Effective simulation and debugging for a high-level hardware language using software compilers

Clément Pit-Claudel
MIT CSAIL
Cambridge, Massachusetts, USA
cpitcla@csail.mit.edu

Thomas Bourgeat
MIT CSAIL
Cambridge, Massachusetts, USA
bthom@csail.mit.edu

Stella Lau
MIT CSAIL
Cambridge, Massachusetts, USA
stellal@csail.mit.edu

Arvind
MIT CSAIL
Cambridge, Massachusetts, USA
arvind@csail.mit.edu

Adam Chlipala
MIT CSAIL
Cambridge, Massachusetts, USA
adamc@csail.mit.edu

Abstract

Rule-based hardware design languages (RHDLs) promise to enhance developer productivity by offering convenient abstractions. Advanced compiler technology keeps the cost of these abstractions low, generating circuits with excellent area and timing properties.

Unfortunately, comparatively little effort has been spent on building simulators and debuggers for these languages, so users often simulate and debug their designs at the RTL level. This is problematic because generated circuits typically suffer from poor readability, as compiler optimizations can break high-level abstractions. Worse, optimizations that operate under the assumption that concurrency is essentially free yield faster circuits but often actively hurt simulation performance on platforms with limited concurrency, like desktop computers or servers.

This paper demonstrates the benefits of completely separating the simulation and synthesis pipelines. We propose a new approach, yielding the first compiler designed for effective simulation and debugging of a language in the Bluespec family. We generate cycle-accurate C++ models that are readable, compatible with a wide range of traditional software-debugging tools, and fast (often 2 to 5x faster than circuit-level simulation). We achieve these results by optimizing for sequential performance and using static analysis to minimize redundant work. The result is a vastly-improved hardware-design experience, which we demonstrate on embedded processor designs and DSP building blocks using performance benchmarks and debugging case studies.

1 Introduction

Most hardware designs are expressed in a spectrum of languages ranging from low-level RTL (Verilog [23] or VHDL [17]) to sequential software languages with annotations for high-level hardware synthesis (Vivado HLS [27], Handel-C [13], Clash [20], etc.). Different points on this scale entail different trade-offs.

Verilog offers limited programming abstractions and composability, which makes it tedious to write and debug, but it offers developers fine-grained control over the resulting circuits. HLS systems, i.e. hardware design systems that start from software languages to generate Verilog, offer rich abstractions and excellent debugging and simulation facilities but poor control over generated circuits. This is not surprising because the sequential computation model of software languages is deeply at odds with the hardware computation models, which try to run all parts of a circuit in parallel all the time.

Rule-based languages, such as Bluespec [18], Kôika [3], and Kami [5], offer an interesting middle ground, with predictable performance and yet high-level, usable, and composable semantics. Rule-based designs describe the manipulation of (hardware) state elements using state-transforming atomic *rules*, which (appear to) execute sequentially. An RTL compiler then introduces concurrency by translating rules into individual circuits that run in parallel while preserving the illusion of sequentiality ("one-rule-at-a-time semantics").

Significant research effort has been dedicated to synthesizing high-quality hardware from Bluespec designs. Comparatively little effort has been expended on cycle-accurate simulation, debugging, and testing of rule-based designs: these tasks are typically performed at the generated-Verilog level. As a result, despite its convenient abstractions, Bluespec is not particularly pleasant to debug: it is mainly done using an enhanced form of printf and wave analysis.

In addition, simulating Bluespec-generated Verilog isn't particularly fast. The key issue is that compilers that target RTL optimize for fast hardware — not fast simulation! Efficient execution in hardware requires maximally parallel designs, so code generators (and hardware engineers) will often introduce additional work (and increase area) to increase concurrency (and reduce critical paths). This directly hurts simulation performance when running on a limited-concurrency platform such as a desktop computer or a server.

This paper tackles both of these issues by decoupling simulation and synthesis but keeping them cycle-accurate with respect to each other. We design a new compilation backend for Kôika, a Bluespec-inspired language that provides

fine-grained control over scheduling, which targets C++ instead of Verilog. Because it is free to employ optimizations specifically geared towards sequential execution and to rely on existing C++ compiler technology, we can produce much-faster software models; speedups of 2x to 3x are typical in our experience. This performance does not come at the cost of intelligibility: by mapping Kôika abstractions to zero-cost idiomatic C++ patterns, we can obtain readable models that closely reflect the structure of the original design.

Concretely, our Kôika simulator, called Cuttlesim, works by compiling each design into a custom C++ program. The baseline version of such a program reads like a fairly direct transliteration. However, we also apply optimizations like the early detection of the abortion of an atomic rule, so that later rule code can be skipped; RTL would compute the whole rule in any case, which comes "for free" in hardware. (Maintaining cycle-level accuracy between RTL and C++ models only requires ensuring that each hardware state element is updated in the same cycle in both the models, and not the amount of computation that is performed in a cycle.) Some of these optimizations depend on static analysis, which is much easier to perform on Kôika than on RTL. We then discover opportunities to specialize the data structures and algorithms used for tracking conflicts between rules. These models are compatible with the whole range of traditional software-debugging tools, enabling a whole new hardware-debugging and verification experience. The information these tools return is easy to map back to the original hardware design (which can be matched nearly line-by-line with the generated C++ code), even when we make no changes to standard tools like `gdb` and `gprof`.

So, overall, we suggest a new workflow for hardware development: write in rule-based hardware-description languages, compile automatically to C++ programs, debug and profile with standard software tools, repeat, and only later synthesize to RTL. This paper presents our Cuttlesim prototype and some representative uses of it in developing embedded-class RISC-V processors and simple DSP components. The paper makes the following contributions:

1. We show how using completely separate toolchains for software simulation and hardware synthesis leads to faster simulation and improved debugging experience.
2. We describe techniques to build fast software models of rule-based designs, using lightweight transactions.
3. We show that rule-based designs are amenable to heavy optimization through static analysis that exploits the simplicity of the input language.
4. We give concrete evidence of the value of this approach using simulation performance benchmarks and debugging and design exploration case studies.

We note that we are focused on designing concrete circuits, not evaluating micro-architectural ideas independently of implementation. As such, we focus on cycle-accurate simulation, not on processor simulation in the style of Gem5 [2] or ZSim [21].

