Acta Cybernetica **26** (2024) 889–912.

RISC-V Based Hardware Acceleration of Interval Contractor Primitives in the Context of Mobile Robotics

Pierre Filiol[ab], Theotime Bollengier[ac], Luc Jaulin[ad], and Jean-Christophe Le Lann[ae]

Abstract

Localization tasks, generally modeled as Constraint Satisfaction Problem (CSP), are recurring problems in mobile robotics. Known approaches rely on software libraries which all have their advantages but also their limitations, among which non-optimal computing performances. This paper proposes a different approach which consists in extending the RISC-V ISA to provide hardware support for interval primitives.

Keywords: intervals, contractor programming, RISC-V, IEEE-1788, mobile robotics, FPGA

1 Introduction

A lot of recurring mobile robotics tasks, such as robust-control or localization, can be modeled as a *Constraint Satisfaction Problem* (CSP) and solved by characterizing the corresponding solution set $\mathbb{S}$. A common way to achieve this is to compute an inner and outer approximation of $\mathbb{S}$, which is usually done with the help of contractor algebra and a paving algorithm such as SIVIA [21]. The paver classifies the search space by recursively calling a contractor to discard parts outside of the target set. In [7, 15, 20] this formalism is illustrated in real-world robotics examples.

A *contractor* C for the set $\mathbb{X} \subset \mathbb{R}^n$ is an operator $\mathbb{IR}^n \to \mathbb{IR}^n$ which satisfies:

$$C([\mathbf{x}]) \subset [\mathbf{x}] \text{ (}\textit{contractance}\text{)}, \tag{1}$$

$$[\mathbf{x}] \subset [\mathbf{y}] \implies C([\mathbf{x}]) \subset C([\mathbf{y}]) \text{ (}\textit{monotonicity}\text{)}, \tag{2}$$

[a]ENSTA Bretagne, Brest, France
[b]E-mail: pierre.filiol@netc.fr, ORCID: 0009-0009-6162-9578
[c]E-mail: theotime.bollengier@ensta-bretagne.org, ORCID: 0009-0006-7315-2736
[d]E-mail: lucjaulin@gmail.com, ORCID: 0000-0002-0938-0615
[e]E-mail: jean-christophe.le_lann@ensta-bretagne.fr, ORCID: 0000-0003-2555-1805

DOI: 10.14232/actacyb.306636

$$C([\mathbf{x}]) \cap \mathbb{X} = [\mathbf{x}] \cap \mathbb{X} \ (\textit{consistency}), \tag{3}$$

where $\mathbb{IR}^n$ is the set of axis-aligned boxes of $\mathbb{R}^n$.

Let C_1 and C_2 be two contractors. The union and intersection operators are defined as follows:

$$(C_1 \cap C_2)([\mathbf{x}]) = C_1([\mathbf{x}]) \cap C_2([\mathbf{x}]), \tag{4}$$

$$(C_1 \cup C_2)([\mathbf{x}]) = C_1([\mathbf{x}]) \sqcup C_2([\mathbf{x}]), \tag{5}$$

where $[\mathbf{a}] \sqcup [\mathbf{b}]$ is the smallest box which contains both $[\mathbf{a}]$ and $[\mathbf{b}]$.

The members of the robotics community usually rely on the use of existing software libraries to implement the contractors required to evaluate solution sets. The available libraries implement the IEEE-1788 standard on interval analysis [18] to various extents and define the most common interval primitives (operators, contractors, ...) for reuse. This paper advocates in favor of a novel approach which adds support for interval primitives directly in RISC-V hardware. We believe that this method can be beneficial to solve some of the portability issues occurring in software approaches, especially for embedded targets. Another advantage is the ability to compute with intervals using a tailored speed/precision compromise which is impossible on general-purpose hardware.

This paper is organized as follows. Section 2 presents a typical mobile robotics localization problem which is used as a reference throughout the article. Then Section 3 discusses the benefits of a hardware approach for intervals. Section 4 compares common hardware acceleration techniques while Section 5 presents the methodology used to create the proof of concept for hardware support of intervals. Section 6 sums up the obtained results and presents future works.

In the rest of the paper, sets $\mathbb{X}$ of $\mathbb{R}^n$ will be represented in *mathbb* font and intervals $[x]$ or boxes $[\mathbf{x}]$ within brackets.

2 State of the art and previous work

Let us consider the following localization problem which will be used for the discussions in this paper. The goal is to solve the CSP which is related to the localization of a robot (green triangle) using 3 landmarks (red circles) with known positions (Figure 1). The robot uses its internal sensors to compute the distances d_i toward landmark i with $i \in \{1, 2, 3\}$. The coordinate of each landmark is defined by a_i with $i \in \{1, 2, 3\}$. Those values are represented by intervals due to measurement uncertainties. The goal is to estimate the position (x_r, y_r).

The state vector of the robot is defined as:

$$\mathbf{x} = (x_r, y_r)^\top \tag{6}$$

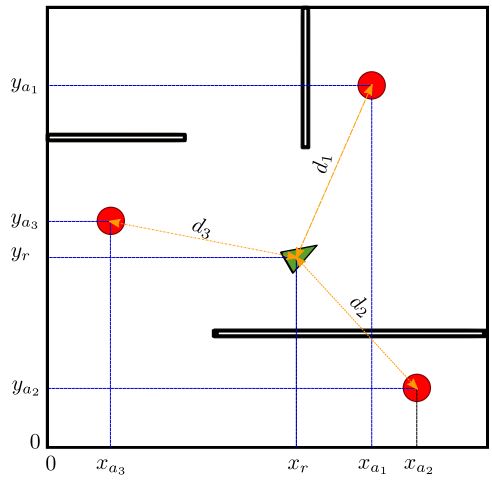

Figure 1: Localization of a robot using 3 landmarks

and its respective domain as:

$$[\mathbf{x}] = [0, \infty] \times [0, \infty]. \tag{7}$$

A couple of coordinates (x_r, y_r) is a potential solution if it satisfies the following constraints (ring equations):

$$a)\ (x_r - x_{a_1})^2 + (y_r - y_{a_1})^2 \in {d_1}^2, \tag{8}$$

$$b)\ (x_r - x_{a_2})^2 + (y_r - y_{a_2})^2 \in {d_2}^2, \tag{9}$$

$$c)\ (x_r - x_{a_3})^2 + (y_r - y_{a_3})^2 \in {d_3}^2. \tag{10}$$

Each constraint gives a corresponding solution sets:

$$\mathbb{S}_1 : \{\mathbf{x} \in [0, \infty]^2 \mid (x_r - x_{a_1})^2 + (y_r - y_{a_1})^2 \in {d_1}^2\}, \tag{11}$$

$$\mathbb{S}_2 : \{\mathbf{x} \in [0, \infty]^2 \mid (x_r - x_{a_2})^2 + (y_r - y_{a_2})^2 \in {d_2}^2\}, \tag{12}$$

$$\mathbb{S}_3 : \{\mathbf{x} \in [0, \infty]^2 \mid (x_r - x_{a_3})^2 + (y_r - y_{a_3})^2 \in {d_3}^2\}. \tag{13}$$

And the solution set for the localization problem becomes:

$$\mathbb{S} = S_1 \cap S_2 \cap S_3. \tag{14}$$

The goal is now to find 3 contractors C_1, C_2, C_3 which respectively characterize sets $\mathbb{S}_1, \mathbb{S}_2, \mathbb{S}_3$ and to perform the intersection of contractors to find $\mathbb{S}$.

