A FIRRTL Backend for the Calyx High-Level Accelerator Compilation Infrastructure

Ayaka Yorihiro
Cornell University
Ithaca, NY, USA
ayaka@cs.cornell.edu

Griffin Berlstein
Cornell University
Ithaca, NY, USA
griffin@cs.cornell.edu

Kevin Laeufer
University of California, Berkeley
Berkeley, CA, USA
laeufer@eecs.berkeley.edu

Adrian Sampson
Cornell University
Ithaca, NY, USA
asampson@cs.cornell.edu

***Abstract*—We build a new translation from Calyx, an open-source intermediate language for compiling high-level programs into hardware accelerators, to FIRRTL, an intermediate representation used in the open-source Chisel and CIRCT projects. Calyx previously targeted Verilog as a *de facto* intermediate language for hardware description; our new backend avoids the complexity of targeting Verilog source code and requires important generalizations for the Calyx compiler. We demonstrate the technical challenges endemic to bridging high-level and low-level hardware compiler infrastructures.**

***Index Terms*—Intermediate Language, Accelerator Design, Compilers**

I. Introduction

Intermediate languages (ILs) for hardware design promise better interoperability between diverse tools: languages, EDA toolchains, testing frameworks, simulators, formal verifiers, and so on. However, these ILs have generally focused on register-transfer level (RTL) design: traditional, low-level hardware specifications. Higher-level compilers, such as high-level synthesis (HLS) tools and accelerator design language (ADL) compilers, typically treat Verilog as a *de facto* IL for generating hardware. While it is a practical choice for interacting with legacy EDA flows, Verilog makes for a cumbersome compilation target—and it leaves HLS tools out from the interoperable ecosystems that these ILs can offer.

This paper connects two existing, open-source compiler infrastructures and ILs: FIRRTL [1], a low-level RTL IL, and Calyx [2], a high-level IL that supports compilers from domain-specific languages. Until now, Calyx has only targeted Verilog, making FIRRTL Calyx's *second* backend. Our primary goal is to demonstrate the potential advantages and engineering challenges for high-level compilers when they opt into a post-Verilog, IL-based hardware design ecosystem. Concretely, our backend brings these benefits to the Calyx compiler:

- Calyx programs can exploit a new pool of FIRRTL-based tooling, including the FIRRTL compiler's optimizations and new simulation infrastructure [3].
- The new backend clarifies the semantics of Calyx's low-level program representation, disentangling it from the nuances of Verilog semantics. We also reveal commonalities between the ILs for Calyx and FIRRTL that may apply to other high-level hardware compilers.
- By generalizing the Calyx compiler, we highlight its implicit dependence on Verilog for *primitives*, built-in hardware modules that all programs use as elementary building blocks, and show how to instantiate the same role in a Verilog-independent way.

Broadly, this paper advocates for further investment in connections between low-level and high-level hardware tooling. By leveraging the complementary strengths in different open-source projects, the community can build better end-to-end tooling that sidesteps Verilog unless it is strictly necessary to interface with proprietary tools. The result will be better interoperability, clearer semantics, and simpler, faster compilers.

This short paper highlights the three main technical challenges in this new backend: (1) the *core language translation* from Calyx's IL to FIRRTL, highlighting the similarities and differences between the two languages; (2) our *primitive library*, which entailed elaborating parameterized descriptions of basic hardware units; and (3) an automated *testing infrastructure*, which makes it possible to execute high-level accelerator designs without manual testbench effort.

II. Core Language Features

The Calyx compiler uses a single intermediate language (IL) to translate high-level, control-based programs to low-level, mostly structural code. Like in FIRRTL, this low-level subset of the IL admits a straightforward translation to Verilog and other HDLs. The core mechanism in our new backend is a translation from this low-level form of the Calyx IL into FIRRTL.

The main construct in low-level Calyx IL is a *guarded assignment*, of this form:

```
cell1.port1 = guard ? cell2.port2;
```

where `guard` is a Boolean expression over port values. Ports are defined whenever *exactly one* such assignment's guard is `1`. *Components* in Calyx are analogous to modules in Verilog or FIRRTL, and they consist only of cell instantiations and a single list of these guarded assignments. Our translation maps guarded assignments onto FIRRTL conditional statements:

```
cell1.port1 is invalid
cell1.port1 <= UInt(0)
when guard:
  cell1.port1 <= cell2.port2
```

We use FIRRTL's `is invalid` initialization to reflect Calyx's semantics: each port may be written by exactly one guarded assignment; otherwise, it is undefined. We also initialize ports to zero to reflect Calyx's semantics for `@control` ports.

Simultaneous assignments to the same port are an error in Calyx, so a cascade of FIRRTL `when` statements suffices for all legal Calyx programs. Because it contains `when`, the generated code is not within the LoFIRRTL subset: we use the FIRRTL compiler to generate muxes for the conditional behavior.

The rest of the translation maps Calyx components onto FIRRTL modules, the ports in Calyx signatures onto FIRRTL `input` and `output` declarations, and Calyx cell instantiations onto FIRRTL `inst` declarations. Calyx does not have a rich type system for values: all signals are "plain" bit vectors. Our backend, therefore, only uses the `UInt<`n`>` types in FIRRTL.

