Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations

Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar,
Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, Jonathan Bachrach
Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
{adamiz, jack.koenig3, psli, rlin, angie.wang, magyar, dgkim, colins, chick, ucbjrl, jrb}@berkeley.edu

***Abstract*—Enabled by modern languages and retargetable compilers, software development is in a virtual "Cambrian explosion" driven by a critical mass of powerfully parameterized libraries; but hardware development practices lag far behind. We hypothesize that existing hardware construction languages (HCLs) and novel hardware compiler frameworks (HCFs) can put hardware development on a similar evolutionary path by enabling new hardware libraries to be independent of underlying process technologies including FPGA mappings. We support this claim by (1) evaluating the degree with which Chisel, an existing HCL, can support powerfully parameterized libraries, and (2) introducing the concept and implementation of an HCF that uses an open-source hardware intermediate representation, FIRRTL (Flexible Intermediate Representation for RTL), to transform target-independent RTL into technology-specific RTL. Finally, we evaluate many hardware compiler transformations, including simplifying transformations, analyses, optimizations, instrumentations, and specializations, which demonstrate the power of a combined HCL and HCF approach.**

***Index Terms*—RTL; Design; FPGA; ASIC; Hardware; Modeling; Reusability; Hardware Design Language; Hardware Construction Language; Intermediate Representation; Compiler; Transformations; Chisel; FIRRTL;**

I. Introduction

The end of Dennard scaling and slowing technology advances have eliminated the associated "free" power, performance, and area improvements for digital circuits. Since specialized hardware implementations have enormous energy and performance improvements over software on a general-purpose processor, specialization is likely the future of hardware design [1][2][3]. This trend will manifest in an increased demand for diverse products containing different specialized RTL. Meeting this demand with existing methodologies has proven difficult [4].

In contrast, the software industry has much faster design cycles than the hardware industry; a small team can go from idea to profitable software in under two weeks. *What can the hardware industry learn from the software community?*

A key contributor to the software industry's productivity is reusable libraries, which amortize development and verification costs of new applications. These libraries are built upon expressive languages with retargetable compilers that perform platform-specific optimizations on general-purpose code.

In comparison, hardware reuse is relatively rare; no widespread reusable hardware library exists. However, if hardware projects reused more code, engineers might spend less time designing and, more importantly, less time verifying the new design. Since the benefits of reusing code are clear, ***why don't hardware engineers write reusable libraries?*** The main contributions of this paper are as follows:

- ***Two hypotheses*** *accounting for the stagnation of hardware library development:* We assert that (1) existing hardware description languages lack the expressivity to support hardware libraries, and (2) diverse underlying implementations require RTL customization, limiting reusability.
- *A re-emphasis and analysis on* ***hardware construction languages (HCLs)*** *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 hardware libraries.
- *An* ***open-source implementation*** *of a* ***hardware compiler framework (HCF)*** *to isolate RTL from implementation constraints*: As software retargetable compilers transform general-purpose code into platform-specific assembly, HCFs transform general RTL into target-specific RTL. By formalizing these transformations into a compiler framework, we can enable robust and reusable RTL transformations.
- *An* ***evaluation of many transformations*** *that demonstrates the wide-ranging applicability of our framework*: Our HCF implementation employs a hardware intermediate representation, FIRRTL (Flexible Intermediate Representation for RTL), as the basis for many different transformations including simplifying transformations, analyses, optimizations, instrumentations, and specializations.

II. Two Hypotheses

Software libraries are pervasive in software development because, through code reuse, they reduce development and verification costs of new applications. Modern software relies on thousands of libraries—Ubuntu 14.04 has approximately 35,000 packages installed natively.

In direct comparison, hardware designers do not commonly reuse modules from project to project, let alone develop extensive and reusable libraries.

Other attempts to reuse hardware has had mixed success. Increased reuse of large complex custom IP blocks at the SoC-level has had many benefits including faster time to market and reduced verification effort. However, custom IP blocks are usually very specialized, as opposed to being basic building blocks of hardware like queues, arithmetic units, multipliers, caches, and so on, and pose more integration challenges than a typical reusable library. To reiterate: ***why don't hardware engineers write reusable libraries?***

A. Incorrect Hypotheses

One could claim the lack of hardware libraries comes from a lack of effort; yet in the authors' experience, many companies try, but fail, to establish internal reusable libraries of hardware components.

One could also claim the lack of hardware libraries comes from a lack of an open-source community; yet, popular open-source software is often written by one or two contributors. D3[5], the popular JavaScript visualization library, was primarily written by a single engineer, but has still seen widespread use.

B. Hypothesis 1—Existing HDLs lack expressivity

Programming languages have seen significant improvements since the 1980s when the majority of popular hardware description languages (HDLs) were designed (Verilog, VHDL). Modern advancements in mainstream programming languages have made languages like Java, C++, Python, Perl, and Ruby very powerful. Object-orientation, polymorphism, and higher-order functions enable the use of good software engineering principles like abstraction, separation of concerns, and modularity; these ultimately encourage and enable code reuse. HDLs have been very slow to adopt these paradigms.

978-1-5386-3093-8/17/$31.00 ©2017 IEEE

An adder reduction tree illustrates this problem: Verilog and VHDL cannot express recursive generate statements, so a designer must manually unroll the loop and calculate indices for every instance. The lack of parameterization precludes re-use when a tree of different width is required.

Another example is a module that filters packets. Either the filter module or an external module must encode the filter condition. The first approach violates the principle of separation of concerns, while the second violates encapsulation. However, higher-order functions provide an elegant software engineering solution to the problem.

SystemVerilog, created in 2002, attempts to improve on existing HDLs by mixing in modern ideas like object-oriented programming with classic Verilog elements. The result is an extremely complicated language—intractable to support and confusing to learn—that is still missing other modern features like higher-order functions. To the authors' knowledge, no commercial SystemVerilog compiler implements the entire specification [6].

