Instruction Extension of a RISC-V Processor Modeled with IP-XACT

Saman Payvar
Computing Sciences
Tampere University
Tampere, Finland
saman.payvar@tuni.fi

Esko Pekkarinen
Computing Sciences
Tampere University
Tampere, Finland
esko.pekkarinen@tuni.fi

Rafael Stahl
Chair of Electronic Design Automation
Technical University of Munich
Munich, Germany
r.stahl@tum.de

Daniel Mueller-Gritschneder
Chair of Electronic Design Automation
Technical University of Munich
Munich, Germany
daniel.mueller@tum.de

Timo D. Hämäläinen
Computing Sciences
Tampere University
Tampere, Finland
timo.hamalainen@tuni.fi

***Abstract*—Short time-to-market and cost consideration of hardware design promotes reuse of ever more complex intellectual property even up to processors. In processor design, the instruction set architecture (ISA) selection is a major design decision driven largely by application requirements. Extendable ISAs enable application-specific adjustments and improved performance at the cost of more complex design. Adding a custom instruction introduces a choice of either utilizing existing hardware or adding new dedicated hardware.**

This work presents an instruction extension flow for a RISC-V processor core modeled in IP-XACT. We demonstrate the workflow by adding three bit manipulation instructions "popcnt", "parity" and "bswap" in the instruction set that executes on an extended processor platform and evaluate their performance in simulation. The simulated instruction count and performance are used to evaluate the benefit of adding dedicated hardware.

The effort analysis of the design flow shows approximately 110 minutes work for adding a new instruction to the RISC-V core. This suggests a straightforward and easy to follow approach that can be extended to other instructions as well. In addition, we propose the workflow to cover adding dedicated hardware in IP-XACT for improved re-usability and design consistency.

***Index Terms*—RISC-V, GCC back end, ISA extension, bit manipulation, IP-XACT**

I. Introduction

The transistor shrinking trend of the CMOS technology introduces higher design complexity of System-on-Chips (SoCs) while competition pressures for short time-to-market at low cost. Modularity addresses the complexity by composing the system of reusable components and their connections. The components are often provided by different vendors which necessitates a standard exchange format to facilitate Intellectual Property (IP) reuse. The most promising is the IEEE 1685 standard (IP-XACT) [1] that is an XML format widely used in the industry for IP description and integration.

Beside increased complexity, transistor size reduction increases the leakage power and thus static power dissipation. Consequently, energy efficiency becomes much more significant. Hardware design energy efficiency is achieved by application specific adjustments which increase the performance and reduce the power consumption. In processor core design, ISA extensions and the hardware design adjustments are the two available options for application specific optimizations.

RISC-V [2] is an open source ISA whose availability, simplicity and adaptability has made it popular in both academia and industry. Studies like [3] motivate further research on the potential of non-standard RISC-V extensions. In this work, we consider adding three bit manipulation instructions to the *PULPino* [4] platform that is an open source implementation of a 32-bit RISC-V single-core microprocessor.

Modeling a processor in IP-XACT facilitates the configurations for different versions such as deciding the standard and non-standard ISA extensions in the hardware design flow. The software compiler must produce compatible binary code for the target hardware, extending the configurability to the compiler, which has lead to *retargetable* compilers [5]. To this end the ISA extensions should be captured in an early phase for consistency in both hardware and software design flows using the IP-XACT model as a single point of entry.

The contributions of this work are:

- Instruction extension design flow of RISC-V ISA modeled with IP-XACT,
- Evaluation of the extensions with RISC-V GCC compilation and simulation, and
- Effort analysis of the proposed design flow

The rest of the paper is organized as follows: Section II introduces the related works. Section III explains the RISC-V non-standard instructions extensions. Section IV depicts the experiments and shows the results. Section V concludes the study and discusses future work.