C_1 can be computed with a forward-backward procedure [25] such as HC4-revised by using the set $\mathbb{S}_1$ AST (Figure 2). Note that throughout the article forward and backward contractors on operator/function $*$ are respectively defined as $\overrightarrow{C_*}$ and $\overleftarrow{C_*}$. The resulting contractors can be found in Algorithm 1 and 2.

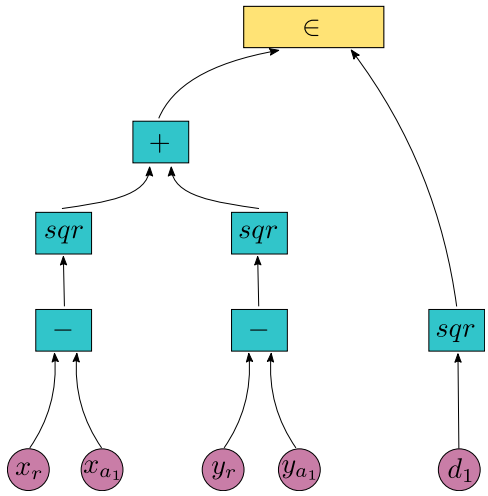

Figure 2: Abstract syntax tree corresponding to $\mathbb{S}_1$

Algorithm 1 Algorithm for $\overrightarrow{C_1}$

Funct $\overrightarrow{C_1}([x_r],[y_r],[x_{a_1}],[y_{a_1}],[d_1])$

1: $[DistX] = \overrightarrow{C_-}([x_r],[x_{a_1}])$
2: $[DistY] = \overrightarrow{C_-}([y_r],[y_{a_1}])$
3: $[DistXSqr] = \overrightarrow{C_{sqr}}([DistX])$
4: $[DistYSqr] = \overrightarrow{C_{sqr}}([DistY])$
5: $[DistSqr] = \overrightarrow{C_{sqr}}([d_1])$
6: **return** $[DistXSqr],[DistYSqr],[DistSqr]$

The following libraries are among the most commonly used in robotics:

- Goualard et al's Gaol [14]. A C++ low-level interval library which is focused on providing most of the common reverse operators for backward propagation.
- Chabert et al's Ibex [4] (IMT Atlantique). A library written in C++ targeting system solving and global optimization problems using interval arithmetic.
- Nehmeier et al's libieep1788 [24]. A C++ template library focused on rigorous implementation of the IEEE-1788 standard.

Algorithm 2 Algorithm for $\overleftarrow{C_1}$

Funct $\overleftarrow{C_1}([DistXSqr], [DistYSqr], [DistSqr])$

1: $[DistXSqr], [DistYSqr] = \overleftarrow{C_+}([DistSqr], [DistXSqr], [DistYSqr])$
2: $[a] = \overleftarrow{C_{sqr}}([DistXSqr], [DistX])$
3: $[b] = \overleftarrow{C_{sqr}}([DistYSqr], [DistY])$
4: $[x_r] = \overleftarrow{C_-}([DistX], [x_r], [x_{a_1}])$
5: $[y_r] = \overleftarrow{C_-}([DistY], [y_r], [y_{a_1}])$
6: **return** $[x_r], [y_r]$

- INRIA's MPFI [29]. A C library implementing interval arithmetic in arbitrary precision by using MPFR reliable floating-points.

The contractor C_1 can be created by implementing Algorithms 1 and 2 using the simple contractor primitives available for example in the aforementioned software libraries.

3 Motivations for a hardware approach

The minimal requirement to perform interval computations in an embedded robotic system is the availability of an hardware *floating-point unit* (FPU) . The industrial standard for that purpose is the IEEE-754 [19] which enables a theoretical portability of the user code to any compliant hardware. However, no guarantee is made about consistency and the final result will not be bit-consistent across multiple {hardware, compiler} pairs. This subject has been extensively researched in contributions such as [9, 13].

This lack of consistency is especially problematic for interval computations which require frequent switches between IEEE-754 rounding modes to perform accurate interval bounds evaluation (respectively round to $-\infty$ and round to $+\infty$ for lower and upper bounds).

Another obstacle lies in the implementation of the interval libraries themselves. Most of them are built using third-party maths libraries which perform fast and accurate floating-point evaluation at the expense of portability. It is frequent for them to inline assembly instructions directly in the C code for optimization but as a consequence the whole library becomes architecture-locked.

For all the reasons detailed above, we believe that relying on a standard floating-point architecture is not a valid approach. This paper presents a dedicated hardware architecture to accelerate the most common interval primitives along with a dedicated compiler. The {hardware, compiler} pair thus tackles the problem of embedded intervals and presents a proof of concept for a hardware accelerator targeted at the most commonly used interval primitives.

3.1 Tailored performances for interval computation

A lot of robotics applications require to repetitively call a contractor to make use of a short-lived sensor input. This is typically the case of the localization example introduced in Section 2 where the distances d_i are used to estimate the robot location. Ideally the contraction fixed-point must be reached before obtaining new inputs which implies to find a compromise between execution speed and result precision. These factors are known to be inversely proportional in the context of floating-point operators [11].

Popular architectures like X86 or ARM use hardware FPUs which are optimized for the IEEE-754 [19] single- and double- precision floating-points. While these formats are good "one size fits all" trade-offs for most of general-purpose computations, they can become a limiting factor in some niche applications such as interval arithmetic. The main reason is that they impose predefined ranges and precisions for floating-point numbers while real-world problems can have varied requirements.

Another drawback is the overall hardware complexity induced by general-purpose FPUs which are required to support all the rounding modes defined in the IEEE-754 standard. An FPU tailored for interval arithmetic, on the other hand, would only require two rounding modes for bound computations (round to $-\infty$ and $+\infty$), thus leading to a lower hardware complexity and power consumption.

3.2 Guaranteed computation at hardware level

Interval arithmetic is especially suited to solve hard non-linear problems in a reliable and efficient way. The IEEE-1788 standard [18] specifies the behavior of interval operators using various flavors which are all organized as depicted in Table 1. From now on, we consider only the set-based flavor which is the most commonly used in robotics.

The mathematical layer describes how interval operations work. It also introduces the notion of decorated intervals comprising a bare interval and a decoration. The latter are meant to implement the standard way of dealing with non-nominal cases during computation. The available decorators for the set-based flavor are presented on Figure 1 and their inclusion relationships are as follows:

$$com \subset dac \subset def \subset trv \supset ill. \tag{15}$$

In a previous article [10] we had shed light on a recurring contractor arithmetic bug which occurs when a function evaluates an interval that is not or partially in its domain of definition. The solution proposed was to mark the incriminated interval with a ι flag, perform the forward propagation normally and execute a modified backward propagation which takes the flag into account. A solution in the spirit of IEEE-1788 would be to create a decorator "out-of-domain" (*ood*) to handle this non-nominal case but doing this without breaking the existing logic depicted in Formula 15 is hard.

