PRINCETON UNIVERSITY

Amelia Dobis

COS 597E Final Project

Modularizing Formal Verification in High-Level Hardware Languages

COS 597E Advanced Topics in Computer Science: Programming Languages (for distributed systems), Distributed Systems

December 13th 2024

Contents

1 Introduction

The ever-increasing demand for performance in modern systems has led to a rise in heterogeneous architectures driven by domain specific accelerators. This need for the fast production of hardware acceleration in various domains has highlighted the gap in efficiency between software and hardware development tools. As a solution, there have been many attempts to modernize hardware description languages [3], either by embedding them into software languages as eDSLs [2, 21, 25], by using software to directly generate hardware descriptions through High-Level Synthesis (HLS) [6, 20, 22, 23], or by introducing new paradigms that are higher-level than traditional register transfer level languages [14,19]. With each new solution comes a new tool-chain, language, and verification system, leading to large, disconnected array of tools that are built from the ground-up for every language. This lack of a shared knowledge-base has greatly reduced the efficiency and adoption of these new design tools. As a solution to this, CIRCT [12] was proposed as a unifying compiler infrastructure and core representation for hardware description languages. This allowed for all optimizations and back-ends to be created exactly once for all possible hardware languages, and many existing languages used widely in industry are now based on CIRCT [2, 5, 7, 8, 17, 19, 20, 22, 23, 27].

While this effort greatly unified the design capabilities of hardware description languages, verification remained primarily done using a disconnected array of mostly proprietary SystemVerilog [1]-based tools, rather than tools built for the source languages. This means that verification is now the main bottleneck in hardware development flows. Formal Verification is a popular method used to verify digital designs, and while its complexity is much greater than dynamic simulation-based methods [11, 13, 15], the correctness guarantees it can produce are often required before the expensive manufacturing process can take place. Typically, formal verification is done using complicated closed-source commercial tools such as Jasper Gold or Synopsis VC Formal, and open-source solutions like Yosys [26] only support System-Verilog as a front-end. Additionally, the scalability of these tools is limited, as abstractions such as modularity are not preserved during verification, leading to a duplication of effort. We believe that we should take inspiration from efforts to unify design capabilities and create a unified modular representation of verification built into the hardware compilation directly. We do so by proposing a unified interface for formal verification as well as a modular specification interface, inspired by Dijkstra's guarded commands [9], called hardware contracts as part of CIRCT's core dialects. These new constructs, alongside our recent addition of formal verification back-ends for CIRCT [10], enable scalable formal verification for all of CIRCT's front-ends *for free.*

This work introduces the concept of hardware contracts and formal test interfaces as parts of CIRCT's core dialects. These abstractions allow for modularity to be retained during verification, thus speeding up the verification process. We present a formal verification back-end introduced to CIRCT and describe how it was updated to support hardware contracts. An initial evaluation is shown using a simple example implemented in Chisel, where a single module with two instances is verified with and without our contracts, achieving a speedup of 1.57x on even a simple example. More elaborate evaluations are left for future work due to time constraints.

2 Overview

To demonstrate our solution, we implement a skeleton of a modular design and specify it using hardware contracts, shown in Figure 1. Here two new concepts are introduced. First a contract region is added to the implementation of module A, this is where we will specify the preconditions

```
class A extends Module {
    val in = IO(Input(UInt(32.W)))
    val out = IO(Output(UInt(32.W)))
    contract {
        requires in >= 0.U
        ensures out === in + 42.U
    }
    // ... Module body ...
}
```

```
// Formal test harness
class B extends Module with Formal {
    val a1 = Instance(A)
    val a2 = Instance(B)
    a1.in := 0.U
    a2.in := 1.U
    assert(a1.out + a2.out === 85.U)
}
```

Figure 1: Modular formal test bench implemented in Chisel using our formal interface. Module A uses a contract that contains pre-conditions that the module's inputs are expected to follow (`requires`) and post-conditions that the module's outputs should guarantee (`ensures`). Module B is a formal test bench that checks the correct interaction between two instances of module A.