978-1-7281-2769-9/19/$31.00 ©2019 IEEE

TABLE I
COMPILER EXTENSION STUDIES

Study	Platform	Compiler	Extension
Tagliavini et al. [3]	PULPino	GCC	SmallFloat SIMD
Sen et al. [8]	ARMv8	N.A.	SPARCE
Murray and Franke [9]	ENCORE	GCC	App. Specific
Sedaghati et al. [10]	x86-64	GCC & ICC	StVEC
Proposed	PULPino	GCC	Bit Manipulation

II. RELATED WORK

The RISC-V ISA has a modular structure where the base ISA covers the minimal instructions and the standard instruction extensions add options for extra functionality. At the time of writing this paper, some standard extensions like bit manipulation [6] and vector extensions [7] are still a work-in-progress. Additionally, the ISA allows adding non-standard extensions, and thus custom instructions, and promising studies and proposals have already emerged. In this section, we compare the current ISA extension works summarized in Table I to our experiments. For the comparison we have considered the platform, the compiler, and the implemented extensions.

A. RISC-V ISA Extensions

The small floating point types for RISC-V ISA study [3] presents extensions for 16-bit and 8-bit floating point types. The extensions are implemented for register size of 32-bit for small floating point (FP) data types. The experiments are performed on RISCY core and show 1.64 times performance boost and 30% energy saving for 16-bits and 2.18 times execution time increase and 50% energy reduction for 8-bits.

The sparsity extension work [8] is targeted for deep neural network (DNN) execution on general purpose processors. The sparsity aware general purpose core extensions dynamically track registers with zero value and prevents fetching redundant instructions. The experiments on image recognition DNNs shows up to 31% performance improvements.

Compiler support for automatic instruction set extension study [9] applies the GCC extension in the middle-end for the ENCORE which is a customizable application specific instruction-set processor. They have reported an average performance improvement of 1.26 for experiments on 179 benchmarks.

The vector instruction extension study [10] focuses on addressing mode of vector instructions for stencil computations. The overhead estimations is considered with a hardware implementation and depicted with the optimistic and the pessimistic simulations. The experiments are performed on four x86-64 platforms using GCC and ICC compiler vectorization options. They reported performance improvement between 20% and 2.47x with optimistic instruction emulation and between 7% and 2.26x with pessimistic instruction emulation for both GCC and ICC.

B. IP-XACT Extensions

Inherently IP-XACT is extendable by vendor extension that contain supplementary information. Extendability is mandatory for wide applicability and industrial acceptance. However, extensive use of vendor extensions will result in exactly what the standard was originally intended to avert, vendor-dependence. In the following we consider proposals for vendor extensions that are not only relevant for this work, but also considered broadly applicable.

The IP-XACT extension targeted for smart systems [11] covers power, temperature and reliability concerns. All standard IP-XACT components are identified by a Vendor, Library, Name, Version (VLNV) 4-tuple, so an optional C tag is proposed to distinguish different concerns resulting in VLNVC identifiers. A smart system is modeled as multiple communicating views where each presents an extra-functional (i.e. non-functional) concern. The IP-XACT views are used for SystemC code generation where instances of all the views are instantiated at the top level. The influence of all concerns of a system are now covered when all views are simultaneously simulated.

IP-XACT is originally intended for hardware description but could be used for both hardware and software data inclusion. In [12], a methodology for both hardware and software IP reuse is proposed for IP-XACT based FPGA design flow. This approach facilitates board support package and software file creations, and results in reducing the design time to one third while doubling automation. In [13] IP-XACT extensions are proposed for covering hardware dependent software.

C. RISC-V IP-XACT Model

The PULPino platform SystemVerilog (SV) implementation is considered for creating IP-XACT models from the SV source code repositories in [14]. The modeling is applied by using the Kactus2 [15] tool to automatically import the modules and further refined manually by the designer by e.g. defining the memory maps. The resulting model covers all four RISC-V configuration options of the PULPino platform and provides the basis for configuration automation which simplifies the rest of the design flow for the different processor variants.

The related works are presented in three subsections covering three aspects of this work. The reported results of RISC-V extension studies promote further investigations of our RISC-V bit manipulation instruction extension work. The IP-XACT studies show that it can be successfully applied to many design topic and promote our core expansion study. The IP-XACT model of RISC-V study facilities our RISC-V core extension explorations.

III. RISC-V INSTRUCTION EXTENSION DESIGN FLOW

The design flow utilized in this study for adding the non-standard instructions to the RISC-V ISA is presented in Fig. 1. This flow has eight levels from top down depicting the stages from inferring the bit pattern of instructions until running the compiled binary on a processor implementation on an FPGA.