High-level synthesis (HLS) takes a different approach by having the user design in a higher level language, with a compiler translating down to RTL. The input language can be C-like [7][8][9][10][11], a parallel C-like language [12][13][14], general purpose [15], or domain specific [16][17][18][19][20][21]. Many HLS tools are evaluated on simplicity of use, performance relative to a hand-coded implementation, succinctness, and resource footprint; their ability to foster reusable hardware libraries is not usually considered.

Unfortunately, HLS approaches suffer from two competing concerns: (1) expressive source languages enable better software engineering (and more reuse); (2) expressive source languages are more difficult to translate to hardware and create more compilation/abstraction layers that hinder users who fine-tune a design.

C. Hypothesis 2—Underlying complexity requires RTL customization

In spite of the success of logic synthesis, many underlying constraints still influence RTL design.

ASIC implementations often require RTL customizations. For example, Verilog lacks an explicit memory construct; users must use a register array. In modern technologies, SRAMs are provided by the fabrication company because large memories often contribute to a design's critical path. RTL designers must rewrite their design to replace these register arrays with black-boxed SRAMs; this eliminates any future reuse that does not use this ASIC technology or performance envelope.

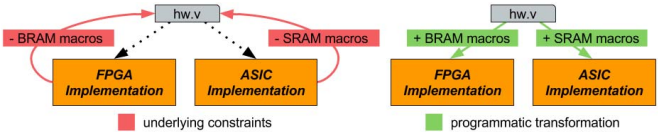

Fig. 1: Underlying constraints for ASIC versus FPGA implementations means the same RTL cannot get good results on both platforms. This limits the reusability of any RTL design. To solve this problem, programmatic RTL transformations must take generic RTL and specialize it for a given platform.

FPGA implementations are no different; many FPGAs have hardened logic blocks to improve design quality. A designer can receive significant performance, power, or utilization advantages by modifying their RTL to be friendlier to a particular FPGA's synthesis tool. These changes, however, may be detrimental to an ASIC implementation or another FPGA implementation.

To solve this problem, some designers write a collection of custom scripts to do ad-hoc programmatic RTL modifications; these scripts are neither reusable, robust, nor composable.

Commercial CAD tools do not completely solve this problem either. While some contain RTL-to-RTL transformations, CAD tools primarily focus on synthesis and place-and-route. They are also not organized in an open-source compiler framework and are insufficient for custom flows that may have unsupported use cases.

One exception is Yosys[22], an open-source framework for Verilog RTL synthesis which maps Verilog to ASIC standard cell libraries or Xilinx FPGAs. Yosys's main focus is logic synthesis, not RTL to RTL transformations. As such, its internal design representation is very low level and cannot represent higher-level constructs like aggregate types, width inference, and conditional assignment.

Separate from CAD tools, there exist stand-alone RTL modifiers, but many are closed source[23] and, like commercial CAD tools, do not easily support custom flows. An exception is PyVerilog[24], which is an RTL-to-RTL modifier tailored specifically to Verilog. As such, it makes it difficult to act upon designer intent that is not directly represented in a Verilog construct. PyVerilog does not support SRAM inference or aggregate types, and these features would be very difficult to support given its internal circuit representation. Another exception is Verific[25], a commercial tool that parses Verilog/VHDL designs and enables users to write transformations based on either the source AST or a language-independent netlist format. Because Verific supports every detail of these languages, a custom transformation must either support all language details (which can increase transformation complexity), or operate on the netlist format (which lacks designer intent).

III. Hardware Construction Languages, Hardware Compiler Frameworks, and Transformations

We assert that expressive languages and programmatic customizations enable reusable libraries. This section emphasizes how hardware construction languages (HCLs) enable expressive hardware designs and how hardware compiler frameworks (HCFs) enable programmatic customizations. We then introduce our open-source HCF implementation and its intermediate representation (IR) and show example transformations to demonstrate its wide applicability.

A. Hardware Construction Languages for Hardware Libraries

Hardware construction languages (HCLs) embed HDL-like hardware primitives in an existing programming language. ***Because an HCL directly uses RTL abstractions, there are no performance/area overheads for using an HCL over any HDL***. HCL designers use the general-purpose language's rich control structures and abstractions to create modular, parameterizable, reusable, and performant designs compared to a equivalent HDL design [26][27][28][29][30][31][32].

Chisel[33] is an open source[1] HCL that is hosted in Scala[34], a modern object-oriented and functional language.

1) HCL structure: All HCLs are software libraries with interfaces for constructing synthesizable RTL. To illustrate, the following toy example HCL has classes representing registers or muxes:

```
// Represents synthesizable piece of hardware
abstract class HW {
  // Emits corresponding HDL representation
  def emit: String
}
class Register(name: String, width: Int)
   extends HW { ...
  def connect(r: HW) = ...
  def emit = s"reg [${width-1}:0] $name;"
}
class Mux(cond: HW, ifTrue: HW, ifFalse: HW)
   extends HW {...}
```

[1]Available: https://chisel.eecs.berkeley.edu/

A designer can then create a register and hook it up by instantiating the `Register` object and calling its `connect` method:

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

Language features like operator overloading can also cut verbosity:

```
class Top { ... // Start of program
  // Equivalent to my_reg.connect(my_mux)
  my_reg := my_mux
}
```

By executing this HCL code, one generates the complete design; this process is called *elaboration*. Each HCL method call builds an underlying data structure representing the hardware design instance. This design can then be emitted to an existing HDL. Developing in a well-designed HCL can closely mimic the experience of writing in an HDL.

2) Enabling Hardware Libraries: **HCLs *by themselves* do not provide any new hardware abstractions**. However, host language features allow designs to be more parameterizable and modular.