Table 1: The set-based flavor decorators (extracted from [18])

Value	Short description	Property	Definition
`com`	common	$p_{\texttt{com}}(f, \boldsymbol{x})$	$\boldsymbol{x}$ is a bounded, nonempty subset of $\mathrm{Dom}(f)$; f is continuous at each point of $\boldsymbol{x}$; and the computed interval $f(\boldsymbol{x})$ is bounded.
`dac`	defined & continuous	$p_{\texttt{dac}}(f, x)$	$\boldsymbol{x}$ is a nonempty subset of $\mathrm{Dom}(f)$, and the restriction of f to $\boldsymbol{x}$ is continuous;
`def`	defined	$p_{\texttt{def}}(f, \boldsymbol{x})$	$\boldsymbol{x}$ is a nonempty subset of $\mathrm{Dom}(f)$;
`trv`	trivial	$p_{\texttt{trv}}(f, \boldsymbol{x})$	always true (so gives no information);
`ill`	ill-formed	$p_{\texttt{ill}}(f, \boldsymbol{x})$	Not an Interval; formally $\mathrm{Dom}(f) = \emptyset$,

The ι flag information could be added to the standard and could be directly used by the hardware implementation.An example of an interval encoding using the ι flag is depicted in Figure 7. This approach comes with a lot of freedom since few things are currently defined in the standard regarding the bit-level representation of intervals (decorated intervals, empty set, NaN handling,...).

4 Evaluation of hardware acceleration strategies for interval arithmetic

4.1 Affordable hardware prototyping with FPGA

The *application specific integrated circuit* (ASIC) technology is nowadays the industrial standard to produce high-end chips. This design process allows to reach the highest performances at the price of tremendous costs in development and manufacturing facilities. The resulting chips are made affordable for customers only through mass-replication which lowers the per-unit cost. As a consequence, this technique is not suitable for hobbyists or small research teams who operate with tight budgets and produce only a few prototypes to evaluate the performances of a specific design.

A solution for this audience lies in the *field programmable gate array* (FPGA) technology which comes at much affordable prices. While ASICs operate directly at transistor level, FPGAs expose an array of interconnected logic cells which contain the digital building blocks for more advanced logic (Figure 3). The desired behavior of the user logic is described through *hardware description languages* (HDL) such as VHDL or VERILOG which are also widely used in ASIC field. With the help of dedicated software, called synthesizers, the user can modify the connections between cells to match the HDL code. This process can be repeated any number of times and allows incremental development whereas ASICs chips are etched once and forever. Table 2 briefly compares both technologies.

From now on, any hardware design mentioned in this paper refers to the production of HDL code for synthesis on an FPGA chip.

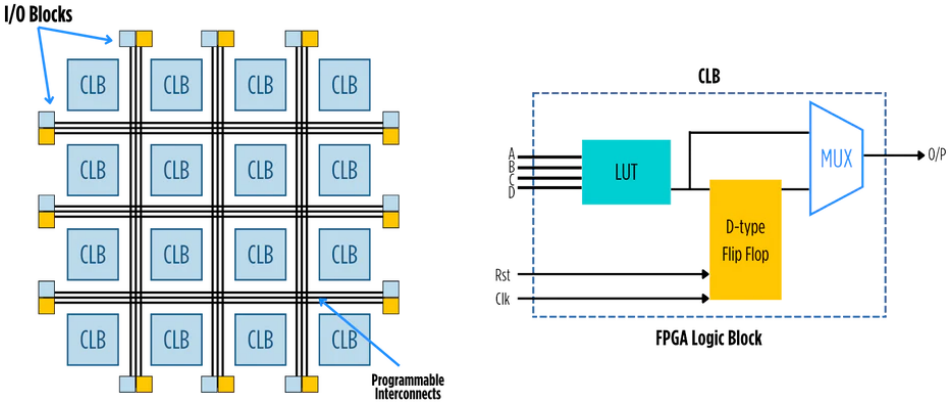

Figure 3: FPGA reconfigurable network and content of a logic cell [6]

Table 2: Comparison of FPGA and ASIC

	FPGA	**ASIC**
Design flow	Reconfigurable circuit (suitable for prototyping)	Permanent circuitery (no room for error)
Design input	HDL	HDL
Entry cost	Low (from 100 $ to 10k dollars)	High (typically millions of dollars)
Typical use	Prototyping of a few units	Mass-production
Energy consumption	Higher than ASIC	Lower than FPGA
Frequencies	Lower than ASIC	Higher than FPGA
Analog designs	Not supported.	Supported (eg transceivers ...).

4.2 Full hardware equation mapping strategy

FPGA are very efficient at implementing highly specialized circuits for niche applications that no manufacturer would mass produce for cost reasons. There are no theoretical issues in translating a set of equations into a dedicated circuit, this approach is very FPGA compliant and has been done very often in literature [8, 12, 16]. While the equations from a contractor such as C_1 are no different and could be implemented in hardware with great performances, this solution will not be the preferred one as the main focus is made on re-usability, ease of prototyping and costs.

Mapping equations from contractors into efficient hardware confronts the designer with difficult FPGA problems such as optimizing the timing and pipelining of

a big combinatorial circuit. The slightest modification of the input contractor (like adding an angular measurement for the landmarks) would require to re-engineer the circuit from scratch and face those difficulties again. In these conditions, it is difficult to imagine that an interval user would be skilled enough both in HDL and in hardware development to be really autonomous with this solution. Moreover, it is frequent to see evolutions in input contractors during the development of a robotic application.

Another drawback of this method is related to high hardware resource consumption. A circuit with no logic folding would require one instantiation for each contractor primitive in the target contractor. For example, Algorithm 1 (forward contraction from C_1) performs 3 square contractions which require duplicated logic. The amount of available hardware resources on a typical FPGA is limited, especially on the low-end and affordable ones. The use of this technique should be reserved to either simple contractors (for which the need of an hardware implementation is dubious) or for production-oriented synthesis on high-end boards.

4.3 Coprocessor strategy

This solution is an alternative to exposing the hardware directly as it was done in previous technique. The user communicates with a hardware device using a software abstraction. In general-purpose computing, this paradigm is often implemented using a coprocessor which performs the computation jobs requested by the main *Central Processing Unit* (CPU). For example, this is the computation model adopted in modern *Graphic Processor Units* (GPUs) where graphic pipeline operations are transferred to a PCI device with the help of APIs such as CUDA or OPENCL.

This technique suits interval computation and a FPGA could be used as a coprocessor to expose data parallelism (SIVIA boxes for example) in a GPU fashion. While this system is theoretically the best compromise between re-usability and performances (logic folding and hardware parallelism), it will not be studied further in this paper as it implies too much complexity for a first approach to hardware intervals. Several problems must be solved to achieve this goal which are beyond building an efficient interval core. The designer must indeed organize the communication between several core instances (for parallelism), optimize host to device communications and produce an efficient compiler.

4.4 Instruction set extension of a general purpose cpu

The two previous subsections have raised the importance of logic folding to lower the hardware complexity and resources utilization. This new strategy aims at adding interval operators directly in the main CPU. An *Instruction Set Architecture* (ISA) defines the supported assembly instructions and their behaviour in an implementation-agnostic way to guarantee binary compatibility between several chip manufacturers. Popular ISAs come with compiler support for high-level

languages such as C thus allowing the development of general-purpose software libraries (libc,...).

