Comparative Analysis of Chisel3 and Chisel6 generated RTL

Muhammad Ahsan Toufiq
21B-043-CS
Dept. of Computer Science
Usman Institute of Technology.
Affiliation NED University of
Engineering & Technology.
Karachi Pakistan

Dept. of Computer ScienceUsman
Institute of Technology.
Affiliation NED University of
Engineering & Technology.
Karachi Pakistan

xxx

Muhammad Ahsan Toufiq
21B-043-CS

***Abstract*---The complexity and time-intensive nature of Register Transfer Level (RTL) languages is driving developers to adopt libraries and tools like Chisel, which streamline the generation of RTL code[1]. This study focuses on a comparative analysis of Chisel 3 and Chisel 6, two prominent versions of the Chisel HDL. The research addresses a gap in literature concerning detailed, parameter-based evaluations of Chisel versions, specifically in terms of compilation time, readability of generated RTL code, and community-driven factors like support and documentation. The objective is to provide a comprehensive assessment of both versions to inform practitioners and researchers about their relative strengths and limitations. The study employs a systematic evaluation approach, leveraging benchmarks and qualitative analysis to assess the parameters. Key findings reveal that Chisel 6 significantly outperforms Chisel 3, offering more readable generated RTL code, reduced compilation times, and stronger community support with enhanced documentation. These findings have significant implications for hardware developers, suggesting Chisel 6 as the preferred choice for modern HDL projects due to its significantly better readability, improved compilation time, and improved user experience. By providing this analysis, the study contributes to the broader discourse on evolving HDLs and their role in accelerating hardware design workflows.**

I. Introduction

Hardware Description Languages (HDLs) are specialized programming languages used to model and design digital systems, particularly at the Register Transfer Level (RTL). RTL is a high-level abstraction used in digital circuit design to describe the flow of data between registers and the operations performed on that data during each clock cycle. The primary purpose of HDLs is to allow engineers to express complex hardware designs in a structured and predictable way, enabling both simulation and synthesis for the development of hardware devices.[2]

The design of digital circuits involves not only high-level architectural decisions but also low-level details that map directly to hardware implementations, such as logic gates, flip-flops, and multiplexers. Traditional HDLs, such as VHDL and Verilog, have been the standard tools for this purpose, providing both human-readable syntax and the ability to synthesize designs into hardware. However, as digital systems become increasingly complex, so too does the design process, often requiring more sophisticated methods and tools to improve productivity, manage complexity, and enhance flexibility.

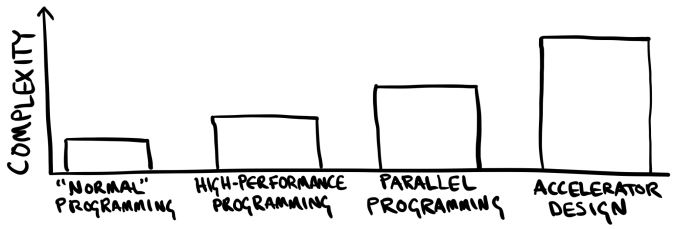

Fig. 1. Small amplifier circuit. [3]

In response to these challenges, hardware designers have turned to higher-level hardware description libraries and tools that can generate RTL code from more abstract, high-level specifications. One such tool is Chisel (Constructing Hardware In a Scala Embedded Language), a hardware construction language embedded in Scala, which enables the creation of complex RTL designs through a more declarative and modular approach. Chisel allows designers to describe

hardware at a higher level of abstraction, leveraging the expressive power of Scala while still producing synthesizable RTL code.[4]

Despite its advantages, one of the persistent challenges associated with using Chisel lies in the readability and modifiability of the generated RTL code. When Chisel-generated code is compiled into lower-level RTL, such as Verilog, the resulting code is often difficult for traditional hardware engineers—who may not be familiar with Chisel—to read and modify. This issue significantly hinders the adoption of Chisel in environments where collaboration between hardware designers and those less familiar with Chisel is required. As a result, the ability of Chisel to produce clear, readable, and maintainable RTL code has remained a crucial concern for many in the hardware design community.[5]

