Improving Hardware Reusability: Software Defined Hardware

Adam Izraelevitz, Jack Koenig, Richard Lin, Chick Markley, Jim Lawson, Christopher Celio, Colin Schmidt, Patrick Li, Elad Alon, Borivoje Nikolić, Jonathan Bachrach

Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
{adamiz, jack.koenig4, rlin, chick, ucbjrl, celio, colins, psli, elad, bora, jrb}@eecs.berkeley.edu

***Abstract*—Modern software engineering techniques such as reusable libraries and retargetable compilers have allowed high productivity and fast design cycles, but hardware development practices lag far behind. We hypothesize that better hardware construction languages and compiler infrastructures will help close the gap, and evaluate this approach with several processor designs written in the Chisel hardware construction language and compiled through the FIRRTL compiler infrastructure.**

***Index Terms*—RTL; Design; FPGA; ASIC; Hardware; Modeling; Reusability;**

Introduction

The end of Moore's law has slowed technology scaling, eliminating the associated power, performance, and area improvements. Given that a specialized hardware implementation has enormous energy/performance improvements over software, specialization is likely the future of hardware design. This trend will manifest in an increased demand for chip diversity containing different RTL designs.

Meeting this demand with existing methodologies will be difficult. Some companies require two years between initial idea and profitable silicon[1], as this process requires design space exploration, RTL development, and verification.

In contrast, software industry's fast design cycle allows for one engineer to go from idea to profitable code in under two weeks. What tricks can the hardware industry learn from the software community?

Software libraries enable code reuse and are pervasive throughout software development. Reusing code via libraries significantly reduces development time and costs of new applications. Reuse has other advantages as well—verification costs of libraries are amortized over all its uses.

In comparison, reuse is pitifully rare in hardware designs, and there does not exist any semblance of a standard library of hardware components. However, if hardware projects reused more code, engineers would spend less time designing and (more importantly) less time verifying the new design. Given these clear benefits, ***why don't hardware engineers write reusable libraries?***

The main contributions of this paper are as follows:

- ***Two hypotheses*** *to account for the stagnation of hardware library development*: As detailed below, we assert that (1) existing hardware-description-languages are deficient in supporting hardware libraries, and (2) that diverse underlying tradeoffs require RTL modifications, thus making general purpose RTL less useful.
- *A reemphasis on* ***hardware construction languages (HCL)*** *as primary tools for* ***hardware libraries***: Previously, many influential works have introduced and expanded upon the concept of a hardware construction language. This paper revisits them in the sole context of providing a platform for which to develop reusable hardware libraries.
- *The concept of a* ***hardware compiler infrastructure (HCI)***: Like how software compilers transform general-purpose code into specialized assembly, we assert that a similar solution can transform general RTL into specialized RTL. By formalizing these transformations into a compiler infrastructure, we can enable robust and reusable RTL transformations.
- *An* ***evaluation*** *of whether HCLs and HCIs meet their claims of superior reusability*: We evaluate RocketChip, an existing hardware library written in Chisel[2] (an HCL) that uses FIRRTL[3] (an HCI) on its parameterization power, code expressivity, code reuse, and transformation reuse.

Two Hypotheses

Software libraries enable code reuse and are pervasive in software development. Reusing code through libraries significantly reduces development time of new applications. Modern software relies on thousands of libraries (Ubuntu 14.04 has >35,000 packages installed natively).

Reuse has a second advantage: the verification costs, not just the development costs, are amortized over all library uses.

In direct comparison, hardware designers don't commonly reuse modules from project to project, let alone develop extensive and reusable libraries of the magnitude seen in software. Recently, we have seen an increase in reusing large complex custom IP blocks, which has had many benefits[4]. However, this is a far cry from the current state of software library ubiquity.

To reiterate: ***why don't hardware engineers write libraries?***

Incorrect Hypotheses: We assert the lack of hardware libraries is not from a lack of effort—many companies have tried internally to establish standard libraries of hardware components to reuse among multiple projects. However, in our experience, these never gain traction and ultimately fail.