This paper aims at extending an existing ISA to support a cleverly chosen set of hardware interval primitives. The main benefit is to produce more efficient libraries that make use of hardware interval instructions instead of the general-purpose ones (Figure 4). This operation requires both the modification of the CPU (circuit) and the compiler (to use the new instructions).

In practice, there is no official mechanisms to alter mainstream CPUs from brand such as INTEL, AMD or ARM to inject new hardware instructions. First and foremost, their architecture design is extremely complex (gate-level design) and the manufacturing process requires industrial tools which are far beyond the budget of hobbyists. Additionally, the corresponding ISAs are distributed under proprietary licenses which prevent any legal modifications. Finally, as older ISAs such as X86 have very limited available opcode space remaining, manufacturers are reluctant to add new instructions to the standard unless there is a major consensus.

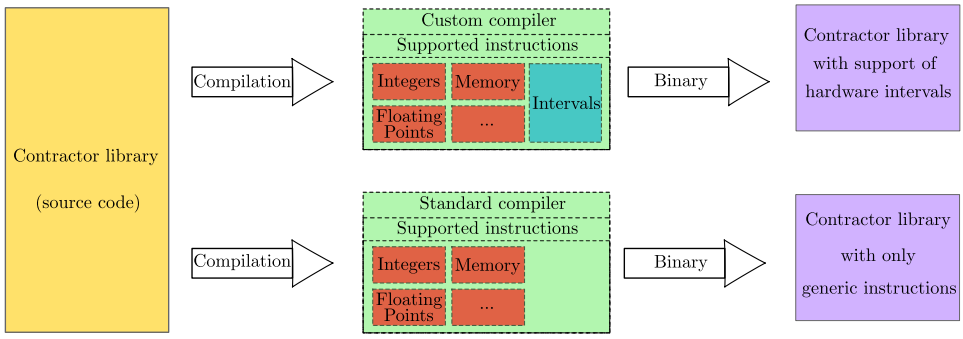

Figure 4: ISA and software stack modification

5 Design of a RISC-V interval custom extension

5.1 The RISC-V standard

The RISC-V standard brings some solutions to the aforementioned problems. Contrary to X86/ARM, it is an open and free ISA [31] whose specification started in 2014 as a purely academic project carried out by the university of Berkeley. As a consequence, anyone is free to implement a core which follows RISC-V standards without paying royalties to any third-party. The main philosophy behind RISC-V is to promote a simple processor model featuring a load-store micro-architecture typical of *Reduced Instruction Set Computer* (RISC) in opposition to the *Complex Instruction Set Computer* (CISC) adopted by X86 processors. The performances and overall simplicity of RISC-V design has gained traction in the last years for

low-power/embedded applications in various fields such as neural-networks [22], cryptography [5] or in our case robotics [32].

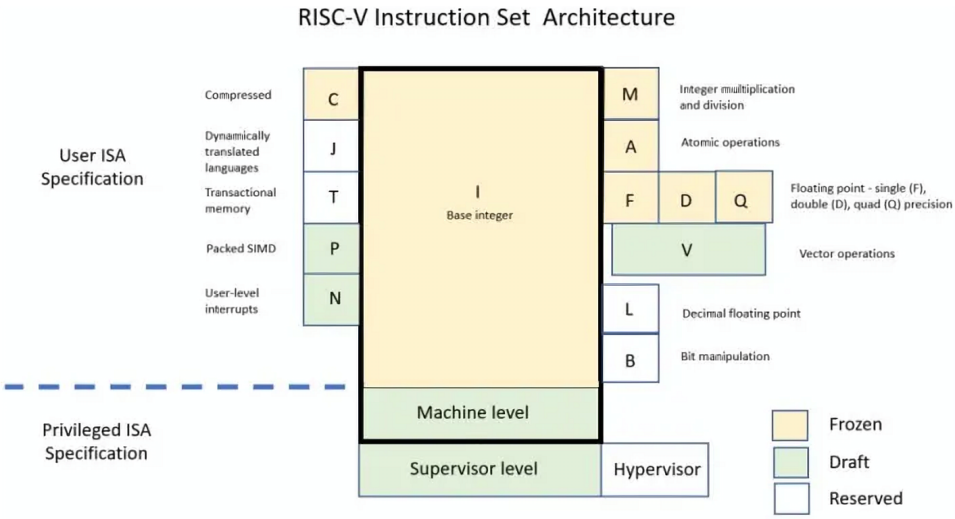

Figure 5: The standard extensions [28]

The standard provides no hints about processor implementation but enforces several design characteristics about the ISA:

- The ISA is designed with modularity in mind. Any RISC-V implementation is composed of a mandatory base ISA (named I) and a number of ISA extensions (identified by a letter). The aim is to allow the user to build a custom processor perfectly tailored for a specific need. All the extensions currently specified in the standard are displayed in Figure 5. A RISC-V core can be described using a naming convention which consists in RV + the register width (in bits) and the supported extensions. For example, a RV32IMFD is a 32 bits core implementing extensions I (base), M (integer multiplication), F (single-precision floating point support) and D (double-precision floating point support).
- The ISA specifies the required registers for each extension as well as their width. For example, base extension (I) comes with 32 registers of width 32 or 64 (depending on the chosen implementation).
- The ISA can be extended with custom extensions to add application-specific operators.

Another strength of the standard lies in its very rich software ecosystem. The user can access a fully-fledged GCC compiler with a dedicated RISC-V toolchain

which takes into account which extensions are actually available in the core to produce optimized binaries.

5.2 Objectives and virtual prototyping

The paper will use the RISC-V standard as a way to demonstrate the feasibility of adding hardware support for intervals and tackle real-world robotic problems. A complete strategy would be as follows:

- Due to the overall simplicity of the standard, it is possible to synthesize an HDL description of a RISC-V core on a middle range FPGA. A lot of core designs can be found in various configurations either in the literature such as [26, 23] or bought from *Intellectual property* (IP) vendors such as Sifive.com. This allows to build a baseline effortlessly and evaluate the performances on the localization problem given in section 2 with only the standard instructions. The target core configuration is set to RV32IMAFD for the reasons exposed in Section 5.3.

- After the initial performance measurement, we take advantage of RISC-V extensibility to add support for a new extension geared toward interval computation named *xinterval*. Two major tasks must be done to achieve this goal. The first one is to implement the *xinterval* instructions in hardware (using an HDL) and to integrate them in the base design synthesized on FPGA (Section 5.3). The second one consists in adding support for instructions in the compiler. To this end, we rely on the available RISC-V GCC which has also been designed with extensibility in mind and allows to define new instructions easily (Section 5.4).

- Evaluate the performances on the localization problem given in Section 2 with the new instructions/compiler and compare with previous results.

The hardware design of a RISC-V extension is a complex topic which demands an extensive knowledge of the underlying micro-architecture and raises various optimization trade-offs. This has been done in literature in works such as [3, 28] and [2]. As a first step we wanted to prove that intervals could be added to RISC-V ISA in an efficient way without worrying too much about hardware implementation. This goal can be achieved through *Virtual Prototyping* (VP) which allows functional evaluation. This approach is often used in system design [1, 17, 27] to decrease development times and help in the upcoming *Register Transfer Level* (RTL) verification step. The development of hardware primitives in an HDL and its integration is a RISC-V core is left as future work and will be the main topic of a next article.