For example, Chisel users can write a recursive Scala function to construct an adder-reduction tree, parameterized on bit-width. Unlike the explicitly unrolled version necessary in Verilog, the same generator could be re-used anywhere an adder tree is desired.

Similarly, a Chisel designer can write a filter module which takes, as a parameter, a higher-order-function that creates the condition-checking hardware. The user of this module then only needs to write the filtering condition, re-using the base filter structure.

Ultimately, an HCL's host language expressiveness is what enables reusable hardware library development.

B. A Hardware Compiler Framework

As software compilers transform general-purpose code into specialized assembly, a hardware compiler transforms general RTL into specialized RTL. By collecting these transformations into a compiler framework, we can enable robust and reusable RTL transformations.

The central part of any compiler is its intermediate representation (IR), upon which all transformations operate. This section gives an overview of our hardware compiler framework and the design of its open-source[2] IR, FIRRTL.

1) HCF Structure: Modern software compiler frameworks like LLVM[35] consist of (1) frontends, (2) transformations, and (3) backends. A frontend parses programs written in a specific programming language (e.g. C++ or Rust) into a compiler-specific IR. IR-to-IR transformations such as optimization passes then can operate on and modify the program's structure. Finally, a backend converts the IR into a program in the target ISA, e.g. ARM or x86. This structure of translating an input language into an IR enables reusing transformations among multiple designs and languages.

Our HCF is similarly structured: Chisel and Verilog frontends translate designs into FIRRTL, transformation passes provide simplification, optimization, and instrumentation, and the resulting FIRRTL can either be simulated directly or passed to one of many Verilog backends tailored for simulators, FPGAs, or ASIC technology processes.

2) FIRRTL Design Justification: Designing an IR is an important part of any compiler, and we evaluate IRs on these three desirable, yet sometimes competing, qualities:

[2]Available: https://github.com/freechipsproject/firrtl

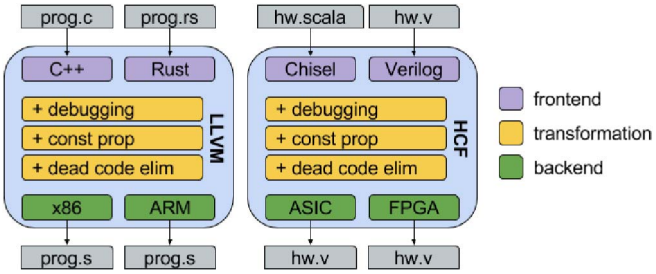

Fig. 2: LLVM can create a C++-to-x86 compiler or a Rust-to-ARM compiler, yet share internal transformations on LLVM-IR. Similarly, our HCF can create a Chisel-to-ASIC-Verilog compiler or Verilog-to-FPGA-Verilog compiler and share internal transformations.

- *clear*: semantically straightforward
- *simple*: small set of IR nodes
- *rich*: captures user-intent

All tools that manipulate RTL- or gate-level designs have an IR that they operate on, whether rigorously defined or not. Each tool's IR makes differing tradeoffs depending on their use: an IR for operating only on behavioral Verilog-2005 should be more rich but less clear and simple than an IR operating solely on netlists.

Our hardware IR, FIRRTL (Flexible Intermediate Representation for RTL), represents RTL digital circuits and is designed to specialize source RTL code from underlying implementations[36]. As such, FIRRTL first prioritizes richness to capture as much source RTL user intent as possible. For example, FIRRTL contains explicit memory nodes, aggregate types, a clock type, and typesafe connections to enable other languages, like Chisel, to map to these constructs and to capture the user's intent.

Since our HCF must eventually emit a less-rich representation for downstream simulators and tools, FIRRTL is also simple. Finally, FIRRTL is clear because it is rigorously defined and has straightforward width inference and type inference rules.

3) FIRRTL Overview: FIRRTL defines hardware modules for encapsulation, registers and memories for state elements, and primitive operations and muxes for combinational logic.

To bridge the gap between capturing user intent and downstream formats, FIRRTL consists of three well-defined forms (*high form*, *middle form*, *low form*) where each uses a smaller, stricter and simpler subset of FIRRTL features than the previous form. FIRRTL's *low form* contains the set of low-level features that map directly to Verilog constructs with straightforward semantics on a variety of targets.

Any transformation can specify which FIRRTL form it consumes, but can always emit a higher form that is subsequently lowered. Less-rich inputs have fewer corner cases, and transformations that modify/generate FIRRTL are simplified with access to rich IR features.

4) In-Memory FIRRTL Representation: The in-memory structure of a FIRRTL design significantly influences how easily transformations are written. As is commonly done in software compilers, a FIRRTL design is internally represented with an abstract syntax tree (AST) structure, where transformations recursively walk nested elements to manipulate the AST. If non-local information is necessary, transformations first walk the tree to build a custom data structure, then walk the tree a second time to manipulate the AST.

Some transformations may require other representations of the design. Combinational loop detection, for example, operates on a netlist-like directed graph. Our compiler framework has an accompanying library, which transformations can use to build these data structures.

The FIRRTL AST consists of IR nodes represented by an in-memory object, each of which is a subclass of one of the following

IR abstract classes: *circuit*, *module*, *port*, *statement*, *expression*, *type*. Each IR node can have children objects of other IR node classes, the relationship of which is shown in Figure 3. Figure 4 demonstrates how a FIRRTL circuit is represented in-memory as an AST of IR nodes.

