Stepwise Debugging for Hardware Accelerators

Griffin Berlstein
Cornell University
USA

Christophe Gyurgyik
Cornell University
USA

Rachit Nigam
Cornell University
USA

Adrian Sampson
Cornell University
USA

ABSTRACT

High-level programming models for hardware design let domain experts quickly produce specialized accelerators. However, tools for debugging these accelerators remain tied to low-level hardware description languages (HDLs). High-level descriptions contain control-flow information that is lost in HDL code. We describe Cider, a stepwise debugger that exploits this information to provide software-like debugging abstractions for languages that compile to hardware. Cider uses Calyx, an intermediate language for accelerator generators that preserves control information. Cider provides breakpoints, watchpoints, state inspection, and source-level position mapping. Using case studies that examine one new and two preexisting accelerator generators, we demonstrate how Cider helps find and localize previously unreported bugs. By directly simulating a control-rich representation, Cider avoids wasting effort on inactive parts of the design and, despite being largely unoptimized, performs competitively with open-source HDL simulators.

CCS CONCEPTS

• Hardware → Hardware description languages and compilation; Bug detection, localization and diagnosis; Bug fixing (hardware); • Software and its engineering → Software maintenance tools.

KEYWORDS

Intermediate Language, Accelerator Design, Debugging, Accelerator Simulation

1 INTRODUCTION

High-level programming models [13, 15, 23, 24, 31] have brought accelerator design within reach of domain experts. Debugging these designs, however, remains challenging. The mainstream option is register-transfer level (RTL) simulation, which records the values of every signal in the circuit on every clock cycle [41, 44]. Scanning through a simulator’s waveform trace is radically different from using a standard software debugger such as GDB [38]: there is no “single-stepping” at any granularity other than clock cycles; there are no breakpoints; and all program state is flattened into wire signals. Worse, RTL simulators do not relate circuit state back to the source code in a high-level language. Waveform debugging is like using a software debugger that can only step through assembly-level instructions, not source-level statements.

An alternative is to debug programs in the source language’s semantics—for example, by writing an interpreter or by compiling it to a software executable and using GDB [34]. High-level execution can find functional problems, but it cannot help with bugs that only arise in hardware. These bugs are common [19]. Generating hardware from high-level descriptions is error-prone: compilers must introduce cycle-level timing, explicit parallelism, custom numerical representations, and physical resource sharing—all of which affect correctness.

This paper describes Cider, a debugging framework for programming languages that compile to hardware accelerators. The key contribution is in implementing a flexible program step abstraction that can correspond to source-level constructs in high-level languages. These steps undergird familiar interactive debugging tools, such as breakpoints, single stepping, and watchpoints, and they enable source-level stepping.

Our framework needs a program representation that can expose these coarse-grained program steps while faithfully simulating hardware-level semantics. Hardware description languages, such as Verilog [1], Bluespec [33], or Chisel [5], do not suffice because they have no information about high-level programs’ control flow. By the time an accelerator compiler has generated Verilog code, all the control information is encoded as state-machine logic, multiplexer signaling, and other unstructured hardware. We instead build Cider on Calyx [32], an existing intermediate language for accelerator compilers that preserves control information from high-level languages.

Cider’s insight is that, by using a higher-level representation rather than an HDL, we can exploit control flow information already present in accelerators to provide stepwise, source-level debugging for languages that compile to hardware. Figure 1 depicts Cider and its relationship to the existing Calyx compiler. Cider executes Calyx
IL directly, without relying on Verilog semantics or simulators, and uses position metadata to provide source-level debugging.

Cider supports three different user populations. It lets domain specific language (DSL) developers quickly prototype new compilers targeting Calyx and can help localize bugs in these compilers. As compilers mature, developers can add source mapping information to provide end users with a native debugging experience through Cider. Finally, Cider can help Calyx compiler engineers localize bugs in the core compiler toolchain and improve the stability of the entire ecosystem.

This paper’s key contributions are:

- We design an interpreter for the Calyx intermediate language [32] by distilling an (informal) IL-level semantics including its undefined behavior. The only previous implementation of Calyx was a compiler to RTL, which left these semantic details under-specified.
- Using the interpreter, we implement a debugger that provides software-like debugging constructs such as single stepping, breakpoints, watchpoints, and state inspection.
- We build flexible source-level mapping for Cider, which lets frontends step at the level of source-code constructs.
- We perform two case studies using Cider to debug accelerator generators: a loop-based imperative language [31] and a machine learning framework [11].
- We compare Cider’s performance to RTL simulation. Cider is 4.2× faster than Icarus Verilog [44], an interpreter-based simulator, and 2.9× slower than Verilator [41], a compilation-based simulator. Cider’s performance advantage stems from exploiting control information to avoid wasted effort simulating irrelevant parts of a design.

2 BACKGROUND

This section summarizes the mainstream debugging tools available to accelerator designers (Section 2.1) and introduces Calyx [32], the program representation that Cider builds upon (Section 2.2).

2.1 Abstractions for Debugging Accelerators

The lingua franca for hardware design is register-transfer level (RTL) descriptions. Programmers can implement computational accelerators directly in RTL or use higher-level languages that compile to it [15, 23, 31, 46]. Designers can therefore debug accelerators at two levels of abstraction: at the functional level, before compiling to RTL; or in RTL, using waveform debugging.

Waveform debugging. RTL code describes how to compute the values in wires and registers each clock cycle. Designers debug a hardware design by either simulating it in software [34, 41, 44] or by running it on an FPGA and recording a subset of the signals using a logic analyzer [16, 21, 45]. Either route produces a waveform trace that exhaustively records, on every clock cycle, the value of every monitored signal. Waveform viewers [9] let designers visualize values while zooming and panning through linear time.

Waveforms reflect RTL semantics, so they can be difficult to associate with bugs that occur in higher-level languages that compile to RTL. They make it hard to temporally localize when a bug happens because the only notion of time step is a clock cycle—not a logical program step. They also complicate spatially localizing bugs because all signals may be equally relevant at any time.

Reacting to this difficulty, tools like Cuttlesim [34] improve post-compilation RTL debugging by raising the abstractions to the level of rule-based hardware design languages, such as Kōkâ [8]. Like HDLs, these languages and tools are built for hardware designers and excel at designing arbitrary circuits and CPUs. Our focus in this paper is instead on high-level algorithmic and domain-specific languages that aim to let domain experts productively design computational accelerators.