One may claim that software libraries are built upon a vibrant open-source community, and since hardware has fewer engineers and a very small open-source community, libraries have not had enough engineers to succeed. We counter this claim by example: D3[5], the popular JavaScript visualization library, was primarily written by a single engineer, but has still seen widespread use. Although a large community helps, writing a useful software library does not require one.

Clearly, there is something more fundamental in modern software engineering which has enabled writing reusable code.

Hypothesis 1—Existing HDLs are deficient: Modern advancements have made programming languages like Java, C++, Python, and Perl very powerful. Using features like object-orientation, polymorphism, and higher-order functions, one can write software using good software engineering principles, including abstraction, separation of concerns, and modularity. Good software is reusable software, and reusable software can easily be turned into a library used by others.

Logic synthesis successfully raised the level of abstraction most hardware designers use to write RTL; gone are the days of manual SPICE modeling and hand-drawn layout. Unfortunately, since the 1980s when the majority of current RTL languages were designed (Verilog, VHDL), programming languages have seen significant improvements. Existing HDLs are sorely lagging behind.

Take the following example: an adder reduction tree, parameterized by the number of elements. Verilog and VHDL cannot express recursive generate statements—instead, the designer must manually roll out the loop and do index calculations. Another example is a module that filters packets. Either the filter module must encode the filter condition (violating the principle of separating concerns) or a separate module must evaluate the filter condition (violating the principle of encapsulation). A modern software engineer would recognize that a higher-order function would elegantly solve this problem.

SystemVerilog is one attempt to improve on existing HDLs. Created in 2002, this language mixes modern ideas like object-oriented programming with classic Verilog elements. The result is a complicated language, intractable to support, confusing to learn, and still missing modern features like functional programming. To the authors' knowledge, no commercial SystemVerilog compiler implements the entire specification.

Hypothesis 2—Underlying complexity influences RTL: In spite of the success of logic synthesis, many underlying constraints still influence RTL design.

For example, in modern technologies, SRAMs are designed specially by the fabrication company, as their size and latency are extremely critical. RTL designers, to make use of these custom blocks, must manually instantiate them in their design. However, this locks the RTL into a specific technology node and eliminates reusability. Other constraints of ASIC technologies can also bleed into the RTL, including buffer insertion and custom layout.

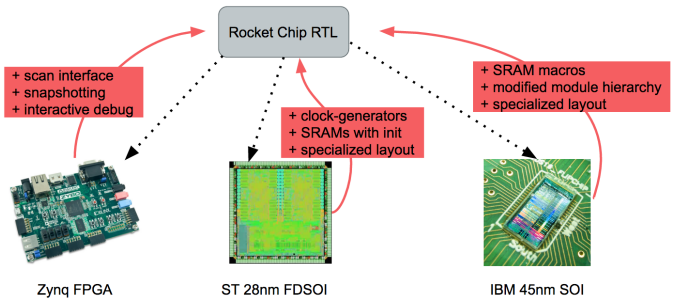

Fig. 1: Underlying Constraints

FPGAs are no different—many have hardened logic blocks to improve design quality. By modifying the RTL to be friendlier to the FPGA synthesis tools, a designer can receive a significant performance advantage. These changes, however, may be detrimental to an ASIC implementation of the RTL.

To address these issues, designers in industry have had to choose from a multitude of existing but insufficient options.

Manually changing the RTL limits reuse by obfuscating and specializing the RTL. CAD tool scripting is insufficient for inevitable unsupported use cases in custom flows. Company-specific closed-source RTL modifiers work but are not shared among academics or between companies. Finally, many designers write custom Python/Perl scripts to automatically edit their RTL. These scripts are not reusable, robust, or composable.

