An Iterative Design and Validation Methodology for RISC-V Custom Extension

Pierre Filiol and Luc Jaulin and Jean-Christophe Le Lann and Théotime Bollengier
ENSTA Bretagne, LabSTICC, Brest, France
{pierre.filiol, luc.jaulin, lelannje, theotime.bollengier}@ensta.fr

***Abstract*—This paper presents an iterative design methodology to guide the development of a RISC-V custom extension from the specification of the instructions down to the micro-architecture implementation. The proposed workflow follows a top-down approach which progressively adds hardware representativeness to a RISC-V architectural simulation while maintaining the ability to execute high level applications. The $xinterval$ custom extension [6] is taken as an example to illustrate how this process can be applied to benchmark real-world robotics problems.**

***Index Terms*—Interval computing, contractors, robotics, RISC-V, FPGA, evaluation methodolgy, toolchain, hardware simulation**

I. Introduction

The RISC-V architecture defines a set of standard ISA extensions, *i.e.* a set of instructions sharing a similar goal, to perform general-purpose computing tasks efficiently. Standard extensions provide support for various types of operands (integers, multi-precision floating-points) or programming paradigms (atomicity, vector computing) and can be freely combined in a processor design. The modular approach adopted by the RISC-V ISA allows chip manufacturers to control the overall complexity of a design and cut down the production costs while closely matching the requirements of the target applications.

Some niche applications fail to perform well enough using only the instructions available from the standard extensions and may greatly benefit from additional hardware acceleration. This situation has been anticipated by the RISC-V architects with the concept of custom extensions. Using this mechanism, a set of highly-specialized instructions can be inserted in the RISC-V opcode space and coexist with the standard extensions. Hardware vendors rely a lot on this feature as it typically involve lower time-to-market to develop efficient application-specific processors. This process involves modifying a RISC-V design on the micro-architecture level and the software toolchain (support of the new instructions in C) [10], [12].

From the system-designer's perspective, specifying a set of additional instructions to optimize an under-performing input problem is a tedious task. The existing bottlenecks must be identified to find the right balance between what needs be handled at hardware level and what should stay in software. This is an iterative process where each micro-architecture candidate is evaluated against metrics such as :

1) The overall speed-up reached with the new instructions on a set of reference applications.
2) The hardware resources required to build the circuit.
3) The energy consumption of the resulting design,...

Attempting to evaluate these metrics raises practical issues right from the beginning. Due to extremely high costs, it is not conceivable to build a fully fledged RISC-V prototype to test each iteration of the micro-architecture. Simulation seems like a mandatory strategy but brings its fair share of challenges such as the degree of representativeness to adopt. Approaches involving RTL-simulation are computationally-demanding (especially when the core grows larger) and are too low-level to execute real-world applications. At the other end of the spectrum, software emulation is well suited for functional evaluation but fails to capture many hardware-related issues such as operation latencies and resources utilization.

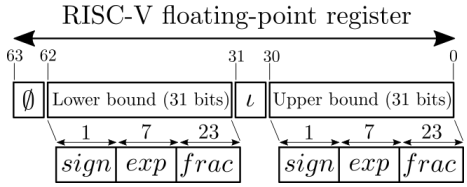

Fig. 1: Interval representation in xinterval

In [6], we designed a custom extension called *xinterval* which adds native hardware support for interval arithmetic (IEEE-1788 standard). The implementation is optimized for mobile robotics applications where a lot of recurring tasks such as localization, robust control or parameters estimation can be modeled by a constraint satisfaction problem involving intervals. The design philosophy is to use the double precision floating-points registers defined by the RISC-V D standard extension to fit the interval model depicted on Figure 1. The instructions defined by xinterval are mostly the interval counterparts for the most recurring floating-point arithmetic operators and transcendental functions.

Contributions: This paper describes an iterative process to specify, develop, test and measure the performances of a custom RISC-V extension by taking *xinterval* as an example. The promoted approach follows a top-down design methodology which gradually introduces hardware representativeness to a software RISC-V ISA simulator. The main benefit of this strategy, which complements the traditional workflows detailed in section II, is to maintain the ability to run a reference application at any stage of the design process to compute fine-grained metrics for various aspects of the custom extension (Figure 2). In the last iteration of the proposed hardware integration (Zynq-based), performances become good-enough for an embedded deployment and real-world tests. This progressive micro-architecture validation naturally eases the transition to a fully-fledged RISC-V in the latest design phases.