The VP used in this study is organized as depicted in Figure 6. The host computer runs a simulation program where a robot tries to estimate its location using 3 landmarks with fixed coordinates (box ③ on Figure 6). At each simulation

step, the distance measurements between the robot and each landmark are sent in the memory of a simulated RISC-V core (box ① on Figure 6). The processor simulator is another program written in C++ which aims at achieving functional emulation of a binary code execution up to the assembly level. It implements the required RISC-V standard extensions and *xinterval* instructions and is able to execute a binary code compiled with the custom GCC presented in Section 5.4. Practically, the simulated core runs an implementation of Algorithms 1 and 2 from section 2 implemented with *xinterval* instructions instead of traditional floating-point.

5.3 Design principle of the `xinterval` custom extension

The simulated core ① from Figure 6 is defined by RISC-V standard as a RV32IMFD. It implements the following standard extensions and registers:

Table 3: Extensions used in the simulated core

Letter	Name	Number of instructions
I	Base Integer	40
M	Integer Multiplication	8
F	Single-precision floating-point	26
D	Double-precision floating-point	26

Table 4: Registers used in the simulated core

Names	Characteristics	Number
x_0-x_{31}	32-bit integer register	32
f_0-f_{31}	32-bit floating-point register	32

The simulated core relies on standard extensions F and D which bring dedicated registers and instructions (Tables 2 and 3) to enable floating-point support respectively in single (32-bit) and double (64-bit) precisions. An interesting feature of the RISC-V standard is the ability to handle 64-bit floating-point numbers on a 32-bit processor (identified by RV32xxx) by working on pairs of 32-bit floating-point registers. The main idea behind *xinterval* integration is to fit the interval representation into one of the aforementioned register, to take advantage of the standard F/D instructions to handle load/store operations and to develop the missing HDL logic to perform interval computations. Section 3.1 recalled that a hardware approach allowed to tailor the performances according to the needs of a specific application. For intervals, this translates into the ability to configure the floating-point format used to encode the bounds of an interval instead of forcing a sub-optimal IEEE-754

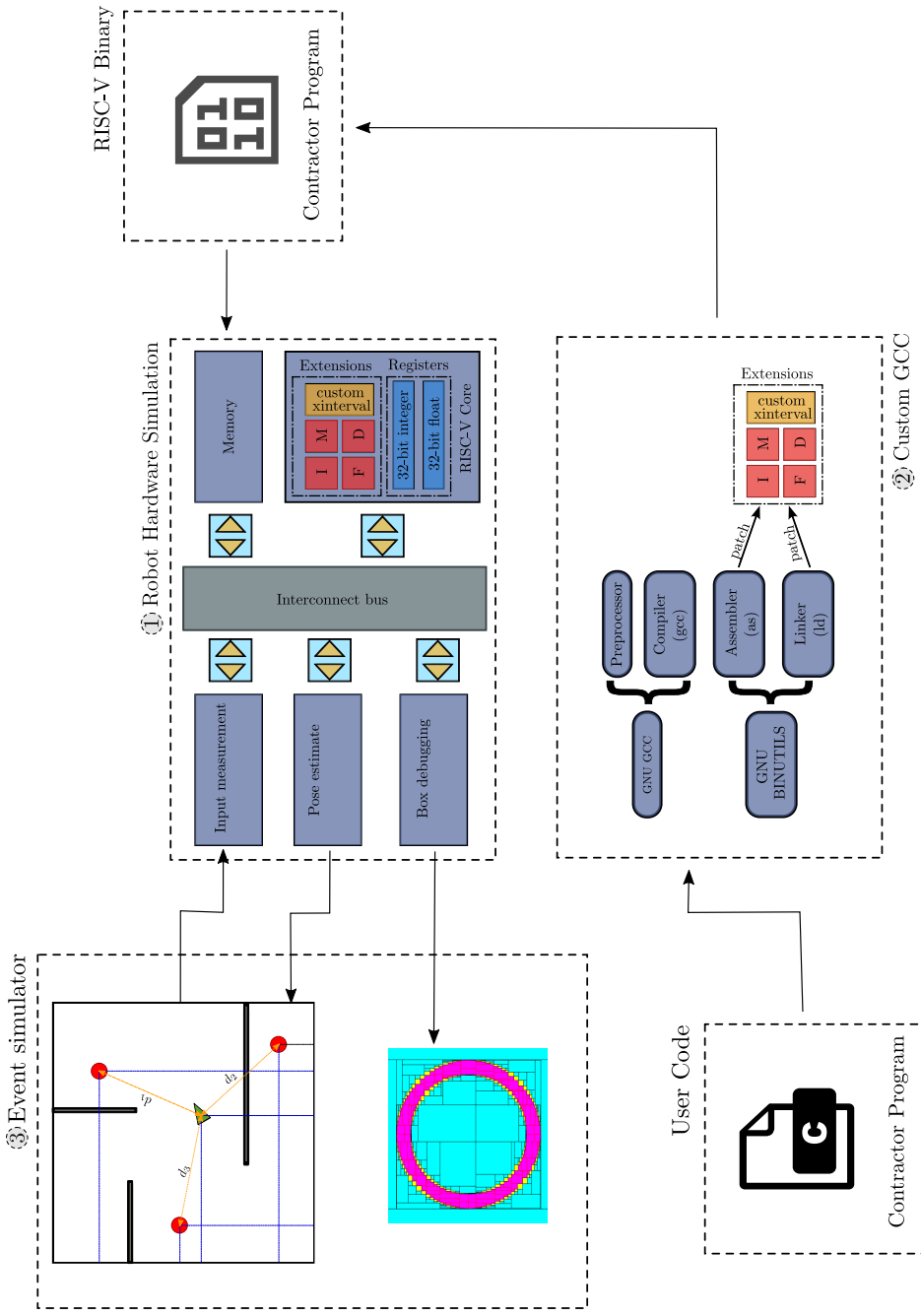

Figure 6: The RISC-V evaluation testbench

format. A direct consequence is the impact on performances in matter of latency, resource utilization and frequency of the resulting hardware design which allows to address a wide range of applications and targets.

In the particular context of RISC-V integration, the interval representation needs to fit a 64-bit double register. We made the choice to use the interval representation depicted in Figure 7. Here, a bound is encoded using 31 bits with a 7 bits exponent field (while 8 are used in IEEE-754 single precision). Practically, this reduces the range of representable numbers (which still remain acceptable for typical robotics applications) but leaves room for two additional 1-bit flags in the 64-bit data. The empty flag marks an empty set and the ι flag is applied to intervals which are subject to the phenomenon illustrated in Section 3.2. If an application requires greater speed at the expense of precision, it would be possible to further reduce the number of bits used for the bounds, even if this means leaving unused offsets in the register.

