Everyone's a Critic: A Tool for Exploring RISC-V Projects

Dustin Richmond, Michael Barrow, and Ryan Kastner
Department of Computer Science and Engineering
University of California San Diego
La Jolla, California 92195
drichmond, mbarrow, kastner at eng.ucsd.edu

***Abstract*—The RISC-V specification is a highly flexible specification for low-cost processors. The RISC-V ISA is royalty free, vendor agnostic, easily portable between development environments, and highly flexible to match the demands of an application. These characteristics make RISC-V a natural ISA choice for an FPGA soft processor and this has led to widespread adoption in academia and industry. However, the sheer number of RISC-V projects can be daunting for potential users.**

This paper describes a tool for exploring RISC-V projects. Our tool provides a web-interface for executing C/C++ code, tests, and benchmarks. Our tool is packaged with interactive tutorials for extending, modifying, and reproducing our work.

I. Introduction

The RISC-V specification is a royalty-free Instruction Set Architecture (ISA) standard for low-cost processors [1]. The specification defines 32-bit, 32-bit embedded, 64-bit, and 128-bit base ISAs and optional extensions for compressed, multiplication, atomic, single, double, and quad-precision floating point instructions. This has lead to led to widespread adoption in academia and industry and an overwhelming proliferation of projects. For example, the Rocket-Chip from UC Berkeley implements the canonical RISC-V System-on-Chip for architecture research [2]. The academic start-up VectorBlox has produced Orca to host the MXP vector processor [3]. RoaLogic has developed RV12, a commerical RISC-V IP [4]. Finally, the VexRiscV project implements a high-performance RISC-V processor for FPGAs [5]. This is flexibility is a major benefit for designers, because it allows them to chose extensions to fit their design goals.

However, the flexibility of RISC-V makes evaluating a single project and comparing between projects difficult. Single projects contain many tuning parameters. Individual projects implement different sets of extensions, different versions of the specification, or worse, deviate from the specification entirely. These cross-project differences can cause code that runs on one processor to hang on another.

We need a tool for studying RISC-V soft-processors so that potential users can evaluate the intra- and inter-project tradeoffs. Such a tool must faciliate experiments by providing an interface to run code, tests, and benchmarks. This tool must be flexible enough to handle inter-ISA variations for comparisons within the RISC-V family. The impact of such an exploration tool would be widespread: Prospective users can make informed choices about RISC-V projects; Researchers can use collections of processor designs for comparative research, and educators can use this tool to build RISC-V curricula [6]. As a result, the RISC-V community can grow using a flexible testing and development tool that encourages competition.

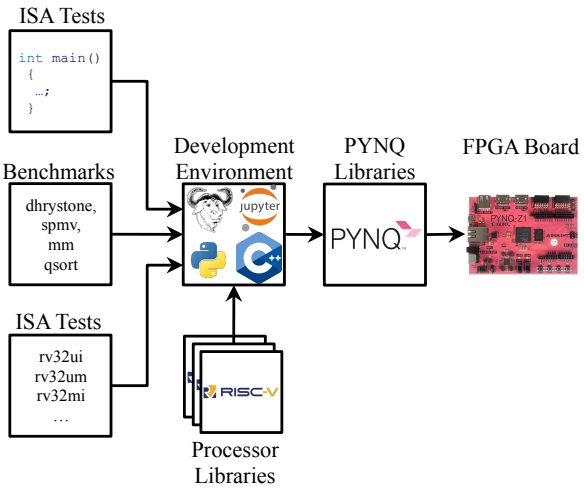

Fig. 1. Our work presented in this paper. Our tool provides a browser-based interface for compiling C/C++ code, benchmarks, and ISA tests targeting a library of RISC-V soft processors. This project is built on top of other widely used software tools and is easily extensible to other RISC-V projects using our provided tutorial.

In this paper we demonstrate our tool for evaluating FPGA-Synthesizable RISC-V processors. A high-level overview of our tool is shown in Figure 1. Our tool provides users with an interface for writing and compiling user C/C++ code, standard ISA tests, and benchmarks. We also provide a tutorial for creating new RISC-V bitstreams. Together these parts provide a flexible environment for evaluating RISC-V designs.

The main contributions of our work are:

- A environment for soft-processor exploration
- A tutorial for adding RISC-V processors to our environment

All the work in this paper can be found in our repositories, under a BSD-3 license.