2 Rule-based designs and their simulation overheads

This section explains why simulating rule-based designs after compilation to Verilog is inefficient and why rule-based designs are in fact amenable to fast simulation. We start with a brief reminder about rule-based designs and their compilation to Verilog. A detailed description of Kôika and hardware synthesis is available in [3].

2.1 Rule-based designs

Languages in the Bluespec family encourage designers to decompose hardware designs into small units of work called *rules*. In a pipelined CPU design, each rule would typically encode one stage (fetch, decode, execute, writeback, etc.).

Rules are written in a simple language with traditional constructs like conditionals, variable bindings, and combinational functions, plus three special primitives: *read*, *write*, and *abort*. These primitives define how rules change the system state and communicate within a cycle. Each read and write is annotated with a port (0 or 1): reads at port 0 observe the value of registers at the beginning of the cycle; reads at port 1 observe the latest write at port 0 if any, or the beginning-of-cycle value otherwise; writes at port 1 are not observable until the next cycle. Aborts cancel a rule's execution.

The semantics of the language specify that rules (should appear to) execute atomically, one-at-a-time: that is, the results computed by a design should be the same as if exactly one rule executed per cycle, with no concurrency and no intra-cycle communication. Implementing these semantics requires ruling out linearity violations, typically using static analysis or dynamic tracking of read-write sets (for example, a write at port 0 precludes a read at port 0 in the same cycle: if both rules ran in the same cycle, the second one would observe the original value, not the result of the write). Kôika, the language that we target, has *rules* as well as explicit *schedulers*, which specify the order in which rules should (appear to) run.

As an example, assume that we are modeling a two-state machine, whose internal state is represented with a register `x`. An additional register `st` keeps track of whether the machine is in state *A* or *B*. The dynamics of the state machine can be described using two rules, `rlA` and `rlB`, each predicated to run only if the machine is in the right state, as follows:

```
rule rlA =
  if (st.rd0 != 'A) abort;
  st.wr0('B);
  let new_x := fA(x.rd0(), get(input)) in
  x.wr0(new_x); put(output, new_x)
```

Here, fA is some combinational function doing potentially complex work, and input and output are external ports used to communicate with the outside world. Rule rlB is similar, and since the two rules are mutually exclusive it does not matter which order we schedule them in. In each cycle, one of the two rules will execute, read input, update the machine's internal state, and write it to output.

To understand why the hardware that Kôika generates for this design does not lend itself to fast simulation, we first need to understand how Kôika compiles designs to Verilog.

2.2 Generating hardware from rule-based designs

Kôika generates hardware by creating one circuit per rule, in isolation, then wiring these rule circuits together as specified by the scheduler. Individual rule circuits track reads and writes to each register, at each port. Scheduling logic ensures that the reads and writes performed by two rules are compatible and that rules did not explicitly abort. In case of conflicts or explicit aborts, the results of the rule are discarded.

The circuit generated by Kôika for the trivial example above is a combinational circuit that computes the new values of st (A or B) and x (the machine's internal state). The resulting Verilog looks roughly like this:

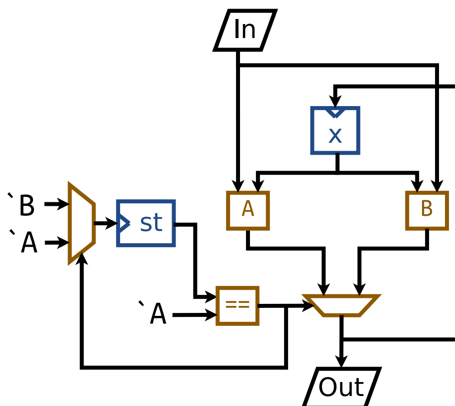
```
module stm(CLK, in, out);
  input wire CLK;
  input wire [31:0] in;
  output wire [31:0] out;

  reg st = 1'b0;
  reg [31:0] x = 32'b0;

  wire [31:0] fB_out, fA_out;
  fA mod_fA(.x(x), .in(in), .out(fA_out));
  fB mod_fB(.x(x), .in(in), .out(fB_out));

  assign out = x;
  always @(posedge CLK) begin
    st <= st == `A ? `B : `A;
    r <= st == `A ? fA_out : fB_out;
  end
endmodule
```

The same circuit is represented in the figure below:



In this simple example, the circuits generated for st and x are both Muxes of the values computed by each rule, predicated by the state that the machine is in at the beginning of the cycle. In a more complex setting, a nontrivial Boolean circuit would decide whether the results of each rule should be committed or discarded.

Note how the circuits corresponding to all rules run in every cycle, though only one of them "commits" (that is, just one of them updates the state).

2.3 Overheads in simulating Kôika

The compilation strategy employed by Kôika [3] is to ensure maximal concurrency between rules by running all rules concurrently in every cycle and reconciling their results a posteriori. Unfortunately, this makes it inefficient to simulate in software that runs on a limited-concurrency platform.

Optimizing for circuits and optimizing for simulation on a CPU are different goals, and it makes sense that Kôika would optimize for good circuit performance. In hardware, the cost (critical path) of the generated circuit will be $|\texttt{Mux(st == A, fA_out, fB_out)}| = |\texttt{Mux}| + \max(|\texttt{st == A}|, |\texttt{fA_out}|, |\texttt{fB_out}|)$, where $|\texttt{X}|$ is the cost of circuit X.

Unfortunately, simulating this circuit as written leads to unnecessary computation. A typical Verilog simulator will generate code to compute *both* fA_out and fB_out in every cycle, *then* mux their results. The sequential running time, thus, would become $|\texttt{Mux}| + |\texttt{st == ...}| + |\texttt{fA}| + |\texttt{fB}|$. We note in passing that at such a small granularity, there is almost nothing to be gained from introducing thread-level concurrency because of the high cost of synchronization and data movement. Decomposing the simulation of a large design to run on multiple cores can make sense, but the amount of concurrency in a circuit is usually much larger than the number of cores in a typical computing platform.

This is a trivial example, and accordingly one might hope to optimize it and run only the relevant branch of the Mux (Verilator doesn't, but other simulators might, and it would be reasonable for it to grow that capability). But on a more complex example, additional rules can easily complicate the picture and make the optimizations intractable. For example, maybe in state A the machine reads from an input FIFO, and the rule fails if this FIFO is empty. Or maybe the whole state machine is embedded in a larger design, and both rules can only execute if no preceding rules generated conflicts: for example, the system might support an external reset command whose execution prevents rlA and rlB from running in that cycle (this would ensure that reset does not need to be sequenced with rlA and rlB and thus does not lengthen the machine's critical path).