Functional debugging. The alternative to waveform debugging is executing programs using their high-level language semantics, before compiling to RTL. For example, high-level synthesis (HLS) compilers translate C to RTL [20, 46]; designers can debug these C programs directly with standard tools like GDB [38]. This approach cannot catch bugs introduced during the compilation to hardware. While users may hope that hardware compilers are always faithful to language semantics, in practice this is rarely the case. Bugs pervade even popular commercial HLS tools [19]. Fundamentally, compilation to RTL needs to introduce hardware-level concerns that are difficult to fully abstract, such as custom numerical formats and pervasive fine-grained parallelism. To bridge this semantic gap, some work has aimed to map waveform data back to high-level program state [12, 16, 18]. However, these efforts are point solutions specific to C-based HLS tools—each new compiler must build such a capability from scratch.

The goal of our work is to provide an infrastructure for debugging in languages that compile to hardware. We build a core debugging engine that exploits the control information from a given high-level language while faithfully executing hardware-level simulation.

2.2 Calyx

Calyx [32] is an intermediate language (IL) for hardware generation. It combines hardware dataflow and control flow. Unlike software ILs [25], it directly represents physical hardware resources; unlike
which encapsulates a set of hardware resources along with a control value of corresponding to the example function’s argument, define their interface. Our counter component has an input port and output description. Components have input and output ports.

Components. Calyx’s analog to a function definition is a component, which encapsulates a set of hardware resources along with a control flow description. Components have input and output ports that define their interface. Our counter component has an input port corresponding to the example function’s argument, x:

```
const step: real = 1.0;
fn counter(x: real) {
  dest = x;
  for _ in 0..4 ( dest += step; }
  return dest;
}
```

Figure 2: Pseudocode that computes dest = x + 4 × step.

```
component counter(go: 1, x: 32) -> (done: 1) {
  cells {
    dest_m = 1D_memory(32, 1);
    step = constant(32, 32768);
    acc_r = register(32); idx_r = register(2);
    fpa = fixed_point_adder(32, 16, 16);
    add = adder(2);
    le = less_than_equal_to(2);
  }
  wires {
    group init { ... } // idx_r <= 0, acc_r <= x
    group incr_idx {
      add.left = idx_r.out; add.right = 2'd1;
      idk_r.in = add.out; idx_r.write_en = 1'd1;
      incr_idx[done] = idx_r.done;
    }
    group incr_st {
      fpa.left = acc_r.out; fpa.right = step.out;
      acc_r.in = fpa.out; acc_r.write_en = 1'd1;
      incr_st[done] = acc_r.done;
    }
    group upd {
      dest_m.write_en = 1'd1; dest_m.addr0 = 1'd1;
      dest_m.in = acc_r.out; upd[done] = dest_m.done;
    }
    // Loop condition
    comb group cond { le.left = idx_r.out; le.right = 2'd3; }
    control {
      seq {
        init;
        while le.out with cond {
          par { incr_idx; incr_st; }
        }
        upd;
      }
    }
  }
}
```

```
Figure 3: A Calyx component generated from Figure 2. It uses a fixed-point approximation of real values with 16 integral and fractional bits.
```

This section explains Calyx by showing how it compiles a simple high-level program. The program in Figure 2 implements a simple counter that iteratively increments the input, x, by four times the value of step, here defined as 1.0. Figure 3 lists a Calyx implementation of this function; we introduce its constructs in this section. Section 3 uses the same example to show how our proposed system, Cider, enables productive debugging of hardware accelerators.

```
// Cider
```

```
component counter(go: 1, x: 32) -> (done: 1)
```

Every port has a bit width. We implement the pseudocode’s real type with a 32-bit fixed-point number. Calyx uses one-bit interface ports go and done to transfer control to and from a component.

Structure. Lines 1 and 3 in Figure 2 declare variables, dest and step. To implement these in hardware, we need a constant and a mutable memory. Calyx components instantiate subcomponents in their cells section (lines 2–9 of Figure 3):

```
dest_m = 1D_memory(32, 1);
step = constant(32, 32768);
```

Here, constant and 1D_memory are components built into Calyx’s standard library. The step declaration includes a bit width (32) and a value; the dest_m memory specifies its element width (32) and size (1).

Groups. We next implement the for loop on line 4. First, the Calyx program instantiates two registers to hold the accumulated dest_m value and the loop counter:

```
acc_r = register(32); idx_r = register(2);
```

The program also needs machinery to increment and check the loop-control register idx_r and to perform the fixed-point accumulation:

```
fpa = fixed_point_adder(32, 16, 16);
add = adder(2);
le = less_than_equal_to(2);
```

The fixed-point adder controls the numerical representation for dest_m in the high-level program (here, 16 bits each for the integer and fraction parts). We also include the add and le comparator cells to implement the loop’s iteration.

Calyx components define groups which implement logical operations by wiring together their cells. Our example uses a group incr_st to implement the loop body, dest_m += step:

```
group incr_st {
  fpa.left = acc_r.out; fpa.right = step.out;
  acc_r.in = fpa.out; acc_r.write_en = 1'd1;
  incr_st[done] = acc_r.done;
}
```

Calyx groups consist of simultaneous (unordered) assignments between ports. This group feeds the fixed-point adder, fpa, with two inputs: acc_r.out, the current value of the accumulator register, and step.out, a constant value. It then writes the sum into acc_r by setting its input port and its write-enable control port, write_en. The final line assigns to a special done signal indicating when the group’s work has finished.

The Calyx implementation in Figure 3 includes a similar group incr_idx to increment the loop index and a combinational (stateless) group cond that uses the le comparator to check the loop condition. Finally, the init and upd groups initialize and finalize the state for the function. The rest of the Calyx code will use these small units of computation to implement the example’s logic.