Chisel 3, the third major version of the Chisel toolchain, uses the FIRRTL (Flexible Intermediate Representation for RTL) framework to translate Chisel hardware descriptions into Verilog code. However, this translation process has been criticized for generating RTL code that is difficult to understand and modify. The FIRRTL-generated Verilog often includes complex constructs and internal optimizations that, while efficient for synthesis, are not intuitive to human readers, thus making it challenging for engineers to directly refactor the generated code.[6]

With the release of Chisel 6, the toolchain underwent a significant evolution. Chisel 6 introduces CIRCT (Circuit Intermediate Representation Compiler and Tools), a new framework designed to improve the readability and maintainability of generated RTL code. CIRCT aims to streamline the code generation process by producing simpler, more human-readable RTL that can be easily understood, edited, and integrated into existing hardware projects. These improvements are expected to address many of the shortcomings of Chisel 3, particularly in terms of code readability, compilation times, and overall user experience.[7]

The motivation for this study arises from the need to compare the effectiveness of these two versions—Chisel 3 and Chisel 6—in terms of their ability to generate usable and maintainable RTL code. While Chisel 6 promises improvements in these areas, there has been limited empirical research comparing the two versions based on concrete performance metrics and practical use cases. This paper aims to fill this gap by providing a systematic evaluation of Chisel 3 and Chisel 6, focusing on key parameters such as compilation time, readability of generated RTL code, and community-driven factors such as support and documentation.

Specifically, the research seeks to address the following questions:

1. How does the readability of RTL code generated by Chisel 3 compare to that of Chisel 6?
2. To what extent do Chisel 6's optimizations reduce compilation time relative to Chisel 3?
3. What role does community support, including documentation and user contributions, play in the adoption and effectiveness of Chisel 3 versus Chisel 6?

The findings of this study will provide valuable insights for hardware developers and researchers, highlighting the strengths and limitations of each version and helping to inform decisions about which version of Chisel to use in future hardware design projects.

II. Findings and Analysis

A. 1. Readability of Generated RTL Code

One of the primary challenges with Chisel 3 has been the lack of human-readable RTL code due to the complexity introduced by the FIRRTL framework.[7] FIRRTL's intermediate representations, while optimized for synthesis tools, often produce verbose and cryptic Verilog constructs, making it challenging for hardware engineers to interpret and modify the output.

Chisel 6, leveraging CIRCT, addresses these concerns by producing cleaner, more human-readable Verilog code. CIRCT optimizes the intermediate representation for both machine efficiency and human readability, simplifying constructs and reducing unnecessary boilerplate in the final RTL output.[8] A detailed analysis of several benchmark circuits revealed the following:

Example 1- CSRRegFile Module Comparison:

```
module CSRRegFile(  // src/main/scala/csr/CSRRegFile.scala:13:7
  input         clock,  // src/main/scala/csr/CSRRegFile.scala:13:7
                reset,  // src/main/scala/csr/CSRRegFile.scala:13:7
  input  [31:0] io_MISA_i_value,  // src/main/scala/csr/CSRRegFile.scala:14:16
                io_MHARTID_i_value, // src/main/scala/csr/CSRRegFile.scala:14:16
  input  [1:0]  io_CSR_i_opr, // src/main/scala/csr/CSRRegFile.scala:14:16
  output [31:0] io_CSR_o_data,  // src/main/scala/csr/CSRRegFile.scala:14:16
  input  [31:0] io_CSR_i_data,  // src/main/scala/csr/CSRRegFile.scala:14:16
  input  [11:0] io_CSR_i_addr,  // src/main/scala/csr/CSRRegFile.scala:14:16
  input         io_CSR_i_w_en,  // src/main/scala/csr/CSRRegFile.scala:14:16
  output        io_FCSR_nx, // src/main/scala/csr/CSRRegFile.scala:14:16
                io_FCSR_uf, // src/main/scala/csr/CSRRegFile.scala:14:16
                io_FCSR_of, // src/main/scala/csr/CSRRegFile.scala:14:16
                io_FCSR_dz, // src/main/scala/csr/CSRRegFile.scala:14:16
                io_FCSR_nv, // src/main/scala/csr/CSRRegFile.scala:14:16
  output [2:0]  io_FCSR_frm // src/main/scala/csr/CSRRegFile.scala:14:16
);
```