A compiler aware of Kôika's semantics, starting from the original design, would save a lot simply by faithfully mapping Kôika behavior to a sequential execution model. It would generate code that stops executing a rule as soon

as it fails. In our example, its cost (running time) would be |st == A| + |fA| + |st == B| if the machine is in state A, and |st == A| + |st == B| + |fB| if the machine is in state B: in no cycle do we need to pay for the cost of executing both rules.

The important concept here is that Kôika's semantics allow rules to exit early, either from conflicts with previous rules or from explicit aborts. In software, the ideal implementation of these early exits is to jump straight to the next rule, skipping whatever remains of the current rule. Our key contribution is showing how to design a compiler that takes advantage of Kôika's "early-exit" semantics and generates C++ models that are perfectly suited to a CPU's mostly sequential execution paradigm. We do this by designing a lightweight transaction system, using static analysis to further reduce transaction costs. Static analysis is particularly easy in this context, because Kôika's semantics are simple and explicit.

There is a surprising benefit to this perspective shift. By designing the simulator carefully, we can produce readable models, amenable to all sorts of software-focused methodologies (step-through debugging, code coverage analysis, etc.). For example, here is how the C++ model rlA might look, annotated with Gcov execution counts, after 500 cycles:

```
-:500 DEF_RULE(rlA) {
-:500   if (READ0(st) != state::A)
-:250     return false;
-:250   WRITE0(st, state::B);
-:250   bits<32> y = fA(READ0(x), get(input))
-:250   WRITE0(x, y); put(output, y);
-:250   COMMIT();
-:250 }
```

It turns out that simply collecting code-coverage statistics on the generated C++ code offers an incredible wealth of architectural information, without having to add a single hardware counter.

We give evidence of the performance of our simulator, and detailed examples of the new design methodologies that our tooling enables, in section 4. But first, we present the technical insights that make these results possible.

3 Compiling Kôika for simulation

The performance of Cuttlesim's models is achieved through two distinct technical contributions: a lightweight implementation of transactions that is shared by all designs, and design-specific optimizations that are derived using static analysis.

Details on the structure of the models that Cuttlesim generates, how they lend themselves to further optimization by C++ compilers, and what techniques we employed to make them both readable and seamlessly debuggable are given in this paper's appendix.

3.1 Matching Kôika's transactional semantics

The original Kôika paper describes Kôika's semantics using *rule logs*, which keeps track of the reads and writes performed by a rule, and a *cycle log*, which keeps track of the reads and writes performed by all the rules that are scheduled in a cycle. Each cycle starts with an empty *cycle log*. Rules are executed one by one, and every time a rule attempts to perform a read or write, a check is made against both the cycle log and rule log up to that point in the cycle to ensure that the action is permitted; if not, the whole rule aborts. When a rule executes successfully, its rule log is appended to the cycle log. Otherwise, the rule log is discarded. At the end of the cycle, the values of the design's state elements (registers) are updated based on the reads and writes accumulated in the cycle log.

The C++ models that Cuttlesim produces closely follow this idea. Each design is compiled into a C++ class. Rules become functions that construct rule logs. The design's scheduler becomes a function that calls each rule in turn. The key difficulty in compiling Kôika is to devise efficient representations for these logs, and to implement the transactional semantics of the rules, which require maintaining shadow states. We start with a naive model, which we then refine incrementally into an efficient implementation. This lets us explain each optimization individually.

In a naive model, Cuttlesim keeps three pieces of data: the values of the design's registers at the beginning of the cycle (its publicly observable state), a cycle log, and a rule log. The logs are arrays containing a structure for each register, which indicates whether the register was read or written at port 0 or 1, plus two data fields (`data0` and `data1`) that keep values written at port 0 or 1 in that register. Reads and writes are implemented as C functions that check whether the operation is permitted by Kôika's semantics and either update the rule log or cause the rule function to abort. Specifically, a read at port 0 checks for writes at any port in the cycle log and returns the beginning-of-cycle value of the register; a read at port 1 checks for writes at port 1 in the cycle log and returns the most recent write0 value from either log, falling back to the beginning-of-cycle state; a write at port 0 checks for reads at port 1 and writes at port 0 or 1 in both logs; and a write at port 1 checks for other writes at port 1 in both logs.

When a cycle begins (resp., entering a rule), Kôika clears the read-write sets of the cycle log (resp., the rule log) and invalidates its data fields. When a rule succeeds, its log is *committed* into the cycle log: read-write sets are or-ed together, and the cycle log's data fields are updated to reflect writes found in the rule log. When a rule fails, nothing needs to be done: the rule log will be reset upon entering the next rule, so the rule simply returns early. When a cycle completes, the model's register values are updated by *committing* the cycle log: if a write occurred at port 1 the `data1` value is copied

from the log to the state; otherwise if a write occurred at port 0 the `data0` value is used instead; and otherwise the state is left unchanged.

3.2 Optimizing transactions

This simple implementation of transactions is slow: models spend inordinate amounts of time checking and copying read-write sets, copying data between logs, and committing results. We improve the design through a sequence of refinements, starting with those that are design-independent.

Separate read-write sets and data. Our naive logs store read-write sets and data fields together. This makes logical sense, but it makes clearing read-write sets costly: we need to zero out parts of a structure interleaved with data. A much better approach is to store read-write bitsets separately from write data: this way, resetting all read-write sets is just a matter of zeroing out a structure, which is cache-friendly and efficient. Concretely, this means changing the type of logs to hold two structures: one for read-write sets and one for values written.

Accumulate logs instead of merging them. Keeping the cycle log L and the rule log ℓ separate makes many operations more costly than they need to be: writes need to perform checks on both logs; reads at port 1 need to look for data in both logs and in the beginning-of-cycle state; and committing a rule log requires or-ing read-write sets together (this is very fast) but also checking for writes at either port in the rule log and optionally copying that data to the cycle log (this is very slow). To speed up these two operations, we change to keeping a cycle log L and an *accumulated* rule log $L +\!\!+ \ell$, ensuring that the full accumulated log is reset upon entering each rule (not just its read-write sets). This makes checks for write operations much simpler (they only need to check the accumulated rule log), and speeds up rule commits significantly (committing a rule is now a plain copy from the accumulated log).

