows hardware designers to (1) reason using shallowly embedded Verilog circuits, (2) automatically translate their shallow circuits to deeply embedded circuits, i.e., a representation using our Verilog semantics, and (3) easily transport correctness properties from shallow circuits to deep circuits.

Beyond the Verilog semantics and the Verilog code generator, it is possible to form two clusters of the artifacts we introduce and extend in this thesis. The first cluster relates to verified stacks and the second cluster to Verilog synthesis. Both clusters relate to both our Verilog semantics and our Verilog code generator.

The first cluster is constituted by the CakeML compiler and the new Silver processor we introduce in this thesis. In our verified stacks, the CakeML compiler and our extensions to it allow us to run CakeML programs on top of the verified Verilog processor Silver. To develop and prove the Silver processor correct, we used our Verilog code generator. By using unverified synthesis tools, we have synthesized the Silver processor to an FPGA board and run our processor and programs on top of this board. The majority of the work building our verified stacks went into the software-hardware interface and the verified Silver processor (i.e. the hardware part of the stacks). The Silver processor – the Silver ISA specifically – is where software and hardware meet. An ISA (although not the Silver ISA) is a standard interface in unverified stacks; what set verified stacks apart from unverified stacks is that all layers of the stacks are themselves verified and that the layers' connections to surrounding layers are verified as well.

The second cluster is constituted by our verified Verilog synthesis tool Lutsig. Lutsig currently targets FPGAs. Since Lutsig can synthesize Verilog artifacts generated by our Verilog code generator (as long as the generated Verilog code is within the subset of Verilog supported by Lutsig), Lutsig is connected to our Verilog code generator. Since both Lutsig and our Verilog code generator are based on our formal Verilog semantics, connecting them was straightforward.

In the future, I hope to be able to connect the two clusters together, i.e., that processor-size Verilog artifacts can be synthesized using Lutsig.

Chapter 2

Verified Compilation on a Verified Processor

Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox

Abstract. Developing technology for building verified stacks, i.e., computer systems with comprehensive proofs of correctness, is one way the science of programming languages furthers the computing discipline. While there have been successful projects verifying complex, realistic system components, including compilers (software) and processors (hardware), to date these verification efforts have not been compatible to the point of enabling *a single end-to-end correctness theorem* about running a verified compiler on a verified processor.

In this paper we show how to extend the trustworthy development methodology of the CakeML project, including its verified compiler, with a connection to verified hardware. Our hardware target is Silver, a verified proof-of-concept processor that we introduce here. The result is an approach to producing verified stacks that scales to proving correctness, at the hardware level, of the execution of realistic software including compilers and proof checkers. Alongside our hardware-level theorems, we demonstrate feasibility by hosting and running our verified artefacts on an FPGA board.

Published at *Conference on Programming Language Design and Implementation (PLDI)*, 2019

2.1 Introduction

A *verified stack* is a computer system that is demonstrably correct. Specifically, it is a system with a formal proof of correctness that covers all layers of the implementation, from the hardware through to the application code. Enabling the construction of verified stacks is a guiding light for the field of formal verification; several projects have made progress towards its achievement [3, 5, 96]. In this paper, we report on a milestone in this tradition: a verified stack consisting of a verified processor that we have synthesized for an FPGA board on which we can run a realistic and verified compiler.

To reach this milestone, we have developed a new verified processor called Silver[1] that is simple but general-purpose, and we have extended the trustworthy chain in the CakeML project with a link to Silver. The Silver processor was verified with ease thanks to a new proof-producing hardware generator that is grounded in a semantics for the hardware description language (HDL) Verilog. To produce machine code for Silver, we use the CakeML translator [43, 79] and compiler [25, 99]. We obtain end-to-end correctness theorems by composing the CakeML compiler's correctness theorem with the Silver processor's correctness theorem.

Our combination of Silver with CakeML yields a general method for verification down to the hardware level. Given a high-level executable specification of behaviour, our method produces machine code for Silver plus an end-to-end correctness theorem stating that the verified Silver hardware will have the observable behaviour of the original high-level specification, provided the generated Silver machine code is initially present in memory.

We demonstrate our method on several applications taken from CakeML's library, including word-count, sort, a proof-checker for OpenTheory proofs [46], and the CakeML compiler itself. To our knowledge, this is the first verified stack development that scales to the point of executing a realistic compiler on top of verified hardware, in a setting with *a single correctness theorem* that covers the full end-to-end composition.

Previously, the CakeML compiler targeted only architectures without verified implementations [25], such as x86 and ARM. When the target architecture has no correctness proof, the hardware and runtime environment must be modelled as assumptions in the compiler's correctness theorem. In this paper, by targeting the verified Silver processor, we address the question "How can the CakeML compiler (and other verified compilers) be extended to reduce assumptions about the hardware and environment?" In explaining this, we

[1] As silverware may be used in consuming cakes and other food, Silver is hardware that can run CakeML as well as other programs.

make the following contributions:

- We exhibit sufficient properties that, if proved about a compiler and a processor, enable them to be used together for constructing verified stacks. The shape of the correctness proofs in our method (§2.2) should be informative for other verified stacks.

- We show how we constructed and verified (§2.3) the Silver processor (§2.4) down to its implementation in Verilog, a mainstream low-level hardware description language. (The software side of our method has been described elsewhere [79, 99].)

- We address claims [59, 99] that the assumptions on the CakeML compiler's correctness theorem (§2.5) are reasonable, by showing how they can be satisfied (§2.6), and highlight minor changes that were required (§2.6.1).

- Finally, we contribute the Silver processor and extensions to CakeML to support Silver as reusable artefacts for constructing verified stacks.

The whole development, including the CakeML compiler and the Silver processor, has been built using the HOL theorem prover [95]; the source code and proofs are available at `https://github.com/CakeML/cakeml` and `https://github.com/CakeML/hardware`.

2.2 Approach

Our approach to building verified stacks divides concerns, just as in the traditional approach to building (unverified) systems, with steps including:

1. Write functional specifications for the application.

2. Implement the specifications as source code in a high-level programming language.

3. Compile the source code to machine code.

4. Link the application machine code with code implementing any required system calls.

5. Run the resulting machine code on a processor (connected to memory and I/O devices).

The main omission is interaction with an operating system: at present, we focus on applications that run on "bare metal". In order to produce *verified* stacks, we have a verification story for each of the steps above, and we produce a single end-to-end theorem that composes the correctness theorems associated with each step.

Now, let us turn to the specific components we use to instantiate the template above and their associated verification story. To make things concrete, we consider an example application, namely, wc, a program that counts the words it receives as input.

2.2.1 Specification

We write formal specifications in higher-order logic, specifically by defining functions in the HOL theorem prover. For wc, these include functions used in logical expressions such as: $|\mathsf{tokens\ is\_space}\ input|$, which is the length $|\ldots|$ of the tokens function applied to the is_space function and $input$. We summarise the specification for wc as a relation, wc_spec $input$ $output$, between input and output strings.

2.2.2 High-level implementation

We implement the application in CakeML, generating code from the specification whenever possible. To do this, we use the CakeML translator [43, 79], which, given specifications that are pure functions or monadic functions representing impure computations, produces both an implementation in CakeML code and a certificate theorem.[2] Ultimately, we obtain a proof that a CakeML program, wc_prog in our example, successfully terminates[3] with output conforming to the specification[4]:

$$\begin{array}{l}\vdash \exists\, io.\\ \quad \mathsf{cakeml\_sem}\ ([\text{“wc”}],\mathsf{fs}_{\mathsf{in}}\ input)\ \mathsf{wc\_prog} = \{\ \mathsf{Terminate\ Success}\ io\ \}\ \wedge\\ \quad \mathsf{wc\_spec}\ input\ (\mathsf{get\_stdout}\ io)\end{array} \tag{2.1}$$

Here [“wc”] represents the command line, and $\mathsf{fs}_{\mathsf{in}}\ input$ represents the filesystem state when the program starts, in this case with no files but with the string $input$ to be read on standard input.

[2] If parts of the desired implementation cannot be translated from the specification, we write CakeML code by hand and verify it using Characteristic Formulae [35] in the tradition of Hoare logic.

[3] CakeML supports correctness proofs for nonterminating programs, but we do not cover them in this paper.

[4] Equality (=) binds tighter than conjunction ($\wedge$); termination and conformance are represented by the two conjuncts.

2.2.3 Compilation to machine code

We compile the CakeML code to Silver machine code using the optimising CakeML compiler, selecting the new target, Silver (ag32), that we have added to the compiler backend. The compiler is proved correct once and for all, with a correctness theorem of this form:

$$\begin{array}{l}\vdash \mathsf{cakeml\_sem}\ (cl,fs)\ prog = behaviours \wedge \\ \quad \mathsf{Fail} \notin behaviours \wedge \\ \quad \mathsf{compile\ conf_{Ag}}\ prog = \mathsf{Some}\ compiled\_prog \wedge \\ \quad \mathsf{installed_{Ag}}\ compiled\_prog\ (\mathsf{basis\_ffi}\ cl\ fs)\ ms \Rightarrow \\ \quad\quad \mathsf{machine\_sem}\ (\mathsf{basis\_ffi}\ cl\ fs)\ ms \subseteq \mathsf{extend\_with\_oom}\ behaviours\end{array} \tag{2.2}$$

Here, basis_ffi cl fs models the behaviour of system calls that access the command line cl and file system fs as expected by CakeML's basis library; machine_sem produces a set of behaviours by repeatedly stepping the machine state ms, applying external interference and executing system calls according to the basis_ffi model [25]; and extend_with_oom adds additional behaviours to the compiled program, namely, allowing it to terminate prematurely with an out-of-memory error, having done only a prefix of the correct I/O events.

We execute the compiler inside the theorem prover (essentially by rewriting with its definition) in order to obtain a compiled program (i.e., bytes of machine code), here wc_ag32, and a compilation theorem:

$$\vdash \mathsf{compile\ conf_{Ag}\ wc\_prog} = \mathsf{Some\ wc\_ag32} \tag{2.3}$$

Instantiating the compiler-correctness theorem (2.2) with the compilation theorem (2.3) and the application-semantics theorem (2.1), we obtain a correctness theorem about the application machine code. For wc, the theorem is:

$$\begin{array}{l}\vdash \mathsf{installed_{Ag}\ wc\_ag32}\ (\mathsf{basis\_ffi}\ [\text{“wc”}]\ (\mathsf{fs_{in}}\ input))\ ms \Rightarrow \\ \quad \exists io. \\ \quad\quad \mathsf{machine\_sem}\ (\mathsf{basis\_ffi}\ [\text{“wc”}]\ (\mathsf{fs_{in}}\ input))\ ms \subseteq \\ \quad\quad\quad \mathsf{extend\_with\_oom}\ \{\ \mathsf{Terminate\ Success}\ io\ \} \wedge \\ \quad\quad \mathsf{wc\_spec}\ input\ (\mathsf{get\_stdout}\ io)\end{array} \tag{2.4}$$

This theorem, relating the semantics of a machine-code program, wc_ag32, to the high-level specification, wc_spec, is what the CakeML project provided prior to this paper. Crucially, this theorem has an assumption constraining the initial machine state, namely $\mathsf{installed_{Ag}}$ wc_ag32 ..., which states that the compiled program is loaded correctly and that the external environment (via system calls and interference) behaves as modelled (§2.5). By targeting a verified processor and verifying the system library, we reduce this assumption, replacing

substantial parts of it by proofs.

2.2.4 Verified system calls

CakeML programs interact with their environment via system calls for reading command-line arguments and reading/writing to standard streams and files. For Silver, we have realised the standard streams `std{in,out,err}`, and the command line, as in-memory devices accessed by Silver machine code that we have verified to implement the system calls required by CakeML. The input devices are prefilled before execution, and the output devices are connected to a text terminal. As we are developing bare-metal applications, the verified system-calls code is included as part of the memory image loaded at startup.

Our theorems about the system calls (§2.6) enable us to replace the $\mathsf{installed}_{\mathsf{Ag}}$ assumption in theorem (2.4) with a simpler assumption, $\mathsf{init}_{\mathsf{Ag}}\ \ldots$, merely stating that the compiled code, system-calls code, and input data are in memory. The resulting theorem has this form:

$$
\begin{array}{l}
\vdash |input| \leq \mathsf{stdin\_size} \wedge \\
\quad \mathsf{init}_{\mathsf{Ag}}\ \mathsf{wc\_ag32}\ ([\text{“wc”}], input)\ ms \Rightarrow \\
\quad\ \exists io\ k. \\
\qquad \mathsf{machine\_sem}\ (\mathsf{basis\_ffi}\ [\text{“wc”}]\ (\mathsf{fs}_{\mathsf{in}}\ input))\ (\mathsf{Next}^k\ ms) \subseteq \\
\qquad\ \mathsf{extend\_with\_oom}\ \{\ \mathsf{Terminate\ Success}\ io\ \}\ \wedge \\
\qquad \mathsf{wc\_spec}\ input\ (\mathsf{get\_stdout}\ io)
\end{array}
\tag{2.5}
$$

Here, $\mathsf{Next}^k\ ms$ is the result of k steps of execution from the initial machine state ms, corresponding to execution of startup code that sets a few registers to satisfy the initialisation assumptions of CakeML (§2.5); and stdin_size is a constant representing the maximum amount of prefilled input we support (about 5 MB).

Working through the definition of machine_sem (essentially, repeated application of Next, using basis_ffi to handle system calls) and using our verified system-calls code, we obtain the following version of this theorem phrased entirely in terms of the Silver ISA and its next-state function Next:

$$
\begin{array}{l}
\vdash |input| \leq \mathsf{stdin\_size} \wedge \\
\quad \mathsf{init}_{\mathsf{Ag}}\ \mathsf{wc\_ag32}\ ([\text{“wc”}], input)\ ms \Rightarrow \\
\quad\ \exists io.\ \mathsf{FG}\ k. \\
\qquad \mathsf{wc\_spec}\ input\ (\mathsf{get\_stdout}\ io) \wedge \\
\qquad \mathsf{is\_halted}_{\mathsf{Ag}}\ (\mathsf{Next}^k\ ms) \wedge \\
\qquad \mathsf{stdout}_{\mathsf{Ag}}\ (\mathsf{Next}^k\ ms).\mathsf{io\_events} \preccurlyeq \mathsf{get\_stdout}\ io \wedge \\
\qquad (\mathsf{exit\_code\_0}_{\mathsf{Ag}}\ (\mathsf{Next}^k\ ms) \Rightarrow \\
\qquad\quad \mathsf{stdout}_{\mathsf{Ag}}\ (\mathsf{Next}^k\ ms).\mathsf{io\_events} = \mathsf{get\_stdout}\ io)
\end{array}
\tag{2.6}
$$

We use the FG operator here to capture the notion that a predicate becomes true at some unspecified point in the future and then remains true thereafter.[5] Thus we see that the Terminate Success io behaviour of machine_sem corresponds to execution for some number of steps, at which point the machine reaches a halting state ($\mathsf{is\_halted}_{\mathsf{Ag}}$ ...), which is a program-specific location where the machine remains for any further steps. Furthermore, at this point, the trace of writes to `stdout` ($\mathsf{stdout}_{\mathsf{Ag}}$ ...) will be a prefix ($\preccurlyeq$) of the specified output, attaining equality if the machine exited successfully ($\mathsf{exit\_code\_0}_{\mathsf{Ag}}$ ...) without running out of memory.

The assumptions on theorem (2.4) that we have proved to reach theorem (2.6) cannot be discharged when developing applications for unverified operating systems. In particular, to reach theorems (2.5) and (2.6), we must prove (see §2.6) that the system calls run correctly under the same ISA semantics, Next, that runs the application code.

2.2.5 Execution on a verified processor

We have proved that the Silver processor, as a hardware circuit, silver_cpu_verilog, implemented in the HDL Verilog, implements the Silver ISA (with next-state function Next). To prove this, we used our new proof-producing Verilog code generator (§2.3). The processor-correctness theorem for Silver is as follows:

$$\begin{array}{l}\vdash \mathsf{let}\ \mathit{vstep}\ =\ \mathsf{verilog\_sem}\ \mathit{env}\ \mathsf{silver\_cpu\_verilog}\ \mathit{init}\ \mathsf{in}\\ \quad \mathsf{is\_lab\_env\ acc\_env\_verilog}\ \mathit{vstep}\ \mathit{env}\ \wedge\\ \quad \mathsf{ag32\_eq\_init\_isa\_verilog}\ (\mathit{env}\ 0)\ \mathit{ms}\ \mathit{init} \Rightarrow\\ \quad\ \forall k.\ \exists m\ \mathit{fin}.\\ \quad\quad \mathit{vstep}\ m = \mathsf{Ok}\ \mathit{fin}\ \wedge\\ \quad\quad \mathsf{ag32\_eq\_isa\_verilog}\ (\mathit{env}\ m)\ (\mathsf{Next}^k\ \mathit{ms})\ \mathit{fin}\end{array} \tag{2.7}$$

Here, $\mathit{vstep}\ m$ is the state of the Silver processor implementation after m clock cycles. The env function is used to represent processor-external entities (is_lab_env ...), such as memory. The two relations ag32_eq_init_isa_verilog and ag32_eq_isa_verilog belong to a family of relations used to express equality between processor states at different abstraction levels and specify the values of various implementation-level registers (see §2.4). The theorem thus states that any number of steps k taken by the ISA can be simulated by a number of steps m by the implementation.

Combining theorem (2.7) with theorem (2.6), and working through some

[5] The FG operator, based on temporal logic, is defined as: $(\mathsf{FG}\ t.\ P\ t) \stackrel{\mathsf{def}}{=} \exists t_0.\ \forall t.\ t_0 \leq t \Rightarrow P\ t$

details about the processor state, we obtain:[6]

$$
\begin{array}{l}
\vdash \mathsf{let}\ \mathit{vstep}\ =\ \mathsf{verilog\_sem}\ \mathit{env}\ \mathsf{silver\_cpu\_verilog}\ \mathit{init}\ \mathsf{in} \\
\quad |\mathit{input}| \leq \mathsf{stdin\_size} \wedge \\
\quad \mathsf{is\_lab\_env}\ \mathsf{acc\_env\_verilog}\ \mathit{vstep}\ \mathit{env} \wedge \\
\quad \mathsf{verilog\_init}_{\mathsf{Ag}}\ \mathsf{wc\_ag32}\ ([\text{“wc”}],\mathit{input})\ \mathit{init}\ \mathit{env} \Rightarrow \\
\quad\ \exists\, \mathit{output}.\ \mathsf{FG}\ m.\ \exists\, \mathit{fin}. \\
\qquad \mathsf{wc\_spec}\ \mathit{input}\ \mathit{output} \wedge \mathit{vstep}\ m = \mathsf{Ok}\ \mathit{fin} \wedge \\
\qquad \mathsf{verilog\_is\_halted}_{\mathsf{Ag}}\ \mathit{fin} \wedge \\
\qquad \mathsf{stdout}_{\mathsf{Ag}}\ (\mathit{env}\ m).\mathsf{io\_events} \preccurlyeq \mathit{output} \wedge \\
\qquad (\mathsf{verilog\_exit\_code\_0}_{\mathsf{Ag}}\ \mathit{fin} \Rightarrow \\
\qquad\quad \mathsf{stdout}_{\mathsf{Ag}}\ (\mathit{env}\ m).\mathsf{io\_events} = \mathit{output})
\end{array}
\tag{2.8}
$$

Theorems of this form are our milestone: working from a high-level specification (wc_spec), theorem (2.8) states that a piece of *hardware* (described in Verilog) implements that specification. Importantly, the creative verification work of the programmer is done at the high level of a CakeML program, not at the hardware level of Verilog.

From the verified circuit silver_cpu_verilog, we have synthesised the Silver processor for an FPGA board. (This is possible because our code generator produces synthesisable Verilog.) If we load a memory image (§2.6) containing the machine code wc_ag32, the input text, and the system-calls code onto the board with the synthesised processor and set it running, the board outputs the number of words in the input (i.e., it runs wc).

The software half of the approach (the production of verified machine code from functional specifications) is explained in detail in previous work [25, 79, 99] that we do not repeat. Our focus is on the hardware side and the software-hardware connection. We start by explaining how we developed and verified Silver (§2.3, §2.4), then describe the connection to CakeML (§2.5) including verification of the system calls (§2.6), and finally present the hardware-level correctness theorem for the CakeML compiler itself (§2.7) and discuss what remains in the trusted computing base (§2.8).

2.3 Producing verified hardware

We have developed a new proof-producing Verilog code generator that translates HOL functions modelling circuits to deeply embedded Verilog programs. The Verilog programs are animated by a new operational semantics for a subset of Verilog that we have developed in parallel with the code generator. The code generator enables relating circuit verification results to a deeply embedded

[6] The predicates with a `verilog_` prefix are analogues of those in theorem (2.6) defined at the Verilog rather than the ISA level.

semantics for a mainstream low-level HDL (here, Verilog), which is novel (see §2.9). The output from the Verilog code generator can be pretty-printed and fed into synthesis toolchains, such as Xilinx's Vivado Design Suite which we have used, to produce FPGA artefacts.