that the module's inputs are expected to follow, marked using the `requires` keyword, and the postconditions that the module's outputs are expected to guarantee, marked using the `ensures` keyword. Second, a formal test harness is introduced, implemented as a Scala trait [24], which is used to signify that the body of module B is simply used to define a test case that should be verified formally using bounded model checking [4], which is the default verification method when no attributes are explicitly given. This annotated design and test are then compiled using CIRCT and first lowered to its core dialects, shown in Figure 2. We can observe the two new constructs introduced to our core dialects: `verif.contract`, which encodes a Hoare triple wrapping an arbitrary operation, or set of operations, and `verif.formal`, which encodes our formal test bench. Our core dialects are a generic form that contain information that can be used for both design and verification. Once we have decided that we want to target a verification back-end, we can lower this generic form of the design into one that specifically encodes our separate bounded model checking problems, which is shown in Figure 3. In this final version, we have reduced our module to a verification problem that checks whether we can prove our contract conditions using the module's body. In our test harness in module B, all of the instances of module A have been replaced with the body of the contract of A. This allows us to only verify module A once, and then use the proven contract as an abstraction of the module itself, allowing us to greatly reduce the effort when verifying instances of a module. Additionally, having formal test constructs, such as symbolic values, contracts and test harnesses directly integrates into our compiler allow for them to easily be accessed in front-end languages, greatly simplifying the process of formally verifying a design. Finally, our bounded model checking back-end allows for CIRCT to directly emit btor2 [18], a format supported by various open-source model checkers, as is shown in Figure 4

3 Background

3.1 Formal Verification of Hardware

This work focuses on integrated creating a unified scalable formal verification framework for high-level hardware languages, by integrating the constructs into the same compiler used to handle the design constructs. Most of the introduced constructs focus on the main formal verification method used in hardware, which is bounded model checking (BMC) [4].

The goal of BMC is to create a first-order logic representation of the design under verification (DUV)

```
hw.module @A(in %clk: !seq.clock, in %reset: i1, in %in : i32, out out : i32) {
    ; ... module body defining %out ...
    %out_ = verif.contract %out : i32 {
        %c0_32 = hw.constant 0 : i32
        %prec = comb.icmp bin ugte %in, %c0_32 : i1
        verif.requires %prec : i1
        %c42_32 = hw.constant 42 : i32
        %in_42 = comb.add bin %in, %c42_32 : i32
        %post = comb.icmp bin eq %in_42, %out : i1
        verif.ensures %post : i1
    }
    hw.output %out_ : i32
}
verif.formal @B {bound = 500, method = BMC} {
    %a1.in = verif.symbolic_value : i32
    %a2.in = verif.symbolic_value : i32
    %c0_32 = hw.constant 0 : i32
    %c1_32 = hw.constant 1 : i32
    %set_a1 = comb.icmp bin eq %c0_32, %a1.in : i1
    %set_a2 = comb.icmp bin eq %c1_32, %a2.in : i1
    verif.assume %set_a1 : i1
    verif.assume %set_a2 : i1
    %a1.out = hw.instance "a1" @A(in: %a1.in) -> (out: i32)
    %a2.out = hw.instance "a2" @A(in: %a2.in) -> (out: i32)
    %add_out = comb.add bin %a1.out, %a2.out : i32
    %c85_32 = hw.constant 85 : i32
    %cond = comb.icmp bin eq %add_out, %c85_32 : i1
    verif.assert %cond
}
```

Figure 2: Result of compiling the input Chisel design into the CIRCT core dialects. Module A is first compiled to a regular `hw.module`, where the contract is lowered to our new `verif.contract` operation, which encodes an arbitrary hoare triple. Module B is marked in Chisel as a formal test harness and is therefore directly lowered to a `verif.formal` operation , which is our unified interface for formal verification, using default attributes such as "bound" or "method" which refer to the bound of the bounded model check and the verification method.