Hardware Construction Libraries

A hardware construction language is simply a software library whose classes mimic true RTL synthesizable constructs. For example, many HCLs have classes to represent registers, muxes, adders, wires, ports, modules, constants, etc. Note that the interpretation of each HCL hardware class is very simple.

```
abstract class HW { // Represents a synthesizable piece of hardware
  def emit: String // Emits corresponding Verilog representation
}
class Register(name: String, width: Int) extends HW { ...
  def connect(r: HW) = ...
  def emit = "reg [" + width + ":0] " + name + ";"
}
class Mux(cond: HW, ifTrue: HW, ifFalse: HW) extends HW {...}
```

Users can manipulate these classes by instantiating them and calling their methods. For example, a designer can instantiate the `Register` class, and call its connect method.

```
class Top { ... // Start of program
  val my_register = new Register("my_register", 32)
  my_register.connect(my_mux)
}
```

Once a design is written, the designer compiles and executes their code. During this execution, the HCL records the sequence of instantiations and method calls to directly build a datastructure representing the hardware design; this process is called *elaboration*. Finally, the string representation is emitted.

Through careful library design, an HCL can closely mimic the experience of writing in a hardware description language. Designers instantiate the HCL's `Register` class (instead of declaring a `reg` in Verilog) or the HCL's `Mux` class (instead of using Verilog's ternary operator).

HCLs themselves do not usually provide any new hardware abstractions; any hardware design written in Verilog can, line by line, be translated into calls and instantiations into the HCL, giving a designer complete control over their design.

Software Engineering Benefits: Many HCLs at baseline do not provide additional hardware abstractions; instead, their strength lies in allowing the designer to use all abstraction mechanisms of the underlying programming language that the HCL is implemented in.

For example, an HCL implemented in Python would be able to use a Python for-loop to instantiate a sequence of wires in replacement of a Verilog generate statement. An HCL

implemented in Java would be able to employ object-oriented verification instead of using SystemVerilog.

Ultimately, the benefit of any HCL is allowing the programmer access to a general purpose programming language to powerfully wrap, modularize, abstract, and encapsulate their construction of hardware.

Chisel—An Example HCL: Chisel is an open-source hardware construction language that is based in Scala [6], a modern object-oriented and functional programming language.

Users of Chisel have built hardware libraries that provide new abstractions, possibilities for parameterizations, and more.

Chisel users created a decoupled interface that supports forward and back-pressure. Built upon this abstraction are functions which generate control logic based off of the number of incoming and outgoing decoupled interfaces. A user of this decoupled interface library can make use of the interface at a higher abstraction level, without knowing the precise underlying Chisel-constructed hardware.

In the previous module filter example, a Chisel user can make a Filter module which takes, as a parameter, a higher-order-function that creates the condition-checking hardware. The user of this module can later, when instantiating the Filter, create and pass in a function which constructs their checking condition. Similarly, a Chisel user can write a recursive Scala function which generates the correct adder-reduction tree. Note that Scala, not Chisel, gives users this power and expressivity.

Hardware Compiler Infrastructure

Background on LLVM—A Modern Software Compiler: LLVM[7] takes general purpose code and specializes it for different machine types. Its structure consists of (1) frontends, (2) transformations, and (3) backends. A frontend parses a programming language into LLVM intermediate representation (LLVM-IR). Transformations then take LLVM-IR and modify it, returning new LLVM-IR. Finally, a backend generates its target ISA assembly, e.g. ARM or x86.

This structure of translating an input language into an intermediate representation enables reusing transformations among multiple compilers. If we applied this concept to hardware, what would it look like?

Hardware Compiler Infrastructure: A hardware compiler infrastructure (HCI) takes general purpose RTL and specializes it for different technology nodes and design requirements. Exactly like LLVM, its structure consists of frontends, transformations, and backends. Again like LLVM, it has an intermediate representation upon which all transformations are developed. This enables reusing transformations between different compilers.

FIRRTL—An HCI: FIRRTL stands for Flexible Intermediate Representation for RTL, and defines the intermediate representation for its corresponding hardware compiler infrastructure. Chisel directly generates FIRRTL, and then this intermediate form is simplified and optimized. Finally, the resulting FIRRTL is passed to a backend tailored for one of three specific targets: (1) a FIRRTL interpreter for simulation (FIRRTL); (2) an FPGA for fast emulation (Verilog); (3) a 28nm ASIC process node (Verilog).

