Zaozi, Reinvent Chisel in Scala 3

Jiuyang Liu
Huazhong University of Science and Technology
China

Ruixing Yang
Institute of Software, Chinese Academy of Sciences
China

JiongJia Lu
Institute of Integrated Circuits, Henan Academy Of Sciences
China

Shupei Fan
Tsinghua
China

Jianhao Ye
University of Chinese Academy of Sciences
China

Xuecheng Zou
Institute of Integrated Circuits, Henan Academy Of Sciences
China

ABSTRACT

Zaozi is a Scala 3-based eDSL for hardware design, directly interfacing with CIRCT via C-APIs to eliminate serialization overhead and accelerate elaboration. By offloading execution to CIRCT, it maintains a minimal JVM footprint as a lightweight type system wrapper for MLIR. Inspired by Chisel[1], Zaozi features a pluggable type system and operation construction, enabling seamless integration with MLIR dialects and diverse backends. It also decouples interface and implementation generation, improving physical design and verification workflows. While not source-compatible with Chisel, it remains IR-compatible via CIRCT[2], ensuring interoperability. This paper introduces Zaozi's architecture, design principles, and performance benefits.

1 INTRODUCTION

1.1 Motivation

Chisel is widely used for large-scale digital design but has several limitations in usability, extensibility, and integration with traditional physical design and verification flows. Its type system embeds hardware value types within Scala but lacks clear distinctions for hardware-specific constructs like IO, registers, and wires, relying on runtime reflection for type management. It also lacks support for user-defined type systems and operations, making features like fixed-point arithmetic difficult to implement. Chisel's global mutable runtime forces full-circuit elaboration, preventing post-elaboration linking and causing inefficiencies in large SoC designs. Furthermore, its incompatibility with traditional hierarchical design and verification methodologies complicates integration with physical design and verification teams.

1.2 Innovation

To overcome Chisel's limitations, **Zaozi** takes a different approach to hardware description. Instead of relying on textual FIRRTL, it directly constructs MLIR using CIRCT's C-API, eliminating serialization overhead and accelerating elaboration. It adopts a Scala 3-based typeclass, leveraging implicit patterns to allow users to define custom type systems and extend operations seamlessly. Unlike Chisel's mutable runtime types, Zaozi introduces a reference-based type system for `Register`, `Wire`, `IO`, and `Probe`, making the Zaozi type system sound and Scala runtime immutable. Additionally, Zaozi enhances physical design and verification by enforcing parameter serialization for both declaration and implementation, requiring explicit interface instantiation before design elaboration, enabling independent interface generation for physical design and verification workflows.

1.3 Contribution

Zaozi is built on extensive experience with Chisel development, incorporating feedback from RTL engineers and physical design teams. It reimagines hardware construction language (HCL) with IR Construction Language (ICL) principles and leveraging MLIR's infrastructure in a Scala 3-based eDSL. Zaozi introduces a state-of-the-art type system and runtime design, offering a scalable and modular approach to hardware description. By bridging the gap between eDSL frontends and CIRCT/MLIR, it provides a unified framework for multiple dialects, simplifying frontend development. These innovations position Zaozi as a forward-looking research effort, advancing next-generation hardware eDSLs while ensuring interoperability with Chisel via CIRCT.

2 SYSTEM DESIGN

Zaozi is designed as a reusable and hierarchical framework, leveraging modular libraries to maximize code reuse while providing fine-grained components that can be utilized by external tools.

2.1 Fundamental Libraries: MLIR and CIRCT Bindings

The core foundation of Zaozi consists of two fundamental libraries: `mlirlib` and `circtlib`. These libraries are generated using `jextract`, which converts MLIR and CIRCT C headers into Java code. Scala 3 is then used to manually wrap these Java bindings, providing a seamless interface for circt API-calls. These bindings serve as the fundamental bridge between Zaozi and CIRCT, enabling direct communication with CIRCT without serialization and deserialization overhead. It can also be compatible with Chisel and other tools that wish to eliminate textual-based exchange interface. To demonstrate the possibility, we upstreamed an experimental Chisel library in Scala 2 that allows Chisel to directly interface with CIRCT using this library.

LATTE '25, March 30, 2025, Rotterdam, South Holland, Netherlands

2.2 Zaozi: A Slim and Extensible eDSL for MLIR

Zaozi itself is a lightweight eDSL that borrows much of its syntax from Chisel to ensure a familiar RTL design experience.3.3

It is composed of several key components:

2.2.1 `Value Type`. The `Value Type` wraps the C++ `mlir::Type` in Scala. It provides familiar types such as `UInt`, `SInt`, `Bits`, `Bundle`, and `Vec`, similar to Chisel. However, compared to Chisel:

- Zaozi maintains a minimal memory footprint in JVM, other than the pointer to the C++ memory segment, nothing is maintained at JVM side.
- Most APIs, including width and field count of `Vec`, are directly accessed by the C-API, allowing MLIR's built-in type inference to be reused, and eliminate any mismatch from the zaozi and circt.
- Users can define their own types by implementing the `toMlirTpe` function, which invokes the C-API and returns an arbitrary `MlirType` wrapped in Scala 3.

2.2.2 `Reference Type`. One of the major limitations of Chisel is the missing of `Reference Type` representing in Scala type system, which is handled at the Scala runtime, leading to significant complexity in Chisel's `Builder`. Zaozi addresses this with a new `Reference Type`, where each reference type encapsulates both a mlir operation and its result. Each `Reference Type` is assigned with a Scala 3 type and a `Value Type` as type parameter, for example, `Register[T <: ValueTpe]`, `Wire[T <: ValueTpe]`, `Constant[T <: ValueTpe]`, and `Probe[T <: ValueTpe]`(for verification), eliminating ambiguities semantic and shifts most error detection to Scala compile-time, reducing runtime errors.