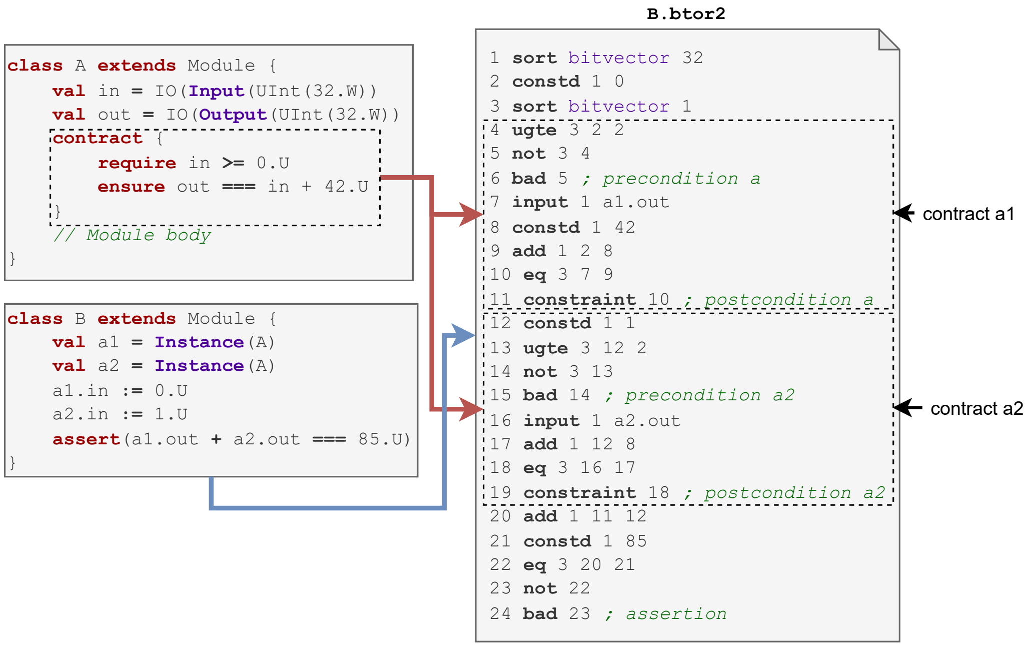
and prove the unreachability of a bad state, which violates a given specification, using a sequence of SMT queries. Specifically, a sequential design, i.e. a design that contains stateful elements that evolve over various cycles, is converted into a state transition system. This encodes the design as a series of states, which represent a set of possible values that each stateful element in the design can have, alongside transitions, which encode the conditions under which a state can advance to another. Figure 5 shows an example of a simple counter circuit encoded as a state transition system. Each transition in a state transition system implicitly encodes a clock tick. In BMC, the bound refers to the number of clock cycles that the state transition system being verified will encode, e.g. a bound of 500 means that we will "unroll" our counter for 500 cycles and check the unreachability of a specified illegal state within that scope. This is the main method used to formally verify hardware, however, it does not allow for the encoding of modularity, an abstraction required to implement

```
verif.formal @A {bound = 500, method = BMC} {
    %in = verif.symbolic_input : i32
    ; ... module body defining %out ...
    %c0_32 = hw.constant 0 : i32
    %prec = comb.icmp bin ugte %in, %c0_32 : i1
    verif.assume %prec : i1
    %c42_32 = hw.constant 42 : i32
    %in_42 = comb.add bin %in, %c42_32 : i32
    %post = comb.icmp bin eq %in_42, %out : i1
    verif.assert %post : i1
}
verif.formal @B {bound = 500, method = BMC} {
    %a1.in = hw.constant 0 : i32 ; reduced via constant prop.
    %a2.in = hw.constant 1 : i32
    ; contract a1
    %a1.out = symbolic_value : i32
    %c0_32 = hw.constant 0 : i32
    %prec_1 = comb.icmp bin ugte %in, %c0_32 : i1
    verif.assert %prec_1 : i1
    %c42_32 = hw.constant 42 : i32
    %in_42 = comb.add bin %in, %c42_32 : i32
    %post_1 = comb.icmp bin eq %in_42, %a1.out : i1
    verif.assume %post_1 : i1
    ; contract a2
    %a2.out = symbolic_value : i32
    %c0_32 = hw.constant 0 : i32
    %prec_2 = comb.icmp bin ugte %in, %c0_32 : i1
    verif.assert %prec_2 : i1
    %c42_32 = hw.constant 42 : i32
    %in_42 = comb.add bin %in, %c42_32 : i32
    %post_2 = comb.icmp bin eq %in_42, %a2.out : i1
    verif.assume %post_2 : i1
    ; test
    %add_out = comb.add bin %a1.out, %a2.out : i32
    %c85_32 = hw.constant 85 : i32
    %cond = comb.icmp bin eq %add_out, %c85_32 : i1
    verif.assert %cond
}
```

Figure 3: Result of lowering the core dialect representation into a form that is suitable to be emitted as a set of btor2 files. Here two formal tests were generated, one for the module, where we check that the body can be used to verify the contract, and one for the formal test harness, where instances of module A are replaced with the body of the contract.

large designs. This is because modules in a designs are inlined during the conversion to a state transition system, meaning that we need to re-verify a module for every one of its instances.