Reset on failure, not on entry. When a rule fails, we just exit from the corresponding function. This works because the next rule will reset the accumulated rule log by copying the cycle log into it. This reset is redundant when a rule completes successfully: committing the accumulated rule log already ensures that the cycle log and the accumulated rule log match. We thus enforce a new invariant: the accumulated log should match the cycle log at the end of each rule. To maintain this invariant we need to reset the rule log upon failures (by copying the cycle log into it) and to reset the read-write set of the cycle log at the beginning of each cycle. This makes failures more costly, but it allows us to eliminate the resets performed upon entering each rule.

Merge `data0` and `data1`. Keeping `data0` and `data1` separately is *almost never* necessary, except in Goldbergian contraptions like the following example:

```
rule rl = r.wr₀(1); r.wr₁(2); r.rd₀(); r.rd₁()
```

Assuming that register `r` held value `0` at the beginning of the cycle and was neither written nor read previously, this rule would execute successfully, with the $\mathtt{rd}_0$ reading 0 and the $\mathtt{rd}_1$ reading 1. This pattern is rare in real designs, as it is widely considered an anti-pattern in Bluespec. Since it is easy to detect and rewrite, Cuttlesim ignores the issue and optionally warns about it. Merging the data fields nearly halves the space occupied by the internal state of Cuttlesim models, and it saves time at the end of the cycle: instead of selectively committing either `data0` or `data1` at the end of each cycle, we can check for either write and commit the same data.

Eliminate beginning-of-cycle state. Merging `data0` and `data1` allows us to further debloat the model by entirely eliminating the beginning-of-cycle state kept in addition to the logs. By changing the model to initialize the data parts of both logs to the registers' initial values (instead of leaving them indeterminate), we establish a new invariant: the data stored in both logs at the end of each cycle matches the values stored in the beginning-of-cycle state. The implementation of reads needs to change to read the logs as appropriate. Eliminating the separately-kept state saves nearly a third of the remaining model memory, eliminates end-of-cycle commits entirely, saves time in reads, and even allows mid-cycle snapshots.

Taken together, these optimizations yield lightweight and relatively efficient transactions that faithfully implement Kôika's semantics. Profiling, however, reveals that models still spend a lot of time copying data: each commit or reset requires copying entire logs, including the data and read-write states of *all* registers. We now show how the cost of commits, resets, and read-write checks can be reduced dramatically using design-specific optimizations.

3.3 Leveraging design-specific knowledge

All the optimizations described below leverage information gathered using a straightforward abstract-interpretation pass. This pass annotates each read, write, and abort within a rule with a conservative approximation of the (non-accumulated) rule log at that point in the program, plus one Boolean per register indicating whether any of the operations on this register might cause failures (due to conflicts) within that rule. Additionally, it generates an approximation of the whole-cycle log by combining the individual rule logs, plus Booleans indicating whether each register might cause failures[1].

[1] The update formula for the flag indicating whether operations on a register can cause failures is essentially a tribool version of Figure 5 from the original Kôika paper.

Minimize read-write sets. Kôika's semantics give each register two read ports and two write ports. To determine whether reads and writes to these elements are valid, Cuttlesim tracks four Booleans per state element, indicating which operations have occurred. In most cases, some of this tracking information is redundant.

First, though Kôika's semantics track reads at port 0, this tracking is only useful as part of a concurrent compilation scheme where rules are all executed in parallel and conflict resolution is delayed. When compiling to a sequential model, as in Cuttlesim, the conflict is flagged as soon as the read is attempted: the `read0` part of read-write sets is unused and can be removed.

More generally, we can use the information gathered in our static analysis to classify registers: "plain registers" are read and written only at port 0; wires are written at port 0 and read at port 1; and "EHRs" ("ephemeral history registers") make more complex use of read and write ports. Then, Cuttlesim can exploit this classification to save memory and simplify conflict checking.

Eliminate read-write sets for "safe" registers. A register is *safe* if the reads and writes performed on it can never fail (the register cannot be a source of inter-rule conflicts). This is the case if all reads and writes on it are ordered in such a way that the corresponding checks always succeed. For a simple register, this would mean never reading after writing; for a wire, this would mean never performing a write (at port 0) after a read (at port 1); for other EHRs, this would mean satisfying the whole set of conditions specified by Kôika's semantics. The static-analysis pass performed by Cuttlesim computes a conservative approximation of whether each register is safe; for safe registers, it completely discards read-write sets and performs reads and writes directly, without checking the usual read and write pre-conditions. This change speeds up reads and writes as well as commits, failures, and the read-write set reinitialization performed at the beginning of each cycle.

Restrict commits and rollbacks to each rule's footprint. The static approximation of each rule log provides an upper bound on its footprint (the set of registers that it reads or writes). Cuttlesim generates custom commit and reset functions that copy and roll back the read-write sets of only those registers that may be read (at port 1) or written, as well as the data parts of only those registers that may be written. However, if a rule touches most of the registers in a design, Cuttlesim reverts to copying whole logs (it is typically faster to perform a single `memcpy` between two logs than to perform a large number of individual field copies).

Speed up early failures. Kôika rules commonly have guards, explicit checks that cause a rule to abort if a condition does not hold. Typically, these checks happen very early, before attempting to write registers; hence, if the rule aborts, there are no modifications to roll back. In that case, Cuttlesim produces code that exits the rule without rollback.

			SLOC			
	M	C	Kôika	Cuttlesim	Verilog	Cycles
`collatz`	✗	✓	38	42	13	1G
Trivial state machine						
`fir`	✓	✓	126	102	18	1G
Finite impulse response filter						
`fft`	✓	✓	266	802	341	30M
Part of a Fast Fourier Transform						
`rv32i`	✗	✗	1436	3462	787	25.1M
Small RISCV core (branch predictor: pc + 4)						
`rv32e`	✗	✗	1436	2648	413	25.1M
Embedded variant of `rv32i` (predictor: pc + 4)						
`rv32i-bp`	✗	✗	1706	9855	3087	23.7M
`rv32i` with a better branch predictor (btb + bht)						
`rv32i-mc`	✗	✗	2047	20575	2855	46.8M
Dual-core variant of `rv32i` (predictor: pc + 4)						

Table 1. Our benchmarks. **M**eta-programming examples use code generation. **C**ombinational examples have a single rule and no scheduling or conflicts. Specialization and heavy optimizations can lead to very small Verilog line counts.