2.2.3 `Scala Dynamic` *on* `Reference Type`. However, a key challenge in the implementation arises after introducing the `Reference Type`. Unlike a straightforward `ref.someField` access to `Bundle` in Chisel, where `someField` is expected to be a member of `Bundle <: ValueTpe` in zaozi. This approach does not work because it refer to the `Reference Type`, and the return type must also be a `Reference Type`. To address this while maintaining a clean and intuitive design, zaozi leverages Scala macros and the `Dynamic` trait to cast `someField` from `Bundle`.

2.2.4 *Operations as* `Type Class` *for* `Value Type`. Zaozi defines a type class-based API for handling data types, allowing operators to be attached flexibly rather than mixed into specific types. It follows a two-level approach: operators are defined via `Type Class`, while implementations are provided using the `given` pattern. This mix-in paradigm enhances extensibility and flexibility, enabling users to introduce new APIs or modify existing implementations, such as replacing an addition operation with a commercial IP block instead of directly generating a `firrtl.add` operator.

2.2.5 `Builder` *as Contextual Function.* In Chisel, hardware elaboration assumes a single global circuit elaborator, maintaining the elaboration state globally. Zaozi challenges this approach by leveraging contextual functions in Scala 3, ensuring that each hardware generator is bound to its own context. This enables parallel hardware elaboration, allowing multiple `Module` contexts to be instantiated and elaborated across multiple threads before linking them in CIRCT. This design significantly improves eDSL runtime efficiency while establishing a more scalable and modular elaboration framework in Zaozi.

Through this modular and extensible design, Zaozi establishes a modern and scalable framework for hardware design, bridging the gap between eDSL frontends and CIRCT while maintaining a familiar user experience.

3 DISCUSSION

3.1 Role of Scala for eDSL

Chisel's success in building a strong community is largely attributed to Scala's language features, including functional programming, a robust type system, and an advanced garbage collector via the JVM.

However, Scala should serve solely as an eDSL rather than a build system, compiler, or linker. Its immutable nature and JVM-based memory overhead make it unsuitable for these roles.

Scala has limited exploration of other host languages, but when considering its evolution as an eDSL, the focus should remain on refining Scala rather than seeking alternatives. Just as the FIRRTL compiler was successfully replaced by MLIR, we should also remove functionalities that do not belong in Scala and rethink how to design a minimal, efficient eDSL.

By clearly defining boundaries, we can delegate non-eDSL tasks to more suitable tools, ensuring a cleaner and more maintainable hardware design flow.

3.2 We need more eDSL in the hardware domain

We need more than just a single hardware description language; the hardware ecosystem demands a broader range of DSLs. UVM should be modeled and transformed into a verification DSL, while SystemC needs to be adapted into a modeling DSL. Additionally, bridging these DSLs is a crucial challenge that must be addressed. The hardware industry requires these capabilities–not just Chisel.

3.3 Lesson from Chisel

Chisel, while successful, lacks elegance. We should learn from its eDSL design challenges:

- **eDSLs are not real languages** - Unlike traditional languages with formal specifications, eDSLs evolve through API changes. As Chisel adoption grew, maintaining backward compatibility and managing releases became increasingly difficult.
- **An eDSL should be as slim as possible** - Bugs can arise at any stage: build systems, the Scala compiler, Chisel's builder, CIRCT compilation, RTL functionality, or EDA tools. Chisel takes on too many responsibilities, increasing complexity and making debugging harder. A hardware eDSL should focus solely on frontend concerns, delegating compilation and toolchain tasks to specialized tools.
- **Don't reliance on Scala's advanced features** - Chisel heavily uses compiler plugins and macros, making maintenance difficult and leading to long-term sustainability issues. A cleaner approach is needed to reduce hidden complexity.

REFERENCES

[1] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *The 49th Annual Design Automation Conference*.

[2] Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth, George Leontiev, Fabian Schuiki, Ram Sunder, et al. 2021. MLIR as hardware compiler infrastructure. In *Workshop on Open-Source EDA Technology (WOSET)*.

```
// Each Geneartor is a contextural function
// user summon Parameter, IO from context.
val p = summon[GCDParameter]
val io = summon[Interface[GCDIO]]
// The Implicit Clock and Reset is set by given
given Ref[Clock] = io.clock
given Ref[Reset] = io.reset
// Return is a NOT UInt, but Reg[UInt], Reg <: ValueTpe
val x: Reg[UInt] = Reg(UInt(p.width.W))
val y: Reg[UInt] = RegInit(0.U(32.W))
val startupFlag: Reg[Bool] = RegInit(false.B)
val busy: Reg[Bool] = y =/= 0.U
io.input.ready := !busy
io.output.bits.z := x
io.output.valid := !busy & startupFlag
val a = x - y

// cond ? (sel, alt) is just a suger, this given is
     introduced by import
given [R <: Referable[Bool]]: BoolApi[R] with
  extension (ref: R)
    def ?[Ret <: Data](
      con: Referable[Ret], alt: Referable[Ret]
    )(using Arena, Context, Block): Node[Ret] = ???


x := io.input.fire ? (
  io.input.bits.x,
  (x > y) ? (
    (x - y),
    x
  )
)
y := io.input.fire ? (
  io.input.bits.y,
  (x > y) ? (
    y,
    (y - x)
  )
)
startupFlag := io.input.fire ? (
  true.B,
  startupFlag
)
```

Listing 1: GCD code example in Zaozi