Our proposed back-ends will focus on targeting bounded model checking, and our proposed contract



```
class A extends Module {
    val in = IO(Input(UInt(32.W)))
    val out = IO(Output(UInt(32.W)))
    contract {
        require in >= 0.U
        ensure out === in + 42.U
    }
    // Module body
}
```

```
class B extends Module {
    val a1 = Instance(A)
    val a2 = Instance(A)
    a1.in := 0.U
    a2.in := 1.U
    assert(a1.out + a2.out === 85.U)
}
```

B.btor2

```
1 sort bitvector 32
2 constd 1 0
3 sort bitvector 1
4 ugte 3 2 2
5 not 3 4
6 bad 5 ; precondition a
7 input 1 a1.out
8 constd 1 42
9 add 1 2 8
10 eq 3 7 9
11 constraint 10 ; postcondition a
12 constd 1 1
13 ugte 3 12 2
14 not 3 13
15 bad 14 ; precondition a2
16 input 1 a2.out
17 add 1 12 8
18 eq 3 16 17
19 constraint 18 ; postcondition a2
20 add 1 11 12
21 constd 1 85
22 eq 3 20 21
23 not 22
24 bad 23 ; assertion
```

Figure 4: High-Level overview of the start and end points of our new formal compilation pipeline. This allows formal verification constructs such as symbolic values, formal test harnesses and contracts to be treated as first class constructs in our hardware langauges.

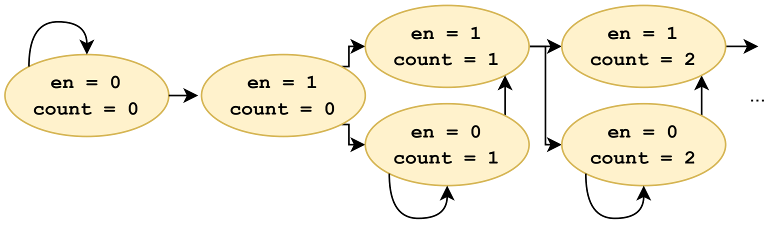


Figure 5: Example of a counter, enabled by the `en` signal, encoded as a state transition system.

construct will allow us to verify each module exactly once.

3.2 CIRCT

Our solution extends the core dialects of CIRCT to introduce a core representation for verification alongside the existing core representation for design.

CIRCT [1] is an MLIR [16]-based hardware compiler infrastructure organized as a set of dataflow dialects, i.e. interoperable domain-specific Intermediate Representations (IRs), passes to transform those dialects, and back-ends to emit optimized SystemVerilog. Figure 6 shows the structure of CIRCT and the relationships between its various dialects. The core dialects are at the center of the entire compilation pipeline and form a generalized representation of hardware that all front-ends target when using CIRCT as their back-end.

[1] https://github.com/llvm/circt

CIRCT unifies hardware compilation into a single tool that supports various paradigms and is the back-end used for several high-level hardware languages, such as Chisel [2], Kanagawa [19], magma [21], and many more.

