Invited: Open-Source EDA Tools and IP, A View from the Trenches

Elad Alon, Krste Asanović, Jonathan Bachrach, Borivoje Nikolić
{elad,krste,jrb,bora}@berkeley.edu
Electrical Engineering and Computer Sciences Department, University of California, Berkeley

ABSTRACT

We describe our experience developing and promoting a set of open-source tools and IP over the last 9 years, including the Chisel hardware construction language, the Rocket Chip SoC generator, and the BAG analog layout generator.

KEYWORDS

open-source CAD, open-source hardware, Chisel, RISC-V

ACM Reference Format:
Elad Alon, Krste Asanović, Jonathan Bachrach, Borivoje Nikolić. 2019. Invited: Open-Source EDA Tools and IP, A View from the Trenches. In *Proceedings of ACM Design Automation Conference (DAC '19).* ACM, New York, NY, USA, 3 pages. https://doi.org/https://doi.org/10.1145/3316781.332348110.1145/1122445.1122456

1 INTRODUCTION

The Berkeley EECS Department has a long history in open-source EDA tools, dating back to the original SPICE distribution in 1972. In this paper, we provide a retrospective and update on a recent set of interconnected projects at Berkeley that have contributed to the recent surge of interest in open-source hardware.

Our original goal in creating the open RISC-V instruction set architecture (ISA), the Chisel hardware construction language, the Rocket Chip generator, and the Berkeley Analog Generator (BAG) systems was to form the infrastructure for a series of research SoC tapeouts. The tools were primarily developed for our own use, but over time as they matured they have been distributed and adopted in a wider community. Each of these contributions is taking a different path to external adoption as we describe below.

2 RISC-V ISA

Modern Systems-on-Chip (SoCs) are heavily dependent on general-purpose or specialized software running on a collection of general-purpose or specialized processing cores. Building a simple processor pipeline is relatively easy, to the point that this forms a common project in many undergraduate courses. However, building and maintaining the software required to run on that processor is considerably more time consuming.

In many earlier processor research projects, we had used variants of existing proprietary ISAs to allow use of existing software stacks. However, as soon as we modified the ISA, we were faced with rebuilding the software stack while still having to deal with the peculiarities of the proprietary ISA. We also could not open-source our designs due to IP issues. In 2010, as part of a new research project focused on application-specific accelerators, we began the design of RISC-V as a simple but efficient ISA that was easy-to-extend with custom acceleration extensions. We also wanted a processor base architecture that was free of IP entanglements, so we could share our designs.

The primary drawback of building a clean-slate ISA was that we had to port all the software. Although we initially thought the ISA design would only take one summer, the ISA evolved over the next four years informed by a string of tapeouts and ports of the compiler and operating system. We made the ISA specification available as a techreport, and made the gcc compiler and Linux ports available from our website, as well as an open-source RTL core written in Chisel and wrapped inside the Rocket Chip SoC generator.

Despite little effort on our part to publicize what we were doing, we soon had external users trying out the ISA and sending comments, and so we decided to make a concerted effort to engage with industry in the summer of 2014 at the Hot Chips conference. We were very surprised at the level of interest in our core and ISA, and so we followed up with an open RISC-V workshop in January 2015. The workshop sold out with over 40 companies attending. A common request from industry attendees was that the ISA needed a home outside of the University to help assure it would be long-lived and stable, so we incorporated the non-profit RISC-V Foundation as a long-term home for the RISC-V specification.

Since that time, RISC-V has become a global phenomonen, with over 140 companies and organizations joining the Foundation, including many of the major semiconductor companies. Nvidia is using home-grown RISC-V cores on all their future GPUs, while Western Digital has announced that all their future products would be based on RISC-V. Many other major companies have RISC-V projects underway around the world. There are at least half a dozen commercial RISC-V core providers and many open-source cores.

Compared to earlier open architecture projects, RISC-V focused on the ISA specification rather than on a particular incarnation of the ISA. One attribute of successful open-source projects is that they provide a complex component that can be developed and maintained by the community and reused and adapted for many different projects. By focusing on the ISA specification, we enable the most complex component in today's SoCs—the software—to be reused across many different projects using many different core designs. An open specification also makes possible open-source cores, but we feel the software ecosystem building up around the RISC-V standard is really the major achievement of the RISC-V

DAC '19, June, 2019, San Francisco, CA

ACM ISBN 978-1-4503-6725-7/19/06. .
DOI: 10.1145/3316781.3323481

standard. Anyone can build a RISC-V core and take advantage of the software base.

3 CHISEL AND FIRRTL

Chisel was designed to be a hardware construction language, not a high-level synthesis (HLS) language, and the project began in 2010 alongside but separate from the RISC-V ISA project. The goal was to allow hardware designers to retain complete control of the microarchitecture of a unit but with much higher-level programming constructs to allow the microarchitecture to be expressed as a parameterized generator. Another goal was to support a unified environment for both architectural simulation and chip fabrication, as we had been discouraged by the lack of accuracy in traditional architectural simulations, and the need to rebuild a chip design in a hardware language after performing architectural design space exploration.