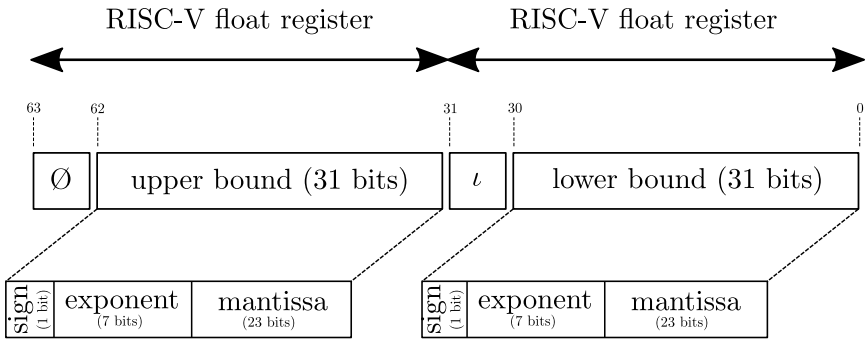

Figure 7: The 64-bit interval representation used in *xinterval*

The custom *xinterval* extension contains instructions to perform the forward and backward contractions of recurring interval primitives. The RISC-V standard introduces limitations that force instructions to adopt one of the encoding shown in Figure 8. Most of the time, we use the R-Type which stands for "register instruction" and encodes a destination register (r_d) and two source registers (r_1 and r_2). Fields *opcode*, $funct_3$, and $funct_7$ are used to discriminate between two R-Type instructions and send them to the right portion of the logic circuit in the processor. For interval primitives with two inputs (addition, ...) the associated backward contractor must have 3 inputs (see Table 5) which forces to use the R4-Type which is a sub-case of R-Type where the $funct_7$ is partly replaced by a third source register r_3. To ease the explanations, instruction formats used in this section will be named using RISC-V GCC terminology (Section 5.4/Table 5).

The general design philosophy used in *xinterval* will now be reviewed through 2 examples:

The addition is a case of a two inputs arithmetic operator for which *xinterval* instructions are summed up in Table 6. The forward contractor performs a simple addition using the input interval operands saved in registers r_{s1},r_{s2} and stores the

Table 5: An extract of GCC terminology for instruction fields

Mnemonic	**Meaning**
D	Fp destination register
S	Fp source register 1
T	Fp source register 2
R	Fp source register 3

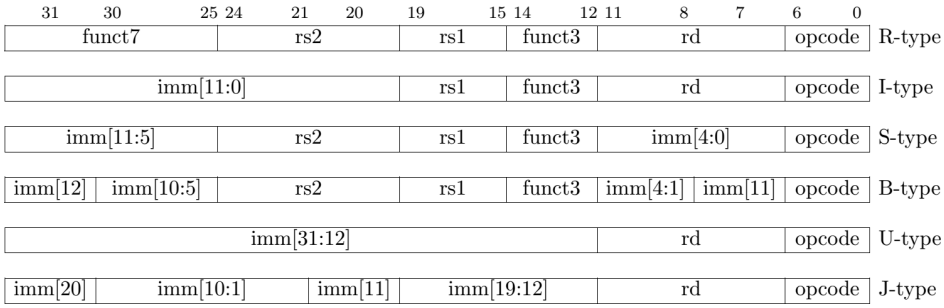

Figure 8: Legal instruction formats in the RISC-V standard

result in destination register r_d. Two corresponding backward-contractors can be defined to contract each input separately. In robotics, the output of the backward contraction is often intersected with the corresponding input pre-propagation to update state variables (x_c and y_c in Table 3). As a consequence, we made the choice to optimize this precise case at the expense of an additional register r_{s3}.

Table 6: Instructions linked to operator +.

Instruction	Prototype	Type	Operation	Register movements
addfwctc	"D,S,T"	+ forward contractor	$z = x + y$	$r_d = r_{s1} + r_{s2}$
addbwctc1	"D,S,T,R"	+ backward contractor 1	$x_c = x \cap (z - y)$	$r_d = r_{s1} \cap (r_{s3} - r_{s2})$
addbwctc2	"D,S,T,R"	+ backward contractor 2	$y_c = y \cap (z - x)$	$r_d = r_{s2} \cap (r_{s3} - r_{s1})$

The principle differs a bit for algebraic and elementary functions, and especially for those which are not defined everywhere and suffer from the bug recalled in Section 3.2. The example of the square root illustrates all these issues. The forward contractor performs a modified version of the square root ($sqrt_\iota$) which sets the ι flag of the output interval when the input is not entirely in the domain of definition.

The backward contractor evaluates the ι flag of the input and performs a modified version of the backward square root function ($sqrt_\iota bw$). It works the same way as the traditional operator but decorates the interval with a iota if the input is partially in the square root domain of definition. This time, square root required only one interval input (in r_{s1}) so the matching prototype in RISC-V assembly for forward and backward are respectively "D,S" and "D,S,T". These operations are described in Table 7.

Table 7: Instructions linked to function $\sqrt{}$.

instruction	Prototype	Type	Operation	Register movements
sqrtfwctc	"D,S"	Square root forward contractor	$y = sqrt_\iota(x)$	$r_d = sqrt_\iota(r_{s1})$
sqrbwctc	"D,S,T"	Square root backward contractor	$x_d = sqrt_\iota bw(x, y)$	$r_d = sqrt_\iota bw(r_{s1}, r_{s2})$

The custom extension *xinterval* proceeds in an analog way for all the operators and algebraic, elementary functions which are often used in robotics applications (Table 8).

Table 8: Supported contractor primitives in *xinterval*.

Addition	Subtraction	Multiplication	Division
Square root	Square	Exponential	Logarithm
Cosine	Sine		

5.4 Modification of the software stack

The GCC RISC-V toolchain is an open-source project bundled as part of the RISC-V software stack. Support for new instructions can be added by modifying the *binutils* module whose purpose is to convert assembly code into an executable binary. This module is then called by GCC as part of the compilation process as depicted in ② in Figure 6. This method is simple because it only requires the modification of *binutils* assembler which is a far less complex code base than the compiler itself.

The encoding of new assembly instructions must be added to the source code of *binutils* and satisfy two rules:

- Uniqueness: two instructions cannot have the same encoding to prevent collisions.
- Format: instruction must have one of the legal types presented in Figure 8.

Let us take the example of two-input forward-contractors of *xinterval* presented in Section 5.3. Since they belong to the R-Type, each new instruction must have a unique $\{opcode,\ funct_3,\ funct_7\}$ combination which does not collide with other standard instructions. Field *opcode* acts as a preliminary filter and is encoded using 7 bits. The standard has officially left some opcode values unused (0xB, 0x2B, 0x5B and 0x7B) by built-in extensions to accommodate custom increments (Table 9). An opcode can only regroup instructions of same formats due to micro-architecture limitations but this is not a problem for us since most of our added operators are R-Type with the opcode value fixed (0xB), all our 2-inputs forward contractors must have unique $\{funct_3,\ funct_7\}$ pairs as depicted in Table 10.

Table 9: The opcode space of the RISC-V standard

inst[1:0] = "11" for RV32

inst[6:5] \ inst[4:2]	000	001	010	011	100	101	110	111
00	LOAD	LOAD FP	custom-0	MISC MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE FP	custom-1	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	reserved	custom-2	48b
11	BRANCH	JALR	reserved	JAL	SYSTEM	reserved	custom-3	≥80b

Available for custom extensions

Table 10: Examples of forward-contractor encoding in *xinterval*

instr name	$funct_7$	rs_2	rs_1	$funct_3$	r_d	*opcode*
addfwctc	0000000	-	-	100	-	0001011
subfwctc	0000001	-	-	100	-	0001011
mulfwctc	0000010	-	-	100	-	0001011
divfwctc	0000011	-	-	100	-	0001011

In the source code of *binutils*, the user can add a relationship between an assembly mnemonic (ie the name of the instruction as written in code) and a machine code encoding [30]. This information is used by GCC during the compilation process GCC to produce efficient machine code from the assembly code.