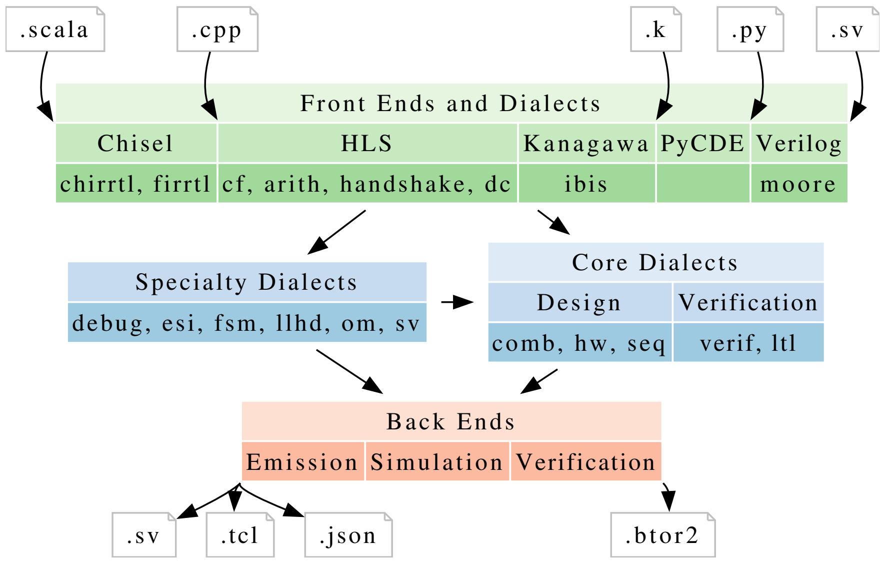


Figure 6: Overview of the CIRCT dialects and the relations between them. In this work, we introduce the core verification dialects, as well as the verification backend.

4 Hardware Contracts

The first contribution of this work is the *hardware contract* abstraction. The goal is to maintain the modularity expressed throughout the design during verification.

Problem Current formal verification tools ignore modularity by inlining module bodies in place of their instances during verification. This is done in order to preserve correctness, however it requires re-verifying modules for each of their instances and leads to a modular design being converted into one big, monolithic bounded model checking problem.

Overview of our solution To prevent this problem, we take inspiration from Dijkstra's Guarded Commands [9] and introduce hardware contracts as a construct that allows the user to create arbitrary Hoare triples in a design by wrapping statements with pre-conditions and post-conditions. These contracts can then be used to create and abstraction of a module, allowing for scalable verification of large modular designs. This also allows us to separate our verification task into multiple smaller problems that can be checked in parallel. Figure 7 shows an abstract overview of our solution.

4.1 Verification using Hardware Contracts

In order to verify a design specified using hardware contracts, there are two types of verification contexts that need to be considered: verifying a module and verifying instances of a module. Once these have been verified, we know that our design satisfies its specification.

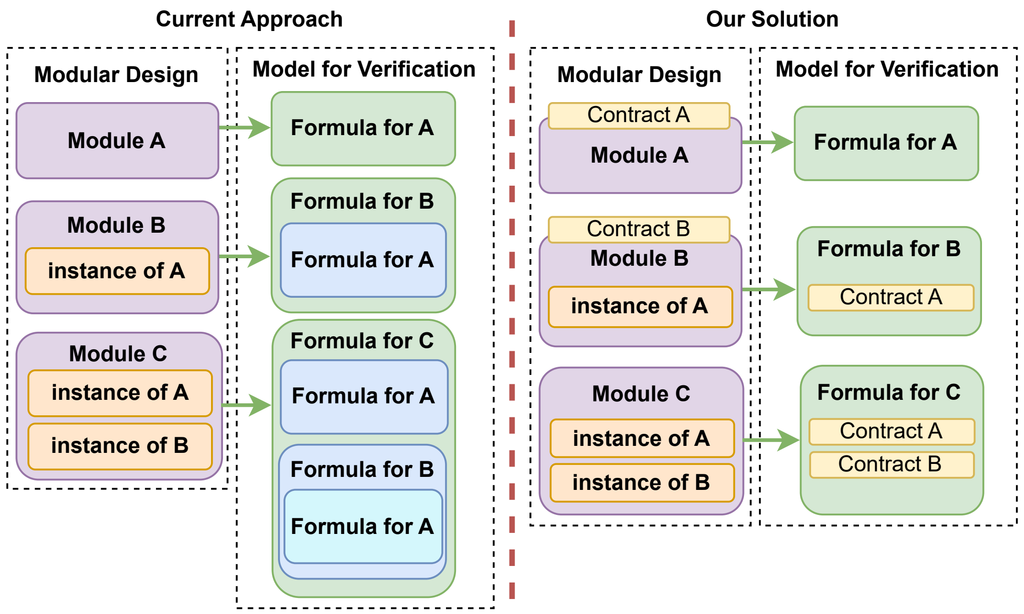


Figure 7: Abstract overview of how hardware contracts are used to simplify verification in highly modular designs.

Verifying a Module The goal when verifying a module is to check that its body satisfies its contract, or more specifically:

Assuming the preconditions, does the body satisfy the postconditions?

To verify this we must replace our module with a formal test by:

- Generating a symbolic value to represent each input.
- Assuming all preconditions on these symbolic inputs.
- Replacing inputs with their symbolic counterparts in the body.
- Asserting all of the postconditions.