4 Evaluation

Cuttlesim models are fast and enable new styles of hardware debugging and design exploration. In this section, we support these claims using performance benchmarks (subsection 4.1) and case studies (subsection 4.2).

4.1 Simulation performance

To evaluate Cuttlesim's performance, we compare primarily against Verilator, an open-source state-of-the-art Verilog simulator[2].

Experimental design. Our evaluation revolves around the simulation of different variants of an embedded processor core supporting the RV32I&E flavors of the RISC-V ISA (minus system instructions, interrupts and exceptions) running a simple integer arithmetic benchmark. To demonstrate that Cuttlesim also behaves well on designs which do not include much control signals, we also evaluate the performance of Cuttlesim on two purely combinational circuits: a small finite impulse response filter, and the butterfly parts of a large FFT design.

Of course, the point of this evaluation is to measure simulation performance, not to showcase the architectural qualities of specific embedded designs. Our benchmarks are described in Table 1.

[2]The authors of Verilator write that "*Verilator has typically similar or better performance versus the closed-source Verilog simulators (Carbon Design Systems Carbonator, Modelsim, Cadence Incisive/NC-Verilog, Synopsys VCS, VTOC, and Pragmatic CVer/CVC).*"

As Kôika has not yet been used to design very large systems, our benchmarks are all small to medium-sized (tens to thousands of lines of Kôika or Bluespec code, or hundreds to tens of thousands of lines of Verilog code). Consequently, we do not make claims about Cuttlesim's performance on very large designs.

Experimental setup. All benchmarks were run on an Intel Core i7-4810MQ CPU @ 2.80GHz with all C++ compiler optimizations turned on, using the settings recommended by Verilator for maximum performance.

Results. We use our benchmarks to answer three questions:

Q1: Can Cuttlesim models run faster than a state-of-the-art Verilog simulator? Yes. We answer this question by comparing the execution time of various Kôika designs when simulated directly with Cuttlesim or indirectly by compiling them to Verilog using the pre-existing Kôika compiler and running them with Verilator. Figure 1 shows our results: on control-heavy designs like CPU cores, Cuttlesim is multiple times faster than Verilator. On combination circuits, Cuttlesim's advantage is narrower, as expected. Other simulators that we benchmarked against (CVC and Icarus) were orders of magnitude slower than Verilator.

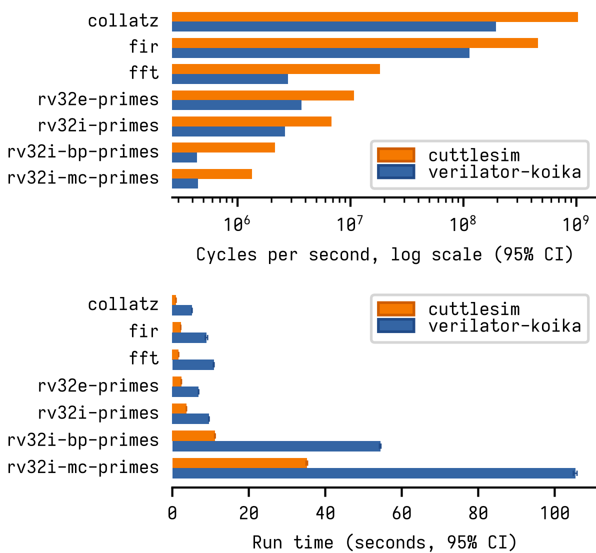


Figure 1. Performance of Verilator and Cuttlesim models

Q2: Is Cuttlesim's advantage only due to Kôika's compiler generating inefficient Verilog circuits? No. We answer this question by benchmarking Cuttlesim models against Verilator simulating Verilog code generated by the commercial Bluespec compiler from equivalent designs. We know that the Verilog generated by Kôika works well with synthesis tools, as the circuits it produces tend to have critical paths and areas comparable to Bluespec-generated ones. However, Kôika targets a much smaller subset of Verilog than Bluespec does. This is for soundness reasons: Kôika's compiler is formally verified against a minimal specification of circuits: the smaller the subset of Verilog used, the more assured we can be that tools will interpret it consistently. Figure 2 shows our results: the Verilog code that Kôika generates simulates roughly within a factor two of that generated by Bluespec for an equivalent design (notably, this number varies depending on the exact version of Verilator used; we notice that the code produced by Bluespec runs faster with Verilator 4, while the code produced by Kôika runs faster with Verilator 3.9). In all cases, it is very likely that a *different* compiler to Verilog could produce significantly better Verilator performance, but optimizing for simulation performance is unnecessary if we have a separate simulator, and indeed in that case we would prefer to focus on the quality of the synthesized hardware (as a concrete example, we could generate much simpler circuits by not introducing concurrency when compiling Kôika's schedulers, but the result would be much slower after synthesis).

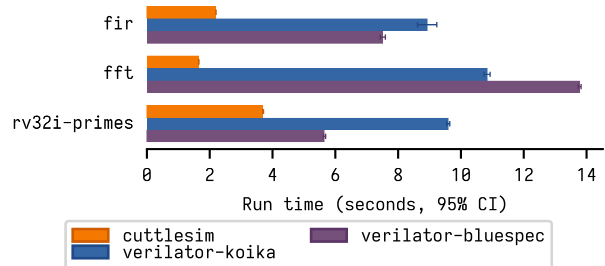


Figure 2. Performance of models on equivalent Bluespec and Kôika designs

Q3: How sensitive is Cuttlesim's performance to external factors, especially compiler choices and tool versions? Somewhat sensitive. We answer this question by compiling Cuttlesim and Verilator models using GCC and Clang and benchmarking the results. Figure 3 shows our results: we find that execution times vary, but that Cuttlesim's speed advantages over Verilator are relatively stable.