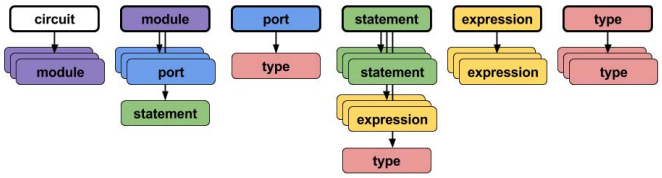

Fig. 3: A FIRRTL circuit is represented using these AST nodes. Each can have one or many different children nodes of various types. For example, a FIRRTL *statement* can have children *statements*, *expressions*, and/or a *type*.

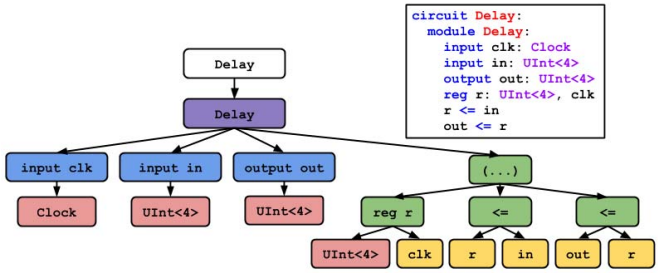

Fig. 4: An example FIRRTL circuit in its AST versus textual representation. This circuit contains a single module that outputs the input signal delayed by one cycle. The `(...)` statement is a block statement that only contains multiple children statements - this node makes it easy to replace a single statement with multiple statements in a single walk of the AST.

The following recursive algorithm visits all *expression* nodes in a circuit: First, visit each module's *statement* nodes. For each visited *statement*, visit each of its children *statement* and *expression* nodes. For each visited *expression*, visit each of its children *expression* nodes.

All transformations use these recursive walks of the FIRRTL AST to modify the circuit.

C. FIRRTL Transformations

Transformations always consume and produce a well-defined AST circuit and easily connect one-after-another. Constraints on the design can be checked after each transformation. This structure makes inserting new transformations straightforward and safe, unlike the use of brittle, ad-hoc scripts.

1) Example Transformation: To express a recursive walk, every IR node has implemented a custom **map** function; a node's **map** applies a user-specified function to the subset of children whose node-type matches the function's input and return node-type.

The following example demonstrates calling a *module*'s **map** with a function that accepts and returns a *port*, and with a function that returns a *statement*.

```
def onP(p: Port): Port = ...
def onS(s: Statement): Statement = ...
val myMod = Module(..., Seq(port1, port2), stmt)
val m1 = myMod.map(onP)
// m1 is Module(..., Seq(onP(port1), onP(port2)), stmt)
val m2 = myMod.map(onS)
// m2 is Module(..., Seq(port1, port2), onS(stmt))
```

While simple, using **map** to recursively walk the FIRRTL AST is extremely powerful.

The following example is an optimization transformation that does constant propagation over muxes with constant predicates. We walk all FIRRTL modules, statements, and expressions recursively by calling **map** on modules, statements, and expressions. For any mux we see, we check our constant propagation condition and, if true, perform the optimization. Note that this code visits expressions in postorder traversal, requiring only one pass through the AST.

```
class SimpleConstProp extends Transform {
  def inputForm = HighForm
  def outputForm = HighForm
  def execute(state: CircuitState): CircuitState = {
    state.copy(circuit = state.circuit.map(walkMod))
  }
  def walkMod(m: DefModule): DefModule = {
    m.map(walkStmt)
  }
  def walkStmt(s: Statement): Statement = {
    s.map(walkStmt).map(walkExp)
  }
  def walkExp(e: Expression): Expression = {
    e.map(walkExp) match {
      case Mux(UIntLiteral(x,_),t,f,_) if x == 1 => t
      case Mux(UIntLiteral(x,_),t,f,_) if x == 0 => f
      case other => other
    }
  }
}
```

2) Simplification Transformations: Simplification transformations take a FIRRTL circuit and simplify it to a lower form. There are two simplification transformations: (1) *high-to-mid*, which takes in high form and emits middle form; (2) *mid-to-low*, which takes in middle form and emits low form.

For example, one task of the high-to-mid transformation is to remove FIRRTL's bulk-connect operator. This operator allows components with aggregate types to be connected in a type-safe manner with a single statement, capturing user intent. However, lower forms only support connections between primitive types, so the high-to-mid transform rewrites the bulk-connect into a series of individual connections.

3) Analysis Transformations: Designers often desire insight into the compiler to understand the degree of optimizations taking place. Node-counting, early area estimations, and module hierarchy depictions are three useful analysis transformations early in the design cycle.

4) Optimization Transformations: The three major optimization transformations implemented are constant propagation, common subexpression elimination, and dead code elimination. Because downstream tools do aggressive logic analysis and other optimizations, these transformations have little effect on the gate-level design, but are critical for code readability.

5) Instrumentation Transformations: Our HCF's modular structure makes it straightforward to add simple instrumentation passes. These can include inserting hardware counters, hardware assertions, or even improving simulation line coverage detection.

6) Specialization Transformations: Different backend targets, especially FPGAs and ASIC process nodes, require RTL modifications to achieve good results.

To solve Verilog's memory problem described in Section II-C, Chisel has a high-level memory construct that directly emits a FIRRTL memory. Our memory transformations either emit a register array or a technology-specific SRAM macro. For FPGAs, we can emit stylized Verilog to ensure the BRAMs are correctly inferred, as well as enable targeting hard macros by directly instantiating FPGA templates. Other ASIC backend specializations include integration

with pad-frame libraries and fine-grained flattening, deduplicating, and moving of modules for floorplanning.

IV. Evaluation

This paper claims that HCLs and HCFs enable flexible hardware libraries by promoting code reuse and isolating source code from backend-specific optimizations/constraints. We demonstrate the following: (1) Chisel provides the necessary expressivity to support a high degree of parameterization and reusability; (2) our HCF implementation's wide-ranging applicability to customize RTL for many different use cases.