Module A in Figure 3 shows an example of a generated module formal test. If the generated transition system is unsatisfiable, then we know that our module satisfies its contract.

Verifying an instance of a module The goal when verifying an instance is simply to check that the inputs connected to the instance satisfy the instantiated module's preconditions. This is checked by asserting the preconditions on the signals driving the inputs. If this holds, then we know that our postconditions hold, as we have already proven that in our module test. We can thus generate symbolic values representing the outputs of our instantiated module and assume the postconditions on them. This models the behavior of the module without require an expensive re-verification of the entire module body. Given that our module test and our instance tests are two independent verification tasks, we can safely verify them in parallel and consider the design to be verified if all of the parallel tasks are successful. Module B of Figure 3 shows an example of this replacement for both instances of module A.

4.2 Implementing hardware contracts in CIRCT

This solution is implemented by introducing five new operations to the `verif` dialect in CIRCT, as well as a `PrepareForFormal` pass that performs the contract replacement.

- `verif.formal @Sym {attr-dict} {<body>}`: declares a formal test block that can contain constructs specific to verification. Each formal test will generate its own btor2 file.
- `%val = symbolic_value : <type>`: declares a free variable that will be interpreted symbolically during verification. This is used to encode inputs and results in contracts, and it can only be used in a formal test.
- `%<outputs> = verif.contract %s {<body>}`: defines a hardware contract around all of the operations reachable by `%s` (in the dataflow graph).
- `verif.requires %<precondition> : <type>`: declares a precondtion.
- `verif.ensures %<postcondition> : <type>`: declares a postcondition.

5 Extending the Formal Backend

The second task in this work is to extend the btor2 backend introduced in a previous work [10] to support hardware contracts.

The existing backend originally only supported synchronous single clock designs and handled modules by inlining their bodies in place of every instance. We extend this in various ways such that it can emit a directory of btor2 files and support multi-clock designs. This is done by:

- Introducing an infinitesimal clock and encoding all clocks as registers updated conditionally on every infinitesimal clock. This way we can reduce a multi-clock design to a single clock without modifying its behavior.
- Running the btor2 emission pass in parallel on each `verif.formal` rather than on an `mlir.module` as it currently does. This also requires creating a file for each instance of this pass, rather than at the beginning of the compilation pipeline as was originally done.
- Adding support for the new symbolic value operations introduced in the previous section.

Once these modifications were made, CIRCT was able to produce modularized btor2 files that can then all be verified in parallel to efficiently verify a large design. This is all integrated into the test-discovery functionalities of the `circt-test` tool.

6 Initial Evaluation

We perform an initial evaluation of our proposed hardware contracts by running the example from Figure 1 through CIRCT, where the implementation of A is a convoluted counter that is incremented until 42 before yielded the result using a ready-valid interface. The output btor2 files are then verified in parallel using **btormc** [18]. Table 1 shows the result. To guarantee the accuracy of the results, parsing time for all files are evaluated using btormc's verbose mode and then removed from the final wall time measurement. This speedup is due to contracts **enabling solver parallelism** and **simplifying verification**. We believe that the speedup will scale with the size of the design. Evaluation on larger designs is left for future work as it first requires adapting an existing large

without contracts	**with contracts**	**speedup**
0.011s	0.007s	1.57x

Table 1: Wall-time average over 100 runs (in seconds) of verifying Figure 1 with the resulting speedup obtained from using hardware contracts.

design to add contract annotations to each module, which did not fit under this project's time constraints.

7 Conclusions

In this work we introduced a unified representation for scalable formal verification as part of CIRCT's core dialects. We improved verification scalability by introducing hardware contracts as an abstraction that enables maintaining modularity during verification. This not only simplifies the overall verification task, but it also enables parallelization across the verification of multiple modules. Additionally, by introducing these contracts, as well as a unified formal test interface, as part of CIRCT's core dialects, we allow all of CIRCT's frontend languages to gain access to scalable formal verification *for free*. In an initial evaluation we show that these abstractions allow for fully open-source tools to gain significant verification speedups even on tiny designs. Our hope is that hardware contracts will make formal verification more accessible and make open-source model checkers and high-level hardware languages viable tools for solving large-scale verification tasks.

References

[1] Ieee standard for systemverilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)*, pages 1–1354, 2024.