After the patch, RISC-V GCC is now able to compile the following C code using our new instructions (Listing 1). The same process can be repeated for all instructions of *xinterval* to lay the foundations of a hardware-accelerated interval library as presented in Figure 4.

```
/* interval is defined as a double (64 bits) */
typedef interval double;

/* inline function which uses of xinterval instruction addfwctc */
/* inputs loaded from double registers */
/* output stored in double register */
inline interval __attribute__((always_inline))
  _addFwCtc(interval itv1, interval itv2) {

  interval result;;
  asm("addfwctc %0, %1, %2" : "=f"(result) : "f"(itv1), "f"(itv2));
  return result;
}
```

Listing 1: Calling *addfwctc* from C

5.5 Solving the localization problem

In order to solve the localization problem, the robot must compute the intersection of contractors C_1, C_2 and C_3 using the landmark distance measurements. We implemented Algorithms 1 and 2 in C with explicit use of *xinterval* operators to mimic an embedded program running inside a robot. It was then compiled with the modified GCC and executed by the simulated core presented in Section 5.2. (① on Figure 6).

The event simulator shown in ③ on Figure 6 is an external tool which generates coherent sensor measurements to simulate a trajectory between landmarks. These data are periodically written to the robot sensor virtual device to be used by the program running on the ISS. After each localization step, the pose estimate and the SIVIA box are written to dedicated robot devices for *a posteriori* analysis. Figure 9 shows examples of localization paving obtained with our virtual prototype.

6 Results and future works

This paper discussed the advantages of using dedicated hardware to perform interval computations in the context of embedded robotics. This approach differs from the usual one which consists in using software libraries in conjunction to general-purpose hardware. We took advantage of the RISC-V standard to design a custom ISA extension regrouping interval primitives such as forward and backward contractors (Section 5.3). Naturally, the existing software stack and especially the GCC compiler has been modified to expose the new assembly instructions in a high-level language such as C (Section 5.4). Finally, we tested the core in a real-world localization problem using a virtual prototype and performed a successful run of the Algorithm 1 and 2 as shown in Figure 9 (Section 5.5).

As stated in Section 5.1, the goal of this paper was not to implement a full hardware accelerator but to demonstrate the possibility of handling interval primitives directly in hardware. The presented virtual prototype demonstrated the model

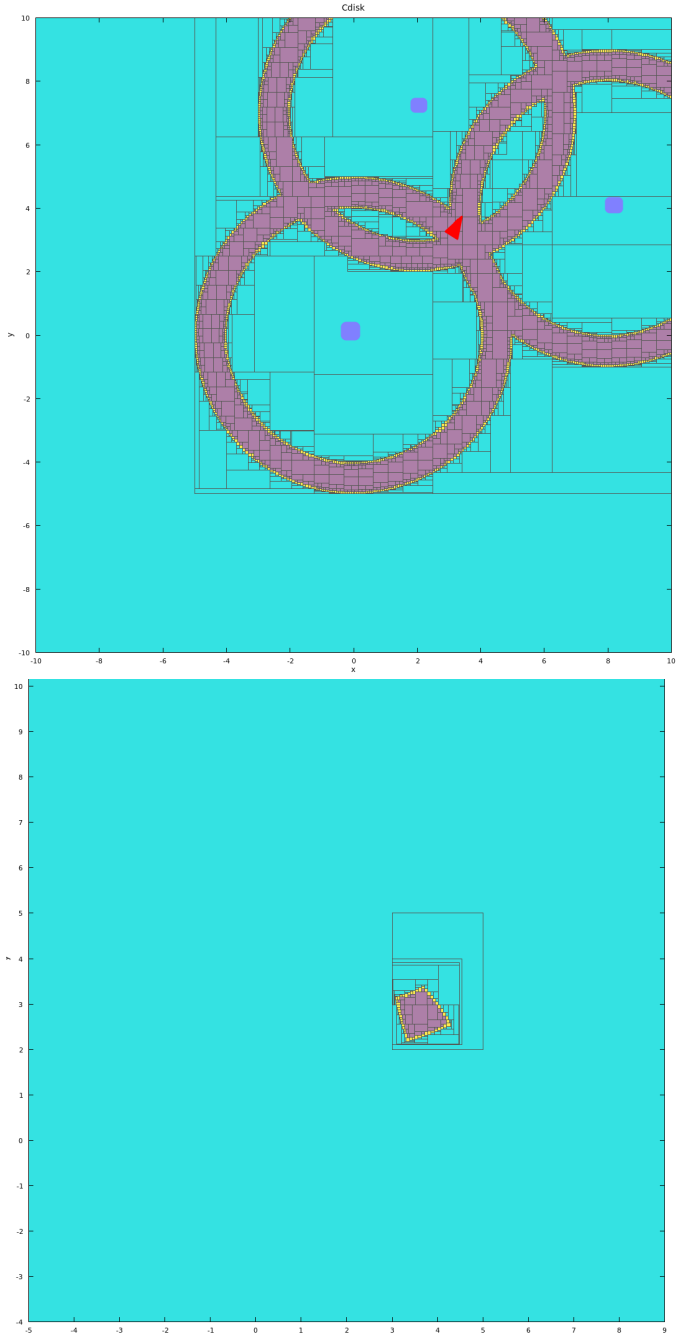

Figure 9: Contractors union and intersection (simulated hardware)

correctness up to the machine code level and is a step toward another refinement of the model to add the RTL layer (under the form of HDL code).

In future works, the focus will be made on the transition between virtual prototype and hardware synthesized on FPGA. The design strategy explained in Section 5.1 will be performed. The main topics are recalled below:

- Synthesis of an existing RISC-V design on a middle-range FPGA. As explained in Section 5, this has been done several times in literature and a lot of designs on shelf are available.

- Execution metrics such as execution time. number of clock cycles and overall frequency will be measured on Algorithm 1 and 2 with only the standard extensions. This process gives a baseline to compare with interval-dedicated instructions.

- Implementation of interval primitives in hardware and integration in RISC-V. The most challenging part is to implement IEEE-754 arithmetic and algebraic operators in an HDL such as VHDL. Once again, such a topic has been widely studied in the literature and the job will be to perform the concatenation of all this work in a dedicated chip. As we target multi-precision, several design trade-offs must be explored for these operators which are mainly hardware resources utilization on FPGA, output frequency and latency.

- Characterization of the performances achieved by the hardware acceleration. We will re-run the Algorithms 1 and 2 (now compiled with *xinterval* support) and compare the results through the prism of the metrics defined in Step 2.

References

[1] Ahmadi-Pour, S., Pieper, P., and Drechsler, R. Virtual-peripheral-in-the-loop : A hardware-in-the-loop strategy to bridge the VP/RTL design-gap. arXiv: 2311.00442, 2023. DOI: 10.48550/arXiv.2311.00442.

[2] Alkim, E., Evkan, H., Lahr, N., Niederhagen, R., and Petri, R. ISA Extensions for Finite Field Arithmetic Accelerating Kyber and NewHope on RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):219–242, 2020. DOI: 10.13154/TCHES.V2020.I3.219-242.