This paper is organized as follows: We describe our environment in Section II. Section III describes related work in

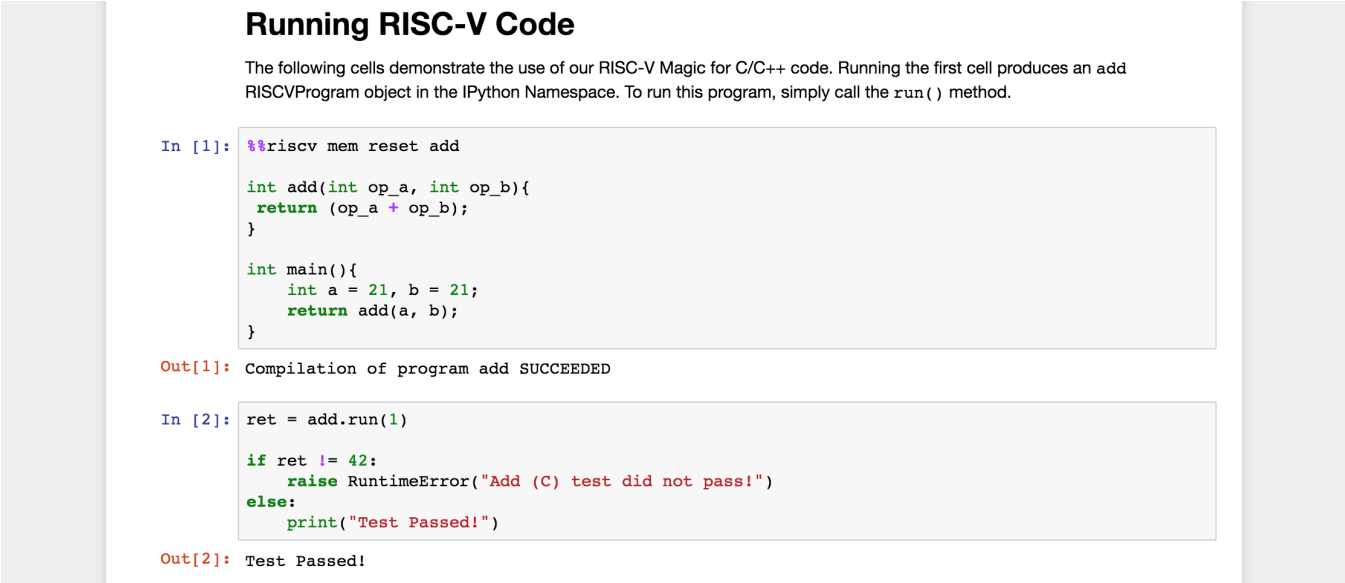

Fig. 2. Our browser-based development environment for compiling C/C++ code to RISC-V processors. The first Jupyter Notebook cell demonstrates compilation, and the second cell demonstrates code execution.

browser-based IDEs and soft-processor research. We conclude in Section IV.

II. A RISC-V Soft-Processor Exploration Tool

Choosing a RISC-V core from the growing list of projects in daunting. The RISC-V specification is highly flexible and lends itself to highly-parameterized projects. Projects implement different versions of the specification, different sets of ISA extensions, or deviate from the base specification. RISC-V projects are also written in many hardware development languages. What users need is a tool for evaluating RISC-V projects that allows them to explore the available RISC-V projects with minimal committment.

An ideal tool for RISC-V exploration has several components: It should provide an interface for writing, compiling, and uploading code to a processor much like Arduino or mbed [7] environments. This tool should include standard benchmarks and ISA tests. This tool should be flexible to handle inter-project variations. It should be able to handle a variety of hardware interface standards (AXI, Avalon, BRAM, and Wishbone). Such a tool would help users navigate the overwhelming buffet of RISC-V projects available.

Figure 1 shows the tool we have developed for RISC-V exploration that meets many of these requierments. All of our work is built on top of PYNQ. PYNQ is a collection of Python libraries for ZYNQ All-Programmable Systems on Chip (APSoCs). We use PYNQ libraries to upload RISC-V bitstreams into the PL, load and compiled binaries into the RISC-V memory, and distribute our tutorial describing how to extend our work with further projects. PYNQ boards come pre-installed with a Jupyter Notebooks server and we use this to provide a broswer-based interface for development.

The components of our work are descibed in the next subsections. Section II-A describes our our tool for writing, compiling, and running user code, tests, and benchmarks from a web browser. Section II-C describes our interactive tutorial for adding RISC-V projects to our library.