Due to FIRRTL's modular framework structure, transformations are commonly reused in spite of a variety of underlying targets.

Preliminary Evaluation

This paper claims that HCLs and HCIs promote reuse, enable flexible hardware libraries, and enable targeting multiple simulation/emulation/fabrication technologies.

Given those assumptions, we would also expect that: (1) HCL/HCI libraries will have powerful parameterization; (2) HCL/HCI codebase will have less code than similar HDL codebases; (3) an HCL/HCI project will reuse more code than it authors; (4) an HCL/HCI project can reuse transformations when targeting different technology nodes, FGPAs, and software simulation.

Parameterization: While difficult to qualitatively evaluate the flexiblility, magnitude, and degree of parameterization that the general purpose programming language provides an HCL, we can qualitatively evaluate the parameters of an existing hardware library, written in an HCL.

RocketChip [8] is an open-source hardware library, written in Chisel, that can generate many different instantiations of a symmetric multi-processor system (SMP).

Some of the parameters available are:

- **Out-of-order parameters:** fetch width (1, 2, 4), issue width (1, 2, 3, 4), branch predictors (BTB, GShare, TAGE)
- **Data parallelism:** number of parallel data operations (4 through 32), precision (half, word, double)
- **Multi-core:** number of cores (1, 2, 4, 16)
- **Datapath:** 5-stage pipeline, 3-stage pipeline, microcoded multi-cycle
- **Cache:** size (64KB to 2MB), associativity (direct-mapped, two-way), type (scratchpad, blocking, non-blocking)

Note that the cross product of these parameters are all valid, and many (but not all) of these design points have been experimented with or even realized in silicon.

Furthermore, many parameters are not simply bit-widths, but impact the control logic, interface definitions, and communication protocols. The generated hardware is therefore very different for each design point, and this is only possible given the power of the underlying programming language, Scala.

Expressiveness: Using software engineering methods enabled by modern programming languages, we should expect fewer lines of code to express similar projects.

OpenPiton [9] is a research project in Verilog that uses OpenSparc cores with a custom interconnect and coherency framework.

OpenPiton and RocketChip have many similarities from 10,000 feet—both are SOC generators, containing cores, caches, network protocols, coherency domains, tests, and much more. Both are used for computer architecture research, have been realized in silicon, and boot Linux.

Transforms	28nm ASIC	FPGA Emulation	Simulation
Constant Propagation	x	x	x
Dead Code Elim.	x	x	x
Sub-Expression Elim.	x	x	x
FIRRTL Backend			x
Verilog Backend	x	x	
Decouple Target Time		x	
Scan Chains		x	
SRAM Replacement	x		

TABLE I: Different downstream targets can still reuse many FIRRTL transformations.

A cursory look at the size of the projects reveals a startling difference—RocketChip contains 40,000 lines of Scala, while OpenPiton has 264,000 lines of Verilog (6.7x more code).

While clearly an apples-to-oranges comparison, the sheer magnitude of code-size differences between OpenPiton and RocketChip cannot be explained solely by their differing feature sets. We believe the choice of a HCL over an HDL has given RocketChip a significant expressiveness advantage.

Hardware Reusability: A major claim in this paper is that HCLs and HCIs enable writing and reusing hardware libraries. We analyzed BOOM [10], an out-of-order machine written in Chisel, to understand how much of its code was actually reused. As shown in Figure 2, BOOM uses more external code (modules from other projects) than internal code.

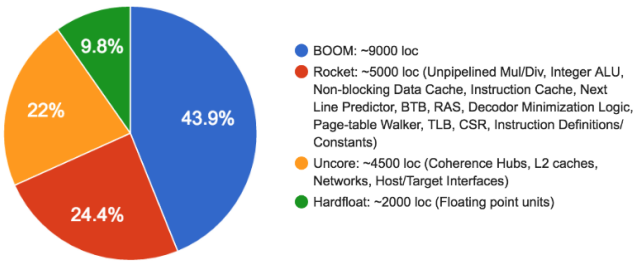

Fig. 2: Lines of code used by BOOM, grouped by project: BOOM (out-of-order core), Rocket (in-order core), Uncore (L2 cache and networks), and Hard oat (oating point units).