II. State of the art and previous works

Emulation is well suited in early design phases to execute the benchmarking applications and perform a functional evaluation of the custom extension. The Spike ISA simulator [11] is the canonical solution promoted by the standard and acts as a golden model for RISC-V ISA simulation especially suited for academic purpose. It can be used in most situations (full coverage of the standard extensions and privileges modes). Alternatively QEMU [2] favors performances over Spike's pedagogic approach. Both tools support the addition of new instructions with some kind of tracing mechanism to log each call and compute metrics.

Architectural simulators allows a more fine-grained analysis of the performances of a RISC-V design. In addition to the functional

verification offered by emulators, they also simulate the underlying hardware architecture with elements such as instruction and memory latencies, pipeline depth, bus frequencies.... Gem5 [3] is a reference in this domain for RISC-V studies. Similarly to the aforementioned emulators, the addition of new instructions is officially supported which makes it a good choice for the benchmarking of custom extensions.

Representativeness can be taken further by integrating the hardware model of the custom extension to an existing RISC-V design and performing the RTL simulation (e.g. GHDL [8]). A lot of RISC-V processor designs are available either royalty-free or commercially and implement the standard to various extents with multiple purposes. A processor suited for xinterval should at least implement the floating-point F and D extensions to host the interval model from Fig. 1. Most designs are synthesizable on medium to high-end FPGAs (depending of the complexity) to create low-cost prototypes for testing and performances evaluation. This approach has been used successfully in the literature to implement standard or custom extensions (e.g. the NOEL-V [7] to implement a vector extension suitable for space applications [5] or multi-precision unums [4] added as a Rocket RISC-V coprocessor [1]).

The integration of custom hardware primitives in a pre-existing processor design is not a trivial task and requires a deep understanding of the underlying micro-architecture. While this step remains necessary to validate a custom extension in real conditions, this paper advocates in favor of a complementary approach which consists in progressively adding hardware representativeness to a software architectural simulation. This offers a reduced complexity compared to direct hardware RISC-V integration but is sufficient to obtain relevant metrics regarding the micro-architecture and to validate the custom extension. Additionally, this hybrid software/hardware approach is deployed on a Zynq board to perform real-world embedded tests with sufficient performances. Fig. 2 details the proposed methodology.

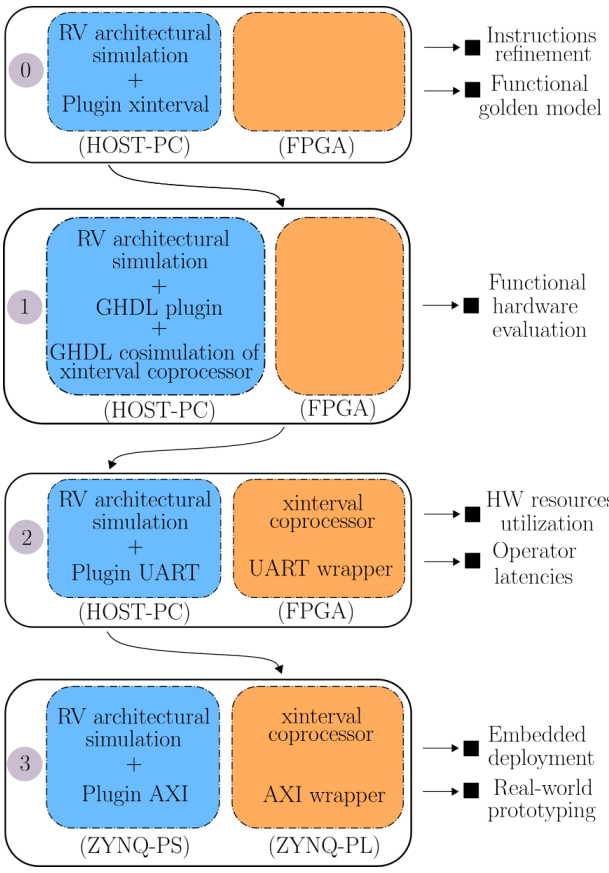

Fig. 2: Promoted design flow

III. Generic building blocks

The architectural simulation used in steps 0 to 3 (Fig. 2) relies on the custom software RISC-V ISA simulator (Fig. 4) The main features of this implementation are:
1) Support for the 64-bit RISC-V standard extensions required by $xinterval$. The simulated core is notably fully-compliant with F/D instructions and uses dedicated floating-point registers to store intervals.
2) Support for $xinterval$ instructions [6] with multiple back-ends. This includes full software (golden-model), GHDL (co-simulation), UART (cheap hardware integration), and AXI (efficient prototyping).
3) Dedicated custom gcc toolchain [6] with native support for $xinterval$ assembly instructions. A minimal syscall emulation is achieved by overloading the Newlib C library to implement virtual peripherals (Fig. 3).
4) Performance estimations for the micro-architecture under test. Each instruction called in a program run is tagged with an estimated clock latency taken from an internal database to build a rough estimate of the overall computation time. The latencies for standard extensions are taken from the GemV reference model and from the real hardware for $xinterval$ instructions.