A. Chisel Support for Hardware Libraries

An expressive language requires fewer lines of code to more fully parameterize a design. This parameterization enables reusing the same code in different contexts with different parameters, potentially generating radically different hardware.

The following evaluates Chisel with regards to its expressiveness, parameterizability, and ultimately its reusability.

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

RocketChip[37] is an open-source hardware library[3], written in Chisel, that can generate many different instantiations of a symmetric multi-processor system (SMP). OpenPiton[38] is another open-source manycore processor and framework[4] used for research, written primarily in Verilog and enhanced with some Python-Verilog generation scripts, that uses an Sun UltraSPARC T1 (OpenSPARC) core 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. OpenPiton's core is a simple in-order design with multi-threaded capability, while RocketChip cores are either in-order (Rocket) or out-of-order (BOOM) but neither have multi-threaded capabilities. As shown in Table I, RocketChip cores have Coremark scores ranging from 2.42 to 4.49, while OpenSPARC has a Coremark score of 1.32[39]. For comparison, a similarly-sized industrial out-of-order core, the Cortex-A9, has a CoreMark score of 3.71[39].

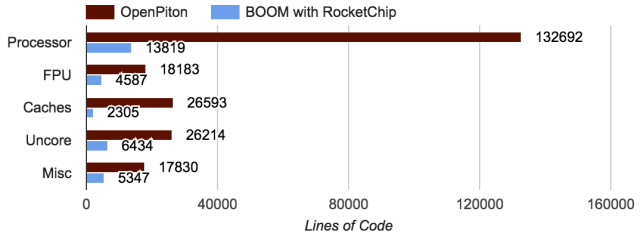

Fig. 5: Similar hardware structures show significant differences in code size, ranging from between 3x to 10x. Because of their differing feature sets, this evaluation should not be taken as a strict comparison; rather we interpret this as a general trend that using Chisel enables a more expressive coding style.

While clearly an apples-to-oranges comparison, Figure 5 depicts a comparison between the code bases. OpenPiton takes between 3x and 10x more code to express similar hardware structures; the sheer magnitude of code size differences between OpenPiton and RocketChip cannot be explained solely by their differing feature sets.

[3]Available: https://github.com/freechipsproject/rocket-chip
[4]Available: http://parallel.princeton.edu/openpiton/

Name	MicroArch	Coremark (per MHz)	Area (mm2)
Rocket	In-Order	2.42	1.62
BOOM	Out-of-Order	4.49	4.99

TABLE I: We pushed two different configurations through our compiler framework and synthesized them with the Synopsis SAED educational standard cell library[41]. The in-order core is a five-stage pipelined processor, while the out-of-order core is a 2-wide fetch, 3-issue integer pipeline, and 2-issue floating-point pipeline with a GShare branch predictor.

In addition, to the authors' knowledge, RocketChip's out-of-order core, BOOM[40], requires the fewest lines of code of any open-source out-of-order core implementation.

While we expect much of the OpenSPARC core was not entirely hand-written (tools like editor extensions could have been used) we feel the comparison of language expressivity remains valid: Chisel is clearly more expressive than Verilog as is shown by the significant reduction in code size for RocketChip.

2) Parameterizability: Parameterization precedes effective reusability - a flexibly parameterized module is more useful, and thus more reusable.

While it is difficult to quantitatively evaluate the flexibility, magnitude, and degree of parameterization that a general purpose programming language provides an HCL, we describe qualitatively the type and degree of RocketChip's parameterizability:

- **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, 8, 16)
- **Cache:** size (64KB to 2MB), associativity (direct-mapped, two-way), type (scratchpad, blocking, non-blocking), coherence policy (MSI, MESI)

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 of these parameters are not simply bit-widths, but impact the control logic, interface definitions, and communication protocols. In Table I, two different parameterizations of our cores result in vastly different designs with very different microarchitectures, performance results, and area numbers.

3) Reusability: We analyzed three processors written in Chisel: (1) BOOM[40], RocketChip's out-of-order machine, (2) Rocket, a single-issue in-order core, and (3) DecVec, a decoupled vector co-processor, to understand whether parameterized designs foster reusability. As shown in Figure 6, approximately 5000 lines of code are shared with all three designs, and even more is shared between pairs of designs. In all, the three designs share half or more of their codebases with one another.

Fig. 6: Three processors Rocket, BOOM and DecVec reuse each other's code. Modules used by all three designs include an ALU, a MulDiv unit, an ICache, a TLB, a Decoder, and an FPU. Modules used by Rocket and BOOM include a non-blocking data cache, a PTW, a CSR, and a BTB.

B. HCF Support for Isolating Source from Backend

When backend-specific customizations are reflected in source code changes, it limits code reusability. Our HCF is designed to enable

many different categories of customizations that transform a design.

First we evaluate the richness of our IR, FIRRTL, and then show it can support a similar degree of optimizations that CAD tools can employ. Then, we demonstrate and evaluate a number of instrumentation and specialization transforms to illustrate the wide-ranging applicability of our framework.

1) FIRRTL Evaluation: What separates FIRRTL from the IRs in Yosys and PyVerilog is its ability to capture user intent. Recursive aggregate types (vectors and bundles) enable grouping related signals. Conditional connection statements enable straightforward floating signal checks. Bulk connections enable typesafe assignments. Finally, its explicit memory node allows straightforward and powerful memory optimizations. All these FIRRTL features should enable concise expressions that would require a less-rich IR to use more lines of code to represent.

To demonstrate the utility of a rich IR, we analyze the following three designs: (1) a reorder-buffer, (2) a branch reorder-buffer, and (3) a register renaming free list. As the design's rich features are simplified into FIRRTL's middle and low forms, we record the lines of code required to represent the design. To ensure for our HCF does not artificially inflate FIRRTL's code size, we run our optimization passes on the final stage of compilation.