A. Development Environment

Modern embedded IDEs like Arduino and mbed [7] provide the ability to write, upload, and run code from a single interface. This provides a unified interface for compiling and running user code. However, these IDEs target well-defined ISAs, and the RISC-V ISA is highly flexible with many extensions and projects that implement variations.

Our tool allows users to build and run C/C++ code on RISC-V processors, and extends the abilities of the IDEs above by providing benchmarks and ISA tests. These features are described below.

1) Broswer-Based IDE: Our browser-based Integrated Development Environment (IDE) is built on top of Jupyter Notebooks as shown in Figure 2. Our work is based on the work in [8]. Jupyter Notebooks are interactive webpages that provide a text-formatting and coding environments in the form of cells. In PYNQ, Jupyter Notebooks are used to deliver documented examples demonstrating PYNQ APIs and peripherals.

Jupyter Notebooks are built on top of the IPython Interpreter. IPython runs Python code from notebook cells. IPython also provides *Magics*, which allow users to run non-Python commands, such as Bash Shell commands, or meta-commands to modify the behavior of an IPython environment. We use IPython Magics to provide a browser-based interface for developing and running C/C++ code complete with syntax highlighting.

An example of our IDE is shown in Figure 2. In this example a simple RISC-V program is written that returns the

sum of two numbers `a` and `b`. The `%%riscvc` command tells Jupyter to interpret this cell as an IPython Magic. The `%%riscv` command three arguments: The PYNQ MMIO obect representing the processor memory, the PYNQ GPIO object representing the reset pin, and the name variable to be created in the IPython namespace. If compilation succeeds, the program can be run in subsequent cells.

The `%%riscvc` command is implemented in two Python classes: `RISCVMagics` and `RISCVProgram`. The `RISCVMagics` class implements the IPython Magic command `%%riscvc`. When the command is typed, this class performs syntax highlighting on C/C++ code in the notebook cell. When the cell is executed this class reads the code and compiles it into a RISC-V binary. The class will print SUCCEEDED, or FAILED (and a compiler error message) depending on the compilation outcome. We also provide a `%%riscvasm` command for RISC-V assembly.

If the compile succeeds, the `RISCVMagic` class will insert a `RISCVProgram` object representing the compiled binary into to the local namespace. This is shown in Cell 2 of Figure 2. The `RISCVProgram` object represents a compiled instance of a `RISCVMagic` cell and can be used to, start, stop and run a compiled cell. When a `RISCVProgram` is run it is uploaded to the memory of the RISC-V processor and the RISC-V processor is released from reset.

B. Tests and Benchmarks

Our IDE also provides an interface for running ISA tests and benchmarks from the standard RISC-V tests repository. These are useful for developers who wish to explore RISC-V projects and better understand the benefits and drawbacks of diverse RISC-V projects.

Our interface is built on top of the browser-based development environment from the previous section. Tests and benchmarks are imported like Python libraries and compiled prior to execution. Like the example in Figure 2, benchmarks and tests can be run from Jupyter Notebook cells. Unlike user program benchmarks and tests rely on standard RISC-V features. For example, benchmarks rely on the presence of cycle and instruction counters, and tests may notify the host using an *environment call* instruction. These features are not standard across projects.

To handle project variations we have modified the RISC-V tests repository to add an interface that allows users to override standard behavior. It is the responsibility of the developer to add their variants so that their processor can be used in our tool.

C. Tutorial

Creating a RISC-V overlay can be a daunting process for a variety of reasons: the inaccessibility of HDLs, interface standards, and complexity of vendor tools. To address this problem we have created a tutorial for adding new RISC-V processors to our environment. Our tutorial is a set of interactive Jupyter Notebooks that teach the reader how to build a PYNQ Overlay with a RISC-V processor for our tool. Each notebook directs the user through a conceptual step, and concludes with notebook cells that verify the notebook has been completed successfully.

By the end of the tutorial readers should understand how a PYNQ Overlay is comprised of Vivado-generated .tcl and .bit files, user-generated Python classes, and packaging scripts. Readers will should also be able to trivially extend this tutorial to other soft-processor projects.

Our 5-notebook tutorial is organized as follows:

1) Configuring the Development Environment: The first notebook of the tutorial teaches the reader how to install dependencies both on a development machine and then on the PYNQ board. The user is directed through installing the Vivado tools, cloning the soft-processor git repository, installing a visual-difftool (such as Meld). On the PYNQ board the user is directed through installing packages from APT, and cloning the necessary RISC-V GNU Toolchain repository.

