Invoking and Linking Generators from Multiple Hardware Languages using CoreIR

Ross Daly
Stanford
rdaly525@stanford.edu

Lenny Truong
Stanford
lenny@stanford.edu

Pat Hanrahan
Stanford
hanrahan@cs.stanford.edu

ABSTRACT

In this paper, we advocate that the EDA community should invest in infrastructure that supports language interoperability and linking, which in turn will promote language diversity, a successful concept in the software world. We begin by observing how the software community has developed standards and infrastructure to support programming language interoperability, followed by discussion of how these techniques can be applied to hardware languages. In particular, this paper proposes applying the concept of software linking to the domain of hardware design. A key issue arises when considering the use of circuit generators, programs that consume parameters and produce hardware circuits. Circuit generators rely on the integration of a programming language and a hardware language, which complicates the notion linking. To address this, we propose the concept of the Foreign Generator Interface (FGI), an adaptation of the of the Foreign Function Interface from software, which enables the construction of hardware designs that use generators from multiple languages. Then, we show how FGI can be used to implement Staged Generator Compilation (SGC), which is a linking-based hardware compilation process inspired by multi-stage programming. Finally, we present CoreIR, a hardware intermediate representation and compiler framework that provides the requisite infrastructure for implementing FGI and SGC.

1 INTRODUCTION: THE FUTURE OF HARDWARE DESIGN

The open-source movement has fostered the collaboration of industry, academia, and hobbyists in developing a diverse set of tools for building software. Considering the success of the open-source model, the researchers and engineers responsible for developing the next generation of EDA tools should commit to the same principles of open collaboration and look to what allowed open collaboration to thrive in the software community

One fundamental component of open collaboration is interoperability, a system design requirement that defines a mechanism with which software tools can interact. When two software systems can interoperate, this results in the merging of two communities of developers and users, who now mutually benefit from the improvements of either system. Another fundamental component is flexible compilation flows like staged compilation. In this paper, we examine the current status of interoperability of hardware languages and propose the requirements of a system that would help move the community towards more language and library diversity. Then, we introduce CoreIR, an open-source intermediate representation and compiler framework that is designed specifically to meet these requirements.

1.1 Vision for the future

The future of open-source EDA tools should include a diverse landscape of language and libraries. This claim is motivated by the success of language diversity in the open-source software community. Software developers are afforded the luxury of choosing the right language for the task at hand, and can compose multiple languages into a single application to balance productivity and performance. In contrast, hardware projects are limited in language choices.

In order to define a path towards a future of hardware language diversity, we observe three ways in which the software community has promoted diversity through language interoperability. First, software languages provide foreign function interfaces, which enable programs to transfer control between program fragments written in different languages. Next, data exchange formats such as JSON provide a means for programs written in different languages to communicate data. Finally, the design of modern software compilers, intermediate representations, and build systems have been optimized to support staged compilation which allows the assembly of program fragments written in multiple languages through a process called linking.

2 BACKGROUND

2.1 Hardware Generators

A *hardware generator* is a program that consumes parameters and produces a hardware circuit. Verilog has primitive generator capabilities in the form of parameterized modules, which can be used to parameterize port widths and make simple instancing decisions using the `generate` statement. However, using parameters to describe optional ports and more complicated instancing and wiring patterns cannot be done. Newer systems overcome these limitations by using more expressive hardware generators for both the circuit *interface* and the circuit structure. Genesis2 [8] presented the concept of a *chip generator*, which uses Perl-based template metaprogramming to construct Verilog strings. Genesis2 enabled more sophisticated parametrization not

found in Verilog and VHDL, such as the optional ports described earlier. Chisel [1] built on this work by embedding a hardware construction language into a general purpose programming language. This approach enabled generator implementations to leverage a larger set of metaprogramming capabilities beyond text-based templating. Furthermore, the embedding approach enabled designers to incorporate programming language concepts such as objected orientation, functional programming, and parameterized types into their hardware generators.