As shown in Figure 7, some designs exhibit huge growths in code size during simplification, in spite of FIRRTL optimizations; this illustrates how a rich IR can concisely express a design, if the designer or frontend chooses to use the rich features.

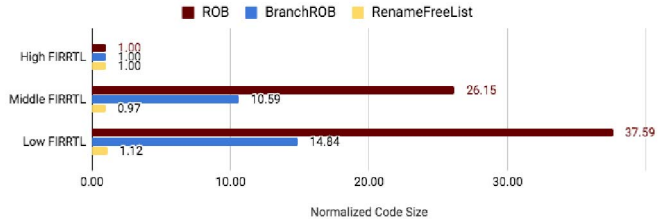

Fig. 7: Code size normalized to size of representation in High FIRRTL. The ROB and BranchROB both use aggregate types, bulk connections, and memory nodes while the RenameFreeList is made primarily of logic and does not use rich FIRRTL features.

2) Optimization Evaluation: A synthesis tool like Yosys does bit-level analysis and can thus perform more aggressive optimizations than FIRRTL can. However, as shown in Figure 8, our optimization passes reduce the cell count by up to 71% compared to up to 76% for Yosys. Running both FIRRTL's and Yosys's optimization passes results in even further cell count reduction.

3) Instrumentation Evaluation: We implemented a FIRRTL line-coverage transform which instruments the circuit to print its coverage information as it is executed in simulation. This instrumentation was necessary for Chisel because some of its constructs cannot map directly to Verilog, and so must first be simplified. This destroys source-level information that Verilog line-coverage tools rely on, making them largely ineffective. This transform works by associating high-level source-line information with low-level execution statements.

Figure 9 shows the percentage of modules in an instance of the RocketChip SoC colored by the percentage of those modules source lines which are tested. We show the results for three different configurations of the SoC; modules with low coverage exist in all three, and there is no clear trend in coverage based on configuration. In general, most modules have a high level of coverage on the given test-suite with a few modules that are very lightly tested. This

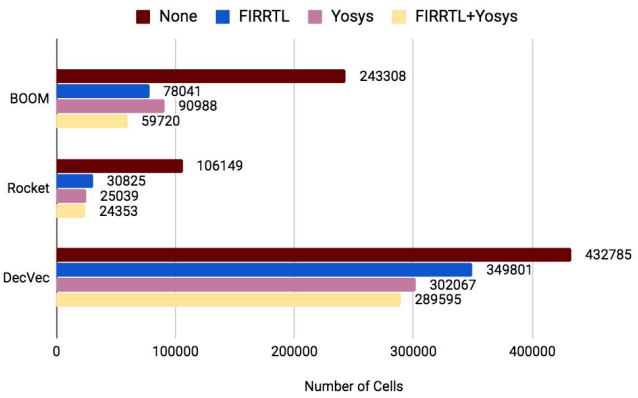

Fig. 8: FIRRTL optimization passes reduce cell count to a similar degree as Yosys optimization passes.

transform enables designers to target low-coverage modules with new tests and improve verification.

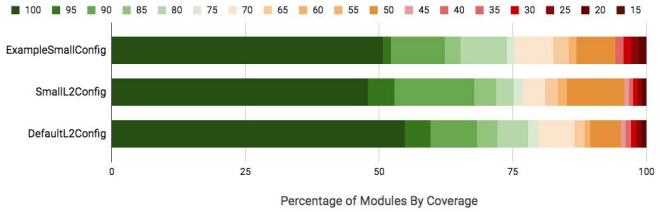

Fig. 9: We show the results for three different configurations of the SoC, a minimally sized configuration, ExampleSmallConfig, a moderately sized configuration with a small L2 cache, SmallL2Config, and the default sized configuration with a 256KB L2 cache, DefaultL2Config. The majority of modules have high coverage, but there remain a few which need targeted testing.

4) Transformations for FPGA Simulation: When simulating on an FPGA, there is little default visibility into a design. Commercial tools like Chipscope[42] enable real-time analysis, but require a long iteration cycle to select specific signals to target. In addition, it does not provide visibility into the BRAM memories on the FPGA, so cannot provide a full "snapshot" of the design at a given cycle.

We implemented an instrumentation transform which enables pausing a design on the FPGA (decoupling host and target time), and a transform that enables reading out a state snapshot of the target design on the FGPA. These transforms involve threading an enable signal to all registers, inserting buffers to record input and output traces, inserting address generation hardware to read out memory state, and attaching a custom daisy chain to scan out register and BRAM state from the FPGA. These types of transforms are a key part of many use cases, including fast and accurate power simulation[43] and debugging a circuit (future work).

In addition to instrumentation, other optimizations aim to provide the most effective use of resources when mapping a design to an FPGA. In particular, BRAMs are a valuable resource that can easily be wasted through replication to accommodate high port counts; instead, one of the FPGA specialization passes can automatically transform memories with high port counts into double-pumped memories with half as many ports by providing clock doubling and glue logic. Although this may reduce the maximum clock rate, the high speed potential of BRAM macros means that many microarchitectures will suffer much less than the worse case halving of throughput. In exchange for this trade off, **the pass attains a 3x reduction in**

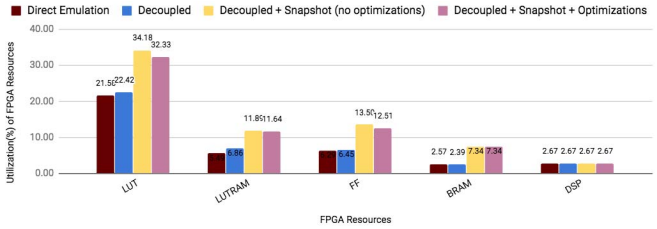

Fig. 10: The decoupling and snapshotting transforms can add significant demands on FPGA resources, but do provide visibility into the design that was previously unobtainable. Baseline and transformed designs run at 40MHz.