Control. Finally, Calyx components use a control program to orchestrate the named groups. A component’s control section resembles an imperative program where the leaf statements are group names. Our example uses a while statement to implement the loop:

```
while le.out with cond {
  par { incr_idx; incr_st; }
```
The incr_idx and incr_st statements are group activations that invoke the connections in the group. Calyx includes a par statement that runs these actions in parallel. The while loop runs until the cond group produces zero on the le.out port. The full program sequence loop with the initialization and clean-up groups:

```c
seq {
  init;
  while ... {...}
  upd;
}
```

Calyx also has a conditional statement if and a call-like operator invoke to run subcomponents.

Figure 3 lists the complete example code. The existing Calyx compiler translates this code to RTL by implementing state machines and other control logic to orchestrate the groups’ assignments; for more detail, see Nigam et al. [32]. Cider debugs Calyx programs by directly executing their control programs instead of translating them to hardware.

3 DEBUGGING WITH CIDER

Figure 3 has a bug. If we use the Calyx compiler to translate it to Verilog and run it with an RTL simulator [41], it runs forever; there is an infinite loop. This section uses this buggy program as an example to demonstrate the use of our new system, Cider, in an accelerator debugging workflow.

The fact that this program runs forever would be immediately confusing to a user debugging solely from the high-level code in Figure 2. After all, the only loop in that program is a bounded for loop! Clearly, there has been an error in translating the high-level for loop into the Calyx while loop that realizes it. Using Cider, we can directly observe this problem.

State inspection and watchpoints. Cider includes an interpreter for the Calyx IL: it can execute the program without compiling it to RTL code first. To start the debugging process, we run the program using Cider; immediately, we see:

```
WARN - Integer overflow, source: counter.add
```

Since Verilog’s integer operators silently allow overflow, the simulator issues no warning. Cider, in contrast, implements Calyx’s semantics, in which implicit overflow is an error. And since all components have an explicit name, Cider can report the specific adder that produces the error.

This warning reveals that the two-bit adder responsible for incrementing the loop counter is overflowing, though it is not yet clear whether this overflowed value gets saved in the counter register. We can start Cider’s interactive debugger and step through the program execution to see how the value in idx_r, the loop counter, changes after each execution of incr_idx, the only group which uses counter.add. Cider’s watchpoints allow this kind of inspection:

```
seq {
  init;
  while ... {...}
  upd;
}
```

Here we see how Calyx groups enable a coarser view of time. Cider knows what groups are running, which lets users inspect signals only at computationally relevant points in time. This watch command instructs Cider to print the value of idx_r.out every time the group incr_idx finishes executing (specified by after). The counter overflows after being incremented to 112 = 3. On closer inspection, we notice that our condition group cond implements the comparison idx_r.out ≤ 3; however, since the maximum value representable with two bits is 3, the condition will always hold. To fix this issue, we can change the counter register idx_r to contain three bits. The corrected line reads:

```
acc_r = register(32); idx_r = register(3);
```

This demonstrates the advantage of building debugging abstractions using coarse-grained (non-cycle-based) temporal schedules. Users can step over several clock cycles of execution and focus on specific sub-circuits while debugging. In contrast, traditional RTL simulation works only at the level of clock cycles, forcing users to untangle logically unrelated signals.

Value representations and breakpoints. After fixing the overflow, we execute it again using Verilator. This time, the program finishes executing, but the output memory dest_m contains the value 0. Cider reports this error:

```
Error (counter.dest_m): Invalid memory access. Given index (1) but memory has size (1)
```

Cider’s runtime checks catch an invalid memory access. Again, such errors are not caught at the RTL level—out-of-bounds indexing silently yields an undefined value. dest_m contains exactly one element which means the only valid index into it is 0. However, the upd group attempts to index it using 1:

```
...; dest_m.addr0 = 1’d1;
```

While an obvious bug, neither Verilator nor Icarus Verilog catch it. For example, Verilator specifies that out-of-bounds accesses are only checked when the memory size is a power of two [42]. To fix this, we can change the assignment’s right-hand side to 1’d0.

Finally, the RTL simulation terminates with this output:

```
"dest_m": [ 100000000000000000 ]
```

The value here is incorrect, but it is hard to tell when reading the bit-level representation. The value is the fixed-point representation of 2.0 whereas it should be 4.0. Fortunately, like some waveform visualizers [9], Cider can reinterpret bit-level values easily.

To debug this incorrect value, we can use breakpoints. Cider supports breakpoints that stop execution at Calyx groups. We interactively set a breakpoint on the group incr_st, which is responsible for incrementing the input value:

```
> break incr_st
> continue
Breaking: counter::incr_st
```

While the break and continue commands let us control the execution, step-over lets us skip logical chunks of the execution. There is no equivalent to step-over in RTL simulation since there is no logical time step beyond a clock cycle. Cider’s print command supports
formatting codes like `\u.16` to print as an unsigned fixed-point value with 16 fractional bits. We can see that, after the first execution of `incr_st`, the value in `acc_r` is 0.5 instead of 1. The likely suspect in this case is our fixed-point constant `step`. We can check this hypothesis:

```plaintext
> print \u.16 step.out
step.out = 0.5
```

Writing fixed-point constants using unsigned integers is tricky; in this case, the `step` value was incorrectly translated and needs to be shifted to the right by one. The corrected definition is:

```plaintext
4 step = constant(32, 65536);
```

Cider leverages the high-level control flow of hardware accelerators to provide software-like debugging abstractions. Unlike debugging flows for RTL or customized accelerator generators, Cider provides a generic set of tools that interact with time and state in a coarse-grained way. The key insight is that plain RTL is missing a representation of an execution schedule that delineates the larger-than-cycle steps in its execution, and it does not decompose the hardware into logical groups that permit local reasoning. Breakpoints, stepping, and watchpoints in Cider rely on the accelerator’s own custom notion of granularity: its groups define the units of work that programmers navigate through while debugging.

4 THE CIDER INTERPRETER

Cider’s execution engine is a new interpreter for Calyx programs. Its goal is to faithfully implement Calyx’s semantics with a fail-stop policy for undefined behavior (UB). Emitting actionable, timely errors for UB is critical for surfaced subtle problems to users. This is in contrast with the only other Calyx implementation, which is a compiler to Verilog that resolves Calyx’s UB in arbitrary ways to prioritize efficiency. Implementing this safe interpreter forced us to design an informal semantics for Calyx independent of compilation to hardware and make concrete decisions about ambiguities in the original language description [32].