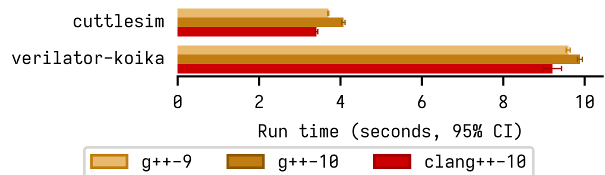


Figure 3. Performance of Verilator and Cuttlesim models

4.2 Debugging, Verification, and Digital Design Exploration Case Studies

Hardware designers spend a significant amount of time simulating designs in order to validate correctness, understand performance properties, and explore design trade-offs. As such, effective design exploration and debugging tools, which enable designers to understand *how a design works* or *why it does not work* as expected, are invaluable in the development process.

In this section, we show how Cuttlesim improves the state of the art in debugging and architectural exploration of rule-based designs. We walk through a series of experiential case studies illustrating various aspects of the hardware design and debugging process: functional-correctness-debugging of a cache coherence protocol, functional validation of a design using randomized testing, performance-debugging of an embedded processor core, and design exploration adding a branch predictor to an existing processor.

All of these case studies focus on cases where cycle-accuracy is paramount: we are working on concrete designs and using Kôika to generate actual circuits. High-level simulators like Gem5 and Zsim would not be applicable.

Case Study 1: Debugging a Cache Coherence Protocol. To illustrate the debugging process with Cuttlesim, we look at a simple case where a programmer is debugging a deadlock in a 2-core machine with L1 "child" caches and a "parent" protocol engine implementing the MSI cache coherence protocol. To determine what state the system is stuck in, the programmer runs the Cuttlesim model of the system in `gdb` until reaching the deadlock state. Next, they use `gdb`'s interactive interface to print out information corresponding to relevant state. In particular, they recall there are status registers such as miss status handling registers (MSHRs), tracking protocol state.

MSHRs are logically structures containing various pieces of information, including a tag that is either `Ready`, `SendFillReq` (indicating a cache miss and needed to send a request to the parent), or `WaitFillResp` (indicating waiting for a response from the parent). The tag type was implemented in Kôika as an enum, and the MSHRs as a struct. The enum names have semantic meaning that are preserved in the generated C++ model, and fields of the structure can be accessed naturally by name without doing bit slicing. Furthermore, the programmer does not have to write custom pretty-printers.

Now, they observe that Core 0's cache is deadlocked in the `WaitFillResp` state and the parent protocol engine is in the `ConfirmDowngrades state` (due to an upgrade request from `Invalid` to `Modified` for an address held by Core1). To determine why the rule corresponding to `ConfirmDowngrades` was not executing (and hence why there was no state transition away from `ConfirmDowngrades`), they set a breakpoint on `FAIL()`, the macro used to exit early from a rule. We consider two possibilities.

Suppose `gdb` indicated the failure was caused by a *conflict* between rules. The programmer puts a watchpoint (hardware breakpoint) on the relevant read-write set and executes in reverse (this is made possible by reverse debugging tools like `rr` [10] or GDB's native recording facilities). The compiler stops where the previous write happened, indicating an accidental `write1` instead of `write0`, conflicting with the rule's `read1`.

Contrast this experience with debugging at the Verilog level. Determining why a rule fails means going through each part of the fail signal to tease out what made it true. Then finding the source of the previous write means wading through a forest of other signals, which is tedious and error-prone.

Alternatively, suppose the failure was caused by an explicit `abort`. There are multiple possibilities here: perhaps the downgrading has not finished, or main memory has not responded with the cache line. To determine the cause, the programmer steps through the rule in `gdb` (they are able to do this due to Cuttlesim's fine-grained simulation of transactions, allowing sequential exploration of what code paths were taken). They realize the rule failed due to intermediate state unexpectedly indicating that downgrading has not finished, despite observing that the other core has indeed downgraded its state. Printing out the intermediate state, they realize that it was erroneously computed and have successfully pinpointed the problem.

Traditional methods of debugging similar bugs might involve adding `$display` statements (so-called `printf()` debugging) or wave-form debugging (e.g. using GTKWave) with Verilog. However, traditional `printf()` debugging often suffers from having to go through multiple (time-consuming) recompilation cycles, as it is not immediately clear what state is relevant and the debugger does not want to be overwhelmed with irrelevant state. With wave-form debugging, we sacrifice the high-level abstractions and close correspondence with original source code, requiring an additional mental step to map the logging output to the original source.

With Cuttlesim, there was no need to recompile the design in order to add logging statements: interactive debugging enables programmers to print just the relevant bits of state, as they become needed — including stopping halfway through the execution of a cycle to print the intermediate stated produced by the execution of a few rules.

Furthermore, the ability to step through (sometimes in reverse) the generated model and observe state at fine-granularity was useful to interactively test local assumptions about the code. Unlike Verilog-level debugging, stepping through interactively makes it very clear which parts of the design execute in a given cycle.

Finally, we note that this process mostly resembled high-level software debugging of state machines.

Case Study 2: Functional Verification with Scheduler Randomization. A good rule-based design should use its scheduler for performance, but not for functional correctness: designs should work regardless of the order that rules are executed in. We would like to verify this property experimentally, using randomized testing to validate the design under many different schedulers. Without Cuttlesim and without significant modifications to Kôika's compiler, a limited approximation of this could be achieved by creating many copies of the design with different schedules and compiling each of them independently. Instead, Cuttlesim's C++ models make it trivial to run this experiment in full generality: in C++, it suffices to write a `cycle()` function that calls rules in random order. We used this methodology to gain confidence in the RISC-V cores that we evaluated for performance.

Case Study 3: Performance Debugging. A standard methodology for designing processors in rule-based languages is to focus initially only on functional correctness in the one-rule-at-a-time semantics (ignoring inter-rule concurrency and cycle boundaries), and then fine-tune the concurrency by deciding which subset of rules happens in each cycle.

This section describes another flavor of hardware "debugging": architectural performance tuning. Concretely, this involves choosing bypassing paths (using inter-rule communication through read-write ports) and a scheduler that, together, maximize performance.

Suppose a programmer observes (on a 4-stage pipelined processor with an idealized single-cycle memory), that retiring 100 NOP instructions took 203 cycles. This suggested suboptimal performance, for one would assume that the pipeline would take one cycle per instruction on each a program containing no branches and thus, no misses.