BRAM utilization for a streaming vector arithmetic block consuming three operands and producing one result per cycle, as shown in Figure 11. This comes at a small **1.43% increase in logic slice utilization.**

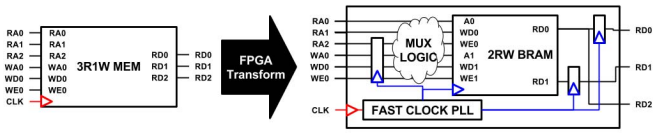

Fig. 11: Automatic double-pumping saves FPGA resources by emulating expensive, highly-ported memories. This approach maintains abstractions that facilitate reuse.

5) Transformations for ASIC Fabrication: ASIC designs benefit from the use of highly optimized hard-IP macros that are targeted towards specific functions. For example, large memory-based designs are typically mapped to vendor-provided SRAMs rather than registers, to improve QoR (in terms of area, power, and timing closure) and tool runtime. As seen in Figure 12, after synthesis a 2048-point memory-based FFT (20-bit real and imaginary) implemented with SRAMs (4 banks of 512-depth memories) is 6x smaller than the same FFT implemented with registers. The savings will increase after place and route due to excess routing penalties incurred in the register based design. Additionally, synthesizing the SRAM-based design takes considerably less time than the register based design, because the tools need to handle significantly fewer hardware instances.

However, specializing RTL to make use of these macros on a per-technology basis is non-trivial. Vendor-provided SRAMs often require properly connecting additional pins for initialization and verification but do not contribute to the functionality at a high-level. To address this issue, our memory-replacement transform replaces a generic FIRRTL memory with a custom black-box that matches the ports of the vendor-provided SRAM. Without running the transform, FIRRTL translates the generic FIRRTL memory into a large register array. When the transform is run, design-specific memory signals (data, address, and enable) are connected to the ports of a vendor-provided SRAM instance, and any additional initialization and verification signals to and from the SRAM are automatically connected, across module boundaries, to top-level ports.

By default, generic FIRRTL memories are mapped to dual-ported SRAMs. However, an additional optimization pass can be run to substitute the memories with single-ported SRAMs. Before making the substitution, this pass traverses the circuit to verify that read and write enables are logically exclusive. This greatly reduces the design effort required to map generic RTL to optimized hardware.

C. Case Study: Custom Design on a new ASIC Process

To illustrate a workflow using an HCL/HCF framework, we created a custom parameterization of RocketChip, synthesized, and place-and-routed on a 28nm process to DRC/LVS-clean GDS.

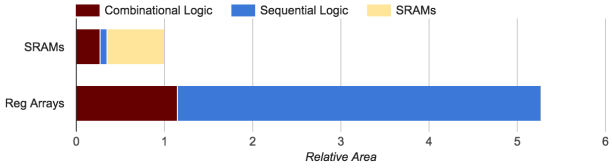

Fig. 12: A hardware FFT Chisel design generated with and without the memory-replacement transform was synthesized in a 16nm process. Our transform improved utilized area by 6x. Designs met timing at 800ps (1.25GHz). Synthesis took ∼6 mins with SRAMs, versus to 1 hr 44 mins without.

The design consists of two cores with a large data-parallel cache-coherent accelerator. The L2 cache is heavily banked, which requires multiple SRAMs. In addition, there are multiple clock and voltage domains, as well as multiple high speed off-chip IOs.

Because of the parameterization and reuse employed by the RocketChip hardware library, it is very easy to specify the desired design - only 1817 new lines of code were added, which consisted of specialized configuration parameters, top-level glue logic, and an associated test harness. Many modules had already been verified and evaluated in previous projects, and thus needed less verification and design effort. Almost all verification effort was spent on the new code as a result, and this reusability was key to reducing design overhead.

Targeting the 28nm process reused the memory-transform and the optimization transforms described in Section III-C. However, this process presented two new problems: (1) our synthesis tools required specifying clock and voltage domains per module; (2) the SRAMs had extra initialization and control pins unique to this process.

Due to the modularity of our HCF implementation, we wrote two custom passes to solve these problems and added them as part of the HCF transform library for use with other designs, backends, and projects, requiring only 680 new lines of code. Our toolchain (and runtimes) included Chisel (12 min), FIRRTL (11 min), synthesis (3.8 hrs), and place-and-route (>40 hrs).

In total, ***94% of this design was reused***, and future work will validate this methodology on other designs with other technologies.

V. Conclusion

Unlike the software industry, the hardware industry is inhibited by the lack of code reuse via libraries. To enable hardware libraries, this paper contributes the following: (1) a reemphasis on how HCLs provide language expressivity to enable reusability, (2) how our hardware compiler framework, based off a new hardware IR, FIRRTL, supports RTL customization, and (3) the wide-ranging applications of a hardware compiler framework.

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. Designers should focus on developing reusable hardware libraries, while researchers and developers should consider reusability as a primary focus of future languages and compilers.

VI. 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; DARPA CRAFT (HR0011-16-C-0052); Intel Science and Technology Center for Agile Design; and ASPIRE Lab industrial sponsors and affiliates Intel, Google, HPE, Huawei, LGE, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

References

[1] G. Venkatesh, J. Sampson *et al.*, "Conservation cores: Reducing the energy of mature computations," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 205–218.

[2] R. Hameed, W. Qadeer *et al.*, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 37–47.

[3] N. Goulding-Hotta, J. Sampson *et al.*, "The greendroid mobile application processor: An architecture for silicon's dark future," *IEEE Micro*, vol. 31, no. 2, pp. 86–95, Mar. 2011.