The decision of re-implementing a RISC-V ISA simulator rather than modifying existing solutions (Spike or Gem-V) is motivated by the necessity of interfacing real hardware directly from the simulator. The price to pay for that approach is rather low, mostly thanks to the overall simplicity of the RISC-V standard compared to other architectures. Thus the whole simulator is made of less than 10k lines of C code (hence maintainable by a small team).

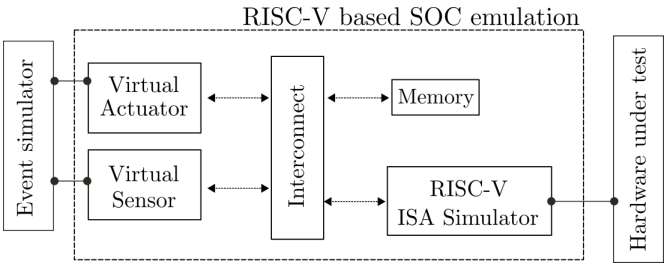

Fig. 3: Simulated SOC for event-driven micro-architecture validation

The $xinterval$ core is a VHDL implementation of the associated custom instructions used as a slave device of the ISA simulator in steps 1 to 3 (Fig. 2). The main features of this implementation are:
1) Simplified version of the classic RISC pipeline (Fig. 5).
2) Control by the ISA simulator with a simple instruction $\leftrightarrow$ result interface. The encoding of the instructions is the same as the one defined in the custom toolchain.
3) The values stored in local registers are replicated from the floating-point registers managed in the ISA simulator.
4) This minimal core can be plugged in several wrappers such as GHDL (step 1), UART (step 2) or AXI (step 3).

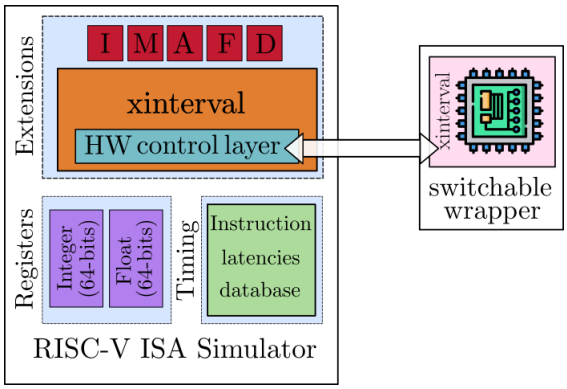

Fig. 4: Custom RISC-V ISA simulator with hardware interfacing capabilities

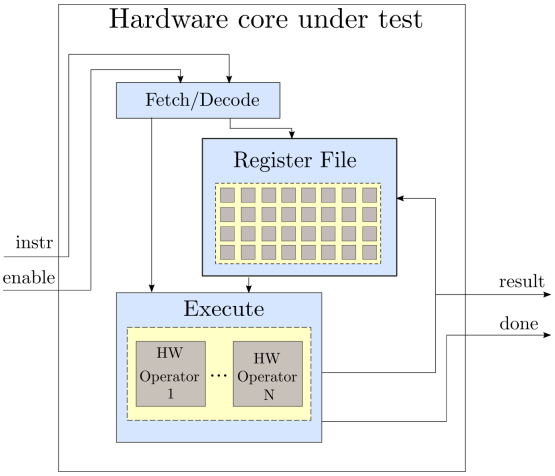

Fig. 5: Internal view of the xinterval core pipeline

IV. Iterative design of the custom extension

This section describes the iterative design strategy from Figure 2.

Step 0: **Implementation of a full software backend for xinterval instructions in the ISA simulator**. The following procedures are cycled until a satisfactory solution is found :

- Integration of the new instructions in a dedicated gcc toolchain.
- Development of the benchmarking applications using the C language. Each program comes in two flavors to provide a basis for performances evaluation. The first one only uses the standard RISC-V instructions while the other is allowed to use the dedicated primitives.
- Establishment of a functional reference model for the upcoming hardware developments.
- Profiling of the benchmarking programs execution and suitability assessment of the chosen instructions.

Step 1: **Functional evaluation of a preliminary simplified hardware model using cosimulation**.