Example. The code generator takes as input a *circuit function* in HOL. A circuit function takes a world-state function env, a circuit-state s, and a clock n and returns the circuit state after n cycles. Circuit functions are expressed in terms of next-state functions representing Verilog processes. Our example circuit AB consists of two processes A and B that count the number of pulses (env.pulse) and set done to true (T) after 10 pulses. Here $1w$ and $10w$ are word literals (with lengths inferred from context); $<_+$ denotes unsigned less-than; with updates a record; and $\langle\!| \ldots |\!\rangle$ constructs a record:

$$
\begin{array}{l}
\mathsf{A}\ env\ s \stackrel{\mathsf{def}}{=} \\
\quad \mathsf{if}\ env.\mathsf{pulse}\ \mathsf{then}\ s\ \mathsf{with}\ \mathsf{count}\ :=\ s.\mathsf{count} + 1w\ \mathsf{else}\ s \\
\\
\mathsf{B}\ s \stackrel{\mathsf{def}}{=} \\
\quad \mathsf{if}\ 10w <_+ s.\mathsf{count}\ \mathsf{then}\ s\ \mathsf{with}\ \mathsf{done}\ :=\ \mathsf{T}\ \mathsf{else}\ s \\
\\
\mathsf{AB}\ env\ s\ 0 \stackrel{\mathsf{def}}{=} s \\
\mathsf{AB}\ env\ s\ (\mathsf{Suc}\ n) \stackrel{\mathsf{def}}{=} \\
\quad \mathsf{let}\ s' = \mathsf{AB}\ env\ s\ n\ \mathsf{in} \\
\quad\ \langle\!|\mathsf{count}\ :=\ (\mathsf{A}\ (env\ n)\ s').\mathsf{count}; \\
\quad\quad \mathsf{done}\ :=\ (\mathsf{B}\ s').\mathsf{done}|\!\rangle
\end{array}
$$

If, as in AB, the input processes do not interfere with each other—i.e., all writes to variables used for communication between processes can be handled by Verilog's nonblocking assignment construct—then the code generator produces a (deeply embedded) Verilog process for each HOL next-state function and then composes them into a complete Verilog module. For the AB example, the code generator produces a Verilog module ABv containing the following code:

```
always_ff @ (posedge clk) // A
 if (pulse) count <= count + 8'd1;
always_ff @ (posedge clk) // B
 if (8'd10 < count) done = 1;
```

The code generator is proof-producing: for each run it produces a correspondence theorem stating that the generated Verilog program has the same behaviour as the input HOL circuit function. This correspondence theorem enables us to transfer properties proved about HOL-level hardware descriptions to the Verilog level. To illustrate, if we assume that the input pulse to AB is high

infinitely often,

$$\mathsf{pulse\_spec}\ env \stackrel{\mathsf{def}}{=} \forall\, n.\ \exists\, m.\ (env\ (n + m)).\mathsf{pulse},$$

then we can easily prove

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

which in turn can be transported to the level of the Verilog semantics using the generated correspondence theorem:

$$\begin{array}{l}\vdash \mathsf{pulse\_spec}\ env \wedge \mathsf{vars\_has\_type}\ s\ \mathsf{ABtypes} \Rightarrow \\ \quad \exists\, n\ s'. \\ \qquad \mathsf{verilog\_sem}\ env\ \mathsf{ABv}\ s\ n = \mathsf{Ok}\ s' \wedge \\ \qquad \mathsf{verilog\_get\_var}\ s'\ \text{“done”} = \mathsf{Ok}\ (\mathsf{VBool\ T})\end{array}$$

Tool implementation. Our Verilog code generator is inspired by the CakeML translator [43, 79], and the Verilog semantics that goes with it is based on the official Verilog standard [48]. We aimed at soundly capturing the standard for a restricted subset of Verilog that we found to be sufficient for describing simple synthesisable synchronous hardware.

In our semantics, we consider a flattened module hierarchy, where all processes correspond to `always_ff` procedural blocks waiting on a program-common clock's `posedge`. We limit the amount of concurrency we need to model by only considering noninterfering processes, where all nonblocking writes are saved in a queue during cycle execution. The contents of this queue are merged into the program state at the end of every clock cycle.

The code generator translates HOL Booleans to Verilog Booleans and HOL words to Verilog arrays. The Verilog Booleans in our semantics only take on the standard Boolean values true (`1`) and false (`0`), as we do not consider wires driven by multiple drivers (`Z`) in our formalisation, and unknown values (`X`) are modelled using quantification inside the logic. The details of the code generator and the semantics are described in more detail in Lööw and Myreen [68].

2.4 The Silver CPU

In this section, we present the Silver ISA, the Silver FPGA implementation and the environment it executes in, and the correctness theorem relating the ISA and its implementation. Figure 2.1 outlines the relationships between the different layers of the implementation of the Silver ISA.

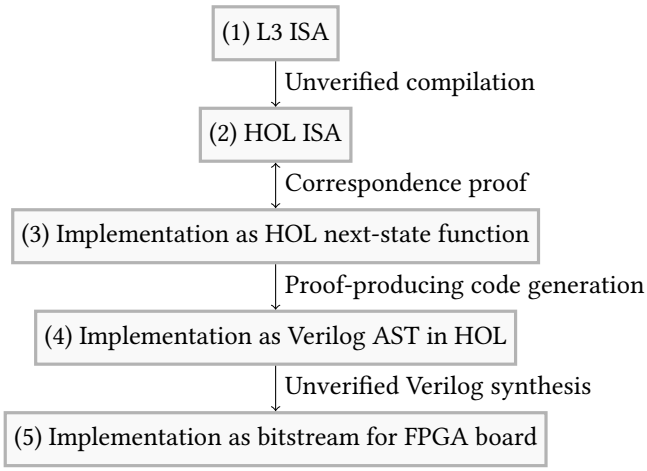

Figure 2.1. The layers involved in the construction and verification of the Silver processor.

2.4.1 The Silver ISA

The Silver ISA (instruction set architecture) is the target of the CakeML compiler's Silver backend. As shown in the topmost layer of Figure 2.1, the ISA is written in the L3 language [24], a domain-specific language for ISAs that is also used for the other CakeML compiler targets and can be transformed into HOL definitions by the L3-to-HOL compiler. Neither the L3 ISA nor the L3-to-HOL compiler are part of the trusted base of the Silver processor: L3 is merely a convenient way to generate HOL definitions of the Silver ISA that can be independently inspected. The generated ISA—the second layer in Figure 2.1—is what is used in our proofs.

The Silver ISA in HOL is an operational semantics over a machine state represented as a HOL record $\langle|\ldots|\rangle$. The machine state contains memory (a function from addresses to bytes), registers (a function from register indices to words), the current program counter (PC), some flags, and a trace of I/O events. The semantics, Next, is a fetch-execute function that retrieves the bytes from memory pointed to by the program counter, decodes them into an instruction, then executes the instruction by updating the machine state. For example, the

execute part of Silver's LoadConstant instruction is specified as follows:

$$
\begin{array}{l}
\mathsf{LoadConstant}\ (reg{,}negate{,}imm)\ s \stackrel{\mathsf{def}}{=} \\
\quad \mathsf{let}\ v\ =\ \mathsf{w2w}\ imm\ \mathsf{in} \\
\quad\ s\ \mathsf{with} \\
\quad\ \langle\!|\mathsf{R}\ :=\ s.\mathsf{R}\langle\!|reg \leftarrow \mathsf{if}\ negate\ \mathsf{then}\ -v\ \mathsf{else}\ v|\!\rangle; \\
\quad\ \ \mathsf{PC}\ :=\ s.\mathsf{PC} + 4w|\!\rangle
\end{array}
$$

Here w2w is an unsigned resizing of words, with updates a record, $f\langle\!|k \leftarrow v|\!\rangle$ denotes a function identical to f except that $f\ k = v$, and $4w$ is a word literal. The reg, $negate$, and imm parameters are information from the instruction decoder specifying which register to update with what content. We see that the function LoadConstant updates the state-record fields R (registers) and PC (program counter). There are similar semantics functions for each instruction in the Silver ISA.

Instruction listing

The Silver ISA is a general-purpose RISC ISA designed to support CakeML. Each instruction is 32 bits long and operates over 32-bit words. The Silver ISA has its roots in Thacker's Tiny 3 computer [101] but has since evolved significantly.

Loading constants into registers. The CakeML compiler frequently wants to load large constants into registers. The Silver ISA supports loading a 23-bit immediate (or its negation) into the lower bits of a register. With another instruction, Silver supports loading a 9-bit immediate value into the upper bits of a register.

ALU operations. The Silver ISA provides instructions for two-argument ALU operations. The ALU supports integer addition, integer add with carry, integer subtraction, increment by one, decrement by one, multiplication (with 64-bit output), logical and, logical or, logical xor, equality, unsigned less-than, signed less-than, retrieving the current carry flag, retrieving the current overflow flag, and simply returning the second operand. The add and subtraction operations update the carry and overflow flags.

Shifts and rotations. Separately from the ALU instructions, there are bit-shift and bit-rotation instructions, in both signed and unsigned variants where necessary.

Memory. Memory can be stored or loaded either as words or individual bytes.

Jumps. The Silver ISA supports conditional and unconditional PC-relative jumps, as well as unconditional jumps to absolute addresses. Jump offsets (or addresses) can be computed (i.e., obtained from a register), which is important when tail-calling a closure or returning from a function (moving the value of the link register into the PC).

Interrupt. The Silver ISA includes an Interrupt instruction, which is used for notifying external hardware of an observable event. In the implementation, Interrupt notifies external hardware and waits for a response before continuing execution. In the semantics of the ISA, Interrupt silently records the current state of memory by pushing it onto the trace of I/O events.

2.4.2 The Silver implementation

We have constructed a Silver processor, implementing the Silver ISA, that is designed for the PYNQ-Z1 FPGA SoC board. The target board is relevant in that any implementation must be adapted to the I/O and memory devices available. The centrepiece of our implementation is an environment-independent processor core, which is connected to the runtime environment by a layer of environment-dependent glue.

The PYNQ board hosts an FPGA chip, an ARM core running Linux which is accessible over SSH, and a DRAM module that is shared between the FPGA chip and the ARM core. In our "lab setup", when we execute a program compiled by the CakeML compiler, we use the ARM core first to load the synthesised Silver processor, as an FPGA bitstream, onto the FPGA chip and then to preload the shared DRAM module with the appropriate memory image (the image (§2.6) contains machine code produced by the compiler, our system-calls code, and data for the command line and standard input).

Formally, we represent the external environment the processor interacts with as a function env from timesteps to the state of the world. The environment is assumed to include a memory interface (is_mem, the DRAM module), an initialisation interface (is_mem_start_interface, notifying when memory is correctly prefilled), and an interrupt-handling interface (is_interrupt_interface, invoked when an Interrupt instruction is executed):

$$\begin{aligned}&\mathsf{is\_lab\_env}\ \mathit{accessors}\ \mathit{step}\ \mathit{env} \stackrel{\text{def}}{=}\\ &\quad \mathsf{is\_mem}\ \mathit{accessors}\ \mathit{step}\ \mathit{env} \wedge \mathsf{is\_mem\_start\_interface}\ \mathit{env} \wedge\\ &\quad \mathsf{is\_interrupt\_interface}\ \mathit{accessors}\ \mathit{step}\ \mathit{env}\end{aligned}$$

The $accessors$ argument is an implementation detail and makes the definitions usable at multiple abstraction levels. As an example, the initialisation interface

is formalised as:

$$\begin{array}{l}\mathsf{is\_mem\_start\_interface}\ env \stackrel{\mathsf{def}}{=} \\ \quad \exists\, n. \\ \quad\quad (\forall\, m.\ m < n \Rightarrow \neg(env\ m).\mathsf{mem\_start\_ready}) \wedge \\ \quad\quad (env\ n).\mathsf{mem\_start\_ready}\end{array}$$

In our lab setup, the interrupt interface is connected to the ARM core and it is used to notify the core of, e.g., system calls it must react to, such as text-output calls.

To produce a hardware description of the Silver processor inside HOL—that is, layer 3 in Figure 2.1—we refined the HOL ISA step by step into a hardware description. The implementation is not pipelined, executes instructions in-order, and is consequently similar, at a high level, to the ISA.

The main difference between the implementation and the ISA is that the implementation must interact with the external interfaces defined above, e.g., instead of updating an abstract memory map as in the ISA, the implementation must access external memory (using the interface defined by is_mem). As a result, the implementation has additional wait states that do not correspond to any state in the ISA, as the processor sometimes has to wait for external interfaces, such as memory, to respond to requests. Because of these additional states, there are two different notions of time. In the ISA, a "step" corresponds to an *instruction cycle*, whereas an implementation-level "step" corresponds to a *clock cycle*; an instruction cycle takes multiple clock cycles to realise in hardware.

Another important step in the refinement process was deduplication of some elements of the ISA. For illustration, consider the definition of the instruction LoadConstant given above. The computation of the next PC is computed directly in the definition of the instruction's semantics. This pattern is repeated for every instruction (except, e.g., jump instructions, where the next PC is computed by more complicated means). One does not want to translate descriptions of instructions such as these to hardware naively, because then the hardware component computing the next PC would be duplicated one time per instruction, wasting hardware resources. Computing the next PC should instead be carried out by a single, shared, hardware component. So, part of the manual refinement process was to identify which parts of the ISA should be implemented by shared hardware components and which could be implemented in a more direct way, similar to the structure suggested by the ISA.

2.4.3 Algorithmic correctness of Silver

To show that the implementation correctly implements the ISA, we have proved a simulation correspondence between the two levels, saying that for any n instruction cycles the ISA can take, these steps can be simulated by running the implementation m clock cycles:

$$\begin{array}{l}\vdash \mathsf{let}\ cstep\ =\ \mathsf{silver\_cpu}\ init\ env\ \mathsf{in}\\ \quad \mathsf{ag32\_eq\_init\_hol\_isa}\ (env\ 0)\ init\ s\ \wedge\\ \quad \mathsf{is\_lab\_env\ acc\_env}\ cstep\ env \Rightarrow\\ \quad\ \forall n.\ \exists m.\\ \quad\quad \mathsf{ag32\_eq\_hol\_isa}\ (env\ m)\ (cstep\ m)\ (\mathsf{Next}^n\ s)\end{array} \tag{2.9}$$

The relation ag32_eq_init_hol_isa belongs to the family of state relations mentioned in §2.2 and says that all ISA-visible state components at the two levels must be equal, e.g., that the memories have the same content and the registers are element-wise equal. The relation also says that some implementation registers must be in their start-up states. The relation ag32_eq_hol_isa is similar to ag32_eq_init_hol_isa in that it also says that all ISA-visible state components (again including memory) must be equal, but it differs by stating that some implementation registers must now be in their in-execution states. The two different state-equality relations are needed as the implementation initially needs to wait for memory to respond with a first instruction before it can proceed in in-execution mode. Lastly, silver_cpu is the HOL hardware description of the processor, in the form of a next-state function expressed such that it is accepted as input by our Verilog code generator.

The simulation correspondence is quite weak, as it does not provide any information about what happens during the implementation's execution of an instruction. In particular, it does not tell us anything about the wait states mentioned in the previous section. For example, in terms of the wc example in §2.2, to prove theorem (2.8), beyond theorem (2.7) and theorem (2.6), we needed a separate lemma saying that the processor "does nothing" after a CakeML program has terminated; or, in other words, a lemma stating that the ISA-visible state is unchanged at any clock cycle after program termination, not just at any instruction cycle.

2.4.4 Correctness of the Verilog implementation

The correspondence between the HOL processor implementation and the Verilog processor implementation is more direct than the correspondence between the HOL processor implementation and the processor ISA. More precisely, with the help of the Verilog code generator invoked on the HOL processor, we can

prove the following theorem:

$$\begin{array}{l}\vdash \mathsf{ag32\_eq\_hol\_verilog}\ \mathit{init}\ \mathit{vs} \Rightarrow \\ \quad \exists\, \mathit{vs}'. \\ \qquad \mathsf{verilog\_sem}\ \mathit{env}\ \mathsf{silver\_cpu\_verilog}\ \mathit{vs}\ n = \mathsf{Ok}\ \mathit{vs}' \wedge \\ \qquad \mathsf{ag32\_eq\_hol\_verilog}\ (\mathsf{silver\_cpu}\ \mathit{init}\ \mathit{env}\ n)\ \mathit{vs}'\end{array} \tag{2.10}$$

Here ag32_eq_hol_verilog is another relation from the state-equality family, again requiring that its two parameters represent the same machine state at two different abstraction levels. As seen previously, the verilog_sem function runs Verilog programs in our Verilog semantics, and silver_cpu_verilog refers to the Verilog program the translator built out of the HOL processor implementation silver_cpu.

The derivation of theorem (2.10) is mostly automated by the code generator; the main obligation to discharge as a user of the code generator is to express the input circuit as a hardware description at the same level as the example from §2.3. To derive the ISA and Verilog implementation-correspondence theorem (2.7) from §2.2, we simply compose theorem (2.10) with the implementation-correctness theorem (2.9).

For synthesis for our FPGA board, we have used the Verilog code generated by the process described in this section in combination with some Verilog glue to connect the processor to its environment (see the discussion in §2.8).

2.5 CakeML's assumptions

The CakeML compiler's correctness theorem makes a long list of assumptions regarding the execution environment of the generated code. This section presents what the assumptions are, while the next section explains how we have met these assumptions with the verified Silver processor and verified implementation of system calls.

The compiler's correctness theorem, an instance of which is theorem (2.2) in §2.2.3, includes the following assumption:

$$\mathsf{installed}_{\mathsf{Ag}}\ \mathit{compiled\_prog}\ (\mathsf{basis\_ffi}\ \mathit{cl}\ \mathit{fs})\ \mathit{ms},$$

which encapsulates the assumptions about the execution environment of the generated code. It relates the generated code $\mathit{compiled\_prog}$, the command-line arguments cl, the state of the file system fs, and the Silver ISA machine state ms. Informally, it requires that ms is set up correctly for execution of $\mathit{compiled\_prog}$ to begin.

The formal definition of $\mathsf{installed}_{\mathsf{Ag}}$ is too long to reproduce in full here, but

the following list outlines its contents:

(i) Registers 1–4 provide accurate information on where the part of memory usable by $compiled\_prog$ is located in machine state ms.

(ii) The read-only data of $compiled\_prog$ is stored in memory, where it is expected to be (based on registers 1–4).

(iii) The machine code of $compiled\_prog$ is stored in memory, and the program counter of ms points at the first address of this machine code.

(iv) The code and data sections do not overlap, and various pointers are aligned to word boundaries.

(v) Calls to external functions (i.e., system calls) behave according to the modelled behaviour of the filesystem fs and command line cl.

The last point (v) above is by far the most complicated assumption. It requires that each time the CakeML-generated code jumps to external code (e.g., code for reading external input), the external code will execute and safely return to the CakeML code according to CakeML's calling convention for external calls. Furthermore, each execution of external code must adhere to the CakeML basis library's assumption basis_ffi cl fs, explained in detail below.

The formal definition (omitted) of (v) is slightly unintuitive because the property is defined in terms of restricting the freedom of an oracle function. This oracle function, which we call the *interference oracle* of the foreign-function interface (FFI), is an argument to the operational semantics machine_sem that is used in the correctness theorem for the CakeML compiler. For the most part, machine_sem executes the next instruction using the Silver ISA's next-state function Next. However, when machine_sem encounters an entry point to external code (an FFI call), the semantics consults the interference oracle to determine what the resulting Silver ISA state should be.

The interference oracle is restricted to leave unchanged the part of the machine state that is private to CakeML code; the oracle is obliged to write the correct return value (according to basis_ffi cl fs, see below) to the shared array that is used for communicating between CakeML code and the external code; and the interference oracle is forced to set the program counter to the correct return address (in order to continue execution of the CakeML code).

So, what is basis_ffi cl fs? It encapsulates the assumptions that the CakeML standard basis library makes of its foreign-function interface. In the context of bare-metal systems, this is the interface to system calls that support the I/O functions in the basis library. Concretely, basis_ffi is defined as a record that defines (1) an oracle function basis_ffi_oracle, which specifies the behaviour of

each call, and (2) the current state of the external world consisting of a command line (cl) and a filesystem (fs). The basis_ffi_oracle recognises calls to: "read", "write", "get_arg_count", "get_arg_length", "get_arg", "open_in", "open_out", "close", and "exit". An excerpt of its definition is shown below:

```
basis_ffi_oracle name (cl,fs) conf bytes def=
 if name = "read" then
  case ffi_read conf bytes fs of
    FFIfail ⇒ Oracle_final FFI_failed
  | FFIreturn bytes fs ⇒ Oracle_return (cl,fs) bytes
  | FFIdiverge ⇒ Oracle_final FFI_failed
 else ...
```

When the FFI with name "read" is called, basis_ffi_oracle delegates the task to ffi_read, which receives configuration $conf$, input $bytes$ and the current state of the filesystem fs as arguments. The $conf$ and $bytes$ values are arguments that the CakeML programmer passed to the call at the source level. From the programmer's perspective, $bytes$ is a byte array that they have made the FFI call with, in this case, expecting it to be filled with characters from reading a file. The function ffi_read is defined as:

```
ffi_read conf (b0::b1::b2::b3::bytes) fs def=
  do
    assert (|bytes| ≥ w22n [b0; b1] ∧ |conf| = 8);
    (l,fs') ← read (w82n conf) fs (w22n [b0; b1]);
    FFIreturn
      ([0w] ++ n2w2 |l| ++ [b3] ++ map c2w l ++
       drop |l| bytes) fs'
  od otherwise (FFIreturn (1w::b1::b2::b3::bytes) fs)
```

When ffi_read receives a $bytes$ argument of sufficient length, it calls a read function from the filesystem model. This read function (definition omitted) is given a file handler, a filesystem state and the maximum length it is allowed to read. The read function returns (l,fs'), where l is the number of bytes that were actually read and fs' is the new filesystem state. The complicated list expression passed to FFIreturn specifies how the length of list l and its content is communicated in the shared array on return. On failure, ffi_read returns $1w$ in the first element of the array. All of the functions involved here are defined in a monadic style (do . . . od, etc.) since there can be assertion failures at many different points. The w22n, w82n, n2w2, and n2w functions are conversions between bytes and natural numbers.