To investigate, the programmer starts a gdb session to step through the execution, following a NOP instruction through the pipeline rule-by-rule. They observe that a NOP instructions never gets decoded in the same cycle that an older NOP is executed. Taking advantage of Cuttlesim's ability to step through individual rules, they observe that the decoding stage checks the scoreboard for outstanding writes to the source registers of the currently-being-decoded instruction (to prevent read-after-write hazards where an instruction would observe a stalled value not yet updated by an older instruction). In this case, the scoreboard marked the previous NOP as a dependency on the NOP being decoded, and so the new NOP could not be decoded.

One could think that NOP instructions do not write and so, do not generate dependencies. However in RISC-V, a NOP is encoded as `ADDI x0 x0 0`, where `x0` is a special non-writeable register always containing `0`. Likely, the processor designer neglected to implement a special case for tracking the dependencies on register `x0`, causing an unintended dependency to be created between the NOP instructions.

Identifying such a bug using traditional tools is particularly difficult because one needs to go *backward*. First, the debugger must ensure that the compiler did not erase any intermediate signals. Next, the programmer must locate and display the stuttering signals (using some Verilog display debugging), and finally they print all the signals used in the computation of the stuttering control signal to identify the cause. With Cuttlesim, the programmer simply steps *forward* through the code and observes the point where the rule fails. Since the generated model corresponds closely to the original source code, this is straightforward and there is no additional mental step to relate the model to the original source.

Case Study 4: Branch Prediction Exploration. In this section, we describe the process of improving the branch prediction mechanism of a baseline processor that only had a simple "PC + 4" predictor. The goal of this section is not to equip the processor with a state-of-the-art branch prediction, but to illustrate the process of architectural refinement in rule-based designs when using Cuttlesim.

Concretely, we need to add a Branch Target Buffer in charge of recording the target addresses of branch and jumps and a Branch History Table in charge of tracking and predicting whether branches are taken or non-taken, and to update the mechanism that handles mispredictions. We name the two designs `baseline` and `bp`.

Traditionally, to measure the improvement achieved by such an architectural change, we would add hardware performance counters, iteratively gathering increasing amounts of data (instructions executed per cycles, number of mispredicted instructions, number of cycles spent waiting for a subsequent instructions, etc.).

In Cuttlesim, we can gather all those statistics at once, without adding a single piece of counting hardware. We use a code coverage tool called Gcov, which measures the number of times that each line of a C++ model was executed. Since the model matches the source design closely, these counts naturally provide detailed architectural information.

The following listing shows a representative snippet of Gcov output:

```
// Snippet of execute stage (baseline)
14890635: bits<32> nextPc = ctrlResult.nextPC;
14890635: if (nextPc != decoded.ppc) {
 2071903:   WRITE0_FAST(pc, nextPc);
...
// Snippet of scoreboard logic (baseline)
21424532: if (score1 != 0 || score2 != 0) {
 9211172:   FAIL();
```

```
// Snippet of execute stage (BP)
14890635: bits<32> nextPc = ctrlResult.nextPC;
14890635: if (nextPc != decoded.ppc) {
  165753:   WRITE0(pc, nextPc);
...
// Snippet of scoreboard logic (BP)
21579776: if (score1 != 0 || score2 != 0) {
 9211302:   FAIL();
```

We learn that the number of mispredictions went down from 2'071'903 to 165'753. From the same Gcov run, we also learn that for this specific program the decoding of instructions is very often stalled by the scoreboard, because of read-after-write hazards. This is explained by missing bypassing paths, forcing the processor to insert bubbles between back-to-back data dependent arithmetic instructions. From this evaluation and more in depth inquiry, the working architect may want to think of ways to reduce that potential bottleneck.

Designing profiling harnesses using hardware counters can take a significant amounts of work. Cuttlesim, in combination with Gcov, enables us to collect these performance numbers with low effort and high speed, making quantitative evaluation of rule-based designs significantly easier.

5 Related Work

Different hardware design communities use very different languages and tools and have different requirements in terms of the quality of generated circuits (when circuits are generated) and the need of timing fidelity of simulation. We discuss several of these communities and their approaches in this section.

Digital Design and Verilog. Digital design usually begins with writing RTL, and Verilog is a de-facto standard for doing so. Verilog was originally designed with explicit concern for software simulations [23] of circuit designs but by nineteen nineties became the standard interface for hardware synthesis as well because it provided a layer of abstraction over gate libraries. Verilog is actually structured around two sublanguages – a purely structural language to describe circuits and a behavioral language with higher-level features that are not intended to be synthesized to hardware, but instead used to write testbenches and behavioral models of the hardware.

Because of its pervasive use, a rich variety of tools that work in conjunction with Verilog are available. For example, as the designs became larger, people started generating structural Verilog using facilities coming from metalevel languages [1, 6, 22]. Nevertheless, users want much better support for both functional and performance debugging of complex designs and quicker evaluation of design alternatives.

When it comes to Verilog simulation, the programmer has several options depending on which subset of Verilog one is interested in simulating. When one is interested in simulating synthesizable designs expressed in the structural subset of Verilog with a single (or few) clocks, the best choice is often to do cycle-based simulation. In this case, the standard approach is to translate Verilog to C to get C functions that are computing the cycle state updates, yielding excellent performance [14, 24]. In contrast, to simulate more of the Verilog language (including behavioral constructs), simulators traditionally use an event-based (or activity-based) approach, e.g. in Icarus [26].

In both cases, the debugging experience is not ideal, leaning heavily on either printf debugging (display statements in Verilog) or direct observation of wave forms. When Verilog is compiled to C, the C generated is not intended to be read, and so C debugging tools are not of much use.

FPGA simulation: Another way to achieve very fast Verilog simulation is to map the design on an FPGA. There are two main inconvenience in doing so: (1) the synthesis, placement, and routing flow is very slow. It is not uncommon when targeting a modern FPGA to wait several hours for the design to be ready to be run on the FPGA; and (2) Some hardware structures don't map well over FPGAs and changing them reduces the fidelity of the simulation. Sophisticated techniques have been devised to preserve the cycle-accuracy of the original RTL even when FPGA implementations uses different hardware structures [19, 25].

Higher-level abstractions for digital designs. Rule-based languages [3, 5, 15, 18] provide an intermediate middleground. They embody a clean concurrency model that is intrinsically useful for debugging, formal verification and digital design exploration, and can be translated predictably and efficiently to RTL. However, they piggyback on RTL simulators for simulation. So far they have not exploited the extra structure coming from the rule abstraction for software simulation - that is what this paper is about.