- HDL design of the *xinterval* core using a simplified RISC pipeline architecture (Fig. 5). At this point no assumption is made about a particular target or clock frequency. It means that the underlying hardware operators should stay at the behavioral level while putting aside problematic such as pipelining or resources utilization.
- Implementation of a GHDL backend in the ISA simulator. The chosen cosimulation strategy (Fig. 6) offers a simple hardware-agnostic setup suitable for functional evaluation in the early design phase. The GHDL simulator runs a testbench instantiating the *xinterval* core and indefinitely exchanging instructions and results with the ISA simulator. This communication is performed using the VHPIDIRECT protocol [9] which allows the monitoring of VHDL signals from an arbitrary C routine compiled as a GHDL plugin. This feature is used to start a TCP server at the beginning of the RTL simulation to receive orders from the ISA simulator and control the instruction/result buffers adequately.
- Profiling of the benchmarking programs execution and functional verification of the hardware model.

Step 2: **Hardware model refinement on a FPGA target using a UART peripheral**.

- Refinement of the hardware core to match the desired performances on the target FPGA. This tedious process requires vendor-specific synthesis tools to find the acceptable trade-off between metrics such as operator latencies, resources utilization and energy consumption. As a matter of example, most of the hardware primitives used in *xinterval* rely on IEEE-754 floating-point algorithms which requires deep pipelines to be implemented efficiently on FPGA and make a high use of scarce resources such as DSP. The latencies of the resulting hardware primitives are added to the latency database maintained in the ISA simulator for upcoming profiling.
- Implementation of the UART backend (ISA simulator) and the UART hardware wrapper (FPGA). In this step, the *xinterval* core is accessed using a serial protocol which allows simple and low-bandwidth communications with a real FPGA.
- Profiling of the benchmarking programs execution and evaluation of the micro-architecture performances using the operator latencies declared in the ISA simulator. This process gives the order of magnitude for the speed-up achieved by the custom extension on a given application.

Step 3: **Embedded deployment on Zynq (Fig. 7) for efficient testing**. The goal is to overcome the limitations of the previous approach which focused on micro-architecture evaluation at the expense of simulation speed. The UART protocol drastically reduced the performances (Table I) by imposing an incompressible latency for each submitted instruction. This step leverages the Zynq architecture to implement a high-performance AXI communication between an ISA simulator running on Zynq-PS and the validated *xinterval* core on the Zynq-PL. The main features of the design are summarized below:

- **Zynq-PL side**: The *xinterval* core is wrapped using an AXI-LITE slave interface and made accessible to the Zynq-PS part.
- **Zynq-PS side**: Lightweight Linux environment to run the ISA simulator modified with a AXI backend. The latter communicates with the *xinterval* on the Zynq-PL part using a dedicated kernel-space device driver.

V. Comparative Performance Results

This section describes some of the results obtained when evaluating the performances of the *xinterval* micro-architecture [6] using the

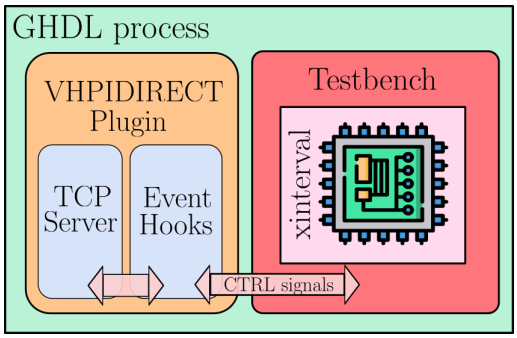

Fig. 6: GHDL cosimulation (step 1)

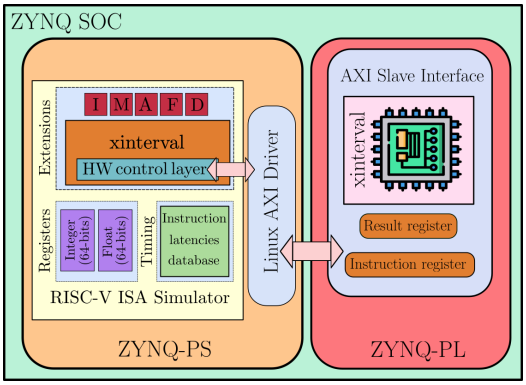

Fig. 7: Zynq deployment (step 3)

iterative method presented in this paper. The reference application computes all the solutions (x, y) in $[-6, 6] \times [-6, 6]$ which satisfies the constraint from (1) using interval contractors [14] and the SIVIA paving algorithm [13].

$$(y-5)\cos(4\sqrt{(x-4)^2+y^2}) - x\sin(2\sqrt{x^2+y^2}) \in [0,\infty] \quad (1)$$

Figure 8 shows a graphic representation of the solution set associated to constraint (1) when the benchmarking application is executed on a Zynq target in Step 3.