[4] O. Shacham, O. Azizi *et al.*, "Rethinking digital design: Why design must change," *IEEE Micro*, vol. 30, no. 6, pp. 9–24, Nov. 2010.

[5] M. Bostock, V. Ogievetsky, and J. Heer, "D3 Data-Driven Documents," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2301–2309, Dec. 2011.

[6] S. Sutherland and D. Mills, "Synthesizing systemverilog: Busting the myth that systemverilog is only for verification," in *SNUG Silicon Valley*, 2013.

[7] Xilinx, "Vivado High-Level Synthesis." [Online]. Available: http://www.xilinx.com/products/silicon-devices.html

[8] Mentor, "Catapult and PowerPro: High-Level Synthesis and RTL Low-Power." [Online]. Available: https://www.mentor.com/hls-lp/

[9] P. Coussy, C. Chavet *et al.*, "GAUT: A High-Level Synthesis Tool for DSP Applications," in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Springer Netherlands, 2008, pp. 147–169.

[10] A. Canis, J. Choi *et al.*, "LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36.

[11] J. L. Tripp, M. B. Gokhale, and K. D. Peterson, "Trident: From High-Level Language to Hardware Circuitry," *Computer*, vol. 40, no. 3, pp. 28–37, 2007.

[12] T. S. Czajkowski, U. Aydonat *et al.*, "From opencl to high-performance hardware on FPGAS," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, Aug. 2012, pp. 531–534.

[13] M. Owaida, N. Bellas *et al.*, "Synthesis of Platform Architectures from OpenCL Programs," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2011, pp. 186–193.

[14] A. Papakonstantinou, K. Gururaj *et al.*, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *2009 IEEE 7th Symposium on Application Specific Processors*, Jul. 2009, pp. 35–42.

[15] J. Auerbach, D. F. Bacon *et al.*, "Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 89–108.

[16] N. George, H. Lee *et al.*, "Hardware system synthesis from Domain-Specific Languages," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2014, pp. 1–8.

[17] F. Hannig, H. Ruckdeschel *et al.*, "PARO: Synthesis of Hardware Accelerators for Multi-dimensional Dataflow-Intensive Applications," in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, Berlin, Heidelberg, Mar. 2008, pp. 287–293.

[18] P. Milder, F. Franchetti *et al.*, "Computer Generation of Hardware for Linear Digital Signal Processing Transforms," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 17, no. 2, pp. 15:1–15:33, Apr. 2012.

[19] A. Hormati, M. Kudlur *et al.*, "Optimus: Efficient Realization of Streaming Applications on FPGAs," in *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '08. New York, NY, USA: ACM, 2008, pp. 41–50.

[20] 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*, Jun. 2004, pp. 69–70.

[21] "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, Jan. 2012.

[22] C. Wolf, *Yosys Open SYnthesis Suite*. [Online]. Available: http://www.clifford.at/yosys/

[23] J. P. Bergmann and M. A. Horowitz, "Vex-a CAD toolbox," in *Proceedings 1999 Design Automation Conference*, 1999, pp. 523–528.

[24] S. Takamaeda-Yamazaki, "Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL," in *Applied Reconfigurable Computing*. Springer, Apr. 2015, pp. 451–460.

[25] "Verific Design Automation." [Online]. Available: http://www.verific.com/

[26] J. Decaluwe, "MyHDL: A Python-based Hardware Description Language," *Linux J.*, vol. 2004, no. 127, p. 5, Nov. 2004.

[27] S. Sato and K. Kise, "ArchHDL: A Novel Hardware RTL Design Environment in C++," in *Applied Reconfigurable Computing*. Springer, Cham, Apr. 2015, pp. 53–64.

[28] S. Takamaeda-Yamazaki, *Veriloggen: A library for constructing a Verilog HDL source code in Python*. [Online]. Available: https://github.com/PyHDI/veriloggen

[29] D. Lockhart, G. Zibrat, and C. Batten, "PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2014, pp. 280–292.

[30] O. Shacham, S. Galal *et al.*, "Avoiding game over: Bringing design to the next level," in *DAC Design Automation Conference 2012*, Jun. 2012.

[31] P. Bellows and B. Hutchings, "JHDL-an HDL for reconfigurable systems," in *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines*, Apr. 1998, pp. 175–184.

[32] Y. Li and M. Leeser, "HML, a novel hardware description language and its translation to VHDL," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 1, pp. 1–8, Feb. 2000.

[33] J. Bachrach, H. Vo *et al.*, "Chisel: Constructing Hardware in a Scala Embedded Language," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1216–1225.

[34] M. Odersky, S. Micheloud *et al.*, "An overview of the Scala programming language," Tech. Rep., 2004.

[35] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *CGO*, San Jose, CA, USA, Mar. 2004, pp. 75–88.

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

[37] K. Asanović, R. Avižienis *et al.*, "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr. 2016.

[38] J. Balkind, M. McKeown *et al.*, "OpenPiton: An Open Source Manycore Research Framework," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 217–232.

[39] EEMBC, "Coremark: an EEMBC benchmark." [Online]. Available: https://www.eembc.org/coremark/

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

[41] R. Goldman, K. Bartleson *et al.*, "Synopsys' open educational design kit: Capabilities, deployment and future," in *IEEE International Conference on Microelectronic Systems Education, MSE '09, San Francisco, CA, USA, July 25-27, 2009*. IEEE Computer Society, 2009, pp. 20–24.

[42] Xilinx, "ChipScope Pro Debugging Overview." [Online]. Available: https://www.xilinx.com/itp/xilinx10/isehelp/ise_c_process_analyze_design_using_chipscope.htm

[43] D. Kim, A. Izraelevitz *et al.*, "Strober: Fast and Accurate Sample-Based Energy Simulation for Arbitrary RTL," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2016, pp. 128–139.