The first Chisel versions were monolithic combinations of the frontend design language, the intermediate representation and transforms, and the backend output generators. In 2016, we completely rewrote Chisel as Chisel 3, an open compiler that specifies an intermediate representation FIRRTL (for Flexible Intermediate Representation for RTL), and allows users to write their own FIRRTL passes. By layering Chisel in this way, we created an ecosystem allowing more front ends, tools, backends and passes. We have also recently completed a Verilog front-end allowing users to convert Verilog designs into FIRRTL. Now that Chisel 3 was in place, considerable work went into making Chisel handle more real-world designs. Multiple clock domains and resets, and asynchronous reset blocks, were added. Black box support was expanded, permitting better parameterization of external Verilog IP blocks.

We learned the hard way that any design methodology is only as good as the slowest phase. We had countless people say that improving the speed of writing RTL is all well and good, but what about simulation, verification or physical design? Now that we had an open compiler with an intermediate representation, we could produce a large array of invaluable tools and facilities for users. In particular, we used the new infrastructure to produce innovations in simulation, testing, verification, and physical design.

From the beginning, we understood the importance of fast and accurate simulation. In simulation during design debugging, the most important factor is the wall-clock time needed to reach simulation cycle n, including simulator compile and build times. For different n, different simulation strategies are appropriate. Simulation speed and compile time are often in conflict. Having an IR allowed us to create a FIRRTL interpreter, called Treadle, that trades runtime for compile time allowing for nearly interactive simulations for low cycle counts.

The original Chisel had a C++ backend that produced fast two-state simulation, but that was dropped in favor of a Verilog-only output to reduce maintenance effort, and to simplify verification by providing only a single output instance to be verified. We then spent effort to speed up Verilog simulation using the Verilator open-source simulator.

Larger simulation cycle counts often mandate use of FPGAs. Unfortunately, obtaining cycle-accurate system simulations on an FPGA is quite a daunting task. One task that is particularly onerous is creating a cycle-accurate DRAM model. To make this more approachable, we created a token-based simulation framework, MIDAS, and synthesizable RTL DRAM model based on a timing model. The simulation framework along with FIRRTL allows one to transform the base RTL into RTL that passes tokens on external signal lines and into a form compatible with an RTL timing model.

A big goal was addressing the hardware verification challenge. We needed to support cover points, assertions, and print statements in the emitted Verilog code to connect with commercial verification tools. We also needed to keep readable source names in the emitted Verilog to allow us to map bug finds to Chisel source code. The original academic version relied on backend gate synthesis to clean up emitted Verilog, but more recently a circuit-optimization pass was added to Chisel before Verilog emission. This optimization helps with both simulation speed and with verification code coverage.

Recently, we have begun transferring successful ideas from software testing to hardware verification. Our recent work ported a software feedback-guided fuzz testing, called AFL, to hardware, a system we call RFUZZ. It effectively puts the timing critical statistics gathering code into hardware so that the fuzzing can happen at MHz speeds. We do this by transforming the resulting FIRRTL into RTL augmented with statistics-gathering hardware.

One of the remaining tall poles in the design loop tent is physical design. We were finding that we had very little reuse of our physical design scripts from tape out to tape out, across different EDA tools and different nodes at different foundries. We have developed a mechanism for connecting RTL design to physical design by programmatically annotating the FIRRTL code and abstracting the physical design artifacts into a reusable framework.

We had always imagined Chisel supporting a layering of domain-specific languages. In 2017, we created a variant of Chisel for digital signal processing called ChiselDSP. ChiselDSP provides a productive path from math to circuits, starting with a straightforward translation of equations to RTL with floating-point unit, to RTL with fixed-point arithmetic, and ultimately high-performance pipeline RTL implementations. ChiselDSP includes support for complex numbers and interval types.

Over many years of trying to explain what we were doing, we came around to explaining our techniques as “reuse”. People could understand that reuse would save time. By writing highly parametrized generators in a full-powered programming language, users could apply them to a wider range of use cases. We then went on to extend reuse to compiler passes and to the scripting of physical design as well. Finally, by structuring our designs around best software practices, we could create a powerful library of reusable hardware components.

Despite, feeling that we were largely successful in attaining radically better reuse, we did learn that there is a sizable adoption challenge based on hardware designers having less software experience and being uncomfortable with powerful programming languages. Organizing design into hardware generators was sometimes challenging for designers to understand. We taught many boot camps and found that users really needed to start from very basic structural descriptions while seeing equivalent Verilog. After many attempts we ended up writing a progression of examples in order of difficulty as Jupyter notebooks using our fast FIRRTL

simulator. Ultimately, we feel that hardware-focused EEs are going to need to take more software-focused CS courses in order to take advantage of the full power of Chisel and BAG.

