A Case for In-situ Hardware Development

Alborz Jelvani
Rutgers University
USA

Richard P. Martin
Rutgers University
USA

Santosh Nagarakatte
Rutgers University
USA

ABSTRACT

Existing hardware design tools have failed to create an environment to support rapid hardware development cycles. Hardware simulators attempt to close this gap by allowing familiar software debugging techniques to be used for simulation, yet debugging for in-hardware deployment still relies on primitive tools such as logic analyzers. We propose the integration of familiar software-based debugging conveniences for in-hardware development. The evaluation of a preliminary prototype is also presented.

1 INTRODUCTION

The differences between hardware and software development are stark. Software development has evolved into a diverse ecosystem, housing various instruments to enable incremental development through a rapid edit-compile-debug loop. Hardware development, by contrast, involves at least *three* separate phases, each lacking diverse tools: simulation, FPGA validation, and final tape-out.

Simulation is often a key step for pinpointing behavioral bugs in a design, and while correct simulation of a design is necessary, it is not sufficient. Not only is simulating large designs infeasible via software, but for many applications the final realization is an FPGA, as product requirements and sales volume may not justify an ASIC. Unfortunately, the length of the edit-compile-debug loop on FPGA's is much longer than software development. Hardware synthesis can take on the order of hours rather than seconds typical for software compilation. Additionally, in-situ debugging for FPGA's relies on primitive tools such as log-based tracing and logic analyzers. In effect, the edit-compile-debug loop cannot support rapid incremental development, which hinges on the ability to rapidly iterate over two steps: (1) making an observation and (2) implementing a change.

Our work proposes bringing rapid development cycles to the FPGA workflow through quicker edit-compile-debug cycles. To enable this, in-situ debugging is necessary, as opposed to using simulation. Key features an in-situ debugger should provide include the ability to pause and restart execution, examine and modify state at the source code level, and observe the effects of incremental changes with low latency.

2 EXISTING DEBUGGING APPROACHES

A common approach for in-situ development is trace-based debugging via tools such as Intel's Signal Tap and Xilinx's ILA IP blocks. While these tools allow a developer to view the state of hardware, and even replay state transitions later in software, the developer cannot *interact* with the design logic during a live debugging session, and any incremental design changes are accompanied with a high latency compilation. Recent works [3, 6] have explored trace-based debugging techniques that provide debugging of communication protocols and finite state machines, however, these tools cater to the *ex post facto* model of incremental development, where effects of computation are observed after a complete execution phase. We advocate for in-situ interactive debugging workflows, similar to the work proposed in [1, 4] - with key differences being support for incremental development and source-level debugging. Without both interactive debugging and incremental development support, inefficient debugging and compilation phases will bottleneck hardware development cycles.

To this end, we propose instrumenting hardware designs with debugging logic. However, unlike traditional approaches that use the instrumented logic only for state visibility, we advocate for instrumenting a debugging engine into the users design that is capable of hardware simulation. Combined with logic to pause the design clock, modify hardware state, and rapidly observe code modifications, support for incremental hardware development can be achieved by shortening the edit-compile-debug cycle. Our approach relies on a synchronous paradigm, but synchronous designs are by far the most common.

3 DEBUGGING ENGINE

Our debugging engine consists of three main components: A design controller, a simulator core, and a command interface. These components facilitate interaction of user with design and enable low-latency compilation during incremental development.

3.1 Design Controller

The design controller is responsible for interacting with debug logic that is automatically instrumented during compile time. To a large extent, this includes logic for halting the design clock, reading/writing to registers, and setting breakpoints or watchpoints. To halt the design clock, a compiler pass replaces each hardware module's clock input with a *virtual clock* that is controlled by the design controller and interacted with through the command interface. Additionally, a compiler pass wires registers of interest in each module to a multiplexer in the design controller, controlled through the command interface. Breakpoints and watchpoints are implementable through latches that halt the design clock on programmable triggers.

3.2 Simulator Core

To lower compile latency in the edit-compile-debug cycle, the debugging engine should be equipped with a simulator core capable of simulating hardware. This is useful during incremental development, as the debugging engine can support a *behavioral recompilation* - when design modification effects are propagated to existing

LATTE '24, April 27-28, 2024, San Diego, California, USA

hardware logic via co-simulation. This means the effects of an incremental change can be observed almost instantaneously, as it involves only a software recompilation. Behavioral recompilation can operate at the module granularity, which would offload HDL modules to the simulator post-modification. To decrease module fragmentation, static analysis can be used to identify fine-grained areas of code that require behavioral recompilation. An example of this mechanism is demonstrated in the following Verilog snippet adopted from a real bug found on GitHub in an open-source hardware library implementing IEEE 754 floating-point units [2].

```
1  always @(posedge clk) begin
2      z[22:0] <= z_m[22:0];
3      z[30:23] <= z_e[7:0] + 127;
4      z[31] <= z_s;
5      if ($signed(z_e) == -126 && z_m[23] == 0) begin
6        z[30:23] <= 0;
7      end
8  +   if ($signed(z_e) == -126 && z_m[23:0] == 24'h0) begin
9  +     z[31] <= 1'b0; // set sign bit to 0
10 +   end
11     //if overflow occurs, return inf
12     if ($signed(z_e) > 127) begin
13       z[22:0] <= 0;
14       z[30:23] <= 255;
15       z[31] <= z_s;
16     end
17 end
```

This Verilog snippet is responsible for handling the addition of floating-point values. The bug results in the incorrect handling of addition for canceling terms, and is caused by the absence of the highlighted code. As defined in the floating-point standard, $-a + a = +0$. Without the highlighted code, the sign bit of the floating-point value remains 1, which denotes a negative value even if the mantissa bits are all 0. This causes $-a + a$ to evaluate to -0.

