Encoding Purely Functional Languages in an RTL-based Compiler

Amelia Dobis[*] Gongqi Huang[*] Mae Milano

Princeton University
Department of Computer Science
Princeton, NJ, USA

Abstract

Unifying compiler infrastructure for high-level hardware languages has long been the goal of the CIRCT project. However, by following trends in hardware language design, CIRCT has specialized itself in HLS and RTL-like paradigms, while requiring elements from other paradigms to be handled mostly in their respective frontends. In this work, we present 3 approaches to support purely functional languages natively in CIRCT. We demonstrate these by creating a CIRCT-based compiler for Clash, a Haskell-as-hardware language. These methods range from simple conversions leveraging existing frontend transformations, to the introduction of full lambda calculus support in CIRCT via the `lc` dialect. By retaining the original functional structure of a Clash design, the `lc` dialect captures the source's high-level structure, enabling optimizations and higher-quality emission by backends. This paves the way for non-RTL paradigms to leverage the power of a unified compiler.

1 Introduction

Non-HLS (High-Level Synthesis) high-level hardware languages have evolved through distinct "waves", each moving further from gate-level netlists. The **first wave** introduced behavioral description of hardware via VHDL and Verilog. This was followed by a **second wave** of embedded Domain Specific Languages (eDSLs), such as Chisel (Scala) [1], pyMTL (Python) [2] , and Lava (Haskell) [3], which embed structural components in a host language to generate hardware. Finally, a **third wave** has emerged, focusing on custom languages with advanced type systems, such as Spade [4], Filament [5], or Anvil [6].

Despite the distinct philosophies of these waves, their paradigms remain close to the Register Transfer Level (RTL), often utilizing first-wave languages as their compilation targets. This motivates a unified, MLIR-based [7] infrastructure, namely CIRCT [8], to consolidate the compilation of these diverse hardware languages via a central RTL-based core dialects. While CIRCT has had success in unifying the compilation of RTL-like paradigms, it is primarily optimized for structural abstractions with explicit stateful elements. As a result, CIRCT struggles to natively encode purely functional languages, where hardware is defined through function composition and Algebraic Data Types (ADTs), without requiring high-level functional constructs to be elaborated away by frontends.

Clash. Being neither an eDSL, a custom language, nor a form of HLS, Clash [9] leverages the power of Haskell by reinterpreting System FC [10], the internal representation of the Glasgow Haskell Compiler (GHC) [11], to produce hardware. As a result, designers are able to describe hardware through purely functional abstractions, such as higher-order functions, polymorphism, and ADTs, without sacrificing the explicit structural nature of hardware languages. To bridge the semantic gap, Clash employs a normalization process that transforms System FC into a flat, "normalized" representation, while only eliminating recursive constructs that lack a direct hardware equivalent. This form is then finally lowered to a synthesizable target in VHDL or Verilog.

In this work, we propose 3 approaches for encoding purely functional hardware languages in a unified RTL-based compiler such as CIRCT. We demonstrate these approaches by creating a CIRCT-based Clash compiler.

2 Building A CIRCT-Based Clash Compiler

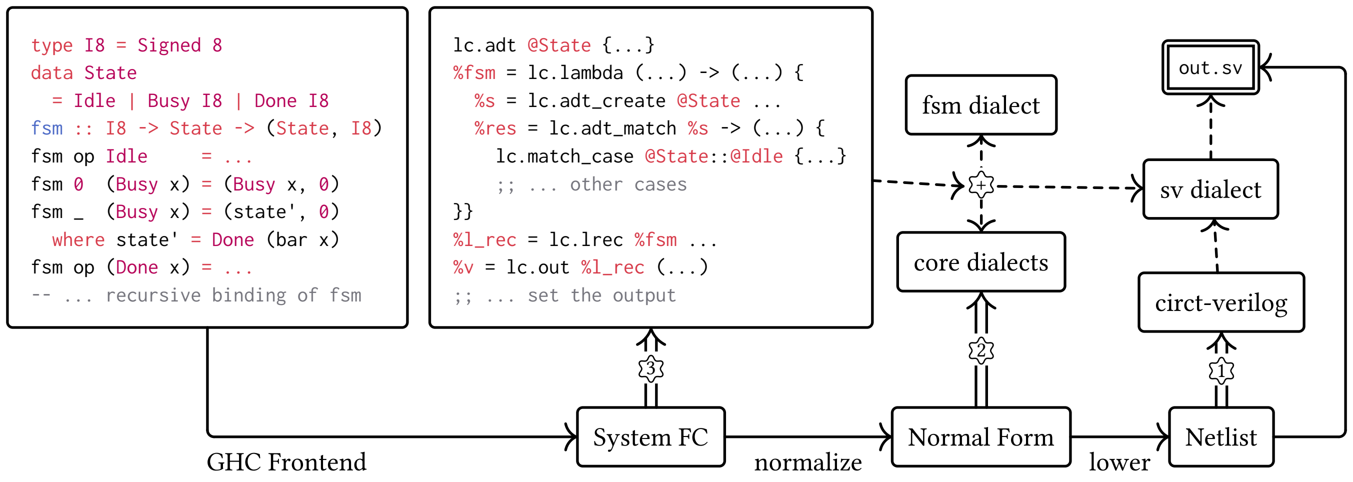
In porting a compiler to CIRCT, we start by identifying potential entry-points, i.e. stages at which we can transfer computation from our frontend (Clash) to our compiler (CIRCT). For Clash, we identify 3 such entry-points. These trade-off utilizing existing lowering methods found in our frontend with preserving the source's high-level structure in backends. Figure 1 illustrates how an example description of control logic, i.e. `fsm`, is compiled using the existing Clash compiler, and then integrated with CIRCT through entry-points ①-③. In particular, `fsm` is a higher-order function that takes a integer `op`, an ADT `State` representing its state, and returns a tuple of the next state and output. `bar` is a function that performs some computation. We omit Clash-specific operations to generate the top-level module to focus on the functional description of a stateful circuit.