[2] 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 *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 1216–1225, New York, NY, USA, 2012. Association for Computing Machinery.

[3] Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Ross Daly, Caleb Donovick, David Durst, Kayvon Fatahalian, Kathleen Feng, Pat Hanrahan, Teguh Hofstee, Mark Horowitz, Dillon Huff, Fredrik Kjolstad, Taeyoung Kong, Qiaoyi Liu, Makai Mann, Jackson Melchert, Ankita Nayak, Aina Niemetz, Gedeon Nyengele, Priyanka Raina, Stephen Richardson, Raj Setaluri, Jeff Setter, Kavya Sreedhar, Maxwell Strange, James Thomas, Christopher Torng, Leonard Truong, Nestan Tsiskaridze, and Keyi Zhang. Creating an agile hardware design flow. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, DAC '20. IEEE Press, 2020.

[4] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, Yunshan Zhu, et al. Bounded Model Checking. *Advances in computers*, 58(11):117–148, 2003.

[5] chipsalliance. Flexible intermediate representation for rtl.

[6] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. Transformations of high-level synthesis codes for high-performance computing. *IEEE Trans. Parallel Distrib. Syst.*, 32(5):1014–1029, May 2021.

[7] John Demme. Elastic Silicon Interconnects: Abstracting Communication in Accelerator Design. 2021.

[8] John Demme. The ESI System Construction Compiler in 2024. 2024.

[9] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[10] Amelia Dobis. Formal verification of hardware using mlir. Master thesis, ETH Zurich, Zurich, 2024.

[11] Amelia Dobis, Kevin Laeufer, Hans Jakob Damsgaard, Tjark Petersen, Kasper Juul Hesse Rasmussen, Enrico Tolotto, Simon Thye Andersen, Richard Lin, and Martin Schoeberl. Verification of chisel hardware designs with chiselverify. *Microprocessors and Microsystems*, 96:104737, 2023.

[12] Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth, George Leontiev, Fabian Schuiki, Ram Sunder, et al. Mlir as hardware compiler infrastructure.

[13] Martin Erhart, Fabian Schuiki, Zachary Yedidia, Bea Healy, and Tobias Grosser. Arcilator: Fast and cycle-accurate hardware simulation in CIRCT. `https://llvm.org/devmtg/2023-10/slides/techtalks/Erhart-Arcilator-FastAndCycleAccurateHardwareSimulationInCIRCT.pdf`.

[14] Google. Xls: Accelerated hw synthesis. `https://github.com/google/xls`.

[15] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 29–42. IEEE Press, 2018.

[16] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, page 2–14. IEEE Press, 2021.

[17] Kingshuk Majumder and Uday Bondhugula. Hir: An mlir-based intermediate representation for hardware accelerator description. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ASPLOS '23, page 189–201, New York, NY, USA, 2024. Association for Computing Machinery.

[18] Aina Niemetz, Mathias Preiner, Claire Wolf, and Armin Biere. Btor2 , btormc and boolector 3.0. In *International Conference on Computer Aided Verification*, 2018.

[19] Blake Pelton, Adam Sapek, Ken Eguro, Daniel Lo, Alessandro Forin, Matt Humphrey, Jinwen Xi, David Cox, Rajas Karandikar, Johannes de Fine Licht, Evgeny Babin, Adrian Caulfield, and Doug Burger. Wavefront threading enables effective high-level synthesis. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.

[20] Morten Borup Petersen. A dynamically scheduled hls flow in mlir, jan 2022.

[21] Lenny Truong and Pat Hanrahan. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, 2019.

[22] Christian Ulmann. Multi-level rewriting for stream processing to rtl compilation. Master thesis, ETH Zurich, Zurich, 2022.

[23] Mike Urbach and Morten B Petersen. HLS from PyTorch to System Verilog with MLIR and CIRCT. 2022.

[24] Bill Venners, Lex Spoon, and Martin Odersky. *Programming in Scala, 3rd Edition.* Artima Inc, 2016.

[25] whitequark. amaranth. `https://github.com/amaranth-lang/amaranth`, 2022.

[26] Claire Wolf. Yosys open synthesis suite. `https://yosyshq.net/yosys/`.

[27] Ayaka Yorihiro, Griffin Berlstein, Kevin Laeufer, and Adrian Sampson. A firrtl backend for the calyx high-level accelerator compilation infrastructure. 2024.