[3] Bandara, S., Ehret, A., Kava, D., and Kinsy, M. BRISC-V: An open-source architecture design space exploration toolbox. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, page 306–306. ACM, 2019. DOI: 10.1145/3289602.3293991.

[4] Chabert, G. IBEX library, 2007. URL: https://github.com/ibex-team/ibex-lib.

[5] Cheng, H., Großschädl, J., Marshall, B., Page, D., and Pham, T. RISC-V instruction set extensions for lightweight symmetric cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(1):193–237, 2022. DOI: 10.46586/tches.v2023.i1.193-237.

[6] C&T Solutions Inc. What is FPGA (Field Programmable Gate Array)? how does it work? URL: https://www.candtsolution.com/news_events-detail/what-is-fpga-field-programmable-gate-array-and-how-does-it-work/ — Accessed: 2024-03-12.

[7] Delafosse, M., Clerentin, A., Delahoche, L., Brassart, E., and Marhic, B. The mobile robot localization problem treated as a constraint satisfaction problem. *IFAC Proceedings Volumes*, 37(8):394–399, 2004. DOI: 10.1016/S1474-6670(17)32008-6.

[8] El-Kurdi, Y., Giannacopoulos, D., and Gross, W. J. Hardware acceleration for finite-element electromagnetics: Efficient sparse matrix floating-point computations with FPGAs. *IEEE Transactions on Magnetics*, 43(4):1525–1528, 2007. DOI: 10.1109/TMAG.2007.892459.

[9] Farnum, C. Compiler support for floating-point computation. *Software: Practice and Experience*, 18(7):701–709, 1988. DOI: 10.1002/spe.4380180709.

[10] Filiol, P., Bollengier, T., Jaulin, L., and Le Lann, J.-C. A new interval arithmetic to generate the complementary of contractors. In Schön, S., editor, *Proceedings of the Summer Workshop on Interval Methods, Hannover, Germany*. Gottfried Wilhelm Leibniz Universität, 2022. URL: https://hal.science/hal-03859346.

[11] Forget, L., Uguen, Y., and de Dinechin, F. Comparing posit and IEEE-754 hardware cost, 2021. URL: https://hal.science/hal-03195756.

[12] Gac, K., Karpiel, G., and Petko, M. FPGA based hardware accelerator for calculations of the parallel robot inverse kinematics. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation*, pages 1–4, 2012. DOI: 10.1109/ETFA.2012.6489717.

[13] Goldberg, D. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991. DOI: 10.1145/103162.103163.

[14] Goualard, F. GAOL — Not just another interval library, 2020. URL: https://github.com/goualard-f/GAOL.

[15] Guyonneau, R., Lagrange, S., Hardouin, L., and Lucidarme, P. Guaranteed interval analysis localization for mobile robots. *Journal on Advanced Robotics*, 28(16):1067–1077, 2014. DOI: 10.1080/01691864.2014.908742.

[16] Gwalani, K. A. and Elkeelany, O. Design and evaluation of FPGA based hardware accelerator for elliptic curve cryptography scalar multiplication. *WSEAS Transactions on Computers*, 8(5):884–893, 2009. URL: https://dl.acm.org/doi/10.5555/1558772.1558786.

[17] Herdt, V., Große, D., Pieper, P., and Drechsler, R. RISC-V based virtual prototype: An extensible and configurable platform for the system-level. *Journal of Systems Architecture*, 109:101756, 2020. DOI: 10.1016/j.sysarc.2020.101756.

[18] IEEE. 1788-2015 — IEEE standard for interval arithmetic, 2015. DOI: 10.1109/IEEESTD.2015.7140721.

[19] IEEE. 754-2019 — IEEE standard for floating-point arithmetic (revision of IEEE Std 754-2008), 2019. DOI: 10.1109/IEEESTD.2019.8766229.

[20] Jaulin, L. Localization of an underwater robot using interval constraints propagation. In Benhamou, F., editor, *Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming*. Springer Berlin Heidelberg, 2006. DOI: 10.1007/11889205_19.

[21] Jaulin, L. and Walter, E. Set inversion via interval analysis for nonlinear bounded-error estimation. *Automatica*, 29(4):1053–1064, 1993. DOI: 10.1016/0005-1098(93)90106-4.

[22] Jin, S., Qi, S., Dai, Y., and Hu, Y. Design of convolutional neural network accelerator based on RISC-V. In Abawajy, J. H., Xu, Z., Atiquzzaman, M., and Zhang, X., editors, *Proceedings of the Tenth International Conference on Applications and Techniques in Cyber Intelligence*, Volume 170 of *Lecture Notes on Data Engineering and Communications Technologies,*, pages 446–454. Springer International Publishing, 2023. DOI: 10.1007/978-3-031-29097-8_53.

[23] Khabarov, S. RISC-V VHDL: System-on-Chip, 2023. URL: https://sergeykhbr.github.io/.

[24] Nehmeier, M. libieeep1788: A C++ implementation of the IEEE interval standard P1788. In *Proceedings of the 2014 IEEE Conference on Norbert Wiener in the 21st Century (21CW)*, pages 1–6, 2014. DOI: 10.1109/NORBERT.2014.6893854, URL: https://github.com/nehmeier/libieeep1788.

[25] Neumaier, A. and Schichl, H. Interval analysis on directed acyclic graphs for global optimization. *Journal of Global Optimization*, 33(4):541–562, 2005. DOI: 10.1007/s10898-005-0937-x.

[26] Nolting, S. The NEORV32 RISC-V processor. Zenodo, 2024. DOI: 10.5281/zenodo.5018888, URL: https://github.com/stnolting/neorv32.

[27] Pieper, P., Herdt, V., and Drechsler, R. Advanced embedded system modeling and simulation in an open source RISC-V virtual prototype. *Journal of Low Power Electronics and Applications*, 12(4), 2022. DOI: 10.3390/jlpea12040052.

[28] Quinnell, R. Creating a custom processor with RISC-V. *EE Times Europe*, 2010. URL: https://www.eetimes.eu/creating-a-custom-processor-with-risc-v/.

[29] Revol, N. The MPFI library: Towards IEEE 1788-2015 compliance. In *Proceedings of the 13th International Conference on Parallel Processing and Applied Mathematics*, number 12044 in LNCS, pages 353–363, 2019. DOI: 10.1007/978-3-030-43222-5_31, URL: https://inria.hal.science/hal-02162346.

[30] Sandid, H. R. Adding custom instructions to the RISC-V GNU-GCC toolchain, 2022. URL: https://hsandid.github.io/posts/risc-v-custom-instruction/.

[31] Waterman, A., Lee, Y., Patterson, D. A., and Asanović, K. The RISC-V instruction set manual, Volume I: User-level ISA, version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html.

[32] Zelensky, A., Alepko, A., Dubovskov, V., and Kuptsov, V. Heterogeneous neuromorphic processor based on RISC-V architecture for real-time robotics tasks. In Dijk, J., editor, *Artificial Intelligence and Machine Learning in Defense Applications II*, Volume 11543, page 115430L. International Society for Optics and Photonics, SPIE, 2020. DOI: 10.1117/12.2574470.