Our interpreter consists of a structural simulator (Section 4.2), which executes individual groups in a setup resembling RTL simulation, and a control simulator (Section 4.3), which executes its imperative control statements and uses the structural simulator as a subroutine.

4.1 Environment Model

While software languages often represent their execution environment with a call stack, heap, and program counter, circuit simulation demands different abstractions. Circuits have neither function calls nor pointers; all state exists in specific storage elements and all wires and ports are in the same execution scope. At every cycle, values on any wire or port could potentially change. Furthermore, there are no program counters because connections are always active, even if quiescent.

Cider’s environment model lies between these two extremes. As in circuit simulation, the environment contains values for all ports in the program—there is a mapping from each cell, such as a register, to its internal state. However, unlike software simulation, there is also a program counter and execution stack embodied in each component’s control schedule.

Figure 4 shows the environment structure: a tree of interpreter environments where the parent-child relationship is structural containment. A child node is a subcomponent of the parent node. A component’s local environment defines a mapping from each port to a value and from each cell to its internal state, if any. The program counter points to the currently active group in the component’s control program. The environment handles `par` blocks by duplicating the program counter and environment mappings, allowing each “thread” to execute in its own parallel world (Section 4.3).

4.2 Structural Simulation

Cider’s structural simulator is its engine that executes individual Calyx groups, ignoring control statements. As Figure 3 illustrates, each group is an unordered set of simultaneous assignments to ports. Interpreting a group therefore resembles classic RTL simulation [6, 17]: the interpreter can treat the assignments as dataflow “wires” that propagate values between cells. Structural simulation takes the form of two nested loops: an inner loop that executes assignments within a single cycle, and an outer one that iterates across cycles.

The inner stabilization loop iteratively executes all assignments until the port values stop changing. Components that represent combinational logic, such as adders, update their output ports based on their input ports. Calyx assumes assignments are non-conflicting, so Cider raises an error if multiple assignments drive the same port (Section 4.5). When the stabilization loop converges, a clock cycle has completed and the outer simulation loop progresses, updating cell state in sequential elements such as registers. The simulation loop runs until the group’s done condition becomes true. Both loops recursively evaluate subcomponents, traversing the environment tree (see Figure 4) to update the state in each cell’s children.

This structural simulation strategy intentionally uses a basic form of classic hardware simulation for clarity: Cider’s novelty stems from the way it orchestrates structural simulation through control. We expect that more advanced hardware simulation techniques could apply analogously to Cider’s structural simulator to improve its performance [7, 43].
4.3 Control Simulation

Cider’s control simulator executes Calyx’s control programs (lines 29–35 in Figure 3). It resembles interpreters for any imperative language and delegates group execution to the structural simulator. For example, it executes Calyx’s seq statement by recursively executing each child statement to update the environment in turn. The invoke statement transfers control to a subcomponent’s control program and waits for it to finish. Other control operators like if and while work similarly.

The exception is par, which runs several statements in parallel. The interpreter simulates parallelism by advancing the child statements together, one cycle at a time. The original Calyx paper [32] left the precise semantics of parallelism largely undefined. In practice, the compiler offers no guarantees about the relative timing between simultaneous arms of a par block. As a result, data races—two accesses, at least one of which is a write, to the same value in different arms of a par statement—yield undefined behavior. Cider detects races and yields an error. It executes each child of a par statement with a separate copy of the initial environment. Figure 4 illustrates a par statement in main.my_mul that yields three of these “parallel universe” environments. When the par finishes, it merges the copies while detecting conflicting updates to signal errors on races.

4.4 Primitive Simulation

Calyx programs can use black-box RTL modules by defining their interfaces using a primitive definition. Cider implements high-level behavioral models for a set of core primitives. These primitives appear at the leaves of the interpreter’s environment tree (see Figure 4). Primitives interact with the interpreter through combinational and stateful interfaces. The combinational interface interacts with the stabilization loop to describe how to compute output values based on input values. The stateful interface lets components store internal state on iterations of the simulation loop. A single component may use both interfaces: for example, a memory may implement combinational reads and sequential writes.

Cider’s primitive models differ from their Verilog implementation by prioritizing fail-stop behavior for UB. For example, arithmetic primitives can signal warnings or errors on numerical overflow. While overflow has truncation semantics in RTL, it is an error in Calyx—surfacing these errors can help identify problems with numerical representations.

4.5 Undefined Behavior

Unlike a compiler, a debugger must predictably simulate incorrect programs. We identify several categories of UB in Calyx and resolve them predictably to aid debugging.

Undriven signals. Reading a port that has not been written is UB in Calyx. HDL simulators typically model these undriven signals with multi-value logic, using special values like “X” or “Z” to indicate that the value is unknown. These special values can have counterintuitive semantics [40] and have no source-level interpretation. Cider, like the Calyx compiler’s existing Verilog backend, uses 0 for all undriven signals.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>break [grp]</td>
<td>Create a breakpoint at group</td>
</tr>
<tr>
<td>step-over [grp]</td>
<td>Advance the execution over a given group</td>
</tr>
<tr>
<td>step [n]</td>
<td>Advance the execution by n steps (defaults to 1)</td>
</tr>
<tr>
<td>watch [grp] [prt]</td>
<td>Watch a given group with a print statement</td>
</tr>
<tr>
<td>continue</td>
<td>Continue till next breakpoint</td>
</tr>
<tr>
<td>display</td>
<td>Display the full state</td>
</tr>
<tr>
<td>print [fmt]</td>
<td>Print target value. fmt code describes how to interpret the value</td>
</tr>
<tr>
<td>print-state [fmt]</td>
<td>Displays the current program location. Uses source metadata if present; otherwise shows Calyx program</td>
</tr>
<tr>
<td>where</td>
<td>Disable/Enable target breakpoint</td>
</tr>
</tbody>
</table>

Parallel conflicts. Calyx has DRF0 semantics [3]: data races between different arms of a par statement have undefined behavior. The compiler is therefore free to choose any relative timing among groups of parallel threads, including running them sequentially. To avoid unpredictable results from racy code, Cider implements race detection to signal an error when different par arms issue conflicting assignments to the same signals.

