Hardware logical equivalence checking with CIRCT

Dragoș Cristian Lizan [1] Fabian Schuiki [2] Martin Erhart [2, 3] Jonathan Balkind [4, 5]

[1]University of Padua [2]SiFive [3]ETH Zürich [4]Free and Open Source Silicon Foundation [5]UC Santa Barbara

The proposal

LLHD[6] is a novel multi-level Intermediate Representation for Hardware Description Languages which can be employed for all aspects of modern circuit design flows by covering both higher-level, behavioral modeling and lower-level, structural modeling and also netlist synthesis.

In particular, it could sustain a habitat of open-source design automation and verification tools by simplifying their development, as opposed to today's landscape of proprietary tools implementing their own disjoint and incompatible IRs.

Originally, this project aimed at demonstrating said verification prowess by implementing a basic LEC (Logical Equivalence Checker) for combinatorial LLHD designs. Specifically, by translating two circuits into their fundamental boolean equations and formally proving or disproving their equivalence through the aid of an existing SMT solver.

Since LLHD has been merged into the CIRCT project, a larger joint effort to develop Circuit IR Compilers and Tools by applying the MLIR[4] and LLVM[3] development methodology and best practices to the domain of hardware design tools, the project's scope changed to developing a LEC tool for the standard CIRCT dialects.

Use cases

This tool can prove to be immensely useful, potentially saving both time and money, or at least giving peace of mind by providing formal proofs to the categories of users who might employ it:

- **Designers** could perform a manual tweak over a circuit then compare it with the previous version to discover newly introduced bugs, or lack thereof.
- **CIRCT developers** could rapidly test a new transformation pass over an extensive design covering many of the possible corner cases.

Architecture

The tool accepts up to two input circuit descriptions; the parsed intermediate representation then gets visited by a pass which will export the relevant logical constraints to a circuit representation. These representations act as an abstraction over the instanced context of a SMT solver backend, in our case Z3[2], which at end gets tasked with the equivalence problem.