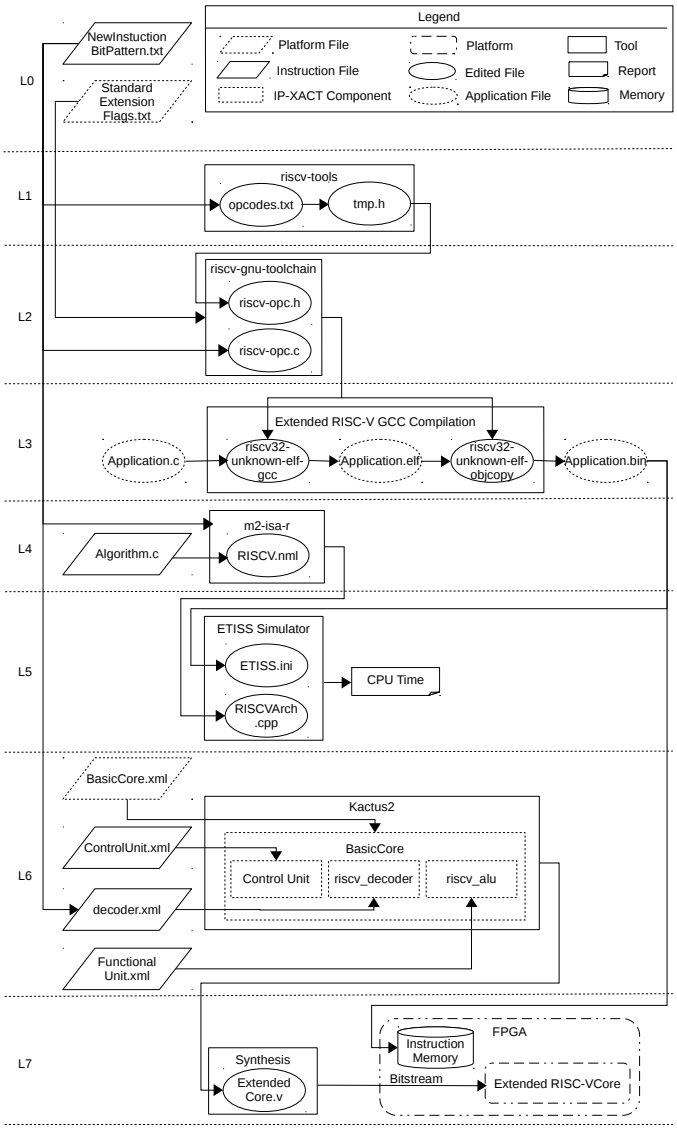

Fig. 1. The proposed design flow of RISC-V instruction extensions targeting simulation and prototyping on FPGA.

A. Standard ISA Selection and Custom Instructions Inclusion

The basic RISC-V core with its standard extensions is determined at the initial stage at level L0 in Fig. 1. The command line flags i.e. *with-arch* and *with-abi* for the compiler generation by *riscv-gnu-toolchain* are stored in a text file. Beside the standard extensions, the bit patterns of the targeted non-standard instructions are stored in another text file using the same format as in the *opcodes.txt* file of *riscv-tools* on L1.

B. RISC-V GCC Compiler Extension

The RISC-V GCC back-end can be edited to cover non-standard RISC-V instructions. Utilizing inline assembly for new instructions requires only minor modification to GCC binutils. Alternatively, the instruction pattern could be added to the automatic code generation in the compiler back-end and the compiler would attempt to utilize it whenever appropriate. However, the latter is vastly more complex requiring in-depth knowledge of the optimizer internals and thus considered outside the scope of this paper.

First, the RISC-V opcodes header file should be edited to include a new instruction's *MATCH* and *MASK* definitions and its *DECLARE_INSN* macro declaration in *riscv-opc.h*. The *MATCH* defines the binary equivalent of an instruction with zeros for the registers while the *MASK* defines a pattern for detecting the instruction. The *MATCH* pattern must be unique for each instruction, so addition of new instruction's *MASK* requires overlap checking which can be performed by tools like *riscv-tools* [16]. This first stage is shown as level L1 where the instruction specification i.e. the binary equivalent of each field of the instruction is considered as input from level L0. The output of level L1 is a temporary header *tmp.h* containing the instructions in compatible format with *riscv-opc.h*.