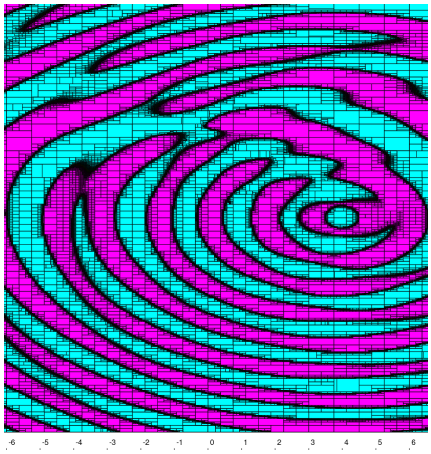

Fig. 8: Solution set (purple area) obtained in step 3 on Zynq

Table I presents the time required to execute the benchmarking application (accelerated flavor), an estimation of the true computation time on a real RISC-V (using the operator latencies) and a relative speed-up to the non accelerated application for all the steps.

	Simulation Time	Rel. CPU Speed-up	Comments
S0	8.2 s (Desktop PC)	/	Golden software model
S1	≈ 18h	/	Slow simulation Functional evaluation only
S2	≈ 8h	× **7.8**	Slow simulation First performance evaluation
S3	**37 s (Zynq)**	× **7.8**	Efficient simulation (target)

TABLE I: Comparative Performance Results

VI. Conclusion

This paper introduced an iterative design methodology to guide the development of a RISC-V custom extension from the specification of the instructions down to the hardware implementation. The greatest benefit of our approach is to maintain the ability to execute high-level applications at any step of the process while providing useful metrics to evaluate the performances of the design. The last step of the methodology provides a simple yet efficient solution for preliminary embedded deployment and lay strong basis for an upcoming fully-fledged RISC-V implementation.

References

[1] Krste Asanovic et al. "The Rocket Chip Generator". Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, April 2016.
[2] Fabrice Bellard. "QEMU, a fast and portable dynamic translator". In Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05). USENIX Association, USA, 41., 2005.
[3] Nathan Binkert et al. 2011. "The gem5 simulator". SIGARCH Comput. Archit. News 39, 2, 1–7, 2011. https://doi.org/10.1145/2024716.2024718
[4] Andrea Bocco, Yves Durand and Florent de Dinechin. "SMURF: Scalar Multiple-precision Unum Risc-V Floating-point Accelerator for Scientific Computing". CoNGA 2019 - Conference on Next-Generation Arithmetic, Singapour, Singapore. pp.1-8, 2019.
[5] Stefano Di Mascio, Alessandra Menicucci, Eberhard Gill and Claudio Monteleone. "Extending the NOEL-V Platform with a RISC-V Vector Processor for Space Applications", Journal of Aerospace Information Systems 2023 20:9, 565-574.
[6] Pierre Filiol. "Efficient Hardware Primitives for Interval Contractors in Robotics and Integration to a Custom RISC-V ISA Extension". 23rd IEEE International NEWCAS Conference, June 22-25th, Paris, 2025.
[7] Frontgrade Gaisler. "NOEL-V Synthesizable VHDL Model". https://www.gaisler.com/products/noel-v
[8] Tristan Gingold. "GHDL, a VHDL compiler". http://ghdl.free.fr/ghdl/index.html
[9] Tristan Gingold. "Co-simulation with GHDL", https://ghdl.github.io/ghdl-cosim/index.html#cosim
[10] OpenHWGroup, "CORE-V Instruction Set Custom Extensions", https://github.com/openhwgroup/cv32e40p/blob/cv32e40p_v1.3.2/docs/source/instruction_set_extensions.rst
[11] RISC-V Software, "Spike RISC-V ISA Simulator", https://github.com/riscv-software-src/riscv-isa-sim
[12] Sifive, "SiFive Int8 Matrix Multiplication Extensions Specification - Version 1.1", https://sifive.cdn.prismic.io/sifive/1a2ad85b-d818-49f7-ba83-f51f1731edbe_int8-matmul-spec.pdf
[13] L. Jaulin, M. Kieffer, O. Didrit and E. Walter, "Applied Interval Analysis", Berlin: Springer, 2001.
[14] R. E. Moore, "Interval Analysis", Prentice-Hall, 1966.