2.6 Setting up Silver for CakeML

In this section, we explain how we transition from theorem (2.4) to theorem (2.6) in §2.2, i.e., how we prove the $\mathsf{installed}_{\mathsf{Ag}}$ assumption in order to move our correctness theorem from a property about CakeML's machine_sem down to Silver's Next. We make this transition by fixing a memory layout that includes code implementing the required system calls and verifying that code. The memory layout that we use is shown in Figure 2.2.

Recall from §2.5 that machine_sem either takes an ordinary step of executing a CakeML-generated Silver instruction or takes an *interference oracle* step representing a call to a foreign function. To move from the machine_sem level to the Silver ISA level, we define a predicate, interference_impltd, which states that the effect of the interference-oracle step can be obtained by normal execution of machine code located somewhere in memory (separate from CakeML-generated code). This predicate bridges the gap between machine_sem and our verification of the system-call code at the ISA level.

The first theorem we prove about the predicate interference_impltd connects it to machine_sem. It states that interference_impltd R_{ffi}, for an arbitrary relation R_{ffi} between the Silver machine state ms and FFI oracle state ffi, implies that a terminating machine_sem can be replaced by a sequence of Next steps that preserve R_{ffi}. The FFI oracle states ffi, ffi' in this theorem are records of the same type as basis_ffi cl fs. Recall that basis_ffi cl fs is the initial FFI oracle state from which a CakeML program is started (featuring, e.g., in theorem (2.6)); ffi and ffi' are a pair of FFI oracle states reached by the CakeML program at runtime. The md argument gives the memory domain (addresses) in which the system-call code is expected to reside:

$$
\begin{array}{l}
\vdash \mathsf{interference\_impltd}\ R_{ffi}\ md\ ms \wedge R_{ffi}\ ms\ ffi \wedge \\
\quad \mathsf{machine\_sem}\ ffi\ ms \subseteq \\
\quad\quad \mathsf{extend\_with\_oom}\ \{\ \mathsf{Terminate\ Success}\ io\ \} \Rightarrow \\
\quad\quad \exists k\ ffi'. \\
\quad\quad\quad R_{ffi}\ (\mathsf{Next}^k\ ms)\ ffi' \wedge \mathsf{is\_halted}_{\mathsf{Ag}}\ (\mathsf{Next}^k\ ms) \wedge ffi'.\mathsf{io\_events} \preccurlyeq io \wedge \\
\quad\quad\quad (\mathsf{exit\_code\_0}_{\mathsf{Ag}}\ (\mathsf{Next}^k\ ms) \Rightarrow ffi'.\mathsf{io\_events} = io)
\end{array}
\tag{2.11}
$$

The second theorem provides a concrete relation, $\mathsf{ffi\_rel}_{\mathsf{Ag}}$, and memory domain, $\mathsf{ffi\_mem\_domain}_{\mathsf{Ag}}$, and proves that they satisfy interference_impltd:

$$
\vdash \ldots \Rightarrow \mathsf{interference\_impltd}\ \mathsf{ffi\_rel}_{\mathsf{Ag}}\ \mathsf{ffi\_mem\_domain}_{\mathsf{Ag}}\ ms \tag{2.12}
$$

The omitted assumptions (...) are routine: e.g., that the FFI oracle state's command line and file system are well-formed, the code is correctly placed in memory (e.g., within the domain), and so on. We omit these routine assumptions

here and below for brevity.

The proof of theorem (2.12) involves showing that each piece of system-call code correctly implements the call as specified by basis_ffi_oracle, which we saw in §2.5. We prove a theorem of the following form (shown here for "read") for each system call:

$$\begin{array}{l}\vdash \ldots \wedge\ md = \mathsf{prog\_mem\_domain}_{\mathsf{Ag}} \quad \ldots \wedge\ \mathsf{ffi\_rel}_{\mathsf{Ag}}\ ms\ \mathit{ffi} \wedge \\ \quad \mathsf{index\_of}\ \text{“read”}\ \mathit{ffi\_names} = \mathsf{Some}\ index \wedge \\ \quad \mathsf{call\_FFI}\ \mathit{ffi}\ \text{“read”}\ conf\ bytes = \mathsf{FFI\_return}\ \mathit{ffi}'\ bytes' \Rightarrow \\ \quad\ \exists\, k. \\ \quad\quad \mathsf{ffi\_interfer}_{\mathsf{Ag}}\ md\ (index,bytes',ms) = \mathsf{Next}^k\ ms \wedge \\ \quad\quad \mathsf{ffi\_rel}_{\mathsf{Ag}}\ (\mathsf{Next}^k\ ms)\ \mathit{ffi}'\end{array} \tag{2.13}$$

Here, call_FFI is a wrapper around basis_ffi_oracle that takes an initial FFI oracle state ffi and returns a new FFI oracle state ffi' along with the $bytes$ returned by basis_ffi_oracle. The conclusion of this theorem has two parts. First, it shows that the FFI call ($\mathsf{ffi\_interfer}_{\mathsf{Ag}}$, described below) is identical to stepping Next k times. Second, it shows that the $\mathsf{ffi\_rel}_{\mathsf{Ag}}$ relation is preserved across these k steps.

So what is the $\mathsf{ffi\_interfer}_{\mathsf{Ag}}$ function? It is a concrete interference-oracle instance for CakeML's FFI semantics that specifies the effect on a machine state ms of running a system call that returns $bytes'$. The md argument indicates the memory domain ($\mathsf{prog\_mem\_domain}_{\mathsf{Ag}}$) that CakeML uses, i.e., the parts with a CakeML prefix in Figure 2.2. The $index$ argument indicates which FFI call is made (in this case "read"). The $\mathsf{ffi\_interfer}_{\mathsf{Ag}}$ function updates the machine state by writing $bytes'$ to the part of md used for communicating with the external call, updating registers and the PC according to the calling convention and, based on $index$, updating memory (outside of md) used for bookkeeping by the external FFI call. Thus to verify each piece of system call code, we must show that executing the code has the effect of $\mathsf{ffi\_interfer}_{\mathsf{Ag}}$.

Each system call is verified in two refinement steps. The first step abstracts from the machine code implementing a system call to a logical specification of its effect. For example, the logical specification for the code implementing "read" is a theorem of the form:

$$\vdash \ldots \Rightarrow \exists\, k.\ \mathsf{Next}^k\ ms = \mathsf{ffi\_read}_{\mathsf{Ag}}\ ms$$

The omitted assumptions (…) ensure, e.g., that the relevant code and data are placed in memory correctly and that the program counter is currently pointing at the start of the code. The theorem's conclusion shows that stepping by k steps yields the machine state given by $\mathsf{ffi\_read}_{\mathsf{Ag}}\ ms$. This logical specification ($\mathsf{ffi\_read}_{\mathsf{Ag}}$) is the glue to the second refinement step.

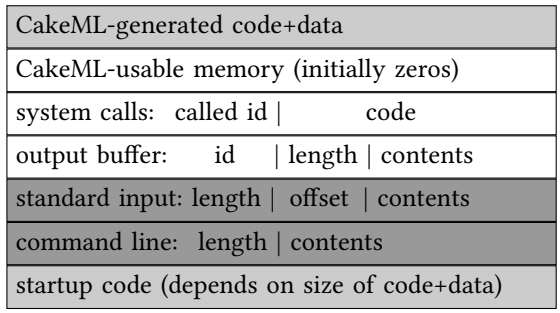

Figure 2.2. The memory layout for running CakeML programs bare-metal on Silver. When preparing the initial memory, parts with a white background are application-independent, parts with an intermediate background are application-dependent, and parts with the darkest background are input for each execution.

The second step connects $\mathsf{ffi\_read}_{\mathsf{Ag}}$ to $\mathsf{ffi\_interfer}_{\mathsf{Ag}}$ assuming the ffi_read specification (§2.5) from CakeML's basis library:

$$\vdash \ldots \wedge \mathsf{ffi\_read}\ conf\ bytes\ fs = \mathsf{FFIreturn}\ bytes'\ fs' \Rightarrow \\ \mathsf{ffi\_read}_{\mathsf{Ag}}\ ms = \mathsf{ffi\_interfer}_{\mathsf{Ag}}\ md\ (index,bytes',ms)$$

As before, the omitted assumptions (...) are routine ones about how the initial machine state ms is set up. At the point of writing, both refinement steps were verified manually with the help of some specially written automation. We are confident, however, that the first step can be fully automated with decompilation tools [77].

As discussed in §2.5, the most complicated part of the existing CakeML assumption are the ones asserting that the system calls are correctly implemented according to CakeML's basis-library assumptions. This assumption is concretely discharged in a few steps but mainly using the composition of theorems (2.11) and (2.12) discussed in this section. Discharging this assumption is what allows us to go from theorem (2.4) to theorem (2.6) in §2.2. It is also in this step where the routine (omitted) assumptions from earlier are discharged. This discharging step is done automatically for concrete compiled programs such as wc_ag32. Crucially, the only remaining assumptions are the ones shown in theorem (2.6).

The remaining assumptions inside $\mathsf{installed}_{\mathsf{Ag}}$ are straightforward compared to the FFI ones. They concern putting the machine in an appropriate initial state for CakeML code to run. Their verification did, however, lead to some minor surprises as we detail next.

2.6.1 Changes to the assumptions

In proving $\mathsf{installed}_{\mathsf{Ag}}$ to move from theorem (2.4) to theorem (2.6), we found that some parts of $\mathsf{installed}_{\mathsf{Ag}}$ were inconsistent, specifically point (iv) in §2.5 about pointers being aligned (which is independent of the `ag32` target). Although the inconsistency was easy to fix, it was not caught previously and had appeared in the final top-level theorem for the CakeML compiler. This highlights the value of reducing the assumptions, ideally by proving them away, in any large formal development.

More substantially, we made some changes to CakeML's target-machine semantics, machine_sem. On the one hand, the design of machine_sem that allows arbitrary interference to non-CakeML parts of the machine state whenever an external call is made is vindicated by our instantiation of the interference oracle with a concrete implementation of system calls. On the other hand, the invariants about both CakeML steps and interference steps needed to be strengthened: we needed to know, in both cases, that memory not used by the currently running code does not change, and that the PC stays within the correct part of memory. The invariants that are now present in our definitions are sufficient for proving correctness of the sequence of calls from CakeML to external code and back on the same machine.

Finally, the extend_with_oom feature of the CakeML compiler's correctness theorem was previously (ab)used to allow compiler-generated startup checks to fail at runtime. They failed unexpectedly when we first tried running programs on Silver, because we had the memory layout and startup code slightly wrong. These dynamic checks have now all been replaced by checks that resort to a valid default configuration instead of causing runtime failures; in other words, CakeML's new startup code will never cause an out-of-memory error.

2.7 Results

The process described so far does not just work for the word-counting (`wc`) application. We can establish the same sorts of connection between other pieces of verified software and the verified Silver platform, creating verified software-to-hardware stacks for a variety of tools. These applications have all been verified previously: the hard intellectual work has already been performed (at the level of HOL and/or CakeML functions). Here, we confirm that the same applications can be compiled for and executed on the Silver platform.

First, we have successfully run all of the programs mentioned in the introduction (§2.1) on our FPGA board. Silver is not a high-performance processor, but small programs such as `sort` complete almost instantaneously when run on small inputs. Running `sort` on a 1000-line file takes a few seconds. Silver's

low performance is more noticeable for larger programs, such as the compiler itself. For example, compiling a one-line hello world program on a modern Intel processor takes around two to three seconds, whereas compiling the same program on Silver takes around four hours.

Second, the verification story (establishing HOL theorems of correctness) for these applications follows the pattern already described in the paper to this point. For example, the correctness statement for the CakeML compiler on Silver (2.14) has much the same assumptions as for the wc example (2.8); that is, the machine has been correctly initialised ($\mathsf{verilog\_init}_{\mathsf{Ag}}$), and the input is not too large, among others:

$$
\begin{array}{l}
\vdash \mathsf{let}\ \mathit{vstep}\ =\ \mathsf{verilog\_sem}\ \mathit{env}\ \mathsf{silver\_cpu\_verilog}\ \mathit{init}\ \mathsf{in} \\
\quad \mathsf{cl\_ok}\ \mathit{cl} \wedge |\mathit{input}| \leq \mathsf{stdin\_size} \wedge \\
\quad \mathsf{is\_lab\_env}\ \mathsf{acc\_env\_verilog}\ \mathit{vstep}\ \mathit{env} \wedge \\
\quad \mathsf{verilog\_init}_{\mathsf{Ag}}\ \mathsf{compiler\_ag32}\ (\mathit{cl},\mathit{input})\ \mathit{init}\ \mathit{env} \Rightarrow \\
\quad \exists\, \mathit{stdout}\ \mathit{stderr}.\ \mathsf{FG}\ k.\ \exists\, \mathit{fin}. \\
\qquad \mathsf{compiler\_spec}\ \mathit{input}\ \mathit{cl}\ \mathit{stdout}\ \mathit{stderr} \wedge \\
\qquad \mathit{vstep}\ k = \mathsf{Ok}\ \mathit{fin} \wedge \mathsf{verilog\_is\_halted}_{\mathsf{Ag}}\ \mathit{fin} \wedge \\
\qquad \mathsf{stdout}_{\mathsf{Ag}}\ (\mathit{env}\ k).\mathsf{io\_events} \preccurlyeq \mathit{stdout} \wedge \\
\qquad \mathsf{stderr}_{\mathsf{Ag}}\ (\mathit{env}\ k).\mathsf{io\_events} \preccurlyeq \mathit{stderr} \wedge \\
\qquad (\mathsf{verilog\_exit\_code\_0}_{\mathsf{Ag}}\ \mathit{fin} \Rightarrow \\
\qquad\quad \mathsf{stdout}_{\mathsf{Ag}}\ (\mathit{env}\ k).\mathsf{io\_events} = \mathit{stdout} \wedge \\
\qquad\quad \mathsf{stderr}_{\mathsf{Ag}}\ (\mathit{env}\ k).\mathsf{io\_events} = \mathit{stderr})
\end{array}
\tag{2.14}
$$

In addition, because the compiler takes the name of its input file as its commandline argument, we have a predicate cl_ok asserting that the commandline is well-formed (essentially, that it is not too large). As before, the conclusion states that execution will eventually result in a final state ($\mathit{vstep}\ k = \mathsf{Ok}\ \mathit{fin}$) satisfying the user-level specification of the compiler behaviour. That specification (compiler_spec) describes how standard output contains a textual representation of the machine code for the input program.

The definition of compiler_spec makes this clear:

$$
\begin{array}{l}
\mathsf{compiler\_spec}\ \mathit{input}\ \mathit{cl}\ \mathit{stdout}\ \mathit{stderr} \stackrel{\mathsf{def}}{=} \\
\quad (\mathit{stdout},\mathit{stderr}) = \\
\qquad \mathsf{if}\ \mathsf{has\_version\_flag}\ (\mathsf{tail}\ \mathit{cl})\ \mathsf{then} \\
\qquad\quad (\mathsf{explode}\ \mathsf{current\_build\_info\_str},\text{“”}) \\
\qquad \mathsf{else} \\
\qquad\quad \mathsf{let}\ (\mathit{cout},\mathit{cerr})\ =\ \mathsf{compile\_32}\ (\mathsf{tail}\ \mathit{cl})\ \mathit{input}\ \mathsf{in} \\
\qquad\qquad (\mathsf{explode}\ (\mathsf{concat}\ (\mathsf{append}\ \mathit{cout})),\mathsf{explode}\ \mathit{cerr})
\end{array}
$$

The compile_32 function mentioned here is a (somewhat complicated) wrapper around a call to the compile function of our initial correctness result for the

compiler (2.2). In this way, our theorem asserts the correctness of the bootstrap of CakeML on Silver.

2.8 Discussion

The promise of a verified stack is the ability to construct systems that have formal evidence for their correct implementation. Such evidence, in the form of mechanically checked proofs, is always subject to implicit and explicit assumptions, which collectively represent the *trusted computing base* (TCB) of the verified stack. The TCB is all the things that need to be trusted if we are to believe the stack operates correctly. In stack constructions, alongside the proof checker itself, only the top and bottom layers contribute to the TCB since there are proofs in between.

In this section, we describe the TCB of the bottom layer of stacks constructed using our methodology and discuss the (necessarily informal) ways in which we can justify the trust we put into these assumptions. We also show where we have reduced the TCB, compared to previous CakeML work, by replacing assumptions with proofs.

Verilog semantics. We assume that our formal model of Verilog is accurate with respect to the Vivado toolchain that takes Verilog input and produces our hardware. Relatedly, we assume that:

- the printing of Verilog abstract syntax trees from HOL is faithful; and
- the Vivado toolchain taking Verilog to FPGA bitstreams is bug-free.

We address these assumptions by using a simple subset of Verilog, one where the semantics is uncontroversial and where we can be relatively confident that the implementation will be straightforward. Code implementing the pretty-printing of ASTs is not complicated, so informal code inspection is helpful with respect to this assumption. Though we have not done this, we could gain assurance by implementing this code in CakeML, developing a parser for the printed syntax, and proving that the composition of parser and printer is the identity. The second item could be further addressed by standard industrial tools such as formal equivalence checkers, but such tools would not produce proofs composable with our formal development.

Hardware. In our lab setup (§2.4.2), we have aimed for convenience rather than a minimal TCB. Consequently, some of the assumptions required by the current lab setup could be significantly reduced with little effort. Concretely,

we are dependent on both the correct operation and the correct initialisation of the various hardware components that realise our final system. For example, we assume that:

- the FPGA chip works correctly;
- the shared DRAM module (and other board modules) works correctly;
- the Python script, running on the ARM core, that we use to preload memory and handle interrupts (such as text output requests) sent to the core is operating correctly; and
- the Verilog glue code used to connect the Silver processor to its environment works correctly. E.g., some interfaces, such as to the DRAM module, are exposed as AXI3 interfaces [65]—but as we are not interested in the details of AXI3 in particular, we expose simplified interfaces to the processor.

The dependence on the ARM core (and the Linux operating system it is running) for preloading memory and interrupt handling is clearly tangential and could be improved by preloading memory by more primitive means and using, e.g., seven-segment displays for text output.

Comparison to previous work. The bottom-layer TCB described above is different to, and a clear improvement on, the bottom-layer TCB accompanying verified software developed with CakeML previously. The assumptions about Verilog and the hardware have replaced enormous, unverified components. In particular, previous work [25, 59] had to assume:

- the correctness of the underlying operating system and its tools to link and run our executables (loading it into memory, connecting it to I/O streams, etc.);
- the correctness of our hardware semantics for targets such as ARM and x86; and
- the correctness of those hardware semantics' realisation in the silicon on which the software was being executed.

2.9 Related work

The CLI stack. An early attempt at constructing a verified stack was made in the late 1980s and early 1990s in the CLI stack project [9, 96], which was built using the Nqthm theorem prover, a precursor to ACL2. The stack included,

among other components, a verified processor and two verified compilers, for Pascal-like and Lisp-based languages, targeting the verified processor. A version of the stack was built for the verified FM9001 processor. FM9001 was described in a custom HDL called DUAL-EVAL, which was translated to LSI Logic's Netlist Description Language for fabrication by LSI Logic on a gate array [16].

Moore [96], one of the stack's principal architects, describes the compilers' languages as "too simple to be of practical use", lacking e.g., I/O mechanisms. Furthermore, it was not possible to run the verified compilers on top of their stack.

The Verisoft stack. A later attempt at a verified stack was made in the 2000s in the Isabelle/HOL-based Verisoft stack project [3]. The processor used in the Verisoft stack is called VAMP, first developed in PVS [11] and later ported to Isabelle/HOL [104]. Beyer et al. [11] call the CLI stack's FM9001 processor "very simple" and note that the VAMP is much more complex as it is both pipelined and capable of out-of-order instruction execution. In comparison with the VAMP processor, our Silver processor must also be described as "very simple". On the other hand, the VAMP processor was also synthesised for FPGAs, but was not verified down to the Verilog code used for synthesis. Instead, for the PVS version, a tool operating outside the formal development called `pvs2hdl` [10] was used to produce the Verilog code out of a gate-level PVS description. Similarly, i.e., also without proof, the Isabelle/HOL VAMP version used an unverified tool called IHaVeIt [104] to translate Isabelle/HOL hardware descriptions to Verilog.

The Verisoft stack also included a verified compiler for C0 [62], a language similar to a subset of C plus garbage collection. The C0 compiler provides similar FFI functionality as the CakeML compiler, called XCalls, allowing programmers to embed VAMP assembly code inside C0 programs. Leinenbach and Petrova [62] describe the compiler as "simple". In contrast with CakeML, it does not include any optimisation passes. The C0 compiler consists of a verified compilation algorithm accompanied by a partly verified C0 implementation. Unlike the CakeML compiler, the implementation is not automatically derived from the compilation algorithm. Instead, a Hoare-logic-based C0 verification environment was used to prove the implementation correct. This means that manual work is needed to keep the compilation algorithm and implementation in sync when new features are added to the compiler. Moreover, only the code-generation implementation (approximately 1500 lines of C0) was proved correct; parsing and I/O were left unverified. A VAMP-machine-code implementation is needed to run the compiler on top of the VAMP processor, but they do not provide a way to compile, with proofs, the C0 implementation to a

machine-code implementation (such as running the C0 compiler in-logic, as the CakeML compiler does when compiling itself to machine code). In other words, not all pieces for running the compiler on top of the VAMP stack are present.