Intra-group conflicts. Calyx defines multiple active drivers to the same port as an undefined behavior. This snippet of a Calyx group contains two guarded Calyx assignments—such assignments are only active when their conditional is true:

```
r.in = cond_a ? 32’h5d;  
r.in = cond_b ? 32’h7d;
```

If cond_a and cond_b are simultaneously true then these two assignments will both write distinct values to the same port and Cider will raise an error.

Loop trip-count bounds. Calyx programs use the attribute @bound to provide loop trip counts for while statements. This attribute provides static information about the number of iterations a loop will conduct, which results in more efficient Calyx lowering. Disagreements between this annotation and the actual loop logic can cause miscompilation. Cider validates the loop bound by tracking the loop iteration count.

Invalid memory indexing. Out-of-bounds memory accesses are UB in Calyx. Whereas RTL simulators silently yield default values for invalid indices [2], Cider eagerly signals an error.

5 DEBUGGING INFRASTRUCTURE

We build Cider’s debugging mechanisms on top of its interpreter from the previous section. Cider can execute any Calyx program, but is most powerful when debugging high-level Calyx programs—those with complex control programs and many fine-grained groups. Cider exploits this control information in accelerators generated by higher-level languages and cannot recover this information from arbitrary RTL code.

5.1 Interactive Steppable Execution

RTL simulators can only advance, or step, the program in clock-cycle increments, if they support interactive execution at all: waveform debugging typically works in batch mode. Cider provides a more software-like debugging experience with manual control.
of program advancement and commands to display the current program counters (or source locations, see Section 6). Cider accomplishes this with a coarser-grained notion of a program step built using the control program and group abstraction in Calyx.

Two central constructs in Cider are a step-over command that advances execution past the current group and a lower-level step (cycle) for finer-grained inspection of a group’s execution. Stepping over lets users examine computation in logical steps that are larger than a single cycle. Even functionally simple groups can take multiple cycles: a multiplication, for instance, might take 3 cycles each time it runs. Being able to step over such a computation means the user can think of it in terms of input/output behavior—the same way we think of functions in software—and only resort to cycle-level timing when necessary. Additionally, users can focus on the set of active groups to understand where bugs may arise instead of having to look at the entire state of the circuit every cycle.

This section describes the three debugging mechanisms in Cider that build on the interpreter and these core constructs: breakpoints, state inspection, and watchpoints.

5.2 Breakpoints

RTL simulators generally do not support breakpoints since there is no discrete computational unit to break on. For example, setting a breakpoint on a line is unlikely to be helpful because, in general, assignments “run” every cycle. This is true even in the presence of conditional blocks in RTL, which compile into multiplexers and thus always compute all their inputs. Worse, a conditional statement in a high-level language is unlikely to correspond to an if statement in Verilog, making the relationship with source code hard to track.

In contrast, Calyx’s groups present a natural opportunity to implement breakpoints since they represent discrete computational blocks that the control program schedules. Cider supports setting breakpoints on any group in the Calyx program and provides a continue command that advances the execution until it hits a breakpoint or the end of the program.

Breakpoints and continue let the user skip program segments that they know to be behaving correctly and keep their attention on potential error sources. As a result, Cider can reduce the signal-to-noise ratio in the debugging experience. This listing shows a simple interaction:

```
> break conv2d::upd20
> continue
Hit breakpoint: conv2d::upd20
> print-state main.conv2d.k1
  k1 = [00000000000000000000000000000001]
> step-over conv2d::upd20
> print-state main.conv2d.k1
  k1 = [00000000000000000000000000000001]
```

This snippet uses a breakpoint and the step-over and print-state commands on a subcomponent’s group to inspect the state update it performs. This prints out the internal state associated with the register k1 in the conv2d component.

Breakpoints are set on component definitions—the global declaration of a component’s behavior—rather than instances, akin to how software debuggers place breakpoints in function definitions rather than specific calling contexts. This means that if there are multiple instances of a single component, any of these instances will trigger the breakpoint.

5.3 State Inspection

The state of a circuit consists of the bit values on each port and wire. While some waveform viewers [9] support custom formats, waveforms generated by RTL simulations usually make this data available in the form of unsigned integers in binary, decimal, or hexadecimal representations. However, designing high-performance accelerators requires using custom data formats, such as fixed-point numbers.

Cider supports two commands for printing data: print, which prints out the value on a port, and print-state, which prints the value stored in stateful primitives like registers. They support format codes which can reinterpret bit values as unsigned or signed fixed-point or decimal numbers. These two commands show two different formats for the same value:

```
> print \u div.out_quotient
  div.out_quotient = 424967279
> print \ls div.out_quotient
  div.out_quotient = -17
```

The format codes \u and \ls represent data as unsigned or signed integers. To use a fixed-point format, we can specify the number of bits in the fractional part. For example, \u.16 formats the port as an unsigned fixed-point number that uses 16 bits for its fractional part.

5.4 Watchpoints

Watchpoints are a common command provided by software debugging tools. Unlike breakpoints, watchpoints do not interrupt the execution of the program and instead just print out the value of a particular port. Watchpoints are particularly useful when the user wants to understand how a value changes throughout execution:

```
> watch incr with print-state \u i
> continue
  i = 0  i = 1  i = 2  i = 3  ...
```

This compressed snippet uses a watchpoint to track the value of a loop counter over the lifetime of a simple program.

Cider supports watchpoints that trigger at the start or end of a group’s execution by desugaring them into a breakpoint, print, and continue command (or a breakpoint, step-over, print, and continue for watchpoints attached after a given group). Cider provides both options to streamline the observation of a group’s effects. Because watchpoints track individual ports or cells, they provide a narrow lens for spatial bug localization, which is ideal for determining if a specific piece of logic is incorrect.

6 SOURCE-LEVEL DEBUGGING

While Cider’s core machinery implements steps at the level of Calyx IL control statements, it can use these steps to provide source-level stepping through frontend language constructs. Cider provides a flexible way to associate Calyx-level positions with source-level positions. We introduce Cider’s source position tracking using Dahlia [31], a loop-based imperative language that compiles to Calyx, as an example.