2) Building a Bitstream: The second notebook of the tutorial teaches a reader how to build an FPGA bitstream for ZYNQ using a RISC-V processor project. It is divided into three parts: *Packaging a RISC-V project as a Vivado IP*, *Creating a Block Diagram*, and *Building a Bitstream*.

The notebook begins with *Packaging a RISC-V project as a Vivado IP* teaching the reader how to package the PicoRV32 RISC-V Processor from [9] as a Vivado IP. We chose this processor because of its ease of use and flexibility. It is highly parameterized, and provides AXI, Wishbone, and BRAM interface standards, and integrates well with Vivado. This section demonstrates how to package two variants of the PicoRV32 processor: a variant with highly-flexible AXI interfaces, and a variant with low-latency BRAM interfaces.

Creating a Block Diagram describes how to build a Block Diagram for PYNQ inside of Vivado IP Integrator. The tutorial provides a "skeleton" .tcl file that opens a Vivado project, creates a bloak diagram, instantiates the correctly-configured ZYNQ Processing System (PS).

Next, the reader selects the PicoRV32 AXI or BRAM variant they want to target and corresponding instructions. The reader adds the processor variant, and all corresponding IP as shown in Figure 3.

- 1 ZYNQ ARM Processing System
- 1 RISC-V Soft-Processor
- 1 Soft-Processor Memory Interconnect
- 1 ARM PS AXI Interconnect
- 1 Soft-Processor Memory
- 2 Reset Controllers (Power-On-Reset, and Warm-Reset)

This section also describes how to connect the processor to the memory system. Our design targets an AXI interface for ease of use with Vivado but we also provide instructions for other memory interface standards.

In *Building a Bitstream*, the reader validates the design and exports the block diagram as a .tcl file, overwriting the skeleton .tcl file that was provided.. The reader is then asked to compile their RISC-V design using the .tcl file they have created.

3) Compiling the RISC-V Toolchain: The third notebook of this tutorial teaches the reader how to compile the RISC-V

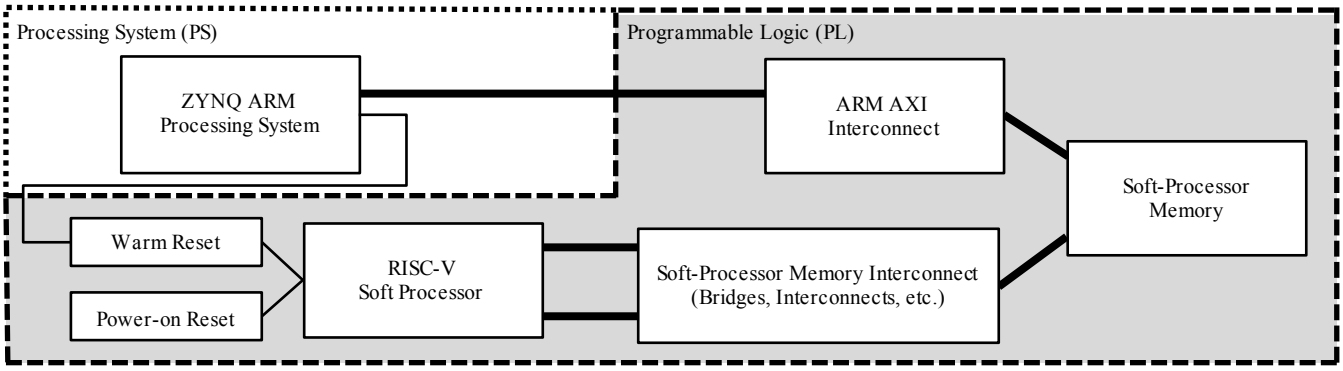

Fig. 3. The high-level system architecture expected by our tool and as described in Section II-A. Processors provide many interface standards and the connection to memory is abstracted by the Soft-Processor Memory Interconnect block.

GNU Toolchain. When the compilation is complete subsequent cells update the PATH environment variable on the reader's PYNQ board.

4) Packaging a Bitstream into an Overlay: The fourth notebook of the tutorial teaches the reader how to package the .tcl file, .bit file, and other newly-generated files into a PYNQ Overlay. The notebook is divided into four parts: *Writing an Overlay Class*, *Creating a Notebooks Directory*, and *Automating Installation / Installing*

Writing an Overlay Class demonstrates how to create a RISC-V overlay class, and declare it as part of a package. This requires creating two files: an __init__.py file that declares the directory as a Python package, and a `riscv` class that inherits from `pynq.Overlay`.