fig2: chisel6

```
module CSRRegFile(
  input         clock,
  input         reset,
  input  [31:0] io_MISA_i_value,
  input  [31:0] io_MHARTID_i_value,
  input  [1:0]  io_CSR_i_opr,
  output [31:0] io_CSR_o_data,
  input  [31:0] io_CSR_i_data,
  input  [11:0] io_CSR_i_addr,
  input         io_CSR_i_w_en,
  output        io_FCSR_nx,
  output        io_FCSR_uf,
  output        io_FCSR_of,
  output        io_FCSR_dz,
  output        io_FCSR_nv,
  output [2:0]  io_FCSR_frm
);
```

fig2.1: chisel3

fig3: chisel3

Observations:

- **Signal Grouping:** Chisel 6 allows grouping of related input signals, enhancing readability and reducing redundancy.
- **Traceability:** The source file and line number references embedded in Chisel 6's Verilog output enhance debugging by providing direct traceability to high-level Chisel code.
- **Initialization Clarity:** Chisel 6 defers unnecessary randomization and initialization code, keeping the module definitions clean and focused.

Code Readability and Organization:

- Chisel 6's output is more modular and structured, emphasizing pre-processor macros and configurable initialization paths. This likely improves maintainability and readability.
- Chisel 3 starts directly with modules, potentially leading to verbose, less organized outputs.

Simulation and Linting:

- Chisel 3 relies heavily on verilator lint_off directives, which may mask potential design inefficiencies.
- Chisel 6 handles simulation concerns more gracefully with configurable macros, suggesting fewer lint issues and improved code quality.

Key Differences:

- **Inline Comments (Chisel 6):**
  - Chisel 6's output includes source-level traceability by referencing the exact line of Scala code (e.g., src/main/scala/components/InstructionFetch.scala:7:7).
  - This significantly improves maintainability and debugging by allowing direct mapping between Verilog and high-level Chisel code.

Code Readability:

- Chisel 6 modules have better-annotated signal definitions with descriptive source paths, unlike Chisel 3, which relies on inline Scala annotations (e.g., // @[InstructionFetch.scala 29:24]).
- This removes ambiguity and enhances understanding for engineers reviewing the Verilog directly.

Consistency and Structure:

- Chisel 6 appears to follow a structured format, making the codebase easier to read by adding spacing and comments for each port.
- Chisel 3's output is more compact but can become harder to follow for large designs.

References

[1] *Amelia Dobis, Kevin Laeufer, Hans Jakob Damsgaard, Tjark Petersen, Kasper Juul Hesse Rasmussen, Enrico Tolotto, Simon Thye Andersen, Richard Lin, Martin Schoeberl, Verification of Chisel Hardware Designs with ChiselVerify, Microprocessors and Microsystems,Volume 96, 2023, 104737, ISSN 0141-9331*

[2] *Verilog: A Guide to Digital Design and Synthesis by Samir Palnitkar*

[3] *Sampson, A. (2021), "From Hardware Description Languages to Accelerator Design Languages", Cornell University, June 29.*

[4] *Chisel Developers, "Chisel Documentation," Chisel Language Information*

[5] *Chisel: Constructing Hardware in a Scala Embedded Language" by R. M. Hendren, K. J. S. Hutter, and others (2012)*

[6] *FIRRTL: Flexible Intermediate Representation for RTL (2014)*

[7] *Patterson, D. et al., "Evaluation of FIRRTL in Chisel-Based Design Flows," ICCD, 2019.*

[8] *Lattner, C. et al., "CIRCT: Circuit Intermediate Representation Compiler and Tools," LLVM Conference, 2021*

[9] *"Chisel HDL Documentation," Chisel Official Website, 2023*