2.2 Hardware Language Interoperability

Currently there exists poor support for interoperability in the hardware language ecosystem. Verilog allows the instantiation of VHDL modules, and vice versa, but compiler support is limited. Since Verilog is the standard input to most EDA tools, the current practical mechanism for using non-Verilog languages is to maintain complex, brittle, project-dependent scripts that generate and combine Verilog files. This is further complicated when trying to swap different implementations of complex generators like memories. IP-XACT [2] is an XML schema for specifying information about reusable circuit components such as file lists, top level ports, protocols, and parameters. This is a good step towards general interoperability, but still requires complex per-project scripts to utilize circuits.

2.3 Software Language Interoperability

In software, language interoperability is treated as a first class citizen. Developers have access to tools and techniques that enable them to compose multiple languages together to construct a single application. One can use a high level language like Python to perform simple glue operations such as processing web requests, composed with a library written in low level languages optimized for performance. The technologies underlying tooling support for interoperability are data exchange formats, foreign function interfaces (FFIs), a common intermediate representation, and linking.

Common data exchange formats such as JSON, XML, or protobuf specify data serialization schemes that includes support for useful data structures. The software community has collaborated in the development of open-source libraries and Application Programing Interfaces (APIs) supporting these formats, making them easy to use in many languages.

A Foreign Function Interface (FFI) is a mechanism for a programming language like Python to directly call functions in another language like C. This is typically accomplished by having the FFI adhere to a specific Application Binary Interfaces (ABI) which specifies how data and program control can be transfered. The success of tools like Python's numpy [7] is directly enabled by using an FFI to interoperate between C and Python.

```
module counter(input clk, input en, output [22:0] cnt);
  (* mylib *)
  Register #(.width(23), .has_en(1)) reg_inst(
    .clk(clk),
    .in(cnt + 1'b1),
    .out(cnt),
    .en(en)
  );
endmodule
```

Figure 1: A Verilog counter module that instantiates the register generator defined in Figure 2. The generator parameters, `width` and `has_en`, are described using Verilog's parameter syntax.

```
# In library mylib
def Register(width, has_en):
   IO = {}
   IO['in'] = In(Bits(width)),
   IO['out'] = Out(Bits(width)),
   IO['clk'] = In(Bit)
   if has_en:
      IO["en"] = In(Bit)
   return NewModule("Reg", IO)
```

Figure 2: Python implementation of a hardware generator that produces a register which is parameterized by `width`, and `has_en` a boolean flag indicating whether to include an enable port.

A common intermediate representation such as LLVM [6] enables interoperability and reuse by providing a common compilation target for different programming languages.

In software, linking is essential to supporting modularity. It enables programs to refer to functions even if the function implementation is not known during compilation. As a result, programs can refer to functions from libraries that may have been written in a different language. To create and run the final executable, a linker is used to resolve the references, and is therefore a key component of the software language interoperability tool chain

3 FOREIGN GENERATOR INTERFACE

In this section, we present the notion of a *Foreign Generator Interface* (FGI), a generalization of the foreign function interface to fit the needs of composing hardware generators written in multiple languages.

To better understand why we must extend the capabilities of a foreign function interface, consider the basic example of designing the Verilog counter in Figure 1. The implementations refers to the register generator implemented

in Python shown in Figure 2. To support this style of multi-language generator composition, we have identified three key capabilities required of the generator infrastructure:

(1) The Verilog design needs a syntactic mechanism for referencing the register generator defined in Python.
(2) The Verilog compiler needs to parse the generator reference, determine the values of the parameters, and invoke the generator with the parameters.
(3) The generator needs to return the interface of the generated circuit to the Verilog compiler so it may be instanced and wired up as a component of the design.

Abstractly, this imples the generator infrastructure needs to support the passing of parameters from hardware language compilers to generators which could be written in a different language. It also needs to provide a mechanism for creating and returning generated circuit interfaces from the generator to the compiler. And finally there needs to be conventions for defining the software environment to run generators. The FGI is a specification for these mechanisms.