Second, the instruction format must be added to the *riscv_opcodes* structure in *riscv-opc.c* file. The format includes data like the instruction's name, type and registers which are inferred from the new instruction's bit pattern text file. Also, the temporary header file *tmp.h* should be applied to *riscv-opc.h*. Then, the RISCV GCC compiler is built with appropriate platform specification data i.e. *with-arch* and *with-abi* configurations received from L0. The second stage is demonstrated as level L2 where the new instruction bit pattern and standard extension flags generated in L0 are considered for header and C file modifications and the compiler build.

Third, the Extended RISC-V GCC compiler is used to compile the application C code to RISC-V assembly. The ETISS [17] project provides a *Makefile* template for compiling the C code using the previously built RISC-V GCC compiler which now recognizes the new instruction's inline assembly. Beside the C code, the *Makefile* accesses the necessary libraries for providing the platform i.e. PULPino compatible binary file. The third stage is demonstrated as level L3 where the C code input for the RISCV GCC compiler is presented on the left and the binary output file on the right.

C. ETISS Simulator Extension

As we are exploring non-standard instructions, no hardware in the market supports them out-of-the-box. Consequently, the ETISS simulator is selected for performance analysis of the studied instructions. ETISS is an ISA-independent Instruction Set Simulator (ISS) based on Dynamic Binary Translation (DBT). ETISS can be used to estimate CPU time of the compiled C code. The simulation is used to evaluate the performance of the processor using base instructions against using custom instructions to evaluate whether the performance gain justifies adding dedicated hardware.

ETISS can support different ISAs using plugins. It already provides a plugin for the basic RISC-V ISA. To extend the RISC-V ISA, the new instruction must be supported by the DBT of the ETISS simulator. This is conveniently done using the *m2-isa-r* tool, which is an Eclipse Modeling Framework (EMF) application that can generate plugins for different ISAs using a model-based flow based on nML description. nML

is a modeling language for compact ISA description. *m2-isa-r* reads the nml file *RISCV.nml* and supplies the ETISS plugin file *RiscvArch.cpp*. The plugin file contains an *InstructionDefinition* for each available instruction and defines the binary encoding and the instruction behavior which enables DBT. The nml file *RISCV.nml* is extended according to the *Algorithm.c* and the new instructions bit pattern text files. The RISC-V plugin *RiscvArch.cpp*, which is extended with the new instructions, is generated using the model-based flow resulting in ETISS supporting the new instructions. The fourth stage is demonstrated as level L4.

In a second step, the ETISS simulator is configured for estimating the performance. The ETISS simulator contains a shell script which accesses the initialization configurations file i.e. *ETISS.ini*. Beside the simulator configuration options, the path to the compiled binary is given in this file. ETISS outputs the text prints of the code and reports CPU time, simulation time, CPU Cycles and MIPS estimation. This is shown as level L5 where the ETISS simulator receives the binary and provides the CPU time report. The experiments in Section IV cover levels L0 to L5 of the presented design flow.

D. Hardware Support

Executing the new instructions is possible by either utilizing existing instructions i.e. a function call to equivalent algorithm or by adding dedicated hardware functional unit (FU) for it in the pipeline. The the latter is presented in the workflow as L6 and L7 where the FU of the new instruction is added at the IP-XACT model level.

First, the IP-XACT model of the selected basic core is packaged in the Kactus2 tool. In the case of PULPino, this is already done. Next, the FU of the new instruction is added in the pipeline and the control unit of the core is replaced with a compatible implementation. In PULPino the FU is added into the ALU module *riscv_alu* and wired to the ALU output selection multiplexer. Then, the module *riscv_decoder* is replaced with one recognizing the new instruction bit pattern and driving the control signals for ALU and optionally the rest of the core. After these editions, the basic core covers the extensions.

Second, the RTL code for the extended processor is automatically created using the generators in Kactus2. The RTL is synthesized using the chosen synthesis tool for the target FPGA platform. Then, the FPGA is programmed with the extended core including the FU of the new instruction. Finally, the binary of the compiled software code which includes the new instructions is loaded into the instruction memory for execution. In Fig. 1 the synthesis and FPGA run stage is depicted as L7.

IV. Experiments

We have studied three non-standard RISC-V bit manipulation instructions including *popcnt*, *parity*, and *bswap*, and their impact on the performance.