Other verified stack work. In the ongoing Coq-based DeepSpec stack project [5], the Kami project [19] enables Bluespec development and verification inside Coq. A pipelined in-order multicore processor has been developed inside the Kami project as a case study but is not yet part of a larger stack. The Coq world's analogue to the CakeML compiler, the CompCert compiler [63], does not have an implementation verified down to machine code, so obtaining a correctness guarantee about running CompCert on top of a Coq-verified processor is nontrivial (as doing this requires having access to a verified machine-code implementation).

There have also been processors developed and (sometimes partly) verified without being part of full-stack projects. Though such components might fit into a verified stack, without actually carrying out the necessary integrations, this remains a "might" rather than a demonstrated "can". Beyer et al. [11] enumerate a few verified processors published before their PVS VAMP paper and note that of the processor papers they cite, only papers about the FM9001 processor (i.e., the processor from the CLI stack) state that the processor has been synthesized. By their account, the remaining processor papers rely on "several simplifications and abstractions". Given the controversies around the Viper processor [20, 70], it is clear that when claiming a processor "verified", one must be precise about what has actually been proved and down to what abstraction level the proofs reach.

Correct hardware. Neither the stack work cited in this section nor other ITP hardware-verification work [15, 19, 45, 52, 87], have combined verification with formal semantics for a mainstream low-level HDL such as Verilog or VHDL (instead relying on, e.g., unverified extraction). Previous formal-semantics work exists for Verilog [55, 73], but those projects do not seriously consider ITP verification.

Verified low-level systems code. The verification of the software programs the stacks we have build are based on is not new (e.g., Goel et al. [28] have verified a x86-machine-code word-counting program). Rather, is it the inclusion of substantial programs in stack constructions that is the focus and novelty in this paper.

Unlike high-level application code which can be compiled by the CakeML compiler down to Silver machine code, we implemented and verified the system calls for our stack by hand. This was manageable for the CakeML basis library, but verifying more complicated system calls would require (or at least, be

significantly aided by) low-level programming and verification frameworks [17, 18, 81] and automated decompilation tools [77].

2.10 Conclusion

This paper has reported on a novel workflow for producing verified stacks that connect verified hardware to verified programs that run on top of it. Our approach connects the CakeML compiler to a new verified Silver processor. We have a unique and novel contribution, which enables the proof of *single end-to-end correctness theorems* for realistic user-level programs, such as the CakeML compiler itself. In other words, not only does the CakeML compiler have a new target, the verified Silver hardware design, but it can itself be run on that hardware. This work is a relief: we now know that the assumptions made at the bottom of the CakeML compiler proof can be met by underlying verified hardware.

At certain points, we have taken the shortest route to our final end-to-end results, which means that there is room for improvement in individual parts of the project. Improvements of one part can be carried out independently of other parts as long as the interfaces between all parts stay the same.

We intend to improve our hardware implementation of Silver. The processor ought to be pipelined and otherwise optimised to support higher clock frequencies in order to produce faster applications. We will do this without fundamentally changing the Silver ISA because we want to keep the ISA at an abstraction level that does not expose implementation techniques in the hardware implementation.

We also want to make it less labour-intensive to develop and set up verified systems code that interfaces with the CakeML-generated code. The set-up work required for this paper was significantly more labour-intensive than expected.

Acknowledgements. This work was partly supported by the Swedish Foundation for Strategic Research.

Chapter 3

A Proof-Producing Translator for Verilog Development in HOL

Andreas Lööw and Magnus O. Myreen

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

Published at *Conference on Formal Methods in Software Engineering (FormaliSE)*, 2019

3.1 Introduction

When building fully verified systems, so called *verified stacks* [3, 5, 96], software and hardware cannot be treated as separate, disconnected worlds. For a system to be fully correct, it is not enough that its software and the hardware constituents are correct considered independently of each other: the software and hardware components must also be integrated correctly. To be able to state and prove all-encompassing system-correctness claims, the software, hardware, and all integrations between them must be made available inside the same formal system.

In a recent paper [67], we present a methodology to construct verified stacks inside the HOL4 interactive theorem prover [95]. For the software side of the stacks, we rely on (and extend) the CakeML project [25]. For the hardware side, we introduce a new hardware-development methodology. In this paper, we provide the technical details of this hardware-development methodology.

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

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

The alternative approach of interactive theorem proving (ITP) has the potential, we claim, to overcome the limitations listed above. This would be because, inside ITP systems, such as HOL4, automatic and manual proofs can be combined in logically safe ways. The ITP approach has been taken many times before (see Sec. 3.8), and there has been previous work on formalizing the semantics of mainstream low-level HDLs, and previous work on formally verifying the correctness of concrete, or parameterized, circuits, inside ITP systems. However, to the best of our knowledge, *there has not been prior work combining these two efforts*, i.e. verification of nontrivial circuits with respect to a detailed explicitly stated formal semantics for one such description language.

Our hardware-development methodology is designed to make hardware verification easy in HOL and, at the same time, support a solid connection

to a formal semantics of a subset of Verilog. The heart of the approach is *an automatic proof-producing tool* that, given a functional version of a Verilog program in HOL, produces Verilog code and proves a *correspondence theorem* stating that the Verilog code and the functional code have the same behavior according to a formal semantics of Verilog we have developed. Furthermore, the correspondence theorem enables transporting system-correctness results for the functional code to the Verilog code, as illustrated in Sec. 3.2.

This paper makes the following contributions:

- We elaborate on our previously published *hardware-development methodology* that is centered around functional versions of Verilog programs in HOL. (Sec. 3.2, 3.3, 3.9)
- We present the details of our *automatic tool* that makes the methodology have a solid connection to our explicitly defined operational semantics for Verilog. (Sec. 3.6)
- The description of our *formal semantics for a subset of Verilog* is also a contribution. The semantics is carefully carved out to be faithful to the Verilog standard, manageable in complexity for HOL proofs, and sufficiently large to express interesting synthesizable hardware. (Sec. 3.5)

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

3.2 Example

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

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

$$\begin{array}{l}
\mathsf{A}\ \mathit{fext}\ s \stackrel{\mathsf{def}}{=} \mathsf{if}\ \mathit{fext}.\mathsf{pulse}\ \mathsf{then}\ s\ \mathit{with}\ \mathsf{count}\ :=\ s.\mathsf{count} + 1w\ \mathsf{else}\ s \\
\mathsf{B}\ s \stackrel{\mathsf{def}}{=} \mathsf{if}\ 10w <_+ s.\mathsf{count}\ \mathsf{then}\ s\ \mathit{with}\ \mathsf{done}\ :=\ \mathsf{T}\ \mathsf{else}\ s \\
\mathsf{AB}\ \mathit{fext}\ \mathit{init}\ 0 \stackrel{\mathsf{def}}{=} \mathit{init} \\
\mathsf{AB}\ \mathit{fext}\ \mathit{init}\ (\mathsf{Suc}\ n) \stackrel{\mathsf{def}}{=} \mathsf{let}\ s' = \mathsf{AB}\ \mathit{fext}\ \mathit{init}\ n\ \mathsf{in} \\
\quad \langle \mathsf{count}\ :=\ (\mathsf{A}\ (\mathit{fext}\ n)\ s').\mathsf{count};\ \mathsf{done}\ :=\ (\mathsf{B}\ s').\mathsf{done} \rangle
\end{array}$$

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

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

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

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

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

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

$$\mathsf{pulse\_spec}\ fext \stackrel{\text{def}}{=} \forall\, n.\ \exists\, m.\ (fext\ (n + m)).\mathsf{pulse},$$

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

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

Properties proved of HOL functions such as AB can easily be transported to properties of its generated Verilog code thanks to the automatically proved correspondence theorems. For our running example, we can prove the following,

(to repeat:) *without manual reasoning about the Verilog semantics*:

$$
\begin{array}{l}
\vdash \mathsf{pulse\_spec\_verilog}\ fext \wedge \mathsf{vars\_has\_type}\ \Gamma\ \mathsf{ABtypes} \Rightarrow \\
\quad \exists n\ \Gamma'. \\
\qquad \mathsf{mrun}\ fext\ \mathsf{ABv}\ \Gamma\ n = \mathsf{Inr}\ \Gamma' \wedge \\
\qquad \mathsf{mget\_var}\ \Gamma'\ \text{“done”} = \mathsf{Inr}\ (\mathsf{VBool\ T}).
\end{array}
$$

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

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

3.2.1 Larger examples

The example above was intentionally kept simple for ease of presentation. However, the methodology has been shown to work on larger examples. Sec. 3.7 describes how we have applied it in our paper on verified stacks [67] to produce a verified implementation of a processor for execution of CakeML programs. In the same section we also describe how to build circuits that perform matching against regular expressions.

3.3 Hardware-development methodology summary

Fig. 3.1 summarizes the flow the translator enables, from high-level specifications down to runnable FPGA application, as exemplified in the previous section.

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

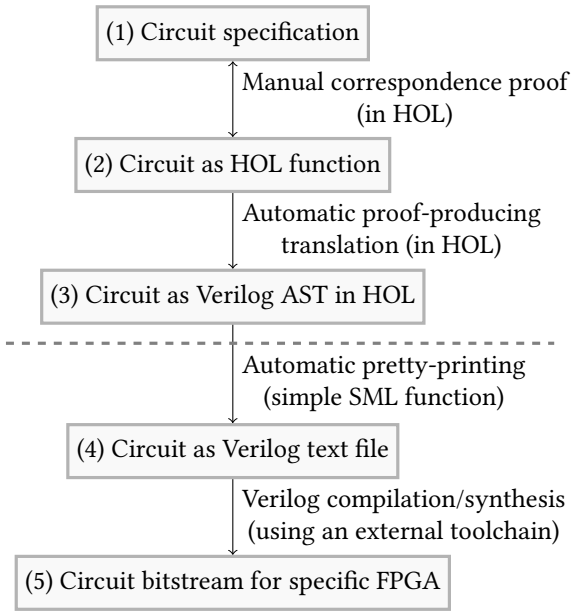

Figure 3.1. An overview of our hardware-development methodology. For non-trivial circuits, human-guided proofs are needed for the connection between layers 1 and 2, whereas the other steps are always automatic. The correspondences above the dotted line are proved functionally correct, and the correspondences below the dotted line are not covered by our formal development. The main focus of this paper is the translator connecting layers 2 and 3.

3.4 Overview

The following sections provide more technical detail: We will first describe the subset of Verilog we target and its semantics (Sec. 3.5), followed by the internals of the translator (Sec. 3.6). We then explain how we have applied our methodology in two case studies (Sec. 3.7). Lastly, we discuss related work (Sec. 3.8) and consider trusted-base issues (Sec. 3.9).

3.5 Verilog

In this section we describe the syntax and formal semantics of the subset of Verilog targeted by the translator. Verilog is a large language; the current standard [48] is more than 1300 pages long. But many parts of the standard are irrelevant for describing hardware: these, nonsynthesizable, parts of the standard include, e.g., concepts used to express test benches and other testing

infrastructure. We have not included all synthesizable constructs in our formalization; rather, the subset we target consists of synthesizable constructs needed in synthesizable behavioral process-based Verilog programs. Our aim is that this subset should be large enough to describe simple synchronous hardware. The subset has already been sufficient for the case studies we present in Sec. 3.7.

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

3.5.1 Abstraction level (Verilog as an output language)

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

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

It is all too easy to fall into the trap of thinking that "real" hardware consists of "the standard gates", such as ANDs, XORs, and flip flops, and consequently that this should be our target representation. This is incorrect, as what real hardware consists of is a technology-dependent question. In the case of FPGAs, real hardware consists of LUTs, DSP blocks, BRAMs, etc., rather than some homogeneous collection of gates. For example, if we compile away the notion of addition by compiling to ANDs, XORs, etc. before handing off our circuits to a synthesis pipeline, the pipeline might fail to exploit special-purpose hardware constructs available for efficient addition computations (as noted by e.g. Beyer et al. [10]).

3.5.2 Subset of Verilog included

We will now discuss the Verilog constructs included in our semantics and how they relate to the standard. Fig. 3.2 gives an overview of the constructs included.

Programs. In Verilog, programs consist of hierarchically connected modules. Each module has a set of input and output ports, which are used to connect the different modules together. Common top-level constructs inside a module

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

Figure 3.2. Verilog values v, expressions e, statements s, and programs p. Variable declarations are not included in the figure.

beyond other instantiated modules include data declarations, procedural blocks and continuous assignments.

In our formalization, to restrict the scope of the initial version of our project, we consider a flattened module hierarchy (i.e., a program consists of a single module). This can be understood as saying that we do not (formally) consider code used to "glue" modules together. (Meredith et al. [73] take the same approach in their Verilog semantics in the K framework; see the discussion in Sec. 3.8.) We did not find an immediate use for continuous assignments in our case studies as they did not include three-state buses and as all combinational logic could be placed inside procedural blocks, so we did not include such assignments in the formalization. As we are only interested in single-clock-domain synchronous hardware, all procedural blocks in our formalization are `always_ff` blocks waiting for a positive edge (`posedge`) from a program-common clock. Consequently, a Verilog program p (Fig. 3.2) in our formalization consists of a data-declarations section and a list of `always_ff` blocks. We use the terms process and procedural block synonymously throughout.

Statements and expressions. In our formalization, statements s and expressions e (both in Fig. 3.2), the syntactical elements inside procedural blocks, consist, beyond nonblocking assignments, mostly of standard imperative-language constructs. The state update of a nonblocking assignment is not visible until the next clock cycle, and such assignments are used for communication between processes.

We only allow pure expressions, because the standard does not enforce an

evaluation order, and we want our generated Verilog programs to be portable between Verilog toolchains. This means, e.g., that our expressions cannot contain assignments.

Variables and nets. In Verilog, there are two kinds of data objects, namely: variables and nets. Variables are included in our formalization as they are used for holding temporary values (in other words, wiring subcircuits) and describing registers capable of holding state between cycles. (Whether a variable corresponds to a register is up to the synthesizer in use to decide, i.e., nothing we have to concern ourselves with on our abstraction level [73].) Nets, however, are mainly useful for handling cases where there are multiple (continuous) drivers, which we are not interested in.

Values. Values v (Fig. 3.2) in our formalization consist of Booleans and nestable (balanced) arrays.

Boolean values. Verilog Booleans can take on four values: `0`, `1`, `X`, and `Z`. However, we can still use standard Booleans in our formalization. We do not include `Z` as a possible Boolean value because it is only used for nets (with multiple drivers), and nets are not covered in our formalization. The value `X` represents an unknown value. Such a value might be useful in simulation-based testing, but for proving we do not need an explicit representation of "unknown", because inside ITPs we already have access to such concepts directly in the logic. For example, if we want to prove that a circuit is correct regardless of the initial value of some particular Boolean variable (as we did in the example in Sec. 3.2), we simply quantify our theorem statement over all possible Boolean values for that variable.

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

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

In particular, some Verilog expressions are context-determined, both with respect to size and signedness. The size and signedness of a context-determined expression are not decided just by the subexpressions of the expression but also by the context the expression is part of. For example, for three Verilog arrays `a`, `b`, and `c`, where `a` and `b` are of length 16 and `c` of length 32, the

addition in the expression `c = a + b` will be a 32-bit addition as `c` is considered part of the context of the addition. To limit the amount of context taken into consideration by context-determined expressions, one can nest expressions inside the concatenation operator, whose operands are self-determined: e.g., `c = { a + b }` expresses a 16-bit addition. As for signedness, i.e. (here) if `a` and `b` are zero-extended or sign-extended, `c` is not considered part of the context even in the concatenation-less expression. That is, if `a` and `b` are signed, then they will be sign-extended, regardless of the signedness of `c`. But if one of `a` and `b` is unsigned and the other signed, then both will be zero-extended.

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

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

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

for unsigned less-than, and in pretty-printing sign casts are introduced for the signed variant.

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

3.5.3 Formal semantics

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

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

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

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

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

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

Here, the parameter s represents the current circuit state, and we recognize fext from the example from Sec. 3.2, allowing modeling of circuit-external behavior, such as e.g. memory modules or nondeterministic input sources. In the expression-level and statement-level Verilog semantics, fext is a function from variable names (strings) to Verilog values, as time is handled on the module level, as will be illustrated. The semantic rule for reading external inputs is

straightforward as well:

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

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

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

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

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

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

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

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

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

Note that mrun runs processes in the order they occur in its input list ps. In Verilog, processes are executed concurrently in an interleaved and nondeterministic manner. But we are only interested in processes that do not "interfere"

with each other, so program execution can be modeled faithfully without considering nondeterministic interleavings. If we let vwrites denote the variables written to blockingly by a process, vnwrites denote the variables written to nonblockingly by a process, vreads denote the variables read by a process, and disjoint denote that two sets are disjoint (i.e., disjoint $s\ t \stackrel{\text{def}}{=} s \cap t = \emptyset$), then we can formalize noninterference as follows:

$$
\begin{array}{l}
\mathsf{valid\_program}\ ps \stackrel{\text{def}}{=} \\
\forall\, i\ j. \\
\quad 0 \leq i \wedge i < \mathsf{length}\ ps \wedge 0 \leq j \wedge j < \mathsf{length}\ ps \wedge i \neq j \Rightarrow \\
\quad \mathsf{let}\ p\ =\ \mathsf{el}\ i\ ps;\ q\ =\ \mathsf{el}\ j\ ps\ \mathsf{in} \\
\quad\ \mathsf{disjoint}\ (\mathsf{vreads}\ p)\ (\mathsf{vwrites}\ q) \wedge \\
\quad\ \mathsf{disjoint}\ (\mathsf{vnwrites}\ p \cup \mathsf{vwrites}\ p)\ (\mathsf{vwrites}\ q \cup \mathsf{vnwrites}\ q).
\end{array}
$$

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

As for the expression-level and statement-level semantics, the only construct that interacts with the queue of nonblocking writes is in fact nonblocking assignments; meaning that reads are always based on Γ. The semantics of (successful) blocking (=) and nonblocking assignments (<=) for, e.g., Boolean variables are given by the following rules:

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

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

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

3.5.4 Validation

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

Another source of validation, which exercises also the statement-level semantics, is that our case studies (Sec. 3.7) worked as expected.

3.6 The translator

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

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

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

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

3.6.1 Input language

The input language of the translator should be thought of as a hardware description language in the same sense as Verilog is a hardware description

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

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

3.6.2 Implementation overview

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

As shown in the example in Sec. 3.2, the translator takes input in the form of a circuit represented as a next-state function consisting of smaller next-state functions. The translator translates the top-level circuit function into a Verilog program, with one process per inner next-state function. In cases where processes communicate, the translator introduces nonblocking assignments.

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

3.6.3 Pass one: process translation

The first pass is a proof-producing SML function, operating through the HOL4 API. To exemplify, the first pass turns the `A` function from the example from Sec. 3.2 into:

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

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

The first pass operates on one function at a time. Each input function is turned into a process (except the top function, which is AB in the example of Sec. 3.2). There is no need to support auxiliary helper functions since all helper functions can be inlined by rewriting rules in HOL before translation.

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

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

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

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

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

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

semantics' Boolean constructor:

$$\mathsf{BOOL}\ b\ v \stackrel{\text{def}}{=} v = \mathsf{VBool}\ b.$$

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

$$\begin{aligned}\vdash\ &\mathsf{Eval}\ \mathit{fext}\ s\ \Gamma\ (\mathsf{WORD}\ (1w \oplus 2w))\\ &(\mathsf{ABOp}\ (\mathsf{Const}\ (\mathsf{w2ver}\ 1w))\ \mathsf{BitwiseXor}\ (\mathsf{Const}\ (\mathsf{w2ver}\ 2w))),\end{aligned}$$

where $(\mathsf{ABOp}\ \ldots)$ is the resulting Verilog code as represented in the internal AST. To produce the above theorem, the translator utilizes the preproved theorem

$$\begin{aligned}\vdash\ &\mathsf{Eval}\ \mathit{fext}\ s\ \Gamma\ (\mathsf{WORD}\ w_1)\ v_1 \wedge\\ &\mathsf{Eval}\ \mathit{fext}\ s\ \Gamma\ (\mathsf{WORD}\ w_2)\ v_2 \Rightarrow\\ &\mathsf{Eval}\ \mathit{fext}\ s\ \Gamma\ (\mathsf{WORD}\ (w_1 \oplus w_2))\ (\mathsf{ABOp}\ v_1\ \mathsf{BitwiseXor}\ v_2).\end{aligned}$$

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

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

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

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

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

One construct that needs special translation treatment is variables. Be-

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

$$\begin{aligned}
\texttt{R}\ s \stackrel{\text{def}}{=} \ &\textsf{let}\ s' \ = \ s\ \mathit{with}\ \langle\!|\texttt{a}\ :=\ 1w;\ \texttt{b}\ :=\ 2w|\!\rangle;\\
&\quad\ s'' \ = \ s'\ \mathit{with}\ \texttt{a}\ :=\ s'.\texttt{a} + 1w;\\
&\quad\ \mathit{tmp} \ = \ 1w\\
&\textsf{in case}\ s''.\texttt{c}\ \textsf{of}\\
&\quad\ 0w\ \Rightarrow\ s''\ \mathit{with}\ \texttt{c}\ :=\ \mathit{tmp}\\
&\ \mid v\ \Rightarrow\ s''\ \mathit{with}\ \texttt{c}\ :=\ 0w
\end{aligned}$$

produces the following output when used as an input circuit:

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

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

Furthermore, in the same example we see that the translator supports nested

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

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

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

The translation approach taken here is inspired by Myreen and Owens' HOL-to-CakeML proof-producing code generator [79]. Translating from HOL to Verilog is both easier and more difficult. It is easier, at least in our case, because we accept a smaller and more specialized subset of HOL as input, and it is more difficult because the distance between HOL and Verilog is larger than the distance between HOL and CakeML.