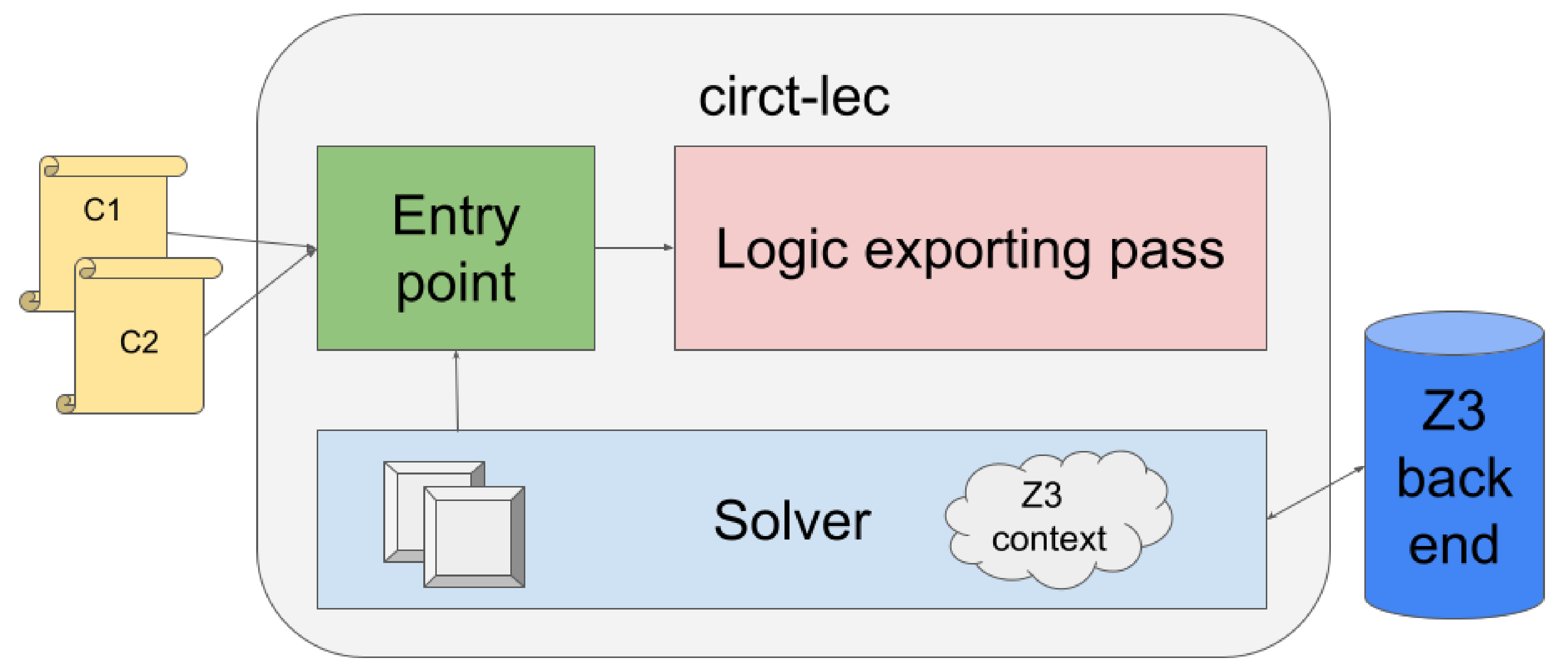


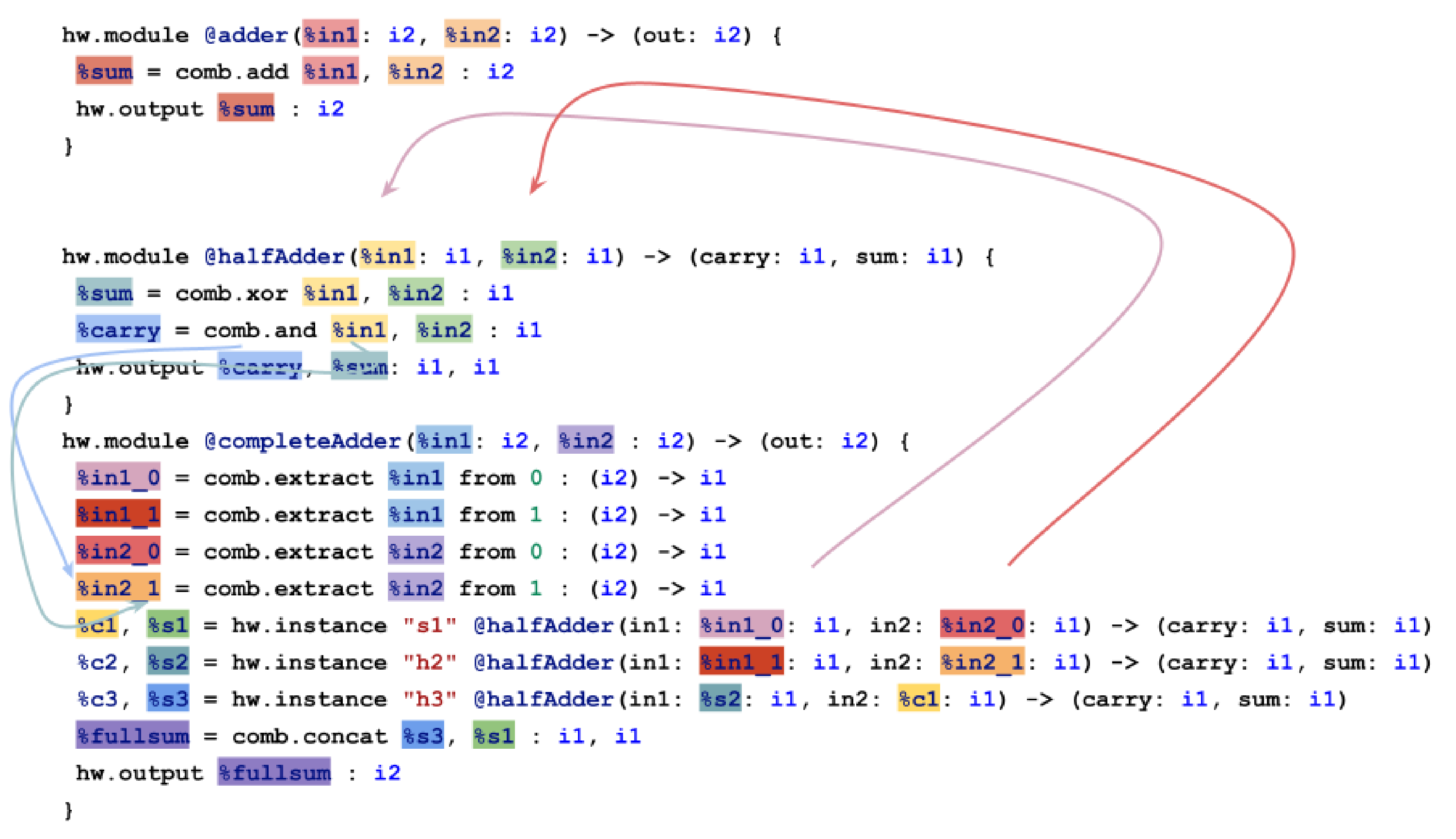
Figure 1. Architectural overview of circt-lec

Logical equivalence checking in action

Checking for the equivalence of two modules is performed by constraining the arbitrary inputs to be the same and attesting it is unsatisfiable for their outputs to differ.