Figure 5 lists an example Dahlia program that doubles the elements of an array. We modify the Dahlia compiler to attach source-position metadata to the IL via Calyx’s metadata attributes. An attribute @pos(n) marks a control statement with an integer tag n.
We also generate a table that maps each tag to a source position. For example, the Dahlia compiler emits this Calyx control program:

```cpp
control {
    seq {
        @pos(0) let0;
        @bound(8) while le0.out with cond0 {
            seq {
                @pos(1) upd0;
                let1;
                upd1;
                @pos(8) upd2;
            }
        }
    }
}
```

And an embedded metadata table:

```c
metadata #( 0: for (let k: ubit<4> = 0..8) { 1: y_int[k] := x_int[k] + 2; }
```  

The @pos tags attach a source position number to leaf nodes in the control program and the metadata table associates each of these with a source position. Tags need not be unique; in this example, `let0` and `upd2` have the same tag because they both map to logic that realizes the `for` loop in Figure 5. Some groups are missing `@pos` tags because they represent intermediate computations without direct correspondence in the source code.

Cider’s `where` command (Table 1) uses the attributes and position table to display the source location for active control statements:

```c
> where y_int[k] := x_int[k] + 2;
```

Multiple locations may be active simultaneously because of parallelism in the design; Cider reports all active positions.

Dahlia’s imperative control maps easily to Calyx’s control operators. To demonstrate Cider’s flexibility, we encode source-level information for a more esoteric frontend language, the systolic array generator.

**Systolic array generator.** Calyx’s systolic array frontend differs from other frontends because its input does not have source positions in a typical sense. The input is a declarative description of a systolic grid of processing engines (PEs) that collectively compute a single linear-algebra operation. Cider’s flexible source-position debugging can nonetheless convey high-level information about the computation’s control flow.

We modify this frontend to generate metadata describing the progress within each PE. The compiler attaches `@pos` tags to Calyx `invoke` statements that advance PEs. Due to the design’s inherent parallelism, multiple positions are active at a time:

```c
> where
    pe 0,1: running iteration 2
    pe 1,0: running iteration 2
    pe 0,2: running iteration 2 ...
```

7 **CASE STUDIES**

We demonstrate Cider using bugs in two case studies: Section 7.1 uses the preexisting Dahlia-to-Calyx compiler [32], and Section 7.2 uses a new TVM-to-Calyx compiler.

7.1 Dahlia Compiler

Dahlia [31] is a loop-based language that uses a novel type system to guarantee that the generated hardware does not have conflicting memory accesses. Like other high-level synthesis tools, Dahlia supports loop unrolling and memory partitioning to express DOALL parallelism. We use the open-source Dahlia-to-Calyx compiler to study the implementation of Polybench [28] and other kernels reported in Nigam et al. [32].

**Parallel writes to memories.** Dahlia’s `unroll` keyword unrolls loops to parallelize computation by duplicating hardware. The Calyx backend for Dahlia relies on the type checker’s guarantee that unrolled loops do not create memory conflicts. However, when executing some of the Dahlia benchmarks, Cider discovers this error:

```
Error: parallel assignments not disjoint: alpha.addr0
```

This message indicates a parallel conflict (Section 4.5), meaning multiple groups are attempting to write to the same port in parallel. We can debug this error by looking at individual groups that write to `alpha.addr0`. Unlike RTL debugging, we do not have to look at every assignment to `alpha.addr0` since only groups that are in a parallel statement can cause this error. The two offending groups are these:

```c
group upd31 { alpha.addr0 = 1'0d; .. } // Conflict
group upd32 { alpha.addr0 = 1'0d; .. } // Conflict
```

As Section 4.5 describes, the parallel writes to `alpha.addr0` are illegal. The position tags help identify the corresponding source line in Dahlia, in turn allowing us to localize and confirm the bug.

An alternative to Cider in this case is instrumenting RTL to catch multiple writes. We implement this strategy by modifying the Calyx Verilog backend. While this technique does let Verilog simulation catch this conflict, it remains challenging to map it back to the source-level Dahlia program since parallel assignments in Verilog do not correspond one-to-one with statements in the original Dahlia.

**Parallel scheduling bugs.** Next, we found a bug in the implementation of a softmax kernel in the Calyx test suite:

```
Error: parallel assignments not disjoint: x_0.in
```

Cider detects another parallel write bug which is, surprisingly, not caught by the above Verilog-level checks:

```
par {
    let1;
    seq { let2;
        while le1.out with cond1 { seq { upd8; upd1; }}
    }
}
```

The groups `let1` and `upd1` write different values to the same register `x_0`. Verilog simulation checks do not catch this bug because the particular compilation strategy used by the compiler schedules

1https://github.com/cucapra/dahlia/issues/384

---

Figure 5: Dahlia code to double the elements of an array.
the two groups in a non-conflicting manner; any change to the compilation scheme might trigger the run-time assertion. This is the key difference between Cider’s checks and the checks in Calyx’s RTL backend—the latter can only find problems in a particular RTL realization of a Calyx program, while the former operates with Calyx’s semantics.

### 7.2 TVM-to-Calyx Compiler

TVM [11] is an open-source machine learning compiler. We built a TVM-to-Calyx compiler to test Cider’s efficacy at debugging large accelerators. Our compiler generates Dahlia kernels from TVM’s Relay IR [36], lowers them to Calyx, and stitches them together to implement complete neural networks. This frontend generates large, fixed-function implementations of CNNs. We do not claim that the accelerators are competitive or even practical to realize on an FPGA; we focus here on their correctness in simulation.

For our case study, we simulate a TVM implementation of the LeNet network [26] compiled to Calyx. GPU-based execution of this kernel uses floating-point operations that are too expensive for FPGAs; we instead use a fixed-point approximation. Because of this representation mismatch, we expect the Calyx program’s output to differ numerically from TVM’s, which makes silently propagating RTL bugs hard to recognize and localize. While our quick spot tests on a few inputs with the Calyx code seem to work, our experience with the Dahlia case study suggests that RTL semantics are too relaxed to catch subtle bugs. We instead use Cider to catch possible problems in the design.