Creating a Notebooks Directory teaches the reader how to build a notebooks directory. The notebooks directory holds the .ipynb metadata files for Jupyter Notebooks, but in our tutorial it also holds the RISC-V compilation script: A Makefile for compilng RISC-V code, a reset.S assembly script for initializing the RISC-V processor at reset, a riscv.ld linker script describing the memory layout for the GNU linker tool.

The final section, *Automating Intallation / Installing*, walks the reader through the step-by-step process of creating a setup.py script that defines their PYNQ Overlay as a Python package that can be installed using Pip. This section starts by verifying the user work from the previous sections: The notebook verifies that the RISC-V compilation files function as expected. Second, the notebook notebook verifies that the Overlay can be imported as a Python package by overriding the Python Package search path. Finally, and finally the notebook runs Pip and verifies that the overlay is installed correctly post-installation.

5) Writing and Compiling RISC-V Code: The fifth and final tutorial of the notebook teaches the reader how to compile RISC-V code, run the resulting programs, and read results from the processor memory.

The notebook starts by demonstrating how to write programs in assembly or C/C++, compile, and then read the results back from memory. This process uses files uploaded manually to the notebook directory and compiled using the makefile from Section II-C4. RISC-V programs are uploaded and run using the PYNQ Overlay class from Section II-C4. Once a program is complete, the ARM processor can read the results from the RISC-V memory space.

Finally, the tutorial introduces the development environment described in Section II-A. Users are directed through installing the IDE package using Pip. These final cells verify that the user has completed the entire tutorial successfully.

III. Related Work

We survey related work in three fields: Broswer-based IDEs, computer architecture education, and soft-processor projects. We demonstrate that there is a gape in these fields that is filled by our work.

A. Broswer-Based IDEs

Browser-based simulation and development environments have seen adoption in recent years. The most familiar example is the mbed platform [7], though academic examples have been created as well [10]. [11] creates a browser-based simulation environment for MIPS. These environments allow users to build, debug, and upload code from their browsers. Browser–based IDEs minimize the installation process and reduce the time to running code.

Our work is derived from PYNQ's IPython-Microblaze feature [8], which allows users to program PYNQ Microblazes using Jupyter Notebooks. This feature allows users to write C/C++ functions and call them directly from Python. Unlike IPython-Microblaze we do not provide Python integration in order to expose the complete RISC-V toolchain.

As a browser-based IDE, our tool has many of the same features. Unlike the examples above, we provide documentation for adding new processors to our tool so that it can grow and expand with the RISC-V community.

B. Computer Architecture Education

The MIPS32 ISA is pervasive in computer architecture curricula. Recently there has been a movement to [12] the RISC-V ISA with the publication of the well-known Patterson and Hennessy book [6] for RISC-V.

With this movement we are presented with opportunities for re-thinking old tools and integrating new tools into the computer architecture curricula. For example, the lowRISC organization supported a Google Summer of Code project to develop a RISC-V simulator [13] to complement the ubiquitious MIPS-32 simulator, SPIM [14].

There is a need for easy-to-use RISC-V environments in education. Our tool would work well in RISC-V curricula. We target a low-cost PYNQ development board and provide extensive documentation for building RISC-V FPGA projects.

C. Soft-Processor Projects

RISC-V is not the first soft processor architecture. MIPS32 is historically the most common soft-processor ISA. OpenRISC [15], [16] has also been proposed as the open hardware replacement for closed ISAs.

RISC-V is growing quickly and many groups have released their RISC-V processors online as open-source repositories. A considerable number of projects are academic: Taiga[17], [18], BOOM [19], and rocket [2]. More targeted research projects have tested security extensions [20], [21], [22] while others have sought smallest implementation size [9], [23], or highest performance [17]. Commercial projects address a particular application, like Pulpino [24] or Orca [3], while other commercial companies, like RoaLogic, provide general-purpose RISC-V IP RV12 [4]. Finally, some projects demonstrate HDL languages like VexRiscV [5] (SpinalHDL) and riscy (Bluespec). This proliferation of open-source RISC-V projects has led to increasinly humorous names, like YARVI (Yet Another RISC-V Implementation).