BOOM is not unique among reuse—our data-parallel accelerator, our three-stage core, and many other designs constantly reuse one-another's codebases. In addition, using approximately 20,500 lines of code to implement a correct out-of-order implementation is extremely impressive.

Transformation Reusability: We claim HCIs give transformation reusability, and the preliminary results in Table I that depict transformation reuse qualitatively demonstrates this fact.

We are currently converting more existing manual changes and downstream scripts into this framework, and as a consequence these results are preliminary.

Related Work

As opposed to HCLs, other work has used modern languages to act as a macro processing language for an underlying HDL, including Genesis2 [11], JHDL [12], and HML [13]. Other HCLs include MyHDL [14] and Bluespec [15]. Yosys [16] is an open-source framework for Verilog RTL synthesis, and maps Verilog to ASIC standard cell libraries or Xilinx FPGAs. Unlike FIRRTL, its main focus is logic synthesis, not RTL to RTL transformations.

Conclusion

In summary, we contributed two hypotheses accounting for the stagnation of hardware library development, reemphasized HCLs ability to enable hardware libraries, and established the concept of a hardware compiler infrastructure (HCI).

Our evaluation demonstrated the ability of HCLs and HCIs to support powerful hardware parameterization, new hardware abstractions, and expressive hardware construction.

Specialization is the future of hardware design, and increasing reusability within our hardware design methodologies is critical to meeting the incoming demand for chip diversity.

Acknowledgements

Research partially funded by DARPA Award Number HR0011-12-2-0016; the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA; and ASPIRE Lab industrial sponsors and affiliates Intel, Google, HPE, Huawei, LGE, Nokia, NVIDIA, Oracle, and Samsung.

References

[1] "Accelerating time to market - RocketSim - rocketick." http://www.rocketick.com/rocketsim/accelerating-time-to-market, 2017.

[2] J. Bachrach, H. Vo, B. Richards, Y. Lee, *et al.*, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, pp. 1216–1225, ACM, 2012.

[3] P. S. Li, A. M. Izraelevitz, and J. Bachrach, "Specification for the firrtl language," Tech. Rep. UCB/EECS-2016-9, EECS Department, University of California, Berkeley, Feb 2016.

[4] "IP vendor selection." http://armipexchange.com/Vendor/ARM, 2017.

[5] M. Bostock, V. Ogievetsky, and J. Heer, "D #x0b3; data-driven documents," vol. 17, no. 12, pp. 2301–2309.

[6] M. Odersky, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger, and et al., "An overview of the scala programming language," tech. rep., 2004.

[7] C. Lattner and V. Adve, "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*, CGO '04, (Washington, DC, USA), pp. 75–, IEEE Computer Society, 2004.

[8] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, *et al.*, "The rocket chip generator," Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[9] J. Balkind, M. Mckeown, Y. Fu, T. Nguyen, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, "OpenPiton : An Open Source Manycore Research Framework," in *ASPLOS*, 2016.

[10] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," Tech. Rep. UCB/EECS-2015-167, EECS Department, University of California, Berkeley, Jun 2015.

[11] O. Shacham, S. Galal, S. Sankaranarayanan, M. Wachs, *et al.*, "Avoiding game over: Bringing design to the next level," in *DAC Design Automation Conference 2012*, pp. 623–629, June 2012.

[12] P. Bellows and B. Hutchings, "JHDL-an HDL for reconfigurable systems," in *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*, pp. 175–184, Apr. 1998.

[13] "HML, a novel hardware description language and its translation to VHDL," *ResearchGate*.

[14] Jan Decaluwe, "MyHDL Manual," July 2016.

[15] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," in *Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings*, pp. 69–70, June 2004.

[16] C. Wolf, "Yosys open synthesis suite." http://www.clifford.at/yosys/.