In the fall of 2018, we held our first Chisel Community Conference. We had over 100 participants and a packed technical program along with a day of tutorials and deep dives. We're looking to turn this conference into a yearly affair and to hold Chisel conferences in other countries with a large number of Chisel and Rocket Chip generator users. At the conference, Google revealed they had used Chisel for the design of their Edge TPU. Intel Research has been exploring Chisel and contributing to the Chisel repo. There has been considerable interest in China around Chisel, partly driven by the interest in Rocket Chip, and the second Chisel Community Conference will be held in China.

4 THE ROCKET CHIP GENERATOR

The first version of RISC-V, Raven-1, was written in Verilog, but after Chisel became usable in 2011, the RISC-V core developers switched to Chisel, and provided significant feedback and code updates to the Chisel code base.

Over time, the processor code base evolved into a full SoC environment including the processor cores. Developed at Berkeley, the Rocket Chip generator included an in-order processor core "Rocket" (named after the first functioning steam locomotive), a cache-coherent interconnect "TileLink", coherent caches, and I/O devices.

The Rocket Chip generator provides a very powerful set of RISC-V processor building blocks. One of the most impressive uses of Chisel (and Rocket Chip generator) was an out-of-order processor generator called BOOM. In BOOM, out-of-order processors could be created by simply combining functional blocks and sizing the datapath. The entire control path would then be automatically synthesized from this high description. The original BOOM was written in only around 10k lines by reusing the rest of the Rocket Chip generator. This is a striking demonstration of the reuse possible in Chisel and showcases the power of hardware generators written in high-level programming languages.

Rocket Chip generator offers multiple types of customization. Besides changing the generator parameters, it is possible to design a new processor core and drop it into the tile, as was done with BOOM. The standard Rocket custom coprocessor (RoCC) interface provides a way to design decoupled coprocessors integrated in the same tile as the core., with two decoupled interfaces connecting coprocessors to the core and the L1 cache or outer memory system. A custom open-source vector coprocessor, Hwacha, has been a feature of several of our prototypes. Rocket Chip's standard bus interface, TileLink2, offers an opportunity for adding peripherals.

The Rocket Chip framework has been used to complete over 17 RISC-V SoC chip tapeouts at Berkeley, in fabrication technologies ranging from IBM 45nm SOI to TSMC 16nm FinFET. Each chip design had objectives of both demonstrating a new functionality, and developing new tools.

Rocket Chip is now widely used outside of Berkeley, in both academia and industry. The Rocket Chip repository is now maintained by SiFive, who contributed a substantial rewrite of TileLink and a new parameter-negotiation framework, Diplomacy. The non-profit lowRISC project is also using Rocket Chip to build a completely open-source SoC. Several Chinese startups are now selling prototype silicon based on Rocket Chip cores. The BOOM core was used as the basis of the Maxion out-of-order core developed by the Esperanto chip startup.

5 BERKELEY ANALOG GENERATOR

Development of analog and mixed-signal (AMS) modules is another time-consuming part of chip development, which consists of iterative refinement of design specifications through layout iterations and post-layout simulations. Changes in the target process technology, which can be as small as a different variant of metal stackup, often trigger a complete redesign. Building AMS generators, rather than instances of AMS blocks is our method for fostering reuse across different chips and technologies. The Berkeley Analog Generator (BAG2) is a Python-based framework for process-portable development of analog generators. The specification-to-verification framework encapsulates a schematic-generation application-programing interface (API), a sizing routine (executed through a design script) and two layout generation engines, Laygo and XBase. The BAG2 framework has been used to generate a wide range of interface blocks, including multi-Gb/s analog-to-digital and digital-to-analog converters as well as high-speed serial interfaces.

6 CONCLUDING THOUGHTS

Reflecting on the success of this set of interconnected open-source projects, we believe that providing solid but malleable components in a reusable form is the key to adoption. Co-developing a comprehensive set of directly usable artifacts (RISC-V, Rocket Chip, SerDes) along with the development tools (Chisel, FIRRTL, BAG) as open-source has led to greater acceptance of the whole suite. For example, it is unlikely the Chisel language would have seen this level of interest without the Rocket Chip generator, which provides a large amount of quality IP in Chisel form. Conversely, the capabilities of the Rocket Chip generator derive from the use of Chisel. Interest in RISC-V was initially driven by the investment we had made in porting and upstreaming key open-source software to RISC-V, and in providing open-source implementations of silicon-proven Unix-capable RISC-V multicores in Rocket Chip. We also have benefited from continual feedback and interaction with the many industrial sponsors of this work over the years.

We continue to build on these frameworks and look forward to many years of future development in collaboration with a growing open-source hardware community.

ACKNOWLEDGMENTS

This research was funded in part by DoE Award DE-SC0003624, and by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), also by DARPA PERFECT Award HR0011-12-2-0016, ASPIRE Affiliates, ADEPT affiliates, BWRC members, TSMC, and ST Microelectronics.