2.1 The Low-Hanging Fruits

The most straightforward integration path of the 3 entry-points is ①, where the fully lowered netlist, generated by Clash, is fed into CIRCT through its Verilog frontend. While this isn't a full integration, we still benefit from all of the extra optimizations provided by CIRCT's LLHD/Moore compiler [12] used in the Verilog frontend, and its industry-strength SystemVerilog (SV) backend. The result is a more consistent and compact output, yet it retains the unintuitive signal names and design structure generated by Clash.

Rather than relying on the existing backend in Clash, another approach, marked as ②, is to directly convert the normalized System FC into CIRCT's core dialects. The ab-

[*]Equal contribution.


```
type I8 = Signed 8
data State
  = Idle | Busy I8 | Done I8
fsm :: I8 -> State -> (State, I8)
fsm op Idle      = ...
fsm 0  (Busy x) = (Busy x, 0)
fsm _  (Busy x) = (state', 0)
  where state' = Done (bar x)
fsm op (Done x) = ...
-- ... recursive binding of fsm
```

```
lc.adt @State {...}
%fsm = lc.lambda (...) -> (...) {
  %s = lc.adt_create @State ...
  %res = lc.adt_match %s -> (...) {
    lc.match_case @State::@Idle {...}
    ;; ... other cases
}}
%l_rec = lc.lrec %fsm ...
%v = lc.out %l_rec (...)
;; ... set the output
```

Figure 1: The Clash compilation pipeline integrated with CIRCT. Starting from an example described using pattern matching and ADTs in Clash, point ③ shows how the high-level structure is preserved in CIRCT using the proposed `lc` dialect.

straction level of the normalized form is close to that of the core dialects, allowing the translation between the two to be mostly syntactic. This is due to the normalization ensuring that all terms are representable in hardware, where, e.g., recursive data types are fully elaborated. With ②, we are no longer tied to backend-specific passes, and now benefit from the sophisticated global optimizations, and all of the other backends, provided for core dialects. While this is an improvement over ①, it is still a shallow integration of our frontend, as none of CIRCT's backends or optimizations have access to the original functional structure of the design. While the original structure directly captures the design's intent, normalization flattens the design making many optimizations difficult without further analyses.

2.2 Preserving High-Level Structure

In our two previous approaches, the source description was structurally flattened before reaching CIRCT, obscuring high-level intent and limiting the scope of downstream optimizations. We thus propose a final approach, which integrates Clash into CIRCT directly from the pre-normalized System FC, i.e. ③. This form preserves the original structure of the design, including all recursive data types and higher-order functions, which can enable otherwise difficult analyses when encoded in CIRCT.

To enable this approach, we introduce the `lc` (lambda calculus) specialty dialect which encodes ADTs, pattern matching, and higher-order functions. Unlike other purely functional MLIR dialects, such as `rise` [13], `lc` captures the recursive and nested structure inherent in functional descriptions, in a manner that interoperates and lowers to existing hardware dialects, in particular the `fsm` dialect.