An existing data exchange format such as JSON does provide a mechanism to pass data between two languages. However, we argue that such a mechanism needs to be augmented for additional data types commonly used as hardware parameters such as a fixed width bit vector.

The specification for exchanging circuit interfaces could be considered analogous to a software Application Binary Interface (ABI). As such it should be designed to be compatible with the circuit type system of a general Hardware Design Language.

We also observe that the process for compiling the circuit in Verilog begins with an invocation of a Verilog compiler which is itself a compiled software binary. This means that:

(a) the compiler binary must be linked at compile time with the Python library, or
(b) the compiler must use run-time linking to dynamically load the Python library, or
(c) the compiler must launch a new process, call the generator via a command-line interface, then parse the result.

Option (a) requires that the compiler either be distributed with support for all languages and all libraries of foreign generators or recompiled for each project which may refer to a different set of libraries. This is not viable because it introduces design flow friction by requiring project-specific custom compilers and difficulty in adding new libraries

Option (b) avoids this issue by extending the compiler to support the required libraries on demand. However, this means that any generator must be accessible either through a dynamic library that a compiler can load at runtime.

Option (c) allows a generic way for generators to be called, but the compiler needs to know what program to use to run the generator. This can be alleviated by requiring generators to conform to a specific command line interface and common runtime environments.

4 STAGED GENERATOR COMPILATION

One decision regarding foreign generators is whether the code is invoked immediately upon reference, so that the generated circuit implementation is available, or if the invocation of the generator is deferred to some later stage. The simplest approach is to invoke the generator immediately upon reference, which is similar to calling an imported function in a Python program. This results in the immediate execution of code. Alternatively, deferring execution of the generator, which we refer to as Staged Generator Compilation (SGC), is analogous to the C compilation process where functions are declared and invoked in a program without needing the definition. The linker resolves references to function definitions at a later stage. The SGC pattern enables useful features such as

(1) Complex compilation passes to determine the exact values of generator parameters needed in a system like Diplomacy [3]
(2) Incremental recompilation with generators.
(3) Swapping out different implementations of generators. For example, one could swap out a simulation model of a memory with a technology-specific memory implementation
(4) Simplified synthesis utilizing a library of generators implementing primitives in terms of technology-specific primitives

One major issue with adapting the linking pattern to hardware is that circuit interfaces cannot just be declared like function interfaces. Instead, their interfaces must be computed based on parameters. This is similar to issues that would arise when trying to link templated C++ functions. So, to support Staged Generator Compilation, the infrastructure needs to provide a mechanism for declaring the interface of a generated circuit without requiring the implementation of the generator.

Two possible solutions for declaring generated circuits are:

(1) Declare the expected generated circuit interface in the host language. Then, at a later stage in compilation, verify that the declaration was correct.
(2) Require the generator to have an additional function to compute the interface without the circuit implementation.

Option 2 could be realized by separating the notion of a generator interface and implementation. This requires that a generator explicitly implements a particular interface. We think that both of these options should be possible to use. Even if a generator library did not implement (2), the generators could still be used eagerly and without linking using the FGI.

5 COREIR

In this section, we introduce CoreIR [4], a system designed to support an ecosystem based on Foreign Generator Interfaces and Staged Generator Compilation. CoreIR is an intermediate representation and compiler framework that facilitates the construction of hardware languages and promotes language interoperability. Heavily inspired by LLVM, CoreIR is written in C++, and language front-ends can use the C API to construct, inspect, and manipulate objects in the CoreIR intermediate representation. CoreIR provides a rich, clear, and simple set of IR nodes and connection semantics, a set of common hardware optimization passes, and different compilation targets; similar to Yosys [9], and FIRRTL [5]. In addition, for interoperability, CoreIR provides the following key services to the hardware language community:

(1) A common target for the development of language and DSL front-ends which results in the reuse of compiler internals such as optimizations and technology mapping.
(2) A common API for implementing Foreign Generator Interfaces in the host language of an embedded hardware language, along with a parameter passing spec.
(3) A simple, but expressive interface type system which serves as a wiring ABI.
(4) APIs for loading libraries and running generators.