**Source locations.** First, we extend the TVM backend to generate source location information to aid debugging. The compiler generates a new component for each TVM operation and uses the @pos tag to call them in the main component. Since each invoke has a direct correspondence in the TVM source, we attach a unique @pos tag for each one:

```plaintext
control { 
  @pos(0) invoke conv2d_1x20x24x24...(.)
  @pos(1) invoke bias_add_1x20x24x24...(.)
}
```

And generate the corresponding metadata table:

```plaintext
metadata # { 
  0: let %x: Tensor[...] = nn.conv2d(%data, ...);
  1: let %x1: Tensor[...] = nn.bias_add(%x, ...);
}
```

**Out-of-bounds access.** Cider immediately reports an out-of-range memory access when we run the kernel:

```plaintext
Error (main.data): Invalid memory access. Given index (0, 1, 0, 0) but memory has dimension (1, 1, 28, 28)
```

The error states that in a four-dimensional memory, the second dimension has size 1 but is being accessed with an out-of-bounds index 1. Unlike RTL simulators, Cider treats this as a hard error (Section 4.5). The debugger points to the main.data memory cell, which is only used by the first component, a two-dimensional convolution kernel.

The likely cause of this bug is an invalid loop nest. We investigate the second index to the memory, which is driven by a loop counter called k1. Setting a watchpoint at the beginning of the appropriate inner loop, we see:

```plaintext
> watch conv2d::let16 with print-state \u conv2d.k1
```

Cider shows us that the inner loop runs multiple times, causing the index to overflow. We investigate the loop-entry condition which, checks if the value of k1 is less than or equal to const30.out:

```plaintext
> print \u conv2d.const30.out
const30.out = 19
```

The TVM-to-Calyx compiler generates incorrect loop bounds, running 20 times instead of once. We track the bug back to a function that incorrectly takes a default value instead of computing the loop bound for the convolution kernel. After fixing this bug in the TVM compiler, this inner loop only runs once, and the error disappears.

**Fixed-point overflow.** Rerunning the LeNet kernel through Cider reports a new warning:

```plaintext
WARN - Overflow in fixed-point multiplier: 81117.5903007190208882 to 15581.5902862548828125
```

All numerical approximation should be explicit—it should not arise from implicit overflow. This is a hardware-specific bug; it does not arise in the original (floating-point) TVM code.

In this case, the warning comes from an exponentiation component in the final softmax layer. The component computes $e^x$, and it overflows on large inputs $x$. We fix the bug by modifying the softmax kernel to normalize the inputs it provides to the exponentiation component. In pseudocode:

```plaintext
def softmax(v: List[float]): List[float]
  e = exp(v - max(v))
  return e / sum(e)
```

Numerical approximation is crucial for high-performance hardware accelerators but challenging to debug for traditional RTL simulators. They cannot quickly localize such problems because the same fixed-point multiplier could be time-multiplexed and used by multiple parts of the circuit. Additionally, they silently propagate overflow errors since they do not consider overflows anomalous. By implementing Calyx’s higher-level semantics, Cider imposes stronger restrictions on program behavior to aid design of computational accelerators.

### 8 PERFORMANCE EVALUATION

Performance is important for a productive debugging workflow. To complement our qualitative case studies that demonstrate Cider’s ability to help find bugs, this section quantitatively measures its performance. We answer these questions:

- Does simulating Calyx programs with high-level control flow improve performance?
- How does Cider compare to state-of-the-art RTL simulators?
- Can Cider scale up to large accelerator designs?

We compare Cider against two open-source RTL simulators: Verilator [41] and Icarus Verilog [44]. Verilator simulates Verilog programs by first compiling them to C++, whereas Icarus Verilog acts as an interpreter. Cider is not heavily optimized and does not seek to outperform these RTL simulators; our evaluation seeks to understand the fundamental performance impact of exploiting control information.
We also compare Cider when running high-level vs. low-level Calyx programs. The existing open-source Calyx compiler [32] lowers the control-heavy Calyx programs emitted by compiler frontends to a control-free form that closely resembles RTL (see Figure 1). Comparing these two forms lets us directly measure the performance impact of Calyx’s control information that is lost during compilation to RTL.

Our benchmarks consist of the Polybench kernels [28] written in Dahlia [31], a Calyx implementation of a ternary content addressable memory (TCAM) for packet processing, and code from our TVM-to-Calyx compiler case study (Section 8.3). Comparing these two forms lets us directly measure the performance impact of Calyx’s control information that is lost during compilation to RTL.

We hypothesize that Cider’s performance advantage over Icarus Verilog stems from its ability to skip executing inactive parts of the program. High-performance RTL simulators also avoid re-executing subcircuits whose inputs do not change, but this change tracking comes with its own overheads that Cider can sidestep.

However, an accurate reflection of a programmer’s experience must incorporate startup time. Modular compilation often does not help with high-level accelerator generators, which typically regenerate the entire RTL design on every change. Figure 6c shows the end-to-end running time, counting both compilation and simulation time. In all cases, Cider finishes simulation before Verilator finishes compilation. On average, Cider’s simulation is 53× faster than Verilator’s compilation phase. This short end-to-end loop makes Cider helpful for fast design iteration.

### 8.3 Scalability: LeNet Benchmark

In our final quantitative study, we measure how well Cider scales to larger accelerator designs. We run the LeNet benchmark from our TVM-to-Calyx compiler case study (Section 7.2), which is 3,598 lines of high-level Calyx across 17 components. Table 2 summarizes the execution time for this benchmark under the three tools: Cider is 8.3× faster than Icarus Verilog, while Verilator outperforms Cider by 3.4×. Verilator’s up-front compilation pays off, but Cider’s advantage over Icarus Verilog suggests that high-level control information still significantly reduces the total work required.

### Table 2: LeNet execution time and slowdown w.r.t. Verilator.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Compilation (sec.)</th>
<th>Simulation (min.)</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Icarus</td>
<td>0.3</td>
<td>215.04 ± 1.89</td>
<td>27.9×</td>
</tr>
<tr>
<td>Cider</td>
<td></td>
<td>26.05 ± 0.06</td>
<td>3.4×</td>
</tr>
<tr>
<td>Verilator</td>
<td>16.5</td>
<td>7.71 ± 0.008</td>
<td>–</td>
</tr>
</tbody>
</table>

Icarus Verilog is an interpreter, like Cider, and therefore starts up quickly but simulates large designs less efficiently. Verilator translates Verilog to C++ and relies on a standard C++ compiler, so it incurs a long startup time but simulates more efficiently. In simulation only, Verilator outperforms Cider by 2.9× while Cider outperforms Icarus Verilog by 4.2×. For the largest benchmark, NTT-64, Verilator is 1.4× faster than Cider and Cider is 59× faster than Icarus Verilog.