The `lc` dialect provides native support for ADTs via `lc.adt` and `lc.adt_case`. Concrete values are instantiated using `lc.adt_create` or `lc.adt_constant`. Individual ADT cases can include block arguments, which function as parameters that can be used during pattern matching, e.g.

```
lc.adt @State {                   %s = lc.adt_create @State
  lc.adt_case @Idle                      init @State::@Idle
  lc.adt_case @Busy(x: i8)        %i = lc.adt_constant
  lc.adt_case @Done(x: i8)                 @State::@Idle
}
```

Pattern matching is supported via `lc.adt_match` and `lc.match_case`, as illustrated in Figure 1. Pattern matches can contain arbitrary logic, including nested pattern matches, using the case's block argument alongside other hardware dialects to produce a result with `lc.yield`. Lambdas can be defined using `lc.lambda`. Unlike `func.func`, these support functions as arguments and recursive value bindings using `lc.rec` and `lc.out`. As a result, the structure of `lc` allows us to identify high-level constructs such as FSMs and encode them using specialty dialects, e.g. the `fsm` dialect, which contain specific optimizations and have direct lowerings to several backends allowing for high-level structure to be retained in the outputs. More specifically, for the example in Figure 1, the FSM structure is identified by the recursive value binding of `%fsm` using `lc.rec`. The `%s` value in `%fsm` tracks the current state represented by each case of the pattern match. These cases yield two results, the next state of the FSM and their return value, whose logics become a transition guard and a state output respectively in the `fsm` dialect.

3 Conclusion

In this work, we add support for purely functional constructs to the otherwise RTL-based unified CIRCT compiler infrastructure. We argue that with the `lc` dialect, we can retain the high-level structure of the source Clash code throughout the compilation pipeline down to the emitted SV, all while enabling more specific optimizations. We also show that shallow integrations with CIRCT can still yield immediate benefits while minimizing the engineering cost. With these approaches, we hope to democratize access to CIRCT, paving the way for full hardware compiler unification, regardless of the source paradigm.

References

[1] J. Bachrach *et al.*, “Chisel: constructing hardware in a Scala embedded language,” in *Proceedings of the 49th Annual Design Automation Conference*, in DAC '12. San Francisco, California: Association for Computing Machinery, 2012, pp. 1216–1225. [Online]. Available: https://doi.org/10.1145/2228360.2228584

[2] S. Jiang, P. Pan, Y. Ou, and C. Batten, “PyMTL3: A Python Framework for Open-Source Hardware Modeling, Generation, Simulation, and Verification,” *IEEE Micro*, vol. 40, no. 4, pp. 58–66, 2020, doi: 10.1109/MM.2020.2997638.

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, “Lava: hardware design in Haskell,” *SIGPLAN Not.*, vol. 34, no. 1, pp. 174–184, Sept. 1998, doi: 10.1145/291251.289440.

[4] F. Skarman and O. Gustafsson, “Spade: An HDL Inspired by Modern Software Languages,” in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, 2022, pp. 454–455. doi: 10.1109/FPL57034.2022.00075.

[5] R. Nigam, P. H. Azevedo de Amorim, and A. Sampson, “Modular Hardware Design with Timeline Types,” *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, June 2023, doi: 10.1145/3591234.

[6] J. Z. Yu, A. R. Jha, U. Mathur, T. E. Carlson, and P. Saxena, “Anvil: A General-Purpose Timing-Safe Hardware Description Language.” [Online]. Available: https://arxiv.org/abs/2503.19447

[7] C. Lattner *et al.*, “MLIR: scaling compiler infrastructure for domain specific computation,” in *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, in CGO '21. Virtual Event, Republic of Korea: IEEE Press, 2021, pp. 2–14. doi: 10.1109/CGO51591.2021.9370308.

[8] S. Eldridge *et al.*, “MLIR as hardware compiler infrastructure,”

[9] C. Baaij, “Digital circuit in CλaSH: functional specifications and type-directed synthesis,” PhD Thesis - Research UT, graduation UT, University of Twente, Netherlands, 2015. doi: 10.3990/1.9789036538039.

[10] M. Sulzmann, M. Chakravarty, S. Peyton Jones, and K. Donnelly, “System F with type equality coercions,” 2007, pp. 53–66. doi: 10.1145/1190315.1190324.

[11] S. P. Jones, K. Hammond, W. Partain, P. Wadler, C. B. Hall, and S. L. P. Jones, “The Glasgow Haskell Compiler: a technical overview,” 1993. [Online]. Available: https://api.semanticscholar.org/CorpusID:56228824

[12] F. Schuiki, A. Kurth, T. Grosser, and L. Benini, “LLHD: a multi-level intermediate representation for hardware description languages,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, in PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 258–271. doi: 10.1145/3385412.3386024.

[13] M. Lücke, M. Steuwer, and A. Smith, “Integrating a functional pattern-based IR into MLIR,” in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, in CC 2021. Virtual, Republic of Korea: Association for Computing Machinery, 2021, pp. 12–22. doi: 10.1145/3446804.3446844.