A. Algorithms of Instructions

We have compared the selected instruction's *LLVM* algorithm implementation [1] to their inline assembly equivalents. Population count algorithm returns the number of one bits in a given value as presented in Algorithm 1. Parity algorithm returns one if the number of ones is odd. Byte swap algorithm performs conversion between big-endian and little-endian.

Theses algorithms are compiled to several existing RISC-V ISA instructions. Consequently, the extended equivalent as only one instruction results in an obvious smaller instruction memory footprint. The disassembler of the *RISC-V GCC* compiler shows 14 *RISC-V* assembly instructions for populations count algorithm, 8 *RISC-V* assembly instructions for parity algorithm and, 6 *RISC-V* assembly instructions for byte swap algorithm. Due to coherency only the population count algorithm is presented. Algorithm 2 depicts disassembly of Algorithm 1 that all together are equivalent to one *popcnt* instruction. For example, the first instruction of Algorithm 1 at line number 2 is compiled to three instructions of *srli*, *and* and, *sub* in the first three lines of Algorithm 2.

Algorithm 1 Population Count

```
1: unsigned int popcount (unsigned int x) {
2:   x = x − ((x >> 1) & 0x55555555);
3:   x = ((x >> 2) & 0x33333333) + (x & 0x33333333);
4:   x = (x + (x >> 4)) & 0x0F0F0F0F;
5:   x = (x + (x >> 16));
6:   return (x + (x >> 8)) & 0x0000003F;
7: }
```

Algorithm 2 Population Count Assembly

```
1:  srli a5, s0, 0x1
2:  and a5, a5, s5
3:  sub a5, s0, a5
4:  srli a1, a5, 0x2
5:  and a1, a1, s2
6:  and a5, a5, s2
7:  add a5, a1, a5
8:  srli a1, a5, 0x4
9:  add a1, a1, a5
10: and a1, a1, s4
11: srli a5, a1, 0x10
12: add a1, a1, a5
13: srli a5, a1, 0x8
14: add a1, a5, a1
```

B. Results and Discussion

Table II shows the *CPU Time* results i.e. the output report of the *ETISS* simulator for running a *for loop* iterating one million times for one function call (see L5 in Fig. 1). The function has three variations i.e. the built-in function, the *LLVM* algorithm, and the inline assembly of the new instruction presented in

[1] https://github.com/sifive/riscv-llvm/tree/master/compiler-rt/lib/builtins

TABLE II
BIT MANIPULATION INSTRUCTIONS PERFORMANCE

Instruction	Built-in A.	Algorithm	Extension	Reduction
popcnt	8.8312 s	8.5187 s	8.0812 s	5.13 %
parity	8.56768 s	8.34893 s	8.09893 s	2.99 %
bswap	8.56493 s	8.25243 s	8.06493 s	2.27 %

TABLE III
NEW INSTRUCTIONS EFFORT ESTIMATION

Effort	L1	L2	L3	L4	L5	Total
Edit	30 min	30 min	10 min	30 min	10 min	110 min

Table II in the second, third and fourth column respectively. The fifth column depicts the performance improvement of the inline assembly compared to the *LLVM* algorithm. The simulation shows up to 5% improvement for new instructions extension in comparison to the *LLVM* algorithms. These results justify the implementation of the expanded *RISC-V* processors.

Table III shows the effort estimation for adding a new instruction according to the presented design flow. These estimations are done with the assumption of already installed and available tools for a first time users who is familiar with compilers and simulators. The estimations show the time required for modifying the files. On L1, one should be familiar with *RISC-V ISA* and instructions bit pattern as each instruction should have a unique bit combination. For L2, one should consider the instructions pattern and should know the targeted *RISC-V* platform. On L3 the only required modification is the Makefile of the *ETISS* compiler to include the new compiler path. For L4 the *RISCV.nml* file should be modified to include the algorithm of the new instruction. Consequently, the *RISCVArch.cpp* file is created automatically. On L5 the *RISCVArch.cpp* file is applied to the *ETISS* tool chain and after re-installation of both the *GCC RISC-V* compiler and the *ETISS RISC-V* simulator, the newly added instruction is recognized by compiler.