To implement FGI, CoreIR defines the set high level parameter types listed below. Each of these types are accompanied by a standard JSON serialization scheme.

```
ParamType = Bool
          | Int
          | BitVector(N)
          | String
          | Json
```

These parameter types were chosen to cover a large variety of parameters seen in hardware languages, along with providing a mechanism for exchanging arbitrary data using the Json type.

The CoreIR type system is designed to provide enough expressibility to declare the interfaces of general circuits, while maintaining a simplicity and preciseness that serves as the foundation for composing circuits defined in different hardware languages. In order to promote diversity in front-end languages with different type systems, the CoreIR type system must be defined in a manner that many type systems can map to.

```
Type = BitIn | BitOut | BitInOut
     | Array(N, Type)
     | Record((string, Type)*)
```

The type system provides the following connection semantics: TypeA can only be connected to TypeB if and only if the recursive type structure is identical and all the leaf types are duals. `BitIn` is the dual of `BitOut` and `BitInOut` is the dual of itself. This type system is a superset of Verilog and VHDL, and is similar to that of Chisel.

To enable SGC, CoreIR has a first-class notion of circuit generators and interface generators within the IR. Circuit definitions can contain instances of generated modules whose generated code need not have been run. CoreIR also provides an API for loading, resolving, and calling generators from external libraries which allows for the user or compiler to make the decision about when to run generators.

6 PRELIMINARY RESULTS

This section details an experiment that demonstrates how CoreIR can be used to implement a circuit design that uses foreign generators. The experiment uses Verilog as a host language and Python as a foreign language which shows how the generated circuit can be instanced and wired like a normal Verilog module. The code for this experiment has been released [4] and is available for use under a permissive open-source license.

First, the Python generator was wrapped in a FGI-compliant dynamic library, and installed on the system. The Yosys Verilog compiler [10] was extended to support referencing foreign generators using Verilog's annotation syntax shown in 1. The compiler loads and calls the foreign generator using the CoreIR API, creates the instance using the interface returned by the Python generator, and then compiles to CoreIR. Finally, at a later stage, the generator is run and linked into the final design.

The CoreIR FGI abstraction allows the details of how to run the generator to be isolated into the creation and installation of the dynamic library rather than mixed in with the details of the host language. Easily wrapping, and installing existing and new generators as dynamic libraries is future work for a hardware library installation tool.

7 CONCLUSION

In this paper, we presented a case for hardware language interoperability and hardware linking motivated by the success of language interoperability in software. We discussed the requirements for a system that enables hardware language interoperability, which focused on the ability to compose and link hardware generators. Then we presented CoreIR, a system that attempts to meet these requirements, alongside an experiment showing a circuit design using a foreign generator. We hope that the ideas presented in this paper enable a more open source future for hardware design.

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 *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. IEEE, 1212–1221.

[2] Victor Berman. 2006. Standards: the P1685 IP-XACT IP metadata standard. *IEEE Design & Test of Computers* 23, 4 (2006), 316–317.

[3] Henry Cook, Wesley Terpstra, and Yunsup Lee. 2017. Diplomatic Design Patterns: A TileLink Case Study. In *First Workshop on Computer Architecture Research with RISC-V (CARRV'17)*.

[4] Ross Daly. 2017. CoreIR. http://coreir.org

[5] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press, 209–216.

[6] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[7] Travis E Oliphant. 2006. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA.

[8] Ofer Shacham. 2011. *Chip multiprocessor generator: automatic generation of custom and heterogeneous compute platforms*. Stanford University.

[9] Clifford Wolf. 2016. Yosys open synthesis suite.

[10] Clifford Wolf, Johann Glaser, and Johannes Kepler. 2013. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*.