Figure 6a compares Cider’s execution time when running high-level Calyx programs, as emitted by frontends, against the same programs when compiled to fully structural (RTL-like) Calyx code. Cider can exploit control information in high-level programs that is lost in the equivalent lowered code, which forces Cider to simulate the entire design on every cycle. High-level simulation is 20× faster on average than structural simulation. For our benchmark with the largest control program, NTT-64, high-level simulation is 124× faster.

### Figure 6: Performance study for Cider. Simulation times are normalized to Cider simulation on high-level Calyx programs.

8.1 Benefits of High-Level Control

Figure 6a compares Cider’s execution time when running high-level Calyx programs, as emitted by frontends, against the same programs when compiled to fully structural (RTL-like) Calyx code. Cider can exploit control information in high-level programs that is lost in the equivalent lowered code, which forces Cider to simulate the entire design on every cycle. High-level simulation is 20× faster on average than structural simulation. For our benchmark with the largest control program, NTT-64, high-level simulation is 124× faster.

### 8.2 Comparison to Verilog Simulation

Figure 6b shows the simulation time only (not counting startup time) for Verilator and Icarus Verilog normalized against Cider.
9 RELATED WORK
The mainstream tools for debugging hardware designs are register-transfer level (RTL) simulation [41, 44, 47] and waveform visualization [9]. As Section 2.1 describes, RTL simulation produces detailed, cycle-by-cycle traces for every signal in a design. This granularity makes it difficult to reconstruct the logic of a high-level accelerator description, which is Cider’s focus. Cider’s primary difference is in the way it exploits coarse-grained temporal information that can illuminate when and where a bug arises.

Some work accelerates simulation using field-programmable gate arrays (FPGAs) by incorporating special debugging logic such as scan chains [39] or using vendor-specific tools to monitor FPGA execution [21, 45]. Recent work [4, 37] addresses the latency of FPGA compilation using a JIT-like method where designs start in software simulation and then migrate individual modules to faster FPGA emulation. While FPGA-accelerated debugging is fast, it still works on RTL descriptions and does not map to high-level language semantics.

Recently, Ma et al. [29] taxonomized RTL bugs found in open-source Verilog code and proposed instrumentation techniques to help detect these patterns. These bugs and tools reflect mistakes in manual RTL design, whereas this paper focuses on supporting high-level languages that generate hardware. Ma et al.’s tools focus on detecting particular bug patterns rather than on general-purpose interactive exploration.

Cuttlesim [34] implements software-like debugging for Köïka [8], a Bluespec-like HDL, by generating human-readable C code and relying on an off-the-shelf software debugger. It lets programmers observe the application of Köïka’s rules, which are sub-cycle units of hardware logic. Cider, in contrast, focuses on stepping through Calyx’s control operators, which are much higher level and can encapsulate many clock cycles. Whereas Köïka and Cuttlesim are suited to manually designing arbitrary hardware such as processors, Calyx and Cider focus on debugging computational accelerators that are automatically generated from high-level languages.

HGDB [48] uses RTL simulation to provide source-level debugging for RTL generators such as Chisel [5]. Like Cuttlesim, HGDB focuses on low-level RTL design, not on high-level languages with software-like control flow.

Some tools for high-level synthesis (HLS) C-to-RTL compilers share Cider’s goal of source-level debugging. Proposals include generating special cycle-level models as part of HLS compilation [12, 30], instrumenting the generated RTL to output position information [10, 16, 27], or pushing source position information through the compiler stack [18]. Such tools are tied to specific HLS input languages; they do not provide a flexible framework for debugging arbitrary input languages. Because they rely on imperative input languages, they are poorly suited to more unconventional domain-specific frontends like Calyx’s systolic array generator.

10 CONCLUSION
A new generation of higher-level programming models should inspire modern tooling that raises the level of abstraction for development, debugging, and deployment of hardware accelerators. Traditional HDL tools are not up to the challenge. The CIRCT project [35] for incubating an ecosystem of interoperable hardware generation IRs has recently included a Calyx MLIR dialect as one of the steps in their compilation flow from high-level machine learning programs to hardware. This both broadens the ecosystem that can take advantage of Cider and suggests a wide audience for similar tools. Cider shows how tools can exploit the computational structure that undergirds accelerators to organize time and information and thus build better mechanisms for understanding their execution.

ACKNOWLEDGMENTS
We thank Andrii Iermolaiev, YoungSeok Na, and Alma Thaler for their contributions to the implementation of Cider’s interpreter. Alexa VanHattum, Drew Zagieboylo, Owolabi Legunsen, and Ryan Doenges generously provided feedback on early drafts of this paper. We also thank the anonymous reviewers and our shepherd for their feedback.

This work was supported in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA. It was also supported by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA). Support also included the Google Research Scholar Program and NSF awards #1845952, #1723715, and #2124045.
To install locally and run the scripts, follow the instructions in the README.md file at the root of the repository.

A.4 Installation
For non-containerized installation, follow the instructions in the README referenced above. Or, use the Docker image linked there.

A.5 Evaluation and expected results
This artifact seeks to reproduce the benchmark results discussed in our performance evaluation as well as the debugging process in Section 3. These are:
- Benchmark Data and Graph Generation: Generate the graphs found in the paper using pre-supplied data:
  - Core benchmark graphs (Figure 6)
  - LeNet comparison (Table 2)
- Benchmark Correctness for each Simulator
- Data Collection
  - Collect timing data for the full suite
  - Generate new graphs and tables from the collected data
- Optional: Interactive Debugging with Cider.
  - Debug the sample program with Cider (Section 3)

A.6 Methodology
Submission, reviewing, and badging methodology.

REFERENCES
[14] Ross Daly, Lenny Truong, and Pat Hanrahan. 2018. Inventing and Linking Generators from Multiple Hardware Languages using CoreIR. In Second Workshop on Open-Source EDA Technology (WISET).

Mike Turpin. 2003. The Dangers of Living with an X (bugs hidden in your Verilog). In Boston Synopsys Users Group Meeting (14).


Received 2022-07-07; accepted 2022-09-22