HLS. High-level synthesis [4, 7–9, 11, 12, 16, 20] is the field of research focused on translating a subset of a software language like C, C++ or Haskell to circuits. The seduction of this approach comes from the fact that for a number of applications, especially in signal processing, the code is already available in some software language and thus, simulating such systems in software is straightforward. The main difficulty is usually in generating circuits of predictably high quality. For fixed-dataflow accelerators, the approach has been quite successful, even seeing commercial success in the recent years [27]. However, there has been little evidence of the success of the approach in designing complex control-heavy machines, like out-of-order processors. In fact, HLS systems usually rely on Verilog to interface with peripherals and other systems.

High-level Architectural Exploration. Computer Architects explore many complex microarchitectural alternatives before building a microprocessor. It is quite difficult to produce an RTL level design for each alternative. A proper evaluation also requires running huge benchmark programs on each design, so simulation speed is paramount. Hence, most architects evaluate ideas with models [2, 21] that do not represent actual machines' cycle count accurately. Evaluating architectural ideas at a higher level of abstraction level is at least three orders of magnitude faster than simulating the corresponding design at the cycle accurate level. It is also relatively easy to change software simulators but the constant danger of this approach is that the simulated design can easily omit critical details of the hardware design. One of the sources of the simulation speed of this approach comes from *direct execution*, that is, if we simulate an X86 machine on an X86 platform most of the simulator instructions can be simulated directly or natively on the underlying hardware.

6 Conclusion

Rule-based languages offer high-level semantics, powerful abstraction facilities, and composability, together with fine-grained control over generated circuits. They enable rapid prototyping and development of hardware designs without sacrificing circuit quality, offering an enticing middle ground between raw RTL and high-level synthesis from software languages.

Until now, most research on rule-based languages had focused on semantics and compiler technology for circuit generation. Little attention had been paid to simulation, debugging, or testing. These tasks were simply performed at the Verilog level, using generic RTL tooling.

We have shown that a much better experience is possible by fully decoupling hardware synthesis from simulation and debugging. We built a specialized compilation toolchain from the Kôika language to C++ that leverages high-level semantic properties to obtain significant speedups over state-of-the-art RTL-level simulators. Because our compiler preserves the structure of the designs, the cycle-accurate models that it generates can be used for debugging, exploration, validation, and testing, enabling hardware designers to leverage the whole ecosystem of software debugging. Through case studies, we have illustrated a new style of hardware development, dramatically improving over the state of the art in hardware-design debugging.

Although we have focused on rule-based designs, our insights have broad applicability, and we hope to see more applications of software tools to improve the hardware-development workflow in the future. The recent explosion of open-source hardware designs, toolchains, and processes, along with widespread availability of cheap FPGAs, has allowed a whole new set of hardware hobbyists to join the hardware-development community. We hope that improved tooling, especially of the kind that software developers are used to working with, will further lower the barriers to entry and make hardware design even more approachable.

References

[1] Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 1216–1225, 2012.

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[3] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. The Essence of Bluespec: A Core Language for Rule-Based Hardware Design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 243–257, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Helge Anderson, Stephen Dean Brown, and Tomasz S. Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays, FPGA 2011, Monterey, California, USA, February 27, March 1, 2011*, pages 33–36, 2011.

[5] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017.

[6] The MyHDL community. Myhdl. http://www.myhdl.org/.

[7] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees A. Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[8] George Economakos, Petros Oikonomakos, Ioannis Panagopoulos, Ioannis Poulakis, and George K. Papakonstantinou. Behavioral synthesis with systemc. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2001, Munich, Germany, March 12-16, 2001*, pages 21–25, 2001.

[9] Conal Elliott. Compiling to categories. *PACMPL*, 1(ICFP):27:1–27:27, 2017.

[10] Jakob Engblom. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, pages 1–6. IEEE, 2012.

[11] Daniel D. Gajski. Specc design environment. *System Design*, page 217–235, 2001.

[12] Abhijit Ghosh, Joachim Kunkel, and Stan Y. Liao. Hardware synthesis from C/C++. In *1999 Design, Automation and Test in Europe (DATE '99), 9-12 March 1999, Munich, Germany*, pages 387–389, 1999.

[13] Mentor Graphics. Handle-C. https://www.mentor.com/products/fpga/handel-c/.

[14] D. J. Greaves. A verilog to c compiler. In *Proceedings 11th International Workshop on Rapid System Prototyping. RSP 2000. Shortening the Path from Specification to Prototype (Cat. No.PR00668)*, pages 122–127, 2000.

[15] David J. Greaves. Further sub-cycle and multi-cycle schedulling support for bluespec verilog. In *Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2019, La Jolla, CA, USA, October 9-11, 2019*, pages 2:1–2:11, 2019.

[16] Sumit Gupta, Nikil D. Dutt, Rajesh Gupta, and Alexandru Nicolau. Loop shifting and compaction for the high-level synthesis of designs with complex control flow. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*, pages 114–121, 2004.

[17] Zainalabedin Navabi. *VHDL: Analysis and modeling of digital systems*. McGraw-Hill, Inc., 1997.

[18] Rishiyur Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.*, pages 69–70. IEEE, 2004.

[19] Michael Pellauer, Muralidaran Vijayaraghavan, Michael Adler, Arvind, and Joel S. Emer. A-port networks: Preserving the timed behavior of synchronous systems for modeling on fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 2(3):16:1–16:26, 2009.

[20] QBayLogic. Clash: A modern, functional, hardware description language. https://clash-lang.org/.

[21] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 475–486, New York, NY, USA, 2013. Association for Computing Machinery.

[22] Jane Street. Hardcaml. https://github.com/janestreet/hardcaml.

[23] Donald E. Thomas and Philip Moorby. *The Verilog hardware description language (3. ed.)*. Kluwer, 1996.

[24] Veripool. Verilator. https://www.veripool.org/wiki/verilator.

[25] Muralidaran Vijayaraghavan and Arvind. Bounded dataflow networks and latency-insensitive circuits. In *7th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2009), July 13-15, 2009, Cambridge, Massachusetts, USA*, pages 171–180. IEEE, 2009.

[26] Stephen Williams. Icarus verilog. http://iverilog.icarus.com/.

[27] Xilinx. Vivado hls. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.