3.6.4 Pass two: full program translation

The second pass takes EvalS theorems produced by the first pass and composes them into a theorem for a whole Verilog program. In terms of the example from Sec. 3.2, the first pass produces two EvalS theorems, one for A and one for B, and the second pass takes them as input and produces a correspondence theorem for the whole circuit AB.

The second pass is verified instead of proof-producing, and it consists of a HOL function intro_cvars and associated proof infrastructure. The function operates over Verilog syntax and takes a user-provided list of "communication variables" (in the Sec. 3.2 example, just `count`) and replaces all assignments to these variables with nonblocking assignments. The proof infrastructure helps to build a whole-program correspondence theorem out of the process theorems produced by the first pass.

The second pass requires that processes do not read from communication variables they have written to earlier in the same cycle. This style requirement should be seen as a strategy to shallowly embed nonblocking assign-

ments, as, process-locally, if a variable is not read after being written to, it does not matter if the writes to it are blocking or nonblocking. More precisely, the style requirement ensures that intro_cvars is semantics preserving in the sense that for any Verilog process p without nonblocking assignments, prun $fext\ (\Gamma, \Delta)\ p = \mathsf{Inr}\ (\Gamma', \Delta')$ implies that there exist Γ'_{cs} and Δ'_{cs} such that prun $fext\ (\Gamma, \Delta)$ (intro_cvars $cs\ p$) $= \mathsf{Inr}\ (\Gamma'_{cs}, \Delta'_{cs})$, and (Γ', Δ') and $(\Gamma'_{cs}, \Delta'_{cs})$ only differ in that writes to communication variables cs have been moved from Γ' to Δ'_{cs}.

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

3.7 Case studies

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

3.7.1 Processor case study

As part of our paper on verified stacks [67] referred to in the introduction, we showed how we have designed, verified, and synthesized a simple processor using the hardware-development methodology that is elaborated in this paper. The processor is capable of hosting programs compiled by the (verified) CakeML compiler [25], including the compiler itself. One of the main results in the paper is a theorem stating that running the compiler correctly compiles programs even when running on top of the processor (down to the Verilog level). The processor and CakeML compiler work that was required for this result is described in more detail in our verified-stacks paper, but we briefly recapitulate some hardware-relevant points here as the case study illustrates that the translator scales beyond the small examples presented in the paper so far.

The processor implements a small custom RISC instruction-set architecture. We designed the processor to be as simple as possible (it is, e.g., not pipelined) but still capable enough to host compiled CakeML programs. For this case study we targeted a PYNQ-Z1 board, using the board's DRAM for data and instruction

memory. The processor has support for hardware accelerators and consists of two processes: one for the processor itself and one for the hardware accelerator currently in use. (We have yet to develop any hardware accelerator beyond a placeholder integer-addition accelerator.) The processor's external environment, such as the DRAM module and an interrupt interface used for communicating with the external world, is modeled by an *fext* function. An earlier unverified but translatable version of the processor consisted of three processes, where the third process modeled a BRAM module used for data and instruction memory; illustrating that BRAMs can also be modeled in the translator's input language (modulo a minor problem with packed vs. unpacked arrays).

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

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

3.7.2 Regexp-matcher case study

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

To show that the flow works, we constructed a matcher implementation for the simple regexp `fo+bar`. The regexp-to-DFA compiler tool is proof-producing, and as our Verilog translator is proof-producing as well, running the flow, we were able to compose the above manual circuit-correctness theorem and the two generated correspondence theorems into a theorem stating that the Verilog circuit accepts the same language as the initial regexp. Being able to compose theorems from different developments in this way illustrates one of the advantages of embedding circuits inside ITPs over relying on traditional

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

verification methods such as model checking. We also synthesized the Verilog circuit with some glue code for a Basys 3 FPGA board connected to a keyboard, and the circuit worked as expected. As only the table differs between different regexp matchers, generating matchers for other regexps is straightforward. (Our driver is register-based; for larger regexps, a BRAM-based driver would be more appropriate. Such a driver would require another round of manual verification.)

3.8 Related work

Producing correct hardware with the help of an ITP has been addressed in many ways. For example, two earlier verified-stack projects [3, 5] report synthesizing for FPGAs: The pre-Verisoft PVS VAMP processor was specified in a custom-built gate-level language which relied on an unverified `pvs2hdl` tool [10] for translation to Verilog for synthesis, and a tool with similar functionality was available for the Isabelle/HOL version of VAMP [104]. In contrast, the DeepSpec Kami project [19] is based on the high-level HDL Bluespec and relies on the usual unverified Bluespec toolchain for synthesis.

More generally, it seems that one common synthesis strategy is to do verified or proof-producing compilation down to a "simple" or "low-level" language (or simply start from such a language) that can "easily", *but without proof*, be translated to some mainstream low-level HDL, such as Verilog or VHDL, and then feed this output to some external synthesis toolchain. For example Iyoda [52], Pizani Flor et al. [87], and Braibant and Chlipala [15] follow this approach. But also the opposite direction is possible; Hunt et al.'s [45] tool instead loads final-product Verilog programs into their ACL2 setup. In the context of hardware security, tools capable of loading (subsets of) VHDL and Verilog into Coq are available [12, 36]. The vital difference between our project and earlier projects is that earlier projects have not provided proofs all the way down to an explicitly stated Verilog/VHDL semantics.

As for Verilog-semantics work, the most complete Verilog formalization we are aware of is Meredith et al.'s rewriting-logic-based K-framework formalization [73], later ported to Isabelle/HOL by Khan et al. [55]. Meredith et al. model a larger subset of Verilog than we do, including e.g. continuous assignments and nonsynthesizable concepts such as delayed statements. This requires a more complete, and complicated, event-queue model and event-execution model. For other previous Verilog-semantics work, see the related-work section of Zhu et al. [115]. But whereas we have applied our Verilog semantics for verification by utilizing it in our translator, by in turn having applied the translator by extracting our case studies, missing from the mentioned and earlier Verilog formalization initiatives are convincing applications, where the authors apply

their semantics in nontrivial circuit-verification work. For example, Khan et al. prove that multiplication by two and left-shifting by one are equivalent in their semantics and then note that "proving more general theorems about complex designs would be extremely difficult".

3.9 Discussion

We will now discuss the trusted computing base (TCB) of our hardware-development methodology. To trust our methodology it is necessary to trust, beyond the usual suspects, an external synthesis toolchain (to be used in the final step of the methodology) and the soundness of our Verilog formalization.

Ideally, analogously to the situation in software development [59], our circuits would exit our ITP at the lowest possible level of abstraction. In hardware development, what the lowest level is depends on what technology we are targeting. For FPGAs, which we are interested in, the lowest level is at the level of FPGA bitstreams. Targeting this level from our input level would require access to a combination of proof-producing and verified tools for technology mapping, placement and routing, etc. At the time of writing, no such toolchain exists. There are multiple reasons for this; one of them being that the FPGA bitstream formats rarely have publicly available documentation. Instead, for transporting our Verilog circuits to FPGA bitstreams, our hardware-development methodology relies on existing unverified tools.

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

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

An inherent risk with targeting a large standard-defined language like Verilog without an official formal semantics is that it is impossible to prove readings of such language standards correct. Consequently, when arguing for our formalization's correctness, we have to fall back on empirical methods, such as testing our semantics against Verilog simulators and carrying out case studies based on our methodology. An alternative means of validation, which we

have not pursued, would be to prove a correspondence between our semantics and a semantics at the level of detail of Meredith et al.'s semantics [73]. In this approach, one would, e.g., show that the order of process execution does, indeed, not matter for Verilog programs satisfying valid_program. This would validate our semantics further, but would not address the question why the more detailed (and therefore more complicated) semantics is correct.

3.10 Conclusion

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

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

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

Chapter 4

Lutsig: A Verified Verilog Compiler for Verified Circuit Development

Andreas Lööw

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.

Published at *Conference on Certified Programs and Proofs (CPP)*, 2021

4.1 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) [59]. For example, the verified CakeML compiler [100] with its accompanying formal-methods tools hosts such a development methodology inside the ITP HOL4 [95]. 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 [95]. 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. 4.8) 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. 4.3). Specifically, we make the following contributions:

- we develop and describe a verified compiler from a subset of Verilog, by far the most widely used HDL today [23], 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 [68]; 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`.

4.2 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 [41]. This problem can to some extent be addressed in non-ITP methodologies by relying on translation validation [88] (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 from 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 [98]). 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 [13], Bluespec [82], and Chisel [6] (see also Gammie [26]). 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 [92] and FIRRTL [53]. Indeed, for communication between tools, compared to Verilog and VHDL, (e.g.) "LLHD's

simplicity offers a much smaller 'surface for implementation errors'" [92]. 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.

4.3 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. 4.1 shows a compilation chain we have made Lutsig part of. The chain goes from HOL circuits down to FPGA bitstreams. In this chain, the compilation from HOL circuits (A) to Verilog circuits (B) is handled by the proof-producing Verilog translator described in Lööw and Myreen [68]. 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. 4.1 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,

[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. 4.2. Nevertheless, if we simply want to use Lutsig as a trustworthy Verilog compiler without transporting proofs, then a simple unverified Verilog parser would suffice as a front-end.

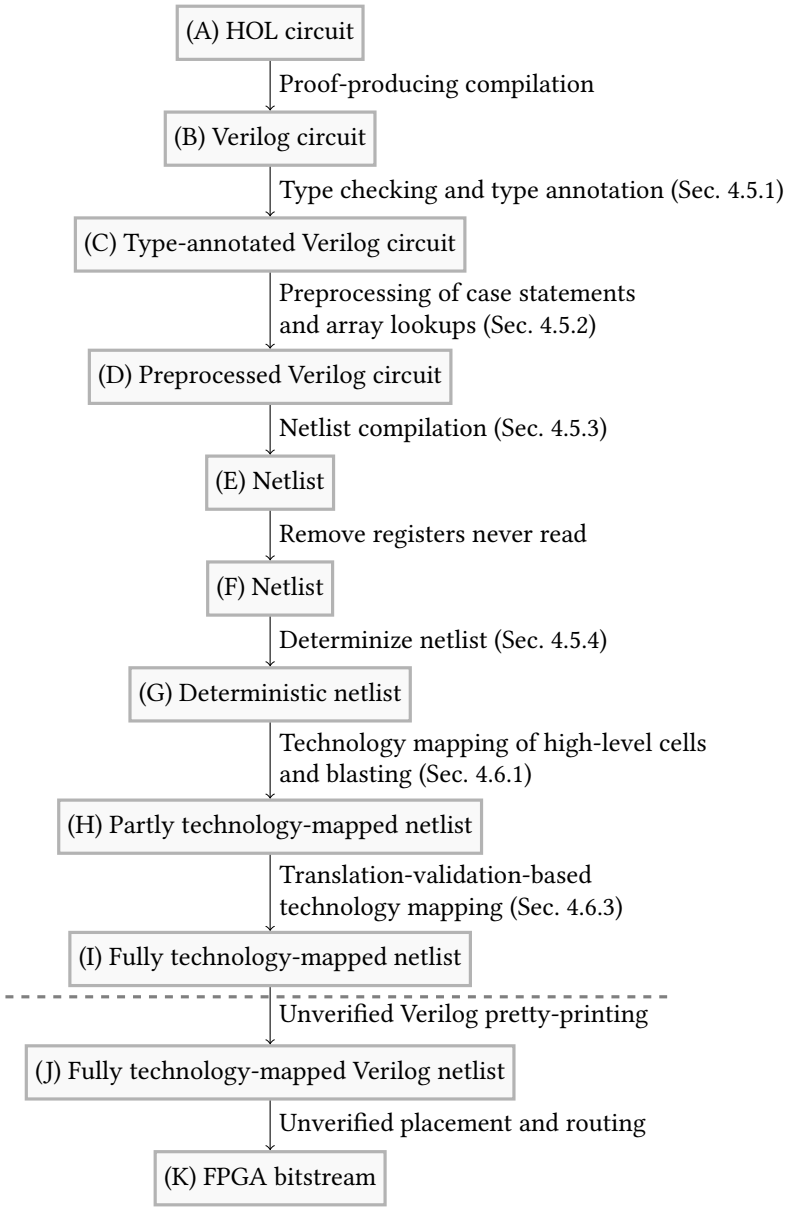

Figure 4.1. An overview of Lutsig's compilation passes and illustration of how Lutsig fits into circuit development.

however, not investigated this in detail yet. In terms of TCB, this means that we are not yet fully independent of Verilog.

In Sec. 4.7, to show how the suggested compilation chain of Fig. 4.1 works in practice, we present a case study following the setup. But first, we describe the different components of Lutsig in the coming sections.

4.4 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 [68]. In the earlier work, it was important that the Verilog semantics soundly captured the Verilog standard [48]. 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.

4.4.1 Source language: Verilog

Lutsig supports the subset of Verilog described in Fig. 4.2. The syntax is animated by a functional big-step operational semantics [83] designed with the aim of being a sound simplification of the simulation semantics provided by the Verilog standard [48]. The semantics is based on previous work [68], 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.

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

[2] The same scope limitations as described in Lööw and Myreen [68] 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	nonblocking assignment
	|	e `=` `X`	blocking `X` assignment
	|	e `<=` `X`	nonblocking `X` assignment
d	::=	$t\ v = c$	
	|	$t\ v$ `=` `X`	
p	::=	`always_ff @(posedge clk)` s	
m	::=	`module` $[(v, t)]$ $[d]$ $[p]$	

Figure 4.2. 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.

environment with the variables in $decls$. In the semantics, nondeterminism 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. 4.6.2 for an example. The function $fext : \mathbb{N} \to \mathsf{string} \to \mathsf{error} + \mathsf{value}$ 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 $fbits : \mathbb{N} \to \mathsf{bool}$ represents an infinite stream of nondeterministic bits and is used to give semantics to nondeterministic 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 nondeterministically. In the compiler, we did not find a use for the additional optimization freedom such nondeterministic interleavings offer, and consequently, a clock cycle in the semantics consists 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 nondeterministic bits from *fbits*. This semantics deviates from the standard, and the deviation is motivated in Sec. 4.4.1. The second construct we highlight is nonblocking 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 nonblocking 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. Nonblocking 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, nonblocking 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.

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 [48] 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 [49][3] had an accompanying synthesis standard [105]. 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

[3] Verilog 2005 [50] was published between Verilog 2001 and the merge of Verilog into SystemVerilog, but Verilog 2005 is a minor update of Verilog 2001.

occur in informal settings. Indeed, Mills and Cummings [75] outline some "RTL coding styles" (antipatterns) that yield simulation and synthesis mismatches.

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. 4.6.2) is stated in terms of Lutsig's formalization of Verilog's simulation semantics.

`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 [74, 97, 103]. 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. 4.5.4, 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. What remains to be explained, then, is `X`. In the Verilog standard [48, 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 `&&` [48, 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 `&` [48, 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" [48, 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" [48, 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 [48, 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 [48, 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 nondeterminism: 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 [48, pp. 148–149, p. 279].[4]

4.4.2 Target language: netlists

The syntax of Lutsig's target netlist language is described in Fig. 4.3. Cells are connected together with members of the cell_input type, called i in Fig. 4.3. A "variable" in the context of 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 [1]. 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 the specific FPGA targeted. carry4 cells represents the carry-chain logic available in the kind of FPGAs we target [107][5]. Lutsig uti-

[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 nondeterministic bits from out-of-bounds accesses.

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

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

Figure 4.3. Netlist values c, types t, inputs i, cells $cell$, register declarations d, and circuits cir.

lizes `carry4` cells to implement addition and wide equality checks (see Sec. 4.6). The semantics of $\texttt{ndet}(t)$ is that the cell nondeterministically (using *fbits*) generates a value of type t. The semantics of $\texttt{mux}(i_0, i_1, i_2)$ is that the cell outputs i_1 when i_0 is true, otherwise i_2. The semantics of $\texttt{array\_write}(i_0, n, i_1)$ 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.

4.5 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. 4.6.2).

4.5.1 Type checking and type annotating

Compilation starts with a type-checking and type-annotation pass. The type annotations are needed in subsequent preprocessing passes (Sec. 4.5.2), and the main Verilog-to-netlist pass (Sec. 4.5.3) 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, otherwise signaling a type error.

The type checker is sound in the following sense:

$$
\begin{aligned}
&\vdash \mathsf{typecheck}\ (\mathsf{Module}\ exttys\ decls\ ps) = \\
&\quad \mathsf{Inr}\ (\mathsf{Module}\ exttys'\ decls'\ ps') \Rightarrow \\
&\quad exttys' = exttys \wedge decls' = decls \wedge \\
&\quad \mathsf{vertype\_prog}\ (\mathsf{K}\ \mathsf{None})\ (\mathsf{Module}\ exttys'\ decls'\ ps') \wedge \\
&\quad \mathsf{run}\ fext\ fbits\ (\mathsf{Module}\ exttys'\ decls'\ ps')\ n = \\
&\quad \mathsf{run}\ fext\ fbits\ (\mathsf{Module}\ exttys\ decls\ ps)\ n
\end{aligned}
$$

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.

4.5.2 Preprocessing

Before the main Verilog-to-netlist pass (Sec. 4.5.3), nonconstant-index array lookups and case statements are compiled away by a series of Verilog-to-Verilog preprocessing passes. We now outline these preprocessing passes.

Array lookups. The preprocessing passes for handling array reads and array writes with nonconstant-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 preprocessing 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 nonconstant index. All read lookups with nonconstant indices are preprocessed 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 preprocessing 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`, (redoing 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 preprocessing pass (which simplifies compilation to netlists further down in the compilation chain).

Case statements. Case statements, both those introduced by the array-lookup preprocessing passes and those from the input program, are transformed into series of if statements by a case statements preprocessing 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`. We, however, leave it as future work to implement a more efficient compilation scheme.

4.5.3 Verilog-to-netlist compilation

The Verilog-to-netlist pass in Lutsig is based on the unverified Verilog compiler CSYN's compilation algorithm [34]. 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 compile Verilog statements and Verilog expressions. We explain compile_exp first. Grossly simplified, (part of) the correctness theorem for compile_exp is:

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

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:

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

Mapping the CAdd cell to cells actually available on FPGAs is the responsibility of technology mapping (Sec. 4.6).

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 postprocessing 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 nonblocking 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:

$$\begin{array}{l}\mathsf{compile\_stmt}\ s\ (\mathsf{BAssn}\ (\mathsf{NoIndexing}\ var)\ (\mathsf{Some}\ e)) \stackrel{\mathrm{def}}{=} \mathsf{do}\\ \quad (s,nl,inp) \leftarrow \mathsf{compile\_exp}\ s\ e;\\ \quad \mathsf{Inr}\ (s\ with\ \mathsf{bsi} := \mathsf{cset\_net}\ s.\mathsf{bsi}\ var\ inp,nl)\\ \mathsf{od}\end{array}$$

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 preprocessing passes (Sec. 4.5.2) have been run, all array indices are in-bound constants.

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

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

$$\mathsf{compile\_exp}\ s\ (\mathsf{Var}\ var) \stackrel{\mathrm{def}}{=} \mathsf{Inr}\ (s,[],\mathsf{cget\_net}\ s.\mathsf{bsi}\ var)$$

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 override 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 nonblocking 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 $inp_{\mathtt{c}}$ and a netlist $nl_{\mathtt{c}}$ for the expression, and `st` and `sf` are compiled by recursively calling compile_stmt in accordance with the following pseudocode (ignoring other state components beyond σ_{b} and σ_{nb} and denoting maps in the two stacks by sigmas as well):

$$\begin{aligned}&\mathsf{compile\_stmt}\ (\sigma_\epsilon :: \sigma_{\mathrm{b}}, \sigma_\epsilon :: \sigma_{\mathrm{nb}})\ \mathtt{st} = \\ &(\sigma_{\mathrm{b}}^{\mathtt{st}} :: \sigma_{\mathrm{b}}, \sigma_{\mathrm{nb}}^{\mathtt{st}} :: \sigma_{\mathrm{nb}}, nl_{\mathtt{st}}), \\ &\mathsf{compile\_stmt}\ (\sigma_\epsilon :: \sigma_{\mathrm{b}}, \sigma_\epsilon :: \sigma_{\mathrm{nb}})\ \mathtt{sf} = \\ &(\sigma_{\mathrm{b}}^{\mathtt{sf}} :: \sigma_{\mathrm{b}}, \sigma_{\mathrm{nb}}^{\mathtt{sf}} :: \sigma_{\mathrm{nb}}, nl_{\mathtt{sf}})\end{aligned}$$

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

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

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 $nl_{\mathtt{c}}$ 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 preprocessing, 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 $\mathsf{a} = \mathsf{b} = 0$ we would get $\mathsf{b} = 1$ from the Verilog program but $\mathsf{b} = 0$ from the generated netlist, because the cell $\mathtt{mux}(\mathsf{a}, 0, \mathsf{b})$ (where a and b refer to the registers for a and b) generated from the if statement would overshadow the blocking assignment (since $\sigma_{\mathrm{nb}}(\mathsf{b})$ will map to the mentioned mux, and $\sigma_{\mathrm{b}}(\mathsf{b})$ 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 nonblocking 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 [108, pp. 233–234] introduces a "usage restriction" saying not to mix blocking and nonblocking assignments (although, Vivado can synthesize programs with mixed assignments "without error"), and (as of this writing) the latest Yosys compiler [113] 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 nonblocking assignments is the same except that σ_{nb} is updated instead.