An incremental change to a design such as this floating-point adder can be reflected inside hardware instantaneously, as the design controller has scaffolding in place to access the register `z[31]`. The added clause on line 8 can run in the simulator core *alongside* the faulty HDL module as follows:

(1) A hardware watchpoint is set for the clause on line 8.
(2) Upon a trigger, the virtual clock is paused, current register values are read, and any logic inside the clause is simulated for 1 cycle.
(3) Simulation side-effects (`z[31] <= 1'b0`) are propagated to corresponding hardware registers and the virtual clock is resumed.

This effectively allows instantaneous modification of hardware at a fine-grained granularity rather than through course-grained fragmentation at module granularity, thus increasing usage of useful hardware resources. An overview of this mechanism is presented in Figure 1.

3.3 Command Interface

The command interface allows interaction and communication with the debugging infrastructure, and consists of two components: a client and a host.

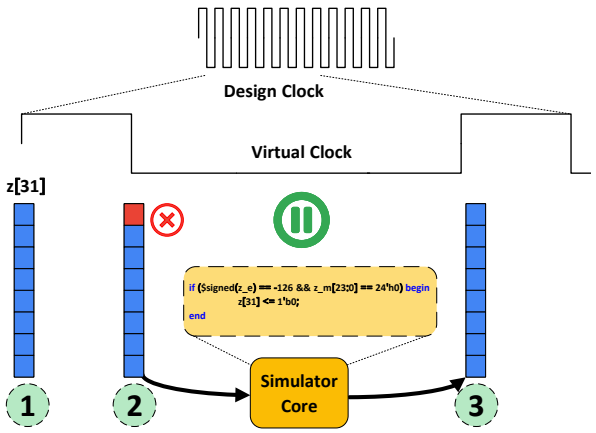

Figure 1: Simulator core operations on a faulty floating-point adder. (1) Hardware logic begins execution on the z register. (2) A hardware watchpoint pauses the virtual clock after a clause trigger; the z register is read into simulation. (3) Simulation executes; the new value of z is written to hardware.

Client side: This is the traditional debugging shell software developers are familiar with, such as that of GDB. During a debugging session, the value of individual design registers can be observed and modified. Breakpoints and watchpoints can also be inserted for various signals. Unlike software debuggers, which abstract event processing through the program counter, the interface for a hardware debugger requires multiple abstraction proxies, such as one that processes events through clock edges and another through source-level hotspots - areas of code contributing to state changes. These abstraction proxies are useful as hardware behavior can inherently be viewed as signals that evolve over time, and also as control of data flow: FSMs, pipelines, and I/O. Implementing abstraction proxies is enabled through the design controller's ability to access registers and control the design clock. For instance, control flow stepping for an FSM can operate by setting a watchpoint on the state signal of the FSM. This allows the design controller to step the virtual clock until the watchpoint on the state signal is triggered.

Host side: The host for the command interface consists of communication logic used in the debugging engine as well as a command execution unit. The host should implement a concise but rich *command set* that is usable by the client interface for implementing debugging abstractions. This command set should also support the handling of subprograms - specifications of hardware modifications that require behavioural recompilation.

4 EVALUATION

Our prototype, hddb (hardware development debugger), is an open-source[1] hardware debugger that includes a compiler for injecting a

[1] https://github.com/Jelvani/hddb

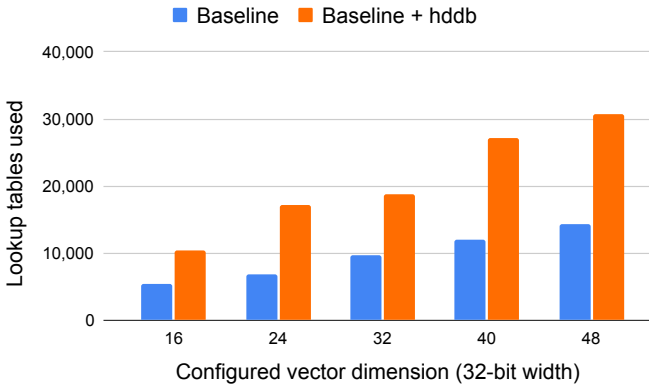

Figure 2: hddb resource overheads for a parameterized vector processing unit core as reported by Yosys version 0.33.