V. CONCLUSION AND FUTURE WORK

Utilizing IP-XACT format as higher level of hierarchy for RISC-V ISA implementation facilitates the creation of different variations of the RISC-V ISA extensions. The variations of the processor implementations forces the compiler and simulator to adjustment accordingly. In this study, we have presented our RISC-V instruction extension design flow and analyzed its utilization effort. Also, we have considered the bit manipulation extension of the RISCV ISA for the PULPino platform and shown how to modify the GCC RISCV compiler accordingly. We have applied the necessary changes to the ETISS RISC-V simulator and experimented with the three instructions. We have compared our implementations with the equivalent algorithms and presented the performance results.

The future work will be to synthesize the extended RISC-V core for FPGA using the different configurations and to run the compiled RISC-V binaries that platform to verify the improvements predicted with simulation.

VI. ACKNOWLEDGMENT

This work is partially supported by the ITEA3 project 16018 COMPACT (Business Finland diary number 3098/31/2017, German ministry of education and research reference number 01IS17028).

REFERENCES

[1] IEEE, "IEEE standard for IP-XACT, standard structure for packaging, integrating, and reusing ip within tool flows," *IEEE Std 1685-2014 (Revision of IEEE Std 1685-2009)*, pp. 1–510, Sep. 2014.
[2] *The RISC-V Instruction Set Manual*, RISC-V Foundation, 2017, version 2.2.
[3] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, "Design and evaluation of smallfloat simd extensions to the risc-v isa," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 654–657.
[4] A. Traber, F. Zaruba, S. Stucki, A. Pullini, G. Haugou, E. Flamand, F. K. Gurkaynak, and L. Benini, "PULPino: A small single-core RISC-V SoC," in *RISC-V Workshop*, 2016.
[5] L. Ghica and N. Tapus, "Optimized retargetable compiler for embedded processors-gcc vs llvm," in *2015 IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*. IEEE, 2015, pp. 103–108.
[6] *RISC-V Bitmanip Extension*, Clifford Wolf, 2019, version 0.90.
[7] *RISC-V Vector Extension*, Andrew Waterman, 2019, version 0.7.1.
[8] S. Sen, S. Jain, S. Venkataramani, and A. Raghunathan, "Sparce: Sparsity aware general-purpose core extensions to accelerate deep neural networks," *IEEE Transactions on Computers*, vol. 68, no. 6, pp. 912–925, 2018.
[9] A. Murray and B. Franke, "Compiling for automatically generated instruction set extensions," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 2012, pp. 13–22.
[10] N. Sedaghati, R. Thomas, L.-N. Pouchet, R. Teodorescu, and P. Sadayappan, "Stvec: A vector instruction extension for high performance stencil computation," in *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2011, pp. 276–287.
[11] S. Vinco, M. Lora, E. Macii, and M. Poncino, "Ip-xact for smart systems design: extensions for the integration of functional and extra-functional models," in *2016 Forum on Specification and Design Languages (FDL)*. IEEE, 2016, pp. 1–8.
[12] A. Kamppi, L. Matilainen, J.-M. Määttä, E. Salminen, and T. D. Hämäläinen, "Extending ip-xact to embedded system hw/sw integration," in *2013 International Symposium on System on Chip (SoC)*. IEEE, 2013, pp. 1–8.
[13] F. Herrera, H. Posadas, E. Villar, and D. Calvo, "Enhanced ip-xact platform descriptions for automatic generation from uml/marte of fast performance models for dse," in *2012 15th Euromicro Conference on Digital System Design*. IEEE, 2012, pp. 692–699.
[14] E. Pekkarinen and T. D. Hämäläinen, "Modeling risc-v processor in ip-xact," in *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE, 2018, pp. 140–147.
[15] A. Kamppi, E. Pekkarinen, J. Virtanen, J.-M. Määttä, J. Järvinen, L. Matilainen, M. Teuho, and T. D. Hämäläinen, "Kactus2: A graphical eda tool built on the ip-xact standard," *The Journal of Open Source Software*, vol. 2, no. 13, p. 151, 5 2017. [Online]. Available: http://dx.doi.org/10.21105/joss.00151
[16] "RISCV Tools," https://github.com/riscv/riscv-tools.
[17] D. Mueller-Gritschneder, M. Dittrich, M. Greim, K. Devarajegowda, W. Ecker, and U. Schlichtmann, "The extendable translating instruction set simulator (ETISS) interlinked with an MDA framework for fast RISC prototyping," in *International Symposium on Rapid System Prototyping (RSP)*. IEEE, 2017, pp. 79–84.