[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.

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 nondeterministic 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. 4.6.2, 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.

4.5.4 Netlist determinization

A determinization pass removes all nondeterminism from the circuit as the target cell set does not include any nondeterministic 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` cell 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 TBD ("to be determined") entries in σ TBD inputs and cell inputs with HBD ("has been determined") entries 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. An 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 an 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 values contained in the inputs' HBD entries. All `ndet` cells and mux 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. 4.6.2. 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.

4.6 Technology mapping

Technology mapping in Lutsig is divided into two passes: The first pass is verified (Sec. 4.6.1), and the second pass is based on translation validation (Sec. 4.6.3). The first pass maps high-level cells and furthermore functions as a preprocessing 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.

4.6.1 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 Boolean cells. The second pass is based on graph covering, and after the first pass' preprocessing 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 nonblasted circuit state s and a blasted state bs is invariant under running a nonblasted circuit and its blasted version:

$$
\begin{array}{l}
\mathsf{blast\_reg\_rel}\ s\ bs \stackrel{\text{def}}{=} \\
\quad \forall\, reg. \\
\qquad \mathsf{case}\ \mathsf{cget\_var}\ s\ (\mathsf{RegVar}\ reg\ 0)\ \mathsf{of} \\
\qquad\quad \mathsf{Inl}\ e\ \Rightarrow\ \mathsf{T} \\
\qquad\quad \mid \mathsf{Inr}\ (\mathsf{CBool}\ v)\ \Rightarrow \\
\qquad\quad\ \mathsf{cget\_var}\ bs\ (\mathsf{RegVar}\ reg\ 0) = \mathsf{Inr}\ (\mathsf{CBool}\ v) \\
\qquad\quad \mid \mathsf{Inr}\ (\mathsf{CArray}\ v)\ \Rightarrow \\
\qquad\quad\ \forall\, i.\ i < \mathsf{length}\ v \Rightarrow \\
\qquad\qquad\ \mathsf{cget\_var}\ bs\ (\mathsf{RegVar}\ reg\ i) = \mathsf{Inr}\ (\mathsf{CBool}\ (\mathsf{el}\ i\ v))
\end{array}
$$

Informally, Boolean registers remain untouched, and each array register is 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 $v = \texttt{array\_write}(i_0, n, i_1)$ 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.

4.6.2 Lutsig's top-level correctness theorem

Composing the correctness theorems for the different passes of the verified compiler (Sec. 4.5) and the verified technology mapper (Sec. 4.6.1) results in the following top-level theorem for the verified part of Lutsig:

$$
\begin{aligned}
&\vdash \mathsf{let}\ m\ =\ \mathsf{Module}\ \mathit{exttys}\ \mathit{decls}\ \mathit{ps}\ \mathsf{in}\\
&\quad \mathsf{compile}\ \mathit{keep}\ m = \mathsf{Inr}\ \mathit{circuit} \wedge \mathsf{writes\_ok}\ \mathit{ps} \wedge\\
&\quad \mathsf{vertype\_fext}\ \mathit{exttys}\ \mathit{vfext} \wedge \mathsf{same\_fext}\ \mathit{vfext}\ \mathit{nfext} \Rightarrow\\
&\quad \exists\, \mathit{cenv}\ \mathit{vfbits}.\\
&\qquad \mathsf{circuit\_run}\ \mathit{nfext}\ \mathit{nfbits}\ \mathit{circuit}\ n = \mathsf{Inr}\ \mathit{cenv} \wedge\\
&\qquad \mathsf{case\ run}\ \mathit{vfext}\ \mathit{vfbits}\ m\ n\ \mathsf{of}\\
&\qquad\ \ \mathsf{Inl}\ e\ \Rightarrow\ \mathsf{T}\\
&\qquad\ \ |\ \mathsf{Inr}\ \mathit{venv}\ \Rightarrow\ \mathsf{verilog\_netlist\_rel}\ \mathit{keep}\ \mathit{venv}\ \mathit{cenv}
\end{aligned}
$$

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

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 nonblocking assignments to the same variable (see Sec. 4.5.3). The predicate verilog_netlist_rel is similar to the predicate blast_reg_rel from Sec. 4.6.1 but only guarantees correspondence for registers in *keep* (as other registers might have been optimized away).

4.6.3 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 [47]. 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 [68] connected to Lutsig. The following predicate

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

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:

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

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 [109]. HOL4's SAT infrastructure relies on the presence of an external (unverified) SAT solver, like e.g. MiniSat [21], 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.

4.7 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. 4.1 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

$$\begin{array}{l}\mathsf{presignal}\ \mathit{fext}\ n\ \mathit{shift}\ \stackrel{\mathsf{def}}{=} \\ \quad \mathsf{if}\ n < \mathit{shift}\ \mathsf{then}\ 0w\ \mathsf{else}\ (\mathit{fext}\ (n - \mathit{shift}))\mathsf{.signal},\end{array}$$

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

$$\begin{array}{l}\mathsf{avg\_spec}\ \mathit{fext}\ n\ \stackrel{\mathsf{def}}{=} \\ \quad \mathsf{if}\ n = 0\ \mathsf{then}\ 0w \\ \quad \mathsf{else\ if}\ (\mathit{fext}\ (n - 1))\mathsf{.enabled\ then} \\ \quad\quad (\mathsf{presignal}\ \mathit{fext}\ n\ 1 + \mathsf{presignal}\ \mathit{fext}\ n\ 2 + \\ \quad\quad\ \mathsf{presignal}\ \mathit{fext}\ n\ 3 + \mathsf{presignal}\ \mathit{fext}\ n\ 4)\ //\ 4w \\ \quad \mathsf{else}\ (\mathit{fext}\ (n - 1))\mathsf{.signal}\end{array}$$

[8] The parenthesized letters refer to the stages introduced in Fig. 4.1.

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

$$
\begin{array}{l}
\mathsf{div\_by\_4}\ s \stackrel{\mathsf{def}}{=} \mathsf{let} \\
\quad s\ =\ s\ \mathit{with}\ \mathsf{sum}\ :=\ (0 :+ \mathsf{word\_bit}\ 2\ s.\mathsf{sum})\ s.\mathsf{sum}; \\
\quad s\ =\ s\ \mathit{with}\ \mathsf{sum}\ :=\ (1 :+ \mathsf{word\_bit}\ 3\ s.\mathsf{sum})\ s.\mathsf{sum}; \\
\quad s\ =\ s\ \mathit{with}\ \mathsf{sum}\ :=\ (2 :+ \mathsf{word\_bit}\ 4\ s.\mathsf{sum})\ s.\mathsf{sum}; \\
\quad s\ =\ s\ \mathit{with}\ \mathsf{sum}\ :=\ (3 :+ \mathsf{word\_bit}\ 5\ s.\mathsf{sum})\ s.\mathsf{sum}; \\
\quad s\ =\ s\ \mathit{with}\ \mathsf{sum}\ :=\ (4 :+ \mathsf{word\_bit}\ 6\ s.\mathsf{sum})\ s.\mathsf{sum}; \\
\quad s\ =\ s\ \mathit{with}\ \mathsf{sum}\ :=\ (5 :+ \mathsf{word\_bit}\ 7\ s.\mathsf{sum})\ s.\mathsf{sum}; \\
\quad s\ =\ s\ \mathit{with}\ \mathsf{sum}\ :=\ (6 :+ \mathsf{F})\ s.\mathsf{sum}; \\
\quad s\ =\ s\ \mathit{with}\ \mathsf{sum}\ :=\ (7 :+ \mathsf{F})\ s.\mathsf{sum} \\
\mathsf{in}\ s \\
\mathsf{avg\_step}\ \mathit{fext}\ s \stackrel{\mathsf{def}}{=} \mathsf{let} \\
\quad s\ =\ s\ \mathit{with}\ \mathsf{h3}\ :=\ s.\mathsf{h2}; \\
\quad s\ =\ s\ \mathit{with}\ \mathsf{h2}\ :=\ s.\mathsf{h1}; \\
\quad s\ =\ s\ \mathit{with}\ \mathsf{h1}\ :=\ s.\mathsf{h0}; \\
\quad s\ =\ s\ \mathit{with}\ \mathsf{h0}\ :=\ \mathit{fext}.\mathsf{signal}; \\
\quad s\ =\ s\ \mathit{with}\ \mathsf{sum}\ :=\ s.\mathsf{h0} + s.\mathsf{h1} + s.\mathsf{h2} + s.\mathsf{h3}; \\
\quad s\ =\ \mathsf{div\_by\_4}\ s \\
\mathsf{in} \\
\quad \mathsf{if}\ \mathit{fext}.\mathsf{enabled}\ \mathsf{then}\ s\ \mathit{with}\ \mathsf{avg}\ :=\ s.\mathsf{sum} \\
\quad \mathsf{else}\ s\ \mathit{with}\ \mathsf{avg}\ :=\ \mathit{fext}.\mathsf{signal} \\
\mathsf{avg}\ \mathit{fext}\ s\ 0 \stackrel{\mathsf{def}}{=} s \\
\mathsf{avg}\ \mathit{fext}\ s\ (\mathsf{Suc}\ n) \stackrel{\mathsf{def}}{=} \mathsf{avg\_step}\ (\mathit{fext}\ n)\ (\mathsf{avg}\ \mathit{fext}\ s\ n)
\end{array}
$$

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

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

Proving the whole implementation correct is also trivial and gives us the following theorem:

$$\vdash (\mathsf{avg}\ \mathit{fext}\ \mathsf{avg\_init}\ n).\mathsf{avg} = \mathsf{avg\_spec}\ \mathit{fext}\ n \tag{4.1}$$

We now derive a Verilog implementation vavg (B) from avg. The core component of the Verilog development tools [68] we have connected to Lutsig is a proof-producing translator from shallowly embedded Verilog-like HOL circuits

to deeply embedded Verilog circuits. For example, as part of the translation of avg, the translator derives the following Verilog code from avg_step:

```
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
```

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. 4.1, we can easily produce a Verilog-circuit-correctness theorem:

$$\begin{aligned} &\vdash \mathsf{lift\_vfext}\ \mathit{vfext}\ \mathit{fext} \Rightarrow \\ &\quad \exists\, s.\ \mathsf{run}\ \mathit{vfext}\ \mathit{vfbits}\ \mathsf{vavg}\ n = \mathsf{Inr}\ s \wedge \\ &\qquad \mathsf{get\_reg}\ s\ \text{“avg”} = \mathsf{Inr}\ (\mathsf{w2ver}\ (\mathsf{avg\_spec}\ \mathit{fext}\ n)) \end{aligned} \tag{4.2}$$

Now having procured the Verilog implementation vavg (B), the verified part of Lutsig (Sec. 4.6.2) can compile vavg to a partly technology-mapped netlist navg' (H). Now, translation-validation-based technology mapping (Sec. 4.6.3) 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. 4.2, Lutsig's correctness theorem (Sec. 4.6.2), and the theorem produced by translation-validation-based technology mapping (Sec. 4.6.3), we have derived

the following correctness theorem for the netlist implementation of the filter:

$$\begin{aligned}&\vdash \mathsf{lift\_nfext}\ \mathit{nfext}\ \mathit{fext} \Rightarrow\\ &\quad \exists\, s.\ \mathsf{circuit\_run}\ \mathit{nfext}\ \mathit{nfbits}\ \mathsf{navg}\ n = \mathsf{lnr}\ s \wedge\\ &\qquad \mathsf{get\_reg\_blasted}\ s\ \text{“avg”}\ (\mathsf{w2net}\ (\mathsf{avg\_spec}\ \mathit{fext}\ n))\end{aligned}$$

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 [40] 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:

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

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

[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.

```
 return acc + 2;
}
```

Vericert compiles this program to two Verilog processes each containing an 11-cases case statement:[10]

```
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: ;
```

[10]The C program and the Verilog program are 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. We also had to slightly modify the Verilog program such that it would fit the subset of Verilog Lutsig supports.

```
    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
```

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 preprocessing-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.

4.8 Related work

In the software world, realistic verified compilers such as the CakeML compiler [100] and the CompCert compiler [64] 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 [15] shares 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-by-cycle behavior. Unlike our source language, Fe-Si does not include `X` values. The Fe-Si compiler, like Lutsig, targets netlists. Neither the Fe-Si compiler nor Lutsig succeeds 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. [14] 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 [61]. 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), and 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 [19, 31, 45, 71, 87].

4.9 Conclusion

We have presented a new verified compiler called Lutsig that compiles Verilog programs down to technology-mapped netlists for FPGAs. We have also illus-

trated 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.

Chapter 5

Lutsig 2.0: Verilog, Synthesis-Tool Verification, and Circuit-Verification Methodology

Andreas Lööw

Abstract. In software development, verified compilers like the CompCert compiler and the CakeML compiler enable a software-development-and-verification methodology that allow software developers to establish program-correctness properties on the compiler's target level. Inspired by verified compilers for software development, the verified Verilog synthesis tool Lutsig enable the same methodology for Verilog hardware development. In this paper, we extend the subset of Verilog supported by Lutsig. Specifically, we extend Lutsig with support for `always_comb` blocks – one of Verilog's features that must be understood *as a hardware construct* rather than *as a software construct*. In extending Lutsig's Verilog support, we are required to revisit Lutsig's circuit-development-and-verification methodology to clarify where hardware constructs such as `always_comb` fit into the methodology. All development for this paper has been carried out in the HOL4 theorem prover.

Paper draft, not yet published

5.1 Introduction

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

1. *develop and compile* your program exactly as when using an unverified compiler;

2. *prove* a source-level correctness theorem about your program (by whatever means are available – the methodology is independent of how the correctness theorem is established); and, lastly,

3. *transport* the source-level program-correctness theorem down to the 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 deployed in many different software contexts, such as e.g. imperative programming [64], functional programming [60], concurrent programming [93], just-in-time compilation [7, 78], compiler-implementation correctness (by compiler bootstrapping) [60, 76], usability such as compositional/separate compilation [85], security such as constant-time preservation [8], and performance such as time/space reasoning [4, 29, 84]. For some contexts, the methodology applies directly. For other contexts, modifications or extensions are required.

In this paper, our interest lies in hardware development rather than software development. Previous work [14, 15, 66] on verified hardware-synthesis tools – also known as hardware compilers – show that VPD is applicable to hardware development, thereby providing a circuit-verification methodology. 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*. Concretely, we extend the verified Verilog synthesis tool Lutsig [66]. Lutsig compiles synchronous Verilog designs to technology-mapped netlists for (a class of) FPGAs.

Specifically, we extend Lutsig with support for `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. Moreover, beyond allowing hardware designers to express design intent, adding support for `always_comb` extends the expressivity of the subset of Verilog Lutsig support: In the new version of Lutsig, any Mealy machine can be encoded (see Sec. 5.8.1).

The main problem in extending Lutsig with `always_comb` support is keeping VPD intact. Both traditional Verilog development (TVD) and VPD fundamentally revolve around compilation; however, at times TVD and VPD do not mix well. Keeping VPD intact turns into a balancing act between TVD, with both its strengths such as synthesis-tool control and its problems like simulation-and-synthesis mismatches, and VPD, allowing transportation of source-level correctness theorems down to the compiler's target level.

All in all, we make the following two contributions:

- We add support for `always_comb` blocks to Lutsig, the Verilog semantics used in Lutsig, and a proof-producing Verilog code generator associated with Lutsig.

- We combine VPD, i.e. the traditional development methodology based on verified compilers, with TVD, i.e. traditional Verilog development, in a way that inherits the strengths of TVD while simultaneously avoiding its main weaknesses.

All the work for this paper has been carried out in the HOL4 theorem prover [95] and all source code and proofs are available at `https://github.com/CakeML/hardware`.

5.2 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.

5.2.1 Verified-program development (VPD)

We now give a more detailed description of VPD, following Leroy's [64] exposition. 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 interacting with a compiler, users are interested in the correctness of the 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 $Spec(B)$ 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.

5.2.2 Traditional Verilog development (TVD)

We now turn to TVD. As Weste and Harris [110, 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 means of communication with the synthesis tool the hardware designer has available is *modeling idioms*, which enable the hardware designer to express not only the function of their design but *what kind of hardware they want*. Modeling idioms is what allows the hardware designer to "write the HDL code that implies" the hardware design they have formed "on paper or in [their] mind."

One example of modeling idioms are the `always_ff` and `always_comb` blocks, allowing hardware designers to specify if combinational or sequential logic

should be inferred by the synthesis tool. In general, what model idioms are available depends on what technology is targeted. For example, the synthesis manual [108, p. 111] for Xilinx's unverified synthesis suite Vivado 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.

5.3 Problems in combining VPD and TVD

In this section, we provide an analysis of the problem of combining VPD and TVD. We identify the two semantics of Verilog as the core of the problem.

Since we are here interested in applying 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 (sometimes abbreviated "ver"), and
- T = technology-mapped netlist for (a class of) FPGAs (sometimes abbreviated "nl").

For L_T, Lutsig uses a simple custom netlist language. What remains to specify is L_S, and this is where our problems begin.

The problems associated with L_S arise from the fact that to support both (1) simulation and other forms of program reasoning and (2) synthesis, Verilog is equipped with two semantics: one *simulation semantics* $L_{\text{ver}}^{\text{sim}}$ and one *synthesis semantics* $L_{\text{ver}}^{\text{synt}}$ (based on, among other things, modeling idioms).

From the perspective of TVD, the presence of a synthesis semantics is motivated by the hardware designers' need for a way to control their synthesis tools (using modeling idioms); however, having two source semantics breaks the theorem-transportation step of VPD since in VPD it is assumed that we have one single semantics L_S for our source language S. The VPD problem is as follows: If we prove our Verilog design P_{ver} correct with respect to $L_{\text{ver}}^{\text{sim}}$, i.e. $P_{\text{nl}} \models_{L_{\text{ver}}^{\text{sim}}} \mathit{Spec}$, and prove our synthesis tool correct with respect to $L_{\text{ver}}^{\text{synt}}$, we can no longer transport the design-correctness result to the netlist level, i.e. derive $P_{\text{nl}} \models_{L_{\text{nl}}} \mathit{Spec}$, by simple composition. In fact, the same problem occurs in TVD, where this composition problem resulting from the two semantics makes itself known in the form of *simulation-and-synthesis mismatches*.

The underlying idea that guided the design of the first version of Lutsig – i.e., that hardware-synthesis-tool verification is just software-compiler verification but for hardware, in other words, straightforward application of VPD, essentially ignoring TVD – runs into problems when Lutsig is extended with features such as `always_comb` which must be understood *as hardware constructs* rather than *as software constructs*. We need the perspective of TVD to even make sense of hardware language features like `always_comb`. (If this is not clear at this point, it will hopefully be more clear after Sec. 5.10.1, where some concrete problems with synthesizing `always_comb` blocks are enumerated.)

Before we describe Lutsig's solution (Sec. 5.5) to the problem outlined in this section, we must first discuss Verilog's two semantics in more detail (Sec. 5.4).

5.4 A closer look at Verilog's two semantics

Having identified the two semantics $L^{\text{sim}}_{\text{ver}}$ and $L^{\text{synt}}_{\text{ver}}$ for Verilog as a source of problems, we now take a closer look at the two semantics and the relationship between them.

5.4.1 Simulation semantics

The simulation semantics $L^{\text{sim}}_{\text{ver}}$ is given by the (System)Verilog standard [48]. The semantics is large, complicated, and full of gotchas [98], but at the end of the day, is an informally specified event-based operational semantics centered around a stratified event queue.

5.4.2 Synthesis semantics

The situation for the synthesis semantics $L^{\text{synt}}_{\text{ver}}$ is less straightforward. Since the Verilog standard does not provide a synthesis semantics and the Verilog synthesis standard [105] has been withdrawn, it is up to each synthesis tool to provide their own synthesis semantics. However, current tool-specific synthesis manuals, such as e.g. the synthesis manuals for Vivado [108] and Quartus [51], largely contain similar material as the withdrawn synthesis standard (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.

Beyond synthesis idioms, the synthesis standard defines Verilog's synthesis semantics in terms of how it relates to Verilog's simulation semantics. Now, we therefore go into the relationship between the two standards.

5.4.3 Relationship between the two semantics

The relationship between Verilog's simulation semantics and synthesis semantics is complicated: The synthesis standard both builds on and deviates from Verilog's simulation semantics. Some parts of the synthesis standard are best understood as describing how to use the syntax and semantics of Verilog, as defined in the Verilog 2001 standard [49], to model hardware; in other words, how to use the simulation semantics to model hardware. Other parts of the synthesis standard, however, clearly deviate from the simulation semantics. Some of these deviations 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 "misordered":

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

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

In the same appendix, we are warned that "making a Verilog x-assignment to a signal 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." This means that e.g. the following Verilog fragment, with `a` of type `logic`, will cause a mismatch:

```
a = 'x;

if (a)
 a = 1;
else
 a = 0;
```

According to Verilog's simulation semantics [48, p. 299], `a` must be 0 after executing the fragment. Clearly, this is not compatible with any "don't care" interpretation of X values.

5.5 Lutsig's VPD methodology for Verilog

We are now at a point where we can present the circuit-verification methodology offered by Lutsig to hardware designers.

[1] Here presented verbatim, using an `always @*` block rather than an `always_comb` block since the synthesis standard was published before the SystemVerilog standard.