This proliferation of RISC-V projects is daunting for end users. In this paper we present a framework for RISC-V exploration that can potentially be used on all of these soft processor processors. This framework includes a tutorial that can be adapted for any RISC-V curricula. Our framework can be used to explore the RISC-V ISA across languages, across architectures, and across projects in one familiar environment.

IV. Conclusion

The RISC-V specification is a highly extensible Instruction Set Architecture (ISA) standard for low-cost processors. This extensibility has led to widespread adoption and many open source projects. The number of RISC-V projects can be daunting, however, and a tool is needed to help users explore their options.

In this paper we demonstrated a tool for exploring RISC-V soft-processor projects. Our tool provides a web interface for writing, compiling, and running user code, standard tests and benchmarks. Our tool is also flexible enough to handle inter-project variations and deviations from the specifications.

Acknowledgments

The authors acknowledge the National Science Foundation Graduate Research Fellowship Program, the Powell Foundation, and ARCS foundation for their support.

References

[1] K. Asanovic and D. A. Patterson, "Instruction sets should be free: The case for risc-v," EECS Department, Univ. of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, Aug 2014.

[2] K. Asanovic *et al.*, "The rocket chip generator," EECS Department, Univ. of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016.

[3] VectorBlox, "Orca," github.com/VectorBlox/orca, 2018.

[4] RoaLogic, "Rv12," github.com/RoaLogic/RV12, 2018.

[5] SpinalHDL, "Vexrisc-v: A fpga friendly 32 bit risc-v cpu implementation," github.com/SpinalHDL/VexRiscv, 2018.

[6] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[7] "The arm mbed iot device platform," Jan 2018. [Online]. Available: www.mbed.com/en/

[8] P. Ogden, "Ipython magic for compiling for microblaze," github.com/PeterOgden/ipython_microblaze, 2018.

[9] C. Wolf, "Picorv32 - a size-optimized risc-v cpu," github.com/cliffordwolf/picorv32, 2018.

[10] A. van Deursen *et al.*, "Adinda: a knowledgeable, browser-based ide," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, May 2010, pp. 203–206.

[11] I. Branovic, R. Giorgi, and E. Martinelli, "Webmips: A new web-based mips simulation environment for computer architecture education," in *Proceedings of the 2004 Workshop on Computer Architecture Education*, ser. WCAE '04. New York, NY, USA: ACM, 2004.

[12] D. Kehagias, "A survey of assignments in undergraduate computer architecture courses," *International Journal of Emerging Technologies in Learning (iJET)*, vol. 11, no. 06, pp. 68–72, 2016.

[13] B. Landers, "Risc-v simulator for education," summerofcode.withgoogle.com/archive/2017/projects/6309390843904000/, 2017.

[14] D. Page, *SPIM: A MIPS32 Simulator*. London: Springer London, 2009, pp. 561–628.

[15] J. Tandon, "The openrisc processor: Open hardware and linux," *Linux J.*, vol. 2011, no. 212, Dec. 2011.

[16] OpenRISC, "mor1kx - an openrisc 1000 processor ip core," Jan 2018. [Online]. Available: https://github.com/openrisc/mor1kx

[17] E. Matthews and L. Shannon, "Taiga: A new risc-v soft-processor framework enabling high performance cpu architectural features," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2017, pp. 1–4.

[18] E. Matthews, "Taiga - a 32-bit risc-v processor," gitlab.com/sfu-rcl/Taiga, 2018.

[19] C. Celio *et al.*, "Boom v2: an open-source out-of-order risc-v core," EECS Department, Univ. of California, Berkeley, Tech. Rep. UCB/EECS-2017-157, Sep 2017.

[20] M. Zimmer *et al.*, "Flexpret: A processor platform for mixed-criticality systems," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014, pp. 101–110.

[21] A. Menon *et al.*, "Shakti-t: A risc-v processor with light weight security extensions," in *Proceedings of the Hardware and Architectural Support for Security and Privacy*, ser. HASP '17. New York, NY, USA: ACM, 2017, pp. 2:1–2:8.

[22] J. Choi *et al.*, "Kami: A platform for high-level parametric hardware specification and its modular verification," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, pp. 24:1–24:30, Aug. 2017.

[23] J. Gray, "Grvi phalanx: A massively parallel risc-v fpga accelerator accelerator," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 17–20.

[24] A. Traber *et al.*, "Pulpino: A small single-core risc-v soc," in *2016 RISC-V Workshop*, 2016.