The figure below highlights the constraints which are placed during value assignment and module instancing.

In particular, LLVM's employment of SSA[5] (Single Static Assignment) vastly simplifies this operation.



```
hw.module @adder(%in1: i2, %in2: i2) -> (out: i2) {
 %sum = comb.add %in1, %in2 : i2
 hw.output %sum : i2
}

hw.module @halfAdder(%in1: i1, %in2: i1) -> (carry: i1, sum: i1) {
 %sum = comb.xor %in1, %in2 : i1
 %carry = comb.and %in1, %in2 : i1
 hw.output %carry, %sum: i1, i1
}
hw.module @completeAdder(%in1: i2, %in2 : i2) -> (out: i2) {
 %in1_0 = comb.extract %in1 from 0 : (i2) -> i1
 %in1_1 = comb.extract %in1 from 1 : (i2) -> i1
 %in2_0 = comb.extract %in2 from 0 : (i2) -> i1
 %in2_1 = comb.extract %in2 from 1 : (i2) -> i1
 %c1, %s1 = hw.instance "s1" @halfAdder(in1: %in1_0: i1, in2: %in2_0: i1) -> (carry: i1, sum: i1)
 %c2, %s2 = hw.instance "h2" @halfAdder(in1: %in1_1: i1, in2: %in2_1: i1) -> (carry: i1, sum: i1)
 %c3, %s3 = hw.instance "h3" @halfAdder(in1: %s2: i1, in2: %c1: i1) -> (carry: i1, sum: i1)
 %fullsum = comb.concat %s3, %s1 : i1, i1
 hw.output %fullsum : i2
}
```

Figure 2. @adder and @completeAdder are provably equivalent in circt-lec

Other higher-level combinational operations are similarly tested for semantic correctness and regression by performing the logical equivalence check with their corresponding decomposition over basic logic gates.

Present work

Considering the time constraints, we convened to focus on reaching a solid and performant foundation for further development, ignoring busywork in favor of advancing in implementing exciting ideas.

Coverage of the HW dialect

All the basic operations, like **module**, **constant** and **output** have been swiftly implemented; **instances** were tricky but important to get right, as they enable checking complex designs.

Integer values, aptly stored internally as **bitvectors**, have been implemented along with the related **extract** and **concat** operations.

Other data structures (e.g. *arrays*) instead were deemed uninteresting as they are abstractions that can be lowered to an integer-only representation if one were to write an appropriate transforming pass.

Coverage of the Comb dialect

Combinational operations are being implemented at a steady pace (e.g. **add**, **xor**, **and**, **mux** and **mul** are already functional), leaving time for further general efforts on developing the tool.

Nonetheless, the project is on track to cover the whole set of operations, consonant with the originally established goal.

Future challenges

Equivalence of sequential circuits

Introducing registers vastly increases the difficulty of the problem and merits particular attention; the two considered approaches are:

- **Finite State Machine equivalence** easily leads to memory problems as the state space explodes (e.g. a simple i64 counter); it could become tractable by constructing the FSM on the fly and performing bounded model checking but it would have to be opt-in as it is an incomplete solution; on the other hand it would ideally also consent checking arbitrary properties expressed in temporal logic.
- **Register correspondence**, that is discovering the register pairs which are equivalent either through heuristics or simulation, thus reducing the problem to equivalence of combinational circuits which the tool can already solve.

Additional ideas

- In case of multiple outputs of which only a portion is wrong, show info on just those **values which are affecting the relevant outputs** rather than the whole model.
- Similarly, it might be interesting to only show interpretation of values at **module boundaries** rather than the whole state.
- It might be possible to localize the introduced bugs by computing the **Craig interpolant** between the circuits.
- Users could submit a list of **instances identifiers to be considered equivalent** (e.g. in case of big designs with localized changes), vastly reducing the time to solve the equivalence check.
- Extend the tool to support **different logical engine backends**, either by integrating an abstract SMT api like Smt-Switch[1] or by employing the SMT-LIB format internally.
- Add support for **multiple-value logic** as used in SV and VHDL.

References

[1] Stanford Centaur. Smt-switch. `https://github.com/stanford-centaur/smt-switch`, 2019.

[2] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[3] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[4] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of moore's law. *CoRR*, abs/2002.11054, 2020.

[5] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 12–27, New York, NY, USA, 1988. Association for Computing Machinery.

[6] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. Llhd: A multi-level intermediate representation for hardware description languages, 2020.