Per the discussion up till now, we cannot simply abandon either VPD and TVD since both provide value to Verilog hardware development. Instead, a balance between the two is needed. In our view, combining VPD and TVD is made complicated by the complicated relationship between Verilog's simulation semantics and synthesis semantics. This complicated relationship also affects informal Verilog development; this can be made clear by looking at today's synthesis tools, in where simulation-and-synthesis mismatches are, within the same tool, handled along the whole spectrum of: silently miscompiling Verilog designs, issuing warnings, and aborting the compilation process entirely. This means that the result of a successful synthesis run is unclear for hardware developers, since a successful run does not mean an actually successful synthesis run.

Lutsig's solution to the above consist of two parts: (1) a multipurpose Verilog semantics (Sec. 5.5.1), (2) a fail-fast synthesis algorithm (Sec. 5.5.2). The guiding design goal for both the semantics and the synthesis algorithm is that hardware designers must be able to trust that if Lutsig runs successfully, then the synthesis process was actually successful.

Clear[2] synthesis-error handling is Lutsig's contribution to TVD: Beyond this new error-handling guarantee, the Verilog design methodology for Lutsig is the same as for unverified-synthesis-based development (i.e., TVD), that is, both approaches follow the same synthesis idioms and guidelines. This is by design, since we want Lutsig-based development to be as familiar to hardware designers as possible.

5.5.1 Part 1: Lutsig's multipurpose Verilog semantics

To allow theorem transportation, as in VPD, and avoid simulation-and-synthesis mismatches, unlike TVD, Lutsig is based on a single Verilog semantics: a *Verilog simulation semantics for verification and synthesis* $L_{\text{ver}}^{\text{Lutsig}}$.

For theorem transportation, we have proved Lutsig semantics preserving with respect to $L_{\text{ver}}^{\text{Lutsig}}$. The full theorem is presented in Sec. 5.11, but in short the theorem states that if Lutsig compiles a Verilog design successfully *there can be no simulation-and-synthesis mismatches.*

The semantics is defined formally in HOL and is the same semantics as in the first version of Lutsig, with the exception that we have added support for `always_comb` blocks. The extension to the semantics is described in Sec. 5.8. The aim of the semantics is to capture Verilog's simulation semantics except where we have deemed Verilog's simulation semantics and synthesis semantics to be

[2]We write "clear" here, rather than "fail fast", since we expect some future Lutsig features, such as BRAM handling, to require more involved error handling than the current fail-fast approach. This, however, is future work.

too incompatible. At such points of disagreement, we must opt for synthesis-oriented behavior, since ultimately the semantics is used for synthesis. Up till now – that is, for the subset of Verilog that Lutsig supports as of this paper – the only intentional incompatibility between Lutsig's Verilog semantics and Verilog's simulation semantics is that X values have "don't care" semantics in Lutsig's semantics.[3]

5.5.2 Part 2: Lutsig's synthesis algorithm

The semantics $L_{\text{ver}}^{\text{Lutsig}}$ is, however, not the full story for avoiding simulation-and-synthesis mismatches in Lutsig. The semantics $L_{\text{ver}}^{\text{Lutsig}}$ is part of Lutsig's synthesis story since it is not feasible in practice to handle all simulation-and-synthesis mismatches in Lutsig's synthesis algorithm (in particular, mismatches based on X values). In the end, it is the *combination* of $L_{\text{ver}}^{\text{Lutsig}}$ and the design of Lutsig's synthesis algorithm that allow for the clean circuit-development methodology offered by Lutsig.

One interesting aspect of Verilog development is that you can write down Verilog designs that simply do not make sense. The construct we add support for in this paper, `always_comb`, provides examples of this. For example, what should be happen when sequential logic is put inside an `always_comb` block? In Lutsig, this is handled in the synthesis algorithm, by aborting the synthesis process. The general approach followed in the synthesis algorithm implementation is to fail fast. If there is a risk for a simulation-and-synthesis mismatch, Lutsig aborts the synthesis process. This approach ensures a uniform treatment of simulation-and-synthesis mismatches, such that a successful synthesis run is always free of mismatches.

The semantics, on the other hand, has no problem with sequential logic inside `always_comb` blocks, and in this sense simulation-and-synthesis mismatches are possible even when using Lutsig. The guarantee that there are no simulation-and-synthesis mismatches only hold for designs for which Lutsig runs successfully. For a design with sequential logic inside an `always_comb` block, Lutsig will abort unsuccessfully (as explained in more detail in Sec. 5.10). Alternatively, one could have handled this in the semantics, but this would complicate the semantics and deviate from Verilog's simulation semantics. Therefore, handling mismatches is done by both Lutsig's Verilog semantics and Lutsig's synthesis algorithm.[4]

[3] Opting for synthesis-oriented behavior at points of disagreement means that Lutsig (at such points) is not compatible with Verilog simulators based on Verilog's simulation semantics. One might ask if this means that Lutsig really is a Verilog synthesis tool. In our view, Lutsig is a Verilog synthesis tool: we simply make the X value convention followed by unverified Verilog synthesis tools explicit.

[4] Here follows a longer design rationale: While this approach allows for a simpler semantics,

5.5.3 Synthesis modeling idioms and Lutsig

The synthesis modeling idioms implemented in Lutsig take largely a background role in the formal development. The idioms influence both the design of the semantics $L_{\text{ver}}^{\text{Lutsig}}$ and how Lutsig's synthesis algorithm interprets Verilog designs, but the synthesis idioms are for example not visible in Lutsig's functional correctness theorem.

We opted for not proving that Lutsig strictly complies with the modeling idioms it implements, since, to not rule out too many optimization opportunities, a synthesis tool might sometimes have to diverge from what the user-provided modeling idioms tell the synthesis tool to do. Indeed, the Verilog synthesis standard's correctness criterion for synthesis tools [105, p. 13] is oriented around the behavioral equivalence of Verilog designs and synthesized netlists rather than strict idiom compliance. The standard "do[es] not take into account any optimizations or transformations" and states that "a specific [synthesis] tool may perform optimization and not generate the suggested hardware inferences or may generate a different set of hardware inferences [...] provided the synthesized netlist has the same functionality as the input model." Moreover, inferring designer intent from Verilog designs is in general fragile. Hence, strictly enforcing idiom compliance is less useful than it may seem at first. Specifically, we do not want to use the same input design for both implementation and specification. E.g., it is easy to accidentally specify latched logic (i.e., level-sensitive stateful logic) when combinational logic was intended by simply forgetting one assignment in a large `always` block.

With that said, in some cases Verilog *does* allow hardware designers to specify their intent independently from the implementation they provide. Ver-

---

one downside of the approach is that there is no clean mathematical description of Lutsig's Verilog synthesis behavior (in other words, Lutsig's Verilog synthesis semantics) – instead, the behavior is given by the combination of Lutsig's Verilog semantics and Lutsig's synthesis algorithm (much like unverified synthesis tools). However, for hardware development, no such description is needed: it follows from Lutsig's correctness theorem that if Lutsig runs successfully, the semantics of Lutsig's Verilog (simulation-like) semantics is preserved. This guarantee is sufficient for circuit-correctness theorem transportation.

What we have is, effectively, four different Verilog semantics: Verilog's simulation semantics $L_{\text{ver}}^{\text{sim}}$ (defined informally by the Verilog standard), Verilog's synthesis semantics $L_{\text{ver}}^{\text{synt}}$ (unclear what should be the authoritative definition: one version of the semantics is defined informally by the Verilog synthesis standard, but this version lacks a succinct description and is instead defined in terms of a combination of synthesis idioms and how it relates to $L_{\text{ver}}^{\text{sim}}$), Lutsig's (multipurpose) Verilog semantics $L_{\text{ver}}^{\text{Lutsig}}$ (defined formally in HOL, aiming to be as close to $L_{\text{ver}}^{\text{sim}}$ as possible while still being fit for synthesis purposes), and Lutsig's Verilog synthesis semantics (which, like $L_{\text{ver}}^{\text{synt}}$, lacks a succinct description, as described in the previous paragraph).

What the lack of a succinct description means in practice is that Lutsig's Verilog synthesis semantics must be presented to hardware designers in the same way as unverified synthesis tools present their Verilog synthesis semantics: by modeling idioms. What we gain from using Lutsig, because of Lutsig's VPD-inspired approach, is that simulation-and-synthesis mismatches will never introduce bugs into our designs.

ilog's `always_comb`, `always_ff`, and `always_latch` are examples of such cases: they allow hardware designers to declare if they intend combinational logic, sequential logic, or latched logic. Furthermore, such intent is robust under optimizations. An unverified synthesis tool can of course check such intent annotations, but a verified synthesis tool can check and enforce the intent annotations with mathematical rigor.

5.6 The rest of the paper

What we should conclude from the discussion up till now is that VPD and TVD mutually benefit from being combined: TVD provides VPD with the conceptual toolbox needed to make sense of Verilog's hardware features such as `always_comb`, and VPD saves TVD from simulation-and-synthesis mismatches and provides circuit-correctness theorem transportation to Verilog developers.

The rest of the paper consist of putting our discussion up till now into practice by adding support for `always_comb` to Lutsig and surrounding components. Previously, Lutsig only supported `always_ff` blocks. After we have presented the updates to Lutsig, Lutsig's full correctness theorem is presented in Sec. 5.11, and, in Sec. 5.12, going beyond functional correctness, we discuss a nonfunctional property about Lutsig we have proved that captures (part of) that `always_comb` must generate combinational logic.

5.7 Using Lutsig in practice

Before heading into technical details, we here make the discussion up till now more concrete 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.[5]

Example module We let the Verilog module in Fig. 5.1, implementing a moving-average filter, serve as a running example in this section. Sec. 5.8 provides more details on Lutsig's Verilog support and the expressiveness of the supported subset. Note that the module utilizes the new `always_comb` support.

[5]Like any Verilog synthesis tool, Lutsig can be made part of different hardware-development flows. E.g., there is also a (unverified) Verilog-text-file front-end for Lutsig available such that Lutsig can be used like a conventional Verilog synthesis tool. But since we in this paper are interested in VPD-inspired flows, in this section we do not describe alternatives flows.

```
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
```

Figure 5.1. Example Verilog module

Proving Verilog designs correct Lutsig is accompanied by a proof-producing Verilog code generator. The code generator is explained in more detail in Sec. 5.9. In short, the code generator constructs a Verilog module P_{ver} given a HOL embedding P_{HOL} of a Verilog circuit. It is important to emphasize that an embedded Verilog circuit is still a Verilog circuit, meaning that for example the synthesis idioms of HOL-embedded Verilog circuits are the synthesis idioms of Verilog. Since 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. $P_{\text{ver}} \models_{L_{\text{ver}}^{\text{Lutsig}}} \mathit{Spec}$, by simple composition.

The Verilog module in Fig. 5.1 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, then the generated Verilog module satisfies the specification (in 8-bit modular arithmetic) $\texttt{avg}[n] = \frac{\sum_{i=1}^{4} \texttt{signal}[n-i]}{4}$, i.e., the module is correct.

Going to the netlist level Now having both a Verilog module (Fig. 5.1) 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 $P_{\text{ver}} \models_{L_{\text{ver}}^{\text{Lutsig}}} \mathit{Spec}$). We discuss Lutsig in more detail in Sec. 5.10 and the functional correctness of Lutsig in Sec. 5.11.

FPGAs At this point, our formal development ends. To run the netlist implementation produced by Lutsig on top of an FPGA, the netlist needs to be placed and routed onto the 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. 5.1 runs correctly on top of an FPGA board we have available.

5.8 Formal semantics

In this section we first describe the updated source language of Lutsig (Sec. 5.8.1 and 5.8.2); that is, we describe the subset of Verilog that Lutsig supports and Lutsig's Verilog semantics $L_{\text{ver}}^{\text{Lutsig}}$ for this subset. We then describe the updated target language of Lutsig (Sec. 5.8.3), that is, Lutsig's netlist language.

5.8.1 Expressiveness

Our initial motivation for working on a new version of Lutsig was to extend the kind of hardware Lutsig can produce. With the new support for `always_comb` blocks, any Mealy machine can be implemented using Lutsig. Recall that a Mealy machine is a tuple $(S, s_0, \Sigma, T, \Lambda, G)$ where S is a finite set of states, $s_0 \in S$ the initial state, Σ an input alphabet, $T : S \to \Sigma \to S$ a transition function, Λ an output alphabet, and $G : S \to \Sigma \to \Lambda$ an output function. Before the support for `always_comb` blocks, every circuit output had to be the direct output of a register, and consequently only trivial output functions G could be encoded in Lutsig's Verilog subset. Now, with support for `always_comb` blocks, any output function G can be encoded.

5.8.2 Lutsig's Verilog semantics

Lutsig's Verilog semantics is designed to be usable for both circuit verification and compiler verification. Lutsig's Verilog semantics is much smaller than the semantics of full Verilog since we are only concerned with synthesizable Verilog code (e.g. test benches are out of scope for our semantics, since HOL already provides all proof and testing infrastructure we need). The semantics presented

here is the same semantics as presented by Lööw [66] but with added support for `always_comb` blocks.

In Lutsig's Verilog, a Verilog module 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.

Handling multiple modules and module instantiations is future work. Lutsig's Verilog semantics is a functional operational semantics that takes four inputs:

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

In Lutsig's Verilog semantics, there are two kinds of values: Booleans (i.e., `logic`) and (packed) 1-dimensional arrays (e.g., `logic[3:0]`). In full Verilog, a bit can take on four different values: 0, 1, X, and Z. In Lutsig's Verilog, a bit can only take on the values 0 and 1. This is because we do not support tristate logic, and no explicit representation of X values is needed since X assignments (and uninitialized variables) are given semantics by assigning nondeterministic bits from $fbits$ instead of a symbolic X value. See Lööw [66] for a longer discussion on this.

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. In short, Lutsig's Verilog process-level 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.

We now describe the process-level semantics of Lutsig's Verilog in more detail. A Verilog module, in Lutsig's view, is not an entity that, like in full Verilog, reacts to a series of external events (by propagating them through the module); rather, a Verilog module is driven by a clock input `clk`, and every new clock cycle a new snapshot of the external world is given to the module through $fext$ in one go. In other words, the semantics is simplified by not modeling modules' combinational behavior between clock cycles.

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 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.[6]

Our intention is that Lutsig's Verilog semantics should coincide with the simulation semantics of full Verilog (modulo X values) as long as good coding style is followed, e.g. not writing blockingly in an **`always_ff`** block to a variable shared between processes as mentioned above. (Formally proving a correspondence between the two semantics would be a worthwhile endeavor.) For using the semantics in proving the implementation of Lutsig correct, coding-style checks are needed in some cases in Lutsig's implementation – in other cases, Lutsig's synthesis algorithm happens to generate netlists behaviorally equivalent to the input design even for suspicious Verilog code so no checks are needed.

5.8.3 Lutsig's netlist semantics

Lutsig's netlists are used for multiple purposes: such as representing intermediate compilation results, representing non-technology-mapped netlists

[6] 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.

consisting of high-level cells such as addition cells, and representing technology-mapped netlists for (a class of) FPGAs. See Lööw [66] for more details.

For this version of Lutsig, to support the compilation of `always_comb` blocks, we split 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. 5.10. 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. We say that a netlist step consists of running all pseudoregister cells, updating all pseudoregisters, and then running all remaining cells. 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.

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 those generated by Lutsig), it is easy to prove that the above semantics collapses into the following clean semantics:

- 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.

5.9 The proof-producing Verilog code generator

The proof-producing Verilog code generator bundled with Lutsig 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 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.

For this paper, we have extended the code generator with support for translating **always_comb** blocks. We have also changed how nonblocking assignments are shallowly embedded, such that a larger set of Verilog designs can be embedded.

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. 5.1 is simply:

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

The next-state function the block is generated from is:

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

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 for the new nonblocking-assignments embedding style. 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

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

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:

$$\begin{aligned}
&\mathsf{mk_circuit}\ sstep\ cstep\ s\ fext\ 0 \stackrel{\mathsf{def}}{=} cstep\ (fext\ 0)\ s\ s\\
&\mathsf{mk_circuit}\ sstep\ cstep\ s\ fext\ (\mathsf{Suc}\ n) \stackrel{\mathsf{def}}{=} \mathsf{let}\\
&\quad s = \mathsf{mk_circuit}\ sstep\ cstep\ s\ fext\ n;\\
&\quad s = sstep\ (fext\ n)\ s\ s\ \mathsf{in}\\
&\quad cstep\ (fext\ (\mathsf{Suc}\ n))\ s\ s
\end{aligned}$$

E.g., the HOL representation of the Verilog module in Fig. 5.1 is

$$\mathsf{mk\_circuit}\ (\mathsf{procs}\ [\mathsf{avg\_ff}])\ (\mathsf{procs}\ [\mathsf{avg\_comb}]).$$

Handling variable initialization requires one more level of encoding but is straightforward and therefore not covered in this paper.

It is important to emphasize that the input language for the code generator is shallowly embedded Verilog, not a separate HOL-based HDL. 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. It is the straightforward translation between shallowly embedded Verilog and deeply embedded Verilog carried out by the code generator that enables this way of thinking. Importantly, since the input language *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.

5.10 Lutsig

We now discuss Lutsig's new support for `always_comb` blocks. In supporting `always_comb` blocks, Lutsig must ensure that all `always_comb` blocks actually represent combinational logic [48, p. 207]. In other words, code inside `always_comb` blocks must never be mapped to registers or other stateful constructs.

Note how, as stated in Sec. 5.5, and as illustrated by example in Sec. 5.10.1, adding support for `always_comb` blocks affects *both* Lutsig's Verilog semantics $L_{\text{ver}}^{\text{Lutsig}}$ and Lutsig's synthesis algorithm. E.g., both the semantics and (as we will see) the algorithm need to sort `always_comb` blocks by dependencies. However, sometimes the two diverge: E.g., sequential logic inside `always_comb` blocks

is not caught in the semantics, instead, checking for sequential logic inside `always_comb` blocks is left entirely to the synthesis algorithm.

Handling sequential logic inside `always_comb` blocks 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.

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 variable level. E.g., all elements of an array variable are either all mapped to one pseudoregister or all to one real register. In full Verilog, the analysis whether an `always_comb` block represents combinational logic is instead based on longest static prefixes [48, p. 282]. Such more fine-grained analysis allows for different parts of an array to be mapped to different kinds of logic. But even with our simplified approach, an analysis on the element level is ultimately 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. Therefore, an analysis on the element level must be carried out at some point in the synthesis process.

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

REM1 Remove unused registers (variable-level analysis)

DET Remove all nondeterminism from the netlist

MAP Compile and technology-map away array cells

REM2 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 REM1/2 and DET, to handle the more complex netlist semantics with pseudoregisters.

5.10.1 Problems in compiling combinational logic

We now highlight some of the problems related to compiling combinational logic and describe how Lutsig handles them. Our presentation is example-driven, and many of the problems relate to avoiding simulation-and-synthesis mismatches (i.e., ensuring behavioral equivalence between (output) synthesized netlists and (input) Verilog modules). 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 $\forall P_S\ P_T,\ Comp(P_S) = \texttt{OK}(P_T) \implies P_S \approx P_T$) do not protect against compiler bugs that cause compilers to fail on valid input code (i.e., bugs causing the the compiler to return `Error` when it should have returned `OK`).

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. REM1 and REM2 make sure the (real) registers for `a` and `b` generated by SYNT are optimized away before the synthesis process is over. REM1 and REM2 are the same pass run twice; we run the pass twice since we want to catch easy cases (such as `a` in the example here) early but at the same time also make sure to catch cases requiring element-level analysis (such as `b` in the example here).