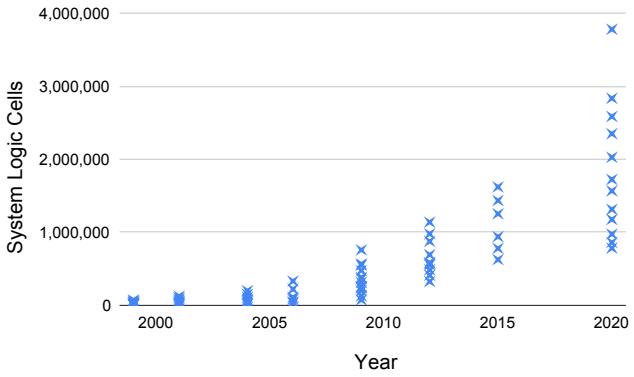

Figure 3: Xilinx Virtex family FPGA logic cell capacities since 1999.

hardware debug engine into designs and a client debugging interface for interacting with the debugging engine. Currently, hddb supports stepping of the design clock and reading/writing to hardware registers; in the future hddb will implement additional mechanisms presented in Section 3.

The hddb compiler instruments hardware designs developed in Migen (a Python based HDL) with debugging logic that communicate with the hddb debugging interface running on the host machine. hddb has currently been tested on the open-source OrangeCrab development board, which is based on the Lattice ECP5 FPGA. The hddb workflow begins with a user design written in the Migen HDL. The design is then passed through the hddb compiler which instruments the design with its debugging engine and outputs a new instrumented source file along with a hardware symbol table used by the client debugging interface. This process is analogous to compiling a program with debug symbol support through a compiler flag. Communication between the host machine and the OrangeCrab board is administered over USB. As the OrangeCrab does not support a native USB interface (the USB pins directly connect to the FPGA), an open-source USB core is instrumented into the users design by hddb. Both the host and client side of the command interface are connected to a Wishbone bus. The client debugging interface on the host machine uses a USB compatible Wishbone bridge utility to communicate with the hddb debugging engine via memory mapped registers accessible over the Wishbone bus. The hddb client running on the host machine provides a text-based user interface for stepping the design clock and accessing hardware registers in the user design.

4.1 Debugging Engine Overheads

Figure 2 presents the resource overheads hddb introduces on a configurable dimension vector processing unit (VPU) core developed in Migen and synthesized by Yosys. The VPU core supports basic operations between two vectors such as element-wise arithmetic and the vector dot-product. hddb currently instruments the entire design; each register in the VPU core is visible and modifiable at runtime through the debugging interface. As expected, there is a considerable resource overhead for instrumenting all registers of the VPU. However, this resource overhead scales linearly in relation to the baseline designs resource usage as the VPU's dimension scales. Additionally, for some designs, partial debugging coverage may be sufficient and resource overheads for debugging can be lowered.

Nonetheless, the overheads of in-circuit debugging can be justified as FPGA logic capacities have been scaling exponentially. Figure 3 presents the logic cell counts of various Xilinx FPGA models of the past two decades (compiled from archived product data sheets). It is common for FPGA logic cell counts to now exceed one million, which can justify the overheads of instrumenting debugging logic into hardware designs.

5 RELATED WORKS

A closely related idea is presented in Cascade [5], a just-in-time compiler for Verilog. Cascade transparently offloads simulation onto hardware, effectively hiding the long compilation latencies that plague hardware design. Their design also allows using non-synthesizable Verilog primitives, such as print statements to be realized in hardware, which is useful for debugging. A crucial difference between Cascade and our proposal is that the former improves the hardware design experience by hiding lengthy compile times, while we propose an improvement to *hardware design visibility*, which is achieved through a debugging framework that can also hide lengthy compile times to support rapid development in the edit-compile-debug cycle.

REFERENCES

[1] Sameh Attia and Vaughn Betz. 2020. StateMover: Combining simulation and hardware execution for efficient FPGA debugging. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 175–185. https://doi.org/10.1145/3373087.3375307

[2] Jonathan P Dawson. 2012. *synthesizable ieee 754 floating point library in verilog*. https://github.com/dawsonjon/fpu

[3] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Haoyang Zhang, Andrew Quinn, and Baris Kasikci. 2022. Debugging in the brave new world of reconfigurable hardware. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 946–962. https://doi.org/10.1145/3503222.3507701

[4] Marco Antonio Merlini, Isamu Poy, and Paul Chow. 2021. Interactive Debugging at IP Block Interfaces in FPGAs. In *The 2021 ACM/SIGDA International Symposium*

on Field-Programmable Gate Arrays. 138–144. https://doi.org/10.1145/3431920.3439305

[5] Eric Schkufza, Michael Wei, and Christopher J Rossbach. 2019. Just-in-time compilation for Verilog: A new technique for improving the FPGA programming experience. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 271–286. https://doi.org/10.1145/3297858.3304010

[6] Gefei Zuo, Jiacheng Ma, Andrew Quinn, and Baris Kasikci. 2023. Vidi: Record Replay for Reconfigurable Hardware. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 806–820. https://doi.org/10.1145/3582016.3582040