III. Primitives

A key difference between RTL descriptions and any high-level hardware compiler is the need for a library of *primitives:* basic building blocks that are implemented outside the language. Verilog and FIRRTL have built-in operators for adders, registers, memories, and so on; any more sophisticated subcircuits must come from user libraries. In Calyx, even extremely basic constructs—including registers, adders, and bitwise logical operations—come from a standard library of primitives. And, as in any high-level accelerator compiler, there are common constructs that do not make sense to implement within the high-level framework itself: for example, Calyx's standard primitive library includes a basic pipelined multiplier and divider. These primitives are critical to the compilation of any Calyx program.

Traditionally, these Calyx primitives have been implemented in parameterized Verilog. For example, Calyx's register primitives, `std_reg`, has a parameter `WIDTH` to dictate the size of the register: the declaration `reg = std_reg(32);` instantiates a 32-bit register.

Our new backend requires a new approach to the primitive library. We implement two options: the existing Verilog implementations and a new primitive library written in FIRRTL.

The first route uses FIRRTL's `extmodule` keyword, which lets FIRRTL code interoperate with modules in other HDLs. Our backend iterates over all cell instantiations to produce a set of unique primitive instantiations. It uses the port and parameter information for each unique instantiation to produce a `extmodule` declaration. Finally, each cell instantiation translates to a FIRRTL instantiation of the corresponding `extmodule`.

The second alternative produces "pure" FIRRTL code. The challenge here is that FIRRTL is monomorphic: i.e., there are no compile-time parameters as in Verilog. We therefore cannot directly translate Calyx's primitives into FIRRTL modules. Instead, our primitive implementations are metaprogramming *templates:* FIRRTL code containing names to be substituted with parameter values. We implement a special-purpose Calyx "backend" that outputs in JSON the names and parameters for every unique primitive instantiation in a program. A separate tool uses this JSON data to fill in the FIRRTL templates and produce monomorphized FIRRTL modules.

In both cases, our new backend revealed the ways Calyx has implicitly relied on Verilog parameters to make primitives work. Our new metaprogramming approach is more general and makes Calyx easier to port to other new backends.

TABLE I: Preliminary Results. T[ms] is simulation time in milliseconds, and S[KB] is the size of the emitted Verilog in kilobytes.

Benchmark	Calyx		FIR_V		FIR_{FIR}	
	T[ms]	S[KB]	T[ms]	S[KB]	T[ms]	S[KB]
3mm	69	194	37	115	34	101
bicg	7	101	5	64	5	50
symm	26	146	21	88	14	73
trmm	14	88	12	55	11	42

IV. Testing Infrastructure

Calyx automatically generates a simple testbench for every program. This built-in testing functionality makes it easy to run programs during development without expending the effort to construct a custom test harness. While custom testbenches are a critical part of traditional hardware development, they are often unnecessary for high-level accelerator design, where programs are typically straightforward input-to-output functions.

In our Verilog backend, this automated testbench assumes the design maps all meaningful inputs and outputs to memories, and it maps these memories onto on-disk files. The Calyx compiler generates Verilog's unsynthesizable `readmemh` and `writememh` constructs, which most simulators support.

There is no analog in the FIRRTL ecosystem: `LoadMemoryAnnotation` and similar annotations existed in some versions of the FIRRTL compiler, but they are supported in neither the new CIRCT-based `firtool` compiler nor universally in simulators [3]. Instead, our strategy is to expose the memories outside of the generated FIRRTL code and rely on Verilog to perform the file reads and writes.

A limitation of this approach is that we cannot simulate our custom testbench (and test our generated FIRRTL designs) on a simulator that does not support Verilog, such as ESSENT [3]. The lack of an official mechanism for reading/writing from files in FIRRTL programs necessitates this workaround and consequently hinders the creation of general cross-simulator testbenches. While it may seem inconsequential, this absence creates a genuine barrier for high-level accelerator design flows targeting FIRRTL and means Verilog unfortunately enjoys a convenience advantage here. We suggest further effort in the FIRRTL ecosystem to close this gap and make it a more convenient target for high-level accelerator design.

V. Evaluation

We compile Calyx programs to FIRRTL, use the FIRRTL compiler 1.6 to generate Verilog, and use Verilator 5.006 to simulate the final design. We compare against the same Calyx program compiled directly to Verilog, and ensure equivalence.

Compiling through FIRRTL can both add overhead and enable new optimizations. We quantify the simulation performance among the different compilation routes. Table I compares (1) the original Calyx-to-Verilog backend (*Calyx*), (2) our new FIRRTL backend with Verilog primitives ($\textit{FIR}_V$), (3) our FIRRTL backend with FIRRTL primitives ($\textit{FIR}_{FIR}$). We measure the wall-clock simulation time and the total size of the generated code. The FIRRTL-compiled designs were simulated faster than the Calyx-compiled designs. Also, FIR_{FIR} yields the smallest Verilog file in all benchmarks. Our backend is open-source in Calyx [4], and our evaluation is public [5].

References

[1] A. M. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017.

[2] R. Nigam, S. Thomas, Z. Li, and A. Sampson, "A compiler infrastructure for accelerator generators," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[3] S. Beamer, T. Nijssen, K. Pandian, and K. Zhang, "ESSENT: A high-performance RTL simulator," in *Workshop on Open-Source EDA Technology (WOSET)*, 2021.

[4] The Calyx Project. (2024) Calyx. [Online]. Available: https://github.com/calyxir/calyx

[5] "Calyx-FIRRTL evaluation," 2024. [Online]. Available: https://github.com/cucapra/calyx-firrtl-evaluation