Sequential logic in `always_comb` blocks Lutsig must check that all `always_comb` blocks actually model combinational logic. E.g., the following block must be rejected by Lutsig:

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

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

MAP is a netlist pass 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. 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 that exemplifies code that 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
 // the comments below illustrate how sigma is updated
 // when MAP iterates over the always_comb block

 // b is a pseudoregister, therefore we start with:
 // sigma(b) = [illegal, illegal]

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

 // no problem reading the full b here since
 // all elements of b covered (i.e., no illegal
 // inputs in sigma(b))
 b = b + 1;
end
```

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.

Intrablock order problems Recall the `andor1b` module with "mis-ordered" assignments discussed in Sec. 5.4. 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
```

Interblock order problems Recall that Lutsig's non-event-based Verilog semantics sorts `always_comb` blocks before execution (see Sec. 5.8). 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;
```

If statements Recall, from Sec 5.5, the discussion on accidentally specifying latched logic. Lutsig handles such situations 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.

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.

5.11 Functional correctness of Lutsig

We now state Lutsig's functional-correctness theorem. 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. Although the only kind of behavior we take into account for Lutsig's correctness is externally visible cycle-per-cycle register and wire states, the presence of nondeterminism ($fbits$) necessitates a slightly complex-looking theorem statement. If we let $P \Downarrow_L^{n,fbits} S$ denote that program 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:

$$\begin{array}{l}\mathtt{Lutsig}(P_{\text{ver}}) = \mathtt{OK}(P_{\text{nl}}) \implies \\ \exists S_{\text{nl}}, P_{\text{nl}} \Downarrow_{L_{\text{nl}}}^{n,fbits} S_{\text{nl}} \wedge \\ \qquad \exists fbits', P_{\text{ver}} \Downarrow_{L_{\text{ver}}^{\text{Lutsig}}}^{n,fbits'} S_{\text{ver}} \implies S_{\text{nl}} = S_{\text{ver}}\end{array}$$

5.12 Nonfunctional correctness of Lutsig

We now state an example nonfunctional-correctness property we have proved about Lutsig. Recall that the main purpose of Verilog's synthesis semantics is to provide a way for hardware designers to express hardware ideas to their synthesis tools through modeling idioms. In our approach, this places the role of the synthesis semantics mainly outside the formal development. Nevertheless, some parts of the synthesis semantics can be treated formally. To illustrate this,

we have proved a property that captures (part of) the (synthesis-semantics) idea that `always_comb` blocks must be mapped to combinational logic [48, p. 207]: 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}.

The above may seem like a trivial property, but note that it relates concepts in the input design (writes) to concepts in the final netlist (registers). This means that we must carry information from the very first compilation phase down to the very last. Moreover, Lutsig did not actually satisfy this property before we started working on the property. This was because the SYNT pass (see Sec. 5.10) 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* – the former does not reliably track the latter since writes may be optimized away in the compilation process!

5.13 Related work

Different hardware languages are associated with differ approaches to hardware design. In turn, different hardware-design approaches require different approaches to taking synthesis aspects into consideration in verification work. In this paper, we consider Verilog-based hardware design. But even within Verilog-based hardware design, multiple development approaches are available. Of particular interest for this paper is different approaches to establish circuit reliability. Verified synthesis tools do not yet belong to the mainstream hardware development toolbox. Instead, e.g. translation validation (known as formal equivalence checking in the hardware world) is more common. The first paper on Lutsig [66] compares Lutsig to different approaches to circuit reliability, so we do not repeat the discussion here.

For this paper, it is not only Lutsig that we had to extend to support `always_comb` blocks. We also had to update Lutsig's Verilog semantics. Verilog has a reputation of being difficult to understand and formalize. One of the problems highlighted by the most detailed Verilog semantics available today, by Meredith et al. [73], actually relate to combinational logic. Their semantics does not properly handle when a variable is assigned multiple times in the same clock cycle (the writes are not propagated through combinational logic like they would in hardware, see the paper for details). To address this issue, Meredith et al. suggest a non-standard alternative semantics (for continuous assignments) based on Gordon's [30] Verilog semantics. Lutsig's Verilog semantics do not suffer from this problem, but Lutsig's semantics also diverges

from the Verilog standard – by unconditionally running all `always_comb` blocks (in dependency order) each clock cycle, rather than following the event-based semantics provided by the Verilog standard.

Non-Verilog development approaches include e.g. generating hardware from software languages like C, so-called high-level synthesis (HLS), and embedding existing or new hardware languages inside general-purpose software languages.

No approach to hardware completely shields the hardware designer from synthesis aspects – not even HLS. For example, the manual [106, p. 17] for Vitis, a 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 [106, 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 the reader 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 manual is 660 pages, reflecting the fact that not even HLS manages to encapsulate the complexities of synthesis.

Approaches in where hardware constructs are embedded inside general-purpose languages include Lava [13] in Haskell and Chisel [6] in Scala. When embedding netlists inside general-purpose languages, most of the complexity of synthesis can be pushed to the general-purpose languages.

As for verified hardware-synthesis tools, Lutsig is not the only verified tool available today. For C, there is Vericert [40]. For Bluespec (more precisely, Bluespec-like languages), there are the Fe-Si project [15] and the Kôika project [14]. (For Verilog, there is only Lutsig.) However, none of the publications for the available verified hardware-synthesis tools discuss constructs that, like `always_comb` blocks, require combining a hardware-oriented synthesis approach (like TVD) with a software-oriented verification approach (like VPD) as we do in this publication (although the Kôika project wanders into similar territory).

5.14 Conclusion

In this paper, we have added support for `always_comb` blocks to Lutsig. This takes Lutsig one step closer to become a usable Verilog synthesis tool. In extending Lutsig, we had to revisit the relationship between VPD and TVD. The discussion on VPD and TVD paves the way for further Lutsig extensions, such as BRAM support (mentioned briefly in Sec. 5.2.2).

Verilog support is not the only kind of Lutsig work needed. An important difference between the unverified synthesis tools available today and Lutsig is the amount of optimization performed by the tools. Since we have not added new optimization passes for this version of Lutsig, previously published data [66] on synthesis-result quality still holds. Clearly, more work lies ahead.

Acknowledgements We thank Magnus Myreen, Wolfgang Ahrendt, Koen Claessen, Warren A. Hunt, Jr., Thomas Melham, Adam Chlipala, and David J. Greaves for comments on draft version of this paper. The comments have substantially improved the presentation.

Bibliography

[1] *7 Series FPGAs Data Sheet: Overview (DS180, v2.6)*. Xilinx. 2018.

[2] Kenneth L. Albin, Bishop C. Brock, Warren A. Hunt, Jr., and Lawrence M. Smith. *Testing the FM9001 Microprocessor*. Tech. rep. 90. Computational Logic, Inc., 1995.

[3] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. "The Verisoft Approach to Systems Verification". In: *Verified Software: Theories, Tools, Experiments (VSTTE)*. 2008. DOI: 10.1007/978-3-540-87873-5_18.

[4] Roberto M. Amadio, Nicolas Ayache, Francois Bobot, Jaap P. Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. "Certified Complexity (CerCo)". In: *Foundational and Practical Aspects of Resource Analysis (FOPARA)*. 2014. DOI: 10.1007/978-3-319-12466-7_1.

[5] Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. "Position paper: The science of deep specification". In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 375.2104 (2017). DOI: 10.1098/rsta.2016.0331.

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

[7] Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, and Jan Vitek. "Formally Verified Speculation and Deoptimization in a JIT Compiler". In: *Proceedings of the ACM on Programming Languages* 5.POPL (2021). DOI: 10.1145/3434327.

[8] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. "Formal Verification of a Constant-Time Preserving C Compiler". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019). DOI: 10.1145/3371075.

[9] William R. Bevier, Warren A. Hunt, J Strother Moore, and William D. Young. "An approach to systems verification". In: *Journal of Automated Reasoning* 5.4 (1989). DOI: 10.1007/BF00243131.

[10] Sven Beyer, Christian Jacobi, Daniel Kroening, and Dirk Leinenbach. *Correct Hardware by Synthesis from PVS*. Tech. rep. 2002. URL: http://www-wjp.cs.uni-sb.de/publikationen/BJKL02.pdf.

[11] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. "Putting it all together – Formal verification of the VAMP". In: *International Journal on Software Tools for Technology Transfer (STTT)* 8.4 (2006). DOI: 10.1007/s10009-006-0204-6.

[12] Mohammad-Mahdi Bidmeshki and Yiorgos Makris. "VeriCoq: A Verilog-to-Coq converter for proof-carrying hardware automation". In: *IEEE International Symposium on Circuits and Systems (ISCAS)*. 2015. DOI: 10.1109/ISCAS.2015.7168562.

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

[14] Thomas Bourgeat, Clément Pit-Claudel, Adam 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. DOI: 10.1145/3385412.3385965.

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

[16] Bishop C. Brock and Warren A. Hunt. "The DUAL-EVAL Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor". In: *Formal Methods in System Design* 11.1 (1997). DOI: 10.1023/A:1008685826293.

[17] Adam Chlipala. "Mostly-automated Verification of Low-level Programs in Computational Separation Logic". In: *Programming Language Design and Implementation (PLDI)*. 2011. DOI: 10.1145/1993498.1993526.

[18] Adam Chlipala. “The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier”. In: *International Conference on Functional Programming (ICFP)*. 2013. DOI: 10.1145/2500365.2500592.

[19] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. “Kami: A Platform for High-level Parametric Hardware Specification and Its Modular Verification”. In: *Proceedings of the ACM on Programming Languages* 1.ICFP (2017). DOI: 10.1145/3110268.

[20] Avra Cohn. “The notion of proof in hardware verification”. In: *Journal of Automated Reasoning* 5.2 (1989). DOI: 10.1007/bf00243000.

[21] Niklas Eén and Niklas Sörensson. “An Extensible SAT-solver”. In: *Theory and Applications of Satisfiability Testing*. 2004. DOI: 10.1007/978-3-540-24605-3_37.

[22] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. “Integration Verification across Software and Hardware for a Simple Embedded System”. In: *Conference on Programming Language Design and Implementation (PLDI)*. 2021. DOI: 10.1145/3453483.3454065.

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

[24] Anthony Fox. “Directions in ISA Specification”. In: *Interactive Theorem Proving (ITP)*. 2012. DOI: 10.1007/978-3-642-32347-8_23.

[25] Anthony Fox, Magnus O. Myreen, Yong Kiam Tan, and Ramana Kumar. “Verified Compilation of CakeML to Multiple Machine-code Targets”. In: *Certified Programs and Proofs (CPP)*. 2017. DOI: 10.1145/3018610.3018621.

[26] Peter Gammie. “Synchronous Digital Circuits as Functional Programs”. In: *ACM Computing Surveys* 46.2 (2013). DOI: 10.1145/2543581.2543588.

[27] Herman Geuvers. “Proof assistants: History, ideas and future”. In: *Sadhana* 34.1 (2009). DOI: 10.1007/s12046-009-0001-5.

[28] Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. “Simulation and Formal Verification of x86 Machine-Code Programs that make System Calls”. In: *Formal Methods in Computer-Aided Design (FMCAD)*. 2014. DOI: 10.1109/FMCAD.2014.6987600.

[29] Alejandro Gómez-Londoño, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O. Myreen, and Yong Kiam Tan. "Do You Have Space for Dessert? A Verified Space Cost Semantics for CakeML Programs". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020). DOI: 10.1145/3428272.

[30] Michael Gordon. "The Semantic Challenge of Verilog HDL". In: *Symposium on Logic in Computer Science*. 1995. DOI: 10.1109/LICS.1995.523251.

[31] Michael J. C. Gordon. "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*. 1986.

[32] Mike Gordon. "From LCF to HOL: A Short History". In: *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[33] Mike J. C. Gordon. "Tactics for mechanized reasoning: a commentary on Milner (1984) 'The use of machines to assist in rigorous proof'". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 373.2039 (2015). DOI: 10.1098/rsta.2014.0234.

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

[35] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. "Verified Characteristic Formulae for CakeML". In: *European Symposium on Programming (ESOP)*. 2017. DOI: 10.1007/978-3-662-54434-1_22.

[36] Xiaolong Guo, Raj Gautam Dutta, Prabhat Mishra, and Yier Jin. "Automatic Code Converter Enhanced PCH Framework for SoC Trust Verification". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.12 (2017). DOI: 10.1109/TVLSI.2017.2751615.

[37] John Harrison, Josef Urban, and Freek Wiedijk. "History of Interactive Theorem Proving". In: *Handbook of the History of Logic, Volume 9: Computational Logic*. 2014. DOI: 10.1016/b978-0-444-51624-4.50004-6.

[38] Jifeng He, C. A. R. Hoare, Martin Fränzle, Markus Müller-Olm, Ernst-Rüdiger Olderog, Michael Schenke, Michael R. Hansen, Anders P. Ravn, and Hans Rischel. "Provably Correct Systems". In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*. 1994. DOI: 10.1007/3-540-58468-4_171.

[39] *Heartbleed*. URL: http://heartbleed.com.

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

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

[42] Mike Hinchey, Jonathan P. Bowen, and Ernst-Rüdiger Olderog, eds. *Provably Correct Systems*. Springer International Publishing, 2017. DOI: 10.1007/978-3-319-48628-4.

[43] Son Ho, Oskar Abrahamsson, Ramana Kumar, Magnus O. Myreen, Yong Kiam Tan, and Michael Norrish. "Proof-Producing Synthesis of CakeML with I/O and Local State from Monadic HOL Functions". In: *International Joint Conference on Automated Reasoning (IJCAR)*. 2018. DOI: 10.1007/978-3-319-94205-6_42.

[44] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Communications of the ACM* 12.10 (1969). DOI: 10.1145/363235.363259.

[45] Warren A. Hunt, Matt Kaufmann, J Strother Moore, and Anna Slobodova. "Industrial hardware and software verification with ACL2". In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 375.2104 (2017). DOI: 10.1098/rsta.2015.0399.

[46] Joe Hurd. "The OpenTheory Standard Theory Library". In: *NASA Formal Methods (NFM)*. 2011. DOI: 10.1007/978-3-642-20398-5_14.

[47] Mike Hutton, Vaughn Betz, and Jason Anderson. "FPGA Synthesis and Physical Design". In: *Electronic Design Automation for IC Implementation, Circuit Design, and Process Technology*. Ed. by Luciano Lavagno, Igor L. Markov, Grant Martin, and Louis K. Scheffer. CRC Press, 2016. Chap. 16.

[48] *IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*. IEEE Std 1800-2017. 2018. DOI: 10.1109/IEEESTD.2018.8299595.

[49] *IEEE Standard for Verilog Hardware Description Language*. IEEE Std 1364-2001. 2001. DOI: 10.1109/IEEESTD.2001.93352.

[50] *IEEE Standard for Verilog Hardware Description Language*. IEEE Std 1364-2005. 2006. DOI: 10.1109/IEEESTD.2006.99495.

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

[52] Juliano Iyoda. “Translating HOL functions to hardware”. PhD thesis. University of Cambridge, 2007.

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

[54] Jeffrey J. Joyce. “Totally verified systems: Linking verified software to verified hardware”. In: *Hardware Specification, Verification and Synthesis: Mathematical Aspects*. 1990. DOI: 10.1007/0-387-97226-9_29.

[55] Wilayat Khan, Alwen Tiu, and David Sanán. “VeriFormal: An Executable Formal Model of a Hardware Description Language”. In: *Singapore Cyber-Security RandD Conference (SG-CRC)*. 2017. DOI: 10.3233/978-1-61499-744-3-19.

[56] Gerwin Klein. “Operating system verification—An overview”. In: *Sadhana* 34.1 (2009). DOI: 10.1007/s12046-009-0002-4.

[57] Carlos Delgado Kloos and Peter T. Breuer, eds. *Formal Semantics for VHDL*. Springer US, 1995. DOI: 10.1007/978-1-4615-2237-9.

[58] Dirk Koch, Frank Hannig, and Daniel Ziener, eds. *FPGAs for Software Programmers*. Springer International Publishing, 2016. DOI: 10.1007/978-3-319-26408-0.

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

[60] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. “CakeML: A Verified Implementation of ML”. In: *Principles of Programming Languages (POPL)*. 2014. DOI: 10.1145/2535838.2535841.

[61] Miriam Leeser, Richard Chapman, Mark Aagaard, Mark Linderman, and Stephan Meier. “High Level Synthesis and Generating FPGAs with the BEDROC System”. In: *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology* 6.2 (1993). DOI: 10.1007/bf01607881.

[62] Dirk Leinenbach and Elena Petrova. “Pervasive Compiler Verification – From Verified Programs to Verified Systems”. In: *Electronic Notes in Theoretical Computer Science* 217 (2008). DOI: 10.1016/j.entcs.2008.06.040.

[63] Xavier Leroy. “A Formally Verified Compiler Back-end”. In: *Journal of Automated Reasoning* 43.4 (2009). DOI: 10.1007/s10817-009-9155-4.

[64] Xavier Leroy. “Formal Verification of a Realistic Compiler”. In: *Communications of the ACM (CACM)* 52.7 (2009). DOI: 10.1145/1538788.1538814.

[65] Arm Limited. *AMBA AXI and ACE Protocol Specification*. Tech. rep. ARM IHI 0022F.b. 2017.

[66] Andreas Lööw. “Lutsig: A Verified Verilog Compiler for Verified Circuit Development”. In: *Conference on Certified Programs and Proofs (CPP)*. 2021. DOI: 10.1145/3437992.3439916.

[67] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. “Verified Compilation on a Verified Processor”. In: *Conference on Programming Language Design and Implementation (PLDI)*. 2019. DOI: 10.1145/3314221.3314622.

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

[69] Donald MacKenzie. *Mechanizing Proof: Computing, Risk, and Trust*. MIT Press, 2001.

[70] Donald MacKenzie. “The fangs of the VIPER”. In: *Nature* 352.6335 (1991). DOI: 10.1038/352467a0.

[71] Thomas F. Melham. *Higher Order Logic and Hardware Verification*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1993.

[72] *Meltdown and Spectre*. URL: https://meltdownattack.com.

[73] Patrick Meredith, Michael Katelman, José Meseguer, and Grigore Roşu. “A formal executable semantics of Verilog”. In: *Formal Methods and Models for Codesign (MEMOCODE)*. 2010. DOI: 10.1109/MEMCOD.2010.5558634.

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

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

[76] Magnus O. Myreen. “A Minimalistic Verified Bootstrapped Compiler (Proof Pearl)”. In: *Conference on Certified Programs and Proofs (CPP)*. 2021. DOI: 10.1145/3437992.3439915.

[77] Magnus O. Myreen. “Formal verification of machine-code programs”. PhD thesis. University of Cambridge, 2009.

[78] Magnus O. Myreen. “Verified Just-in-Time Compiler on X86”. In: *Symposium on Principles of Programming Languages (POPL)*. 2010. DOI: 10.1145/1706299.1706313.

[79] Magnus O. Myreen and Scott Owens. “Proof-producing translation of higher-order logic into pure and stateful ML”. In: *Journal of Functional Programming* 24.2-3 (2014). DOI: 10.1017/S0956796813000282.

[80] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. “A Survey and Evaluation of FPGA High-Level Synthesis Tools”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2016). DOI: 10.1109/TCAD.2015.2513673.

[81] Zhaozhong Ni and Zhong Shao. “Certified Assembly Programming with Embedded Code Pointers”. In: *Principles of Programming Languages (POPL)*. 2006. DOI: 10.1145/1111037.1111066.

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

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

[84] Zoe Paraskevopoulou and Andrew W. Appel. “Closure Conversion is Safe for Space”. In: *Proceedings of the ACM on Programming Languages* 3.ICFP (2019). DOI: 10.1145/3341687.

[85] Daniel Patterson and Amal Ahmed. “The Next 700 Compiler Correctness Theorems (Functional Pearl)”. In: *Proceedings of the ACM on Programming Languages* 3.ICFP (2019). DOI: 10.1145/3341689.

[86] Lawrence C. Paulson, Tobias Nipkow, and Makarius Wenzel. “From LCF to Isabelle/HOL”. In: *Formal Aspects of Computing* 31.6 (2019). DOI: 10.1007/s00165-019-00492-1.

[87] João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. "Pi-Ware: Hardware Description and Verification in Agda". In: *Types for Proofs and Programs (TYPES 2015)*. 2018. DOI: 10.4230/LIPIcs.TYPES.2015.9.

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

[89] Robert Pollack. "How to Believe a Machine-Checked Proof". In: *Twenty Five Years of Constructive Type Theory*. 1998.

[90] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. "QED at Large: A Survey of Engineering of Formally Verified Software". In: *Foundations and Trends in Programming Languages* 5.2-3 (2019). DOI: 10.1561/2500000045.

[91] Adrian Sampson. *FPGAs Have the Wrong Abstraction*. 2019. URL: https://www.cs.cornell.edu/~asampson/blog/fpgaabstraction.html.

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

[93] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. "CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency". In: *Journal of the ACM* 60.3 (2013). DOI: 10.1145/2487241.2487248.

[94] Peter Sewell. *Rigorous Engineering of Mainstream Systems web page*. URL: https://www.cl.cam.ac.uk/~pes20/rems/.

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

[96] J Strother Moore. "A Grand Challenge Proposal for Formal Methods: A Verified Stack". In: *Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST*. 2003. DOI: 10.1007/978-3-540-40007-3_11.

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

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

[99] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. “A new verified compiler backend for CakeML”. In: *International Conference on Functional Programming (ICFP)*. 2016. DOI: 10.1145/2951913.2951924.

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

[101] Chuck Thacker. “A Tiny Computer”. Unpublished memo, available online. 2007.

[102] Lenny Truong and Pat Hanrahan. “A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity”. In: *Summit on Advances in Programming Languages (SNAPL)*. 2019. DOI: 10.4230/LIPIcs.SNAPL.2019.7.

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

[104] Sergey Tverdyshev. “Formal Verification of Gate-Level Computer Systems”. PhD thesis. Saarland University, 2009.

[105] *Verilog Register Transfer Level Synthesis*. IEEE Std 62142-2005. 2005. DOI: 10.1109/IEEESTD.2005.339572.

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

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

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

[109] Tjark Weber and Hasan Amjad. “Efficiently checking propositional refutations in HOL theorem provers”. In: *Journal of Applied Logic* 7.1 (2009). DOI: 10.1016/j.jal.2007.07.003.

[110] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. 4th ed. Pearson, 2011.

[111] Freek Wiedijk. “Pollack-inconsistency”. In: *Electronic Notes in Theoretical Computer Science* 285 (2012). Proceedings of the International Workshop On User Interfaces for Theorem Provers. DOI: 10.1016/j.entcs.2012.06.008.

[112] Wikipedia contributors. *Pentium FDIV bug – Wikipedia*. URL: https://en.wikipedia.org/wiki/Pentium_FDIV_bug.

[113] Clifford Wolf. *Yosys Open SYnthesis Suite*. URL: http://www.clifford.at/yosys.

[114] Zhiru Zhang, Hongbo Rong, and Yu Wang. “Introduction of Special Issue on FPGA-Based Computing”. In: *IEEE Circuits and Systems Magazine* 21.2 (2021). DOI: 10.1109/mcas.2021.3071606.

[115] Huibiao Zhu, Jifeng He, and Jonathan P. Bowen. “From algebraic semantics to denotational semantics for Verilog”. In: *Innovations in Systems and Software Engineering* 4.4 (2008). DOI: